Table des matières

Applications et concepts avancés

Pourquoi ce volume?

Ayant entre autres couvert dans [POOv00] ce que sont les classes et leurs instances, et dans [POOv01] ce que sont l'héritage, le polymorphisme et l'abstraction, le temps est venu d'attaquer quelques pratiques et quelques idées et pratiques plus avancées de la POO.

Ce volume couvrira diverses thématiques avancées de la POO, principalement à l'aide de C++ mais aussi avec Java et C#.

On y verra, entre autres choses :

- la **programmation générique** et les ***templates***;
- ce qu'offrent les **bibliothèques standards** de C++;
- les **objets opaques**;
- les **objets partageables** et les **pointeurs intelligents**;
- quelques schémas de conception tels que le **singleton** et la **fabrique**;
- les **conteneurs standards** et les **itérateurs**;
- quelques **applications de la programmation générique**, dont les **lieurs** et les **expressions lambda**;
- la technique de la **programmation par théorème**;
- les **classes imbriquées**;
- les **foncteurs**;
- les **délégués**;
- les **classes anonymes**; et
- le **contrôle de l'accès aux mécanismes d'instanciation et de destruction**.

Schémas de conception et idiomes de programmation

Ce document, tout comme les documents qui le suivront, présentera un certain nombre de schémas de conception et d'idiomes de programmation. Ces deux termes désignent des pratiques reconnues et répandues, à un point tel qu'elles en sont venues à porter un nom.

⇒ Un **schéma de conception** (en anglais : *Design Pattern*) est une pratique applicable à la vaste majorité des langages de programmation (typiquement, des langages OO).

⇒ Un **idiome de programmation** est une pratique applicable aux langages de programmation respectant certaines prémisses. Pensez à l'idiome RAII, présenté dans [POOv00] et exploité dans [POOv01], qui requiert le support à des mécanismes de finalisation déterministes.

Nous identifierons ces pratiques lorsque nous les rencontrerons. Les schémas de conception et les idiomes de programmation représentent un langage commun dans le monde du développement logiciel, une codification des pratiques. Reconnaître et savoir nommer ces pratiques est un atout pédagogique qu'il ne faut pas négliger.

À propos de la forme

Ce document, plus sophistiqué que ses prédécesseurs [POOv00] et [POOv01], explore diverses avenues et pratiques utiles en POO, mais tend à rester près des bases tout en visant l'application efficace de la POO. Les volumes ultérieurs de cette série poursuivront l'exploration de thématiques plus poussées.

Vous trouverez à divers endroits des petits encadrés indiquant ***Réflexion 02.n*** (pour divers entiers ***n***). 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 00 – 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.

Programmation générique et *templates*

Cette section se veut une description opérationnelle de ce qu'est un modèle, ou ***template*** en C++.

Le thème des *templates* ne constitue pas tant un sujet de l'approche OO qu'un complément logique de cette approche, dans la mesure où elle inclut l'équivalence opérationnelle des types : le développement de classes et d'algorithmes dont les types sont paramétriques.

De manière plus générale, on devrait parler de **programmation générique**. Les *templates* sont l'implémentation C++ du concept, mais Java offre une version différente, à certains égards moins large, du concept (les *Generics*) depuis la version 5, et C# fait de même depuis sa version 2 en adaptant entre autres certains éléments des travaux de C++ 0x sur les ***concepts*** [POOv03].

Combiner POO et généricité permet une généralisation rien de moins que fabuleuse, mais soulève toute une gamme de problèmes nouveaux. L'importance de l'équivalence opérationnelle des types sera mise en relief par le fait que nous écrirons des types et des algorithmes applicables à *quelque chose*, que ce quelque chose soit primitif ou qu'il s'agisse d'une classe. La POO contemporaine va de pair avec la généricité, et ce couple conceptuel ouvre à la pensée des possibilités au préalable insoupçonnées.

Note :	c'est à contrecœur que j'utiliserai le terme anglais *template* dans ce document; les équivalents français (modèle, patron) sont un peu galvaudés, ce qui risquerait (à mon avis) d'introduire de la confusion dans le texte.

L'idée derrière les templates

Les *templates* sont d'abord et avant tout des classes et des sous-programmes conçus pour des ***types paramétriques***. L'idée de types paramétriques date de bien avant la norme ISO (1998), à partir de laquelle elle fit son entrée dans le standard officiel du langage, mais n'avait pas été bien testée et validée auparavant. Le concept est délicat, et son implantation devait être solide.

C++ est un langage dit ***fortement typé***, qui met fortement l'accent sur le volet statique de la programmation : mieux le travail est fait à la compilation et moins d'effort il restera à déployer à l'exécution. Comme pour la plupart des langages OO, avec des hiérarchies de classes, le contrôle strict des types y est primordial.

Ne confondez pas fortement typé, qualificatif qui sied pleinement C++, et *Type Safe*, qui ne lui sied pas. En effet, tel que vu dans [POOv01], C++ permet de convertir un pointeur (typé) en adresse brute (`void*`), totalement abstraite, et permet de ce fait de mentir sur la nature des objets, ce que ne permet pas Java par exemple. Ceci n'est cependant permis que suite à un transtypage explicite.

Un type paramétrique typique est un conteneur, par exemple une pile de *quelque chose* ou une liste de *quelque chose*. Un algorithme générique typique serait trouver le plus petit (au sens de l'opérateur <) d'entre deux éléments de même type, ou trouver l'élément d'un type donné qui satisfait un certain prédicat dans une séquence d'éléments de même type.

Énoncé en ces termes, on ne se surprendra pas que la bibliothèque standard de C++ regorge de conteneurs (piles, listes, files, tableaux dynamiques) et d'algorithmes (tris, recherches, partitionnements) génériques. Ces composants logiciels de base sont ceux à partir desquels les programmes efficaces sont construits.

Il existe des opérations relativement élémentaires qui s'implantent très mal dans un contexte non générique.

En fait, la suivante (aussi pénible en C qu'en C++, d'ailleurs) est un cas concret et commun de fonction simple qui ne s'implante pas proprement :

```
int a = 3, b = 5, c;
float fa = 3, fb = 5, fc;
// impossible sous forme de fonction en
// C, pénible en C++ sans les templates
c = min(a, b);
fc = min(fa, fb);
```

Attention : ici, le mot *impossible* dans le commentaire à droit de l'appel signifie impossible à implanter sous forme de fonction pour tous les types. En effet, en langage C, on ne peut rédiger deux fonctions ayant le même nom mais des paramètres différents, en type ou en nombre. En C++, il est possible (et essentiel) qu'on puisse le faire.

On ne peut donc pas avoir, en langage C, les deux fonctions suivantes dans le même programme :

```
int min(int a, int b) {
   int resultat;
   if (a < b)
      resultat = a;
   else
      resultat = b;
   return resultat;
}
```

```
float min(float a, float b) {
   float resultat;
   if (a < b)
      resultat = a;
   else
      resultat = b;
   return resultat;
}
```

```
// plus concis
int min(int a, int b) {
   return a < b? a : b;
}
```

```
// plus concis
float min(float a, float b) {
   return a < b? a : b;
}
```

En langage C++, même en omettant la programmation générique, on peut y arriver. Toutefois, si on veut vraiment avoir une fonction `min(a,b)` qui fonctionne pour chaque type pertinent, il faut écrire la fonction propre à chacun de ces types. Or, *on parle à chaque fois du même code, où seuls les types de données impliqués changent*.

Solution traditionnelle : les macros

La solution traditionnelle, en langage C comme en langage C++, était d'avoir recours à une macro-instruction, plus souvent appelée ***macro***.

Un exemple de macro est donné à droite, où `min(a,b)` exprime le plus petit, au sens de l'opérateur `<`, de deux éléments `a` et `b`.

```
#define min(a,b) \
   (((a)<(b))?(a):(b))
```

Les macros étant sous la responsabilité du préprocesseur, pas du compilateur, elles consistent en un simple remplacement lexical (et paramétrique) dans le code. Ainsi, le code proposé à droite…

```
int main() {
   int i = 3, j = 5;
   int k = min(i, j);
}
```

…devient, une fois complétée la tâche du préprocesseur, le code proposé à droite. Les macros constituaient donc la méthode passe-partout de contourner ces contraintes particulières du langage C.

```
int main() {
   int i = 3, j = 5;
   int k = (((i)<(j))?(i):(j));
}
```

Avantages des templates *sur les macros*

Il y avait (et il y a toujours, d'ailleurs) des risques, sérieux et nombreux, associés au recours à des macros, le principal étant que les macros agissent à l'insu du compilateur et à l'extérieur des règles du langage. Le préprocesseur modifie le *texte* des programmes; il n'a pas (ou, à vrai dire, n'a que très peu) d'intelligence propre au langage lui-même, et n'a *surtout* pas d'intelligence propre aux types.

Ainsi, la ligne proposée à droite passe sans peine l'étape du préprocesseur mais n'a aucun sens (ou, du moins, n'a pas le sens attendu, quel qu'il soit) en langage C.

```
int *p = min("Allo!",22);
```

De plus, les macros, en transformant les sources, en viennent à générer du code carrément épouvantable à déverminer (et à optimiser, quand elles génèrent beaucoup de code). Pour éviter les ennuis, on tend à enrober chaque paramètre de parenthèses[1], ce qui fait que le code intermédiaire généré par le préprocesseur est bardé de parenthèses, rendant difficile la tâche d'y repérer les erreurs.

La solution sous forme de *template* offre le contrôle des types à la compilation, ce qui constitue un avantage immense sur les macros, tout en permettant une fonction qui :

- sera capable d'opérer peu importe les types impliqués, dans la mesure où les contraintes du *template* en question sont respectées;
- ne demandera pas d'écrire le code de fonctions pour chaque type et chaque combinaison de types possibles;
- n'impliquera pas plus de charge administrative (*Overhead*) à l'exécution qu'une fonction aux types non paramétriques.

Les *templates* constituent un levier de génération de code, un peu comme le sont les macros, mais agissent à l'intérieur du système de types et des règles du langage.

[1] Par exemple, on préférera `((a)<(b)?(a):(b))` à `(a<b?a:b)`.

Rédiger un template

Pour rédiger un *template* de la fonction `min()`, il faut en analyser la fonctionnalité attendue. Ici, à titre d'illustration, nous viserons une fonction qui :

- recevra en paramètre deux entités d'un même type (deux `int`, deux `float`, deux `std::string` si on vise une comparaison lexicographique, *etc.*);
- vérifiera, à l'aide de l'opérateur `<` défini sur ce type, laquelle des deux est la plus petite (ou, pour être plus précis, laquelle précède l'autre);
- affectera à une variable temporaire du même type, à l'aide de l'opérateur `=` défini sur ce type, la valeur du plus petit des deux (de celui qui précède l'autre); et
- retournera cette variable temporaire. Il faudra donc que le type retourné par la fonction soit le même que celui de ses paramètres.

À partir de cette analyse, nous sommes en mesure de constater qu'il n'y a en tout temps qu'un seul type (nommons-le ***T***, pout *type*) qui entre en jeu[2]. C'est précisément de ce type ***T*** que nous voulons faire abstraction, tout en conservant la fonctionnalité derrière la fonction `min()`.

Ainsi, le principe de la fonction `min()` ressemblerait au pseudocode visible à droite, exprimé en indiquant (contrairement aux usages) aux endroits opportuns un type générique *T*.

```
T min(T a, T b)
   T Résultat;
   si a < b
      Résultat ← a;
   sinon
      Résultat ← b;
   retourner Résultat;
```

En identifiant par ***T*** le type utilisé, on conserve précisément la fonctionnalité voulue (remplacez systématiquement ***T*** par `int`, par exemple, et vous retrouverez la fonction `min()` applicable à des `int`). Remarquez que nous déplaçons la réflexion et l'analyse à un niveau d'abstraction plus élevé : peu importe le type utilisé ici, dans la mesure où ce type supporte les opérations que nous lui appliquons.

Si `T` est une classe, par exemple, nous aurons besoin :

- d'un constructeur par défaut (définition de la variable `Résultat`);
- de la comparaison à l'aide de l'opérateur `<`, dont le résultat sera booléen (à cause de son utilisation dans une alternative);
- de l'affectation; et
- de la construction par copie (pour retourner le résultat de l'exécution de la fonction).

La démarche que nous entreprenons ici est une démarche **générique**.

Notre algorithme `min()` est applicable à tout type ***T*** pour lequel les opérations dont il se sert sont définies.

[2] Notez qu'un *template* impliquant plusieurs types est tout à fait légal; ce n'est tout simplement pas le cas dans notre petit exemple, puisqu'on n'appliquera habituellement pas `min(a,b)` si `a` et `b` sont de types différents. Il y a cependant des nuances possibles à l'aide de techniques plus avancées (comme admettre des instances de deux types `T` et `U` si un `U` peut être implicitement converti en un `T`), que nous ne couvrirons pas ici mais qui deviendront accessibles à partir de techniques abordées dans ***POO—Volume 03***.

La syntaxe exacte, en C++, de l'implantation par *template* de la fonction `min()` décrite par le pseudocode plus haut est celle proposée à droite.

```
template <class T>
   T min(T a, T b) {
      T resultat;
      if (a < b)
         resultat = a;
      else
         resultat = b;
      return resultat;
   }
```

Notez les termes en caractère gras :

- le mot clé **`template`** signifie que ce qui suit sera un *template* au sens C++, c'est-à-dire une unité de code faisant appel à des ***types paramétriques*** – une unité de code influencée par les types en fonction desquels elle sera générée;
- la mention **`<class T>`** signifie que `T`, tel qu'utilisé dans cette unité de code, sera un type paramétrique. *Le mot `class` signifie ici que `T` est un type servant de paramètre au `template`*, et non pas qu'il s'agit d'une classe[3]. Notre algorithme `min()` fonctionnera aussi très bien avec des `int` et des `float`;

Une notation alternative (et parfois nécessaire) utilise **`typename`** au lieu de `class`, mais les deux fonctionnent de manière identique dans les cas qui nous intéresseront ici[4].

- ainsi, partout dans ce *template* où on retrouvera le symbole `T`, on pourra remplacer `T` par n'importe quel type, dans la mesure où il s'agit du même type dans tous les cas pour une seule et même utilisation[5]. Dans la version proposée ici, `min(3,4.3)` serait un appel illégal[6] car le paramètre de gauche est un `int` et le paramètre de droite est un double, mais notre algorithme utilise le même type, `T`, pour les deux paramètres que sont `a` et `b`;
- le `T` qui précède le mot `min` signifie que la fonction retournera une valeur du type signifié par `T`. Le `T` qui précède `a` et `b` est porteur du même sens : il indique qu'il s'agit dans chaque cas du même type, peu importe lequel. Ceci s'applique aussi au `T` précédant le symbole `Resultat`, qui sert à déclarer `Resultat` comme étant du même type que `a`, `b` et que la valeur retournée par la fonction.

Nulle part dans cette spécification de *template* peut-on voir ce qu'est le type `T`. Ce n'est que lorsque l'algorithme `min()` sera utilisé concrètement, donc lorsqu'il sera appliqué à des entités d'un certain type, qu'il deviendra possible de générer le code lui correspondant.

Un *template* n'est pas du code mais bien un guide, un modèle permettant à un compilateur de générer du code lorsque les types auxquels ce code sera appliqué seront connus. Si personne ne s'en sert, alors aucun code ne sera généré pour lui.

---

[3] Les concepteurs du langage ont jonglé avec l'idée d'introduire un nouveau mot clé au langage pour ceci, et ont fini par juger que cela n'était pas nécessaire.

[4] Il y a une distinction entre `class` et `typename` dans la programmation à l'aide de *templates*. Nous l'examinerons un peu plus loin mais elle deviendra plus importante pour nous dans [POOv03].

[5] Si vous êtes familière/ familier avec le *λ-calcul*, idée liée de près au développement de l'informatique, vous reconnaîtrez ici son idée maîtresse qu'est celle de *substitution*.

[6] Avec une macro, ce serait légal, mais beaucoup plus subtil à analyser et à déboguer… Quel serait le type de l'expression « `((3)<(4.3)?(3):(4.3))` »? Serait-ce `int`? Serait-ce `double`? Il y a une réponse à cette question, mais est-ce que nous souhaitons que notre code repose sur des trucs aussi obscurs?

Le programme à droite est un exemple complet de déclaration et d'utilisation d'une fonction *template* simple. Le compilateur rencontrera deux utilisations de l'algorithme générique `min()`, l'un prenant des `int` en paramètre et l'autre prenant des `float`, et générera le code à la compilation pour prendre en charge ces types.

```
template<class T>
   T min(T a, T b) {
      T resultat;
      if (a < b)
         resultat = a;
      else
         resultat = b;
      return resultat;
   }
int main() {
   int i0 = 3, i1 = 5;
   float f0 = 12.5f, f1 = -27.3f;
   // min(int.int) retourne un int
   int i = min(i0, i1);
   // min(float,float) retourne un float
   float f = min(f0, f1);
}
```

Suite à la compilation, donc, le *template* n'existera plus, mais on trouvera à sa place deux fonctions globales, soit :

```
int min(int,int);
float min(float,float);
```

Il est aussi possible de déclarer un *template* à l'aide d'un prototype puis de le définir ultérieurement. Dans ce cas, on obtiendra quelque chose comme le code proposé à droite.

```
// prototype
template<class T>
   T min(T, T);
int main() {
   int i = min(3, 5);
   double d = min(12.5, 3.5);
}
template<class T>
   T min(T a, T b) {
      T resultat;
      if (a < b)
         resultat = a;
      else
         resultat = b;
      return resultat;
   }
```

Bémol d'importance : avec un *template*, seul le compilateur est sollicité[7]. Ainsi, on ne peut placer un prototype ou une déclaration de *template* dans un fichier d'en-tête et son implémentation dans un fichier source, puisque l'éditeur de liens n'est pas conscient de l'existence même de *templates*.

Avant C++ 11, il existait un mot clé, **`export`**, destiné à permettre la définition d'un *template* dans un fichier source distinct de celui dans lequel il est utilisé. Ce mot clé a été peu implémenté par les compilateurs contemporains dû à des difficultés techniques extrêmement importantes, alors la plupart des gens définissent les *templates* directement dans les fichiers d'en-tête. Depuis C++ 11, le mot clé `export` et ce qu'il devait supporter sont considérés non seulement dépréciés, mais ne sont tout simplement plus supportés (le mot clé `export` demeure réservé mais ne fait plus partie du standard), alors évitez de les utiliser.

[7] Une des règles d'or de C++ est que les éditeurs de liens simples, utilisés avec des modules rédigés en langage C, doit suffire à résoudre les liens d'un programme C++. C'est pourquoi le compilateur C++ lui-même doit savoir comment résoudre les appels de sous-programmes les classes à types paramétriques dès la compilation.

Généricité et contrats

Exprimer un algorithme ou une classe sous forme générique implique la réalisation d'opérations sur des entités dont on ne connaît pas les types au préalable. C'est de cette abstraction que vient le qualificatif ***générique*** donné à ce type de programmation.

Les opérations utilisées dans ces algorithmes présument certaines particularités de ces types, particularités qui varieront d'un algorithme à l'autre. On peut par exemple penser à :

- la possibilité de générer une copie d'une instance du type;
- la capacité de créer une instance par défaut du type;
- la capacité de comparer deux instances du type à l'aide de l'opérateur `<`;
- la capacité d'affecter une instance du type à une autre;
- la capacité d'invoquer une méthode portant un nom précis; *etc.*

⇒ Ces contraintes qui doivent être respectées par tout type auquel s'appliquera un algorithme générique donné constituent le **contrat** de cet algorithme.

Le mot *contrat* a aussi un sens plus technique; nous l'utilisons ici dans une acception informelle

⇒ Les **concepts**, une idée clé des travaux menant au standard C++ 17, sont un formalisme important de ce que nous entendons ici à titre de contrats sur les types génériques [POOv03].

Plusieurs contrats sont possibles pour un même algorithme, selon l'implémentation choisie, d'où l'importance de détailler le contrat d'un algorithme donné à même sa documentation.

Prenant à titre d'exemple l'algorithme générique proposé à droite (version `A`), le contrat de `T` est très simple : il doit être possible de lui appliquer l'opérateur `-` unaire (donc à un seul opérande) et `T` doit supporter l'affectation.

```
// version A
template <class T>
   void inverser_signe(T &obj) {
      obj = -obj;
   }
```

La version `B` impose un contrat différent : il doit être possible d'appliquer à `T` l'opérateur `*=` prenant comme opérande de droite un entier signé.

```
// version B
template <class T>
   void inverser_signe(T &obj) {
      obj *= -1;
   }
```

Le contrat de `T` pour la version `C` de l'algorithme est qu'il doit être possible d'appliquer à `T` l'opérateur `*` prenant comme opérande de droite un entier signé et que le résultat de cette opération doit pouvoir être affectée à un `T`.

```
// version C
template <class T>
   void inverser_signe(T &obj) {
      obj = obj * -1;
   }
```

La version `D` impose encore un autre contrat : `T` doit offrir un constructeur par copie (utilisé deux fois, soit une pour le passage du paramètre par valeur et une autre pour générer la valeur de retour) et on doit aussi être capable de lui appliquer l'opérateur `-` unaire.

```
// version D
template <class T>
   T inverser_signe(T obj) {
      return -obj;
   }
```

Détails d'implantation

Derrière chaque ajout au langage C++, on trouve des préoccupations d'efficacité en temps et en espace. Pour que les *templates* soient acceptés dans la norme du langage, il fallait qu'ils offrent un niveau de performance équivalent (ou supérieur) à leurs équivalents non génériques, et que leur coût en espace soit aussi petit que possible.

Dans le cas d'un *template*, qui doit être compilé si on veut qu'il soit d'exécution rapide, il est impossible de connaître *a priori* toutes les utilisations possibles qu'on puisse en faire. Aussi, le code propre à une utilisation précise (une combinaison de types spécifique) de *template* ne sera généré que si au moins une utilisation en est faite.

Dans le cas de nos exemples plus haut, le compilateur aura généré le code pour `float min(float,float)` et pour `int min(int,int)`, mais pas pour `string min(string,string)`. Comme à l'habitude, avec C++, un programme ne paie que pour ce qu'il utilise.

Il est possible de combiner des types paramétriques et des types non paramétriques dans une même unité de code.

Par exemple, le code à droite déclare une classe `Tampon` contenant un tableau de type paramétrique `T` de `N` éléments, avec `N` un entier. L'attribut d'instance `nelems` indique le nombre d'éléments, alors que `N` est la capacité d'un `Tampon<T,N>`.

Ici, la méthode `capacite()` est une méthode de classe puisque tous les `Tampon<T,N>` sont du même type et partagent la même capacité (mais pas les mêmes données).

```
template<class T, int N>
   class Tampon {
      T valeurs[N] {};
      int nelems = 0;
   public:
      Tampon() = default;
      static int capacite() noexcept {
         return N;
      }
      int size() const noexcept {
          return nelems;
      }
      // ...
   };
```

Avec cette déclaration de la classe paramétrique `Tampon`, il est possible de déclarer des instances de `Tampon` de différents types et de différentes tailles. Par exemple, observons le programme à droite, qui déclare `tf` comme étant un `Tampon` de `20 float` et `ts` comme étant un `Tampon` de `15 string`.

Remarquons au passage que, contrairement à l'utilisation du *template* `min(a,b)` plus haut, nos déclarations des instances `tf` et `ts` incluent une mention explicite présentant pour quel type chacune de ces instances de `Tampon` sera créée.

```
#include <string>
int main() {
   using std::string;
   Tampon<float,20> tf;
   Tampon<string,15> ts;
   // ...
}
```

Une fois déclarées, ces instances de `Tampon` peuvent être utilisées comme l'est n'importe quel objet, dans la mesure où les types impliqués sont respectés.

Chaque ensemble de types paramétriques appliqués à un *template* donné crée un type distinct. Pour donner un exemple concret, les types `Tampon<int,3>` et `Tampon<int,4>` sont deux types différents. Ceci influence la programmation et a entraîné le développement de techniques propres à la programmation générique.

Exemples typiques d'utilisation de templates *de fonctions et de classes*

Examinons quelques cas où l'application de *templates* s'avère toute désignée.

Les fonctions `min(a,b)` et `max(a,b)`, qui identifient et retournent respectivement le plus petit d'entre `a` et `b` et le plus grand d'entre `a` et `b`. Les écritures choisies à droite ont des contrats minimalistes, n'exigeant que la construction par copie et la comparaison avec l'opérateur relationnel approprié; comparez ces contrats avec celui de la version utilisée en exemple, un peu plus haut.

```
template<class T>
   T min(const T &a, const T &b) {
      return a < b? a : b;
   }
template<class T>
   T max(const T &a, const T &b) {
      return a > b? a : b;
   }
```

Les fonctions `std::min()` et `std::max()` sont livrées avec le standard, alors utilisez-les plutôt que de pondre vos propres versions (à moins d'avoir une *excellente* raison de ne pas le faire).

La fonction `std::swap(a,b)`, qui échange le contenu de `a` et de `b`. Une version officielle fait d'ailleurs partie des bibliothèques standards de C++. Cette procédure, en plus d'être un exemple académique des plus classiques, est utile dans la plupart des algorithmes de tri.

```
template<class T>
   void swap(T &a, T &b) {
      T temp = a;
      a = b;
      b = temp;
   }
```

Un raffinement technique important de C++ 11, les **références sur des *rvalues***, permet une amélioration de performance importante sur des fonctions fondamentales comme `std::swap()`. Voir ***Sémantique de mouvement*** pour plus d'informations à ce sujet.

Sachant cela, si le choix s'offre à vous d'utiliser des versions standards des fonctions essentielles ou d'écrire vos propres versions, pensez-y longuement et faites des tests pour vous assurer que le jeu en vaille la chandelle.

Une classe utilitaire `Cumulateur`, capable de cumuler des valeurs d'un type donné (dans la mesure où ce type possède un opérateur `+=` et un constructeur de copie) dans une variable qui lui est fournie par référence à la construction.

Cette classe a de particulier qu'elle ne dessert que la fonctionnalité cumulative, et ne se charge pas de la gestion de la vie de la variable qui sert à cumuler les valeurs.

Cette classe est opérationnelle mais, en pratique, la sémantique qu'elle propose n'est pas optimale pour plusieurs tâches courantes. Nous verrons une stratégie plus sophistiquée pour réaliser la même tâche dans la section ***Foncteurs***, un peu plus loin dans le présent document.

```
template <class T>
   class Cumulateur : Incopiable {
      T &cumul;
   public:
      Cumulateur(T &var) noexcept
         : cumul{var}
      {
      }
      void ajouter(const T &val) {
         cumul += val;
      }
      T valeur() const {
         return cumul;
      }
   };
```

Une classe `Pile`, qui permet d'empiler et de dépiler des objets de même type, implantée à l'aide d'un tableau. Une solution reposant sur de l'allocation dynamique de mémoire et permettant d'empiler un nombre arbitraire d'éléments est un exercice divertissant à écrire; je vous invite à le faire, ou (mieux à long terme) à examiner la classe `std::stack` de la bibliothèque standard (fichier d'en-tête `<stack>`).

Le code à droite présente à la fois une classe permettant de créer une pile de type paramétrique, avec un nombre maximum d'éléments pouvant être spécifié à la déclaration (`100` par défaut), et deux exemples (plus bas) de déclaration d'une `Pile` paramétrique sur la base de types différents.

Notez que les méthodes `empiler()` et `depiler()`, si simples soient-elles en apparence, ne sont pas qualifiées `noexcept` du fait qu'elles utilisent l'affectation sur un type `T` inconnu au préalable. Comment assurer que cet opérateur ne lèvera pas d'exception?

Ne sachant pas sur quels types `T` le type générique `Pile<T,MAX>` sera utilisé, il est impossible de garantir quoi que ce soit quant aux opérations propres au type `T`. On peut par contre offrir des garanties sur des `T*` ou sur des `T&` car les pointeurs et les références sont des types primitifs.

```
class Oups {};
template<class T, int MAX=100>
   class Pile {
      T contenu[MAX] {};
      int nelems {};
   public:
      Pile()= default;
      bool plein() const noexcept {
         return nelems==MAX;
      }
      bool vide() const noexcept {
         return !nelems;
      }
      void empiler(const T &elem) {
         if (plein()) throw Oups{};
         contenu[nelems+1] = elem;
         ++nelems;
      }
      void depiler(T &elem) {
         if (vide()) throw Oups{};
         elem = contenu[nelems-1];
         --nelems;
      }
   };
```

Vous avez peut-être remarqué que la généricité offre, sur une base statique, une souplesse similaire à ce que le polymorphisme offre sur une base dynamique. Combiner types génériques et types polymorphiques est une combinaison très expressive; pensez, ici, à une `Pile<Forme*>` où `Forme` serait une abstraction…

```
int main() {
   Pile<float,50> PileFloat;
   Pile<char> PileChar;
}
```

Constructeur par défaut d'un type primitif

Appliquer efficacement des techniques de programmation générique demande un système de types très homogène.

Pour cette raison, tel que vu dans [POOv00], C++ considère que les types primitifs ont un constructeur par défaut, qui insère dans la donnée l'équivalent de zéro pour son type (donc `0` pour un `int`, `0.0` pour un `double`, `false` pour un `bool`, `nullptr` pour un pointeur, *etc*.).

Pour des raisons de compatibilité avec C, il est important de solliciter explicitement ce constructeur. En effet, déclarer une donnée d'un type primitif n'entraîne, comme le veut la tradition, aucune initialisation.

Dans le code à droite, lors de la construction de l'instance de `Cumulateur` nommée `c0`, l'attribut `cumul_` est initialisé à zéro. Conséquemment, la valeur résultant du cumul d'entiers sera `45`. Si nous avions voulu conserver un produit plutôt qu'une somme, cette valeur initiale aurait été bien mal choisie.

Évidemment, il est possible de cumuler pour tout type respectant le contrat proposé par la classe générique `Cumulateur`. À droite, le `Cumulateur` nommé `c1` cumule, par concaténation, dans une `std::string` à l'aide de l'opérateur `+=` exposé par ce type.

```
template <class T>
   class Cumulateur {
      T cumul {};
   public:
      Cumulateur() = default;
      void ajouter(const T &val) {
         cumul += val;
      }
      T valeur() const {
         return cumul;
      }
   };
#include <iostream>
#include <string>
int main() {
   using namespace std;
   Cumulateur <int> c0;
   for (int i = 0; i < 10; ++i)
      c0.ajouter(i);
   cout << c0.valeur() << endl;
   Cumulateur<string> c1;
   c1.ajouter("Coucou ");
   c1.ajouter("les ");
   c1.ajouter("amis!");
   cout << c1.valeur() << endl;
}
```

Ici, une instruction comme `c1.ajouter("Coucou ")` reçoit en fait un paramètre de type `const char*`, qui sera converti en `std::string` pour réaliser l'appel parce que la méthode `ajouter()` de `Cumulateur<T>` prend un `const T&` en paramètre, donc `const std::string&` pour un `Cumulateur<string>`.

Méthodes génériques

Il est aussi possible avec C++ d'avoir, dans un type, qu'il soit générique ou non, une ou plusieurs méthodes génériques sur la base d'autres types.

Dans l'exemple à droite, on pourrait avoir une instance de `Cumulateur<int>` et appeler sa méthode `ajouter()` en lui passant autre chose qu'un `int`, par exemple un `double`, un `short` ou tout autre type pouvant être ajouté à un `int` par l'opérateur `+=` qui y est utilisé.

La méthode `Cumulateur<int>::ajouter(double)` sera générée par le compilateur parce qu'on l'aura utilisée, tout simplement.

```
template <class T>
   class Cumulateur {
      T cumul {};
   public:
      Cumulateur() = default;
      template <class U>
         void ajouter(U val) {
            cumul += val;
         }
      T valeur() const {
         return cumul;
      }
   };
```

Types internes et publics (brève introduction)

Parmi les pratiques répandues de la programmation générique, on trouve celle de documenter à même les types les pratiques et politiques internes.

Reprenant l'exemple de la classe `Pile<T,N>` plus haut et adaptant (en partie) ce type aux usages répandus de la programmation générique, on ajoutera des types internes et publics à la classe pour définir aux vu et au su du code client les types à utiliser pour exprimer certaines abstractions internes de cette classe.

Ici, suivant les conventions, nous utiliserons `value_type` pour représenter la valeur contenue dans une `Pile` et nous utiliserons `size_type` pour indiquer le type choisi pour y représenter les tailles (ici, cela inclura à la fois la représentation de la capacité d'une `Pile` et du nombre effectif d'éléments qui y sont entreposés).

Cette pratique a énormément d'avantages :

- elle documente les types (tout conteneur conforme utilisera un type interne `value_type` pour représenter les valeurs qu'il entrepose);
- elle donne une permanence interne à des idées qui n'apparaissent autrement qu'à la compilation (ici, `T` est connu à la compilation seulement mais le type interne `value_type` peut servir pour exprimer l'idée de valeur entreposée dans ce conteneur partout dans le programme);
- elle invite le code client à s'exprimer selon des abstractions contrôlées par l'objet, ce qui facilite énormément l'entretien du code. C'est pourquoi ces types sont publics : on souhaite que le code client les privilégie!

```
class Oups {};
template<class T, int N=100>
   class Pile {
   public:
      using value_type = T;
      using size_type = int;
   private:
      value_type elems[N] {};
      size_type nelems {};
   public:
      Pile() = default;
      size_type size() const noexcept {
         return nelems;
      }
      bool plein() const noexcept {
         return size()==N;
      }
      bool vide() const noexcept {
         return !size();
      }
      void empiler(const value_type &elem) {
         if (plein()) throw Oups{};
         elems[nelems] = elem;
         ++nelems;
      }
      void depiler(value_type &elem) {
         if (vide()) throw Oups{};
         elem = elems[nelems-1];
         --nelems;
      }
   };
```

Ceci n'est évidemment qu'une brève introduction à l'idée de types internes et publics; nous y reviendrons ultérieurement puisque cette pratique s'insère dans une gamme de techniques avancées extrêmement puissantes qu'on nomme la métaprogrammation. Nous faisons mention ici de la pratique puisqu'elle est omniprésente dans la bibliothèque standard du langage, que nous couvrirons dans [POOv03].

Mot clé `typename` et la désambiguïsation du code générique

Imaginons le code proposé à droite. Il s'agit d'un exemple pervers[8] mais possible, dans lequel une déclaration délibérément ambiguë est construite pour illustrer un point.

```
template <class T>
   struct X {
      using t = T;
   };
// spécialisation de X
// pour le type void
template <>
   struct X<void> {
      static const int t = 3;
   };
template <class T>
   class Y {
      int y;
   public:
      void f() {
         X<T>::t * y;
      }
   } ;
```

La classe générique `X<T>` en général définit un type interne `X::t` équivalent à `T` (donc, pour `X<int>`, `X<int>::t` sera `int`).

Cependant, une spécialisation de `X` pour le type `void` définit le nom `t` comme étant une constante de classe. Une spécialisation est un cas particulier à privilégier dans certains cas pointus (ici, lorsque `T` est `void`). Voilà deux sémantiques distinctes pour un même nom, selon le contexte d'utilisation. Rien de gentil ou de recommandable, nous l'admettrons, mais cela reste une possibilité.

Examinons maintenant la classe `Y` plus bas dans laquelle on trouve une méthode d'instance `f()` à l'intérieur de laquelle apparaît une seule ligne. Mettons-nous maintenant dans la peau d'un compilateur : devrions-nous générer une déclaration de pointeur (si `X<T>::t` est un type) ou une multiplication (si `X<T>::t` est une constante)?

Voilà précisément le genre de question qui brime le sommeil des auteurs de compilateurs : le compilateur n'est pas en mesure de décider du code à générer ici sans connaître le type `T`, et ce savoir ne sera disponible qu'au moment de l'utilisation de `Y` avec un type `T` précis. Échec et mat, en quelque sorte.

Un dira de `X<T>::t` qu'il s'agit d'un **nom dépendant** puisque `X<T>::t` est un nom défini dans une entité générique `X<T>` et peut, conséquemment, varier selon le type `T`. Nous faisons face à un problème d'ambiguïté face aux noms dépendants.

```
template <class T>
   class Y {
      int y;
   public:
      void f() {
         typename X<T>::t * y;
      }
   } ;
```

La solution choisie en C++ pour résoudre ce problème est la suivante : si un nom dépendant dénote un type, alors il *doit* être précédé du mot clé **`typename`**. Dans l'exemple ci-dessus, donc, on parle bien d'une multiplication (valide si `T` est `void` mais pas pour un `T` quelconque), alors que le cas à droite est une déclaration de pointeur (illégale si `T` est `void` puisque, dans un tel cas, le nom `X<T>::t` ne correspond pas à un type).

[8] Exemple emprunté sans vergogne à ***David Vandevoorde*** et à ***Nicolai M. Josuttis*** dans [CppTemp].

Le principe SFINAE[9]

Un principe fondamental de la programmation générique à l'aide de *templates* en C++ est le principe `SFINAE`, qui signifie *Substitution Failure is not an Error*. Par *substitution*, on entend une tentative par le compilateur de trouver la bonne combinaison de types et de valeurs à à appliquer à un cas d'utilisation d'un *template* donné.

Le principe `SFINAE` sert à plusieurs applications sophistiquées de **métaprogrammation**, un sujet avancé que nous ne ferons qu'effleurer ici. Voir [hdSFINAE] pour plus de détails.

Selon ce principe, il est possible que certaines substitutions génériques échouent dans un programme et il se trouve qu'*une telle situation n'a rien de dramatique*. Le compilateur, lorsqu'il cherchera à identifier les substitutions à utiliser dans la liste des possibilités qui s'offrent à lui pour un *template* donné, rencontrera inévitablement plusieurs cas qui ne conviennent pas à ses besoins. Ces cas seront simplement exclus de la liste des substitutions possibles sans que cela ne génère d'erreurs à la compilation.

Un exemple simple serait celui de la petite fonction `val_negative()` proposée à droite. On y trouve deux versions possibles de cette fonction, soit une opérant sur une simple constante de type `int` et une autre opérant sur un type quelconque dans lequel on trouve une méthode d'instance constante nommée `negation()` retournant une valeur dont le type est un type interne et public dans `T` nommé `type_negatif`. Le compilateur devra chercher à vérifier laquelle des versions privilégier pour un `int` comme pour une instance de `Zero`.

Ce programme compilera correctement même si le type `int` n'offre pas une méthode d'instance `negation()`, du fait que la tentative avortée de substitution d'un `int` dans la version générique reposant sur une invocation de `negation()` n'est pas une erreur.

Le compilateur avait au départ deux possibilités, mais l'une d'entre elles s'est envolée au passage.

```
int val_negative(int val) {
   return -val;
}
template <class T>
   typename T::type_negatif
       val_negative(const T &val) {
      return val.negation();
   }
struct Zero {
   using type_negatif = Zero;
   type_negatif negation() const {
      return *this;
   }
};
int main() {
   auto val = val_negative(3);
   auto z = val_negative(Zero{});
}
```

Voir [BoostGen] pour plusieurs applications intéressantes de la programmation générique en C++. Plus loin, ***Programmation générique appliquée*** donne un exemple concret et détaillé d'application de ces techniques.

[9] L'exemple qui sera utilisé ici a été inspiré d'exemples plus complexes pris sur le site de la bibliothèque *Boost*. Il est facile de concevoir des exemples intéressants du principe `SFINAE` mais il est plus costaud d'en concevoir des exemples simples.

Dans d'autres langages

D'entrée de jeu, ni Java, ni les langages .NET ne permettent d'exprimer des types internes et publics sous la forme rencontrée en C++. Un tel niveau d'encapsulation et de généralité n'est pas atteignable par programmation dans ces langages au moment d'écrire ces lignes.

La programmation générique est permise en Java depuis la version 1.5 du langage, et l'est aussi dans les langages .NET depuis la version 2.0 de l'infrastructure. L'ajout du support à la généricité pour ces divers langages est un atout indéniable, bien que la généricité y soit chaque fois un peu moins flexible qu'en C++.

En Java, la syntaxe utilisée pour la généricité rappelle fortement celle de C++ mais sans avoir recours à un mot clé spécial comme template.

Java ne supporte toutefois la généricité que sur des objets, pas sur des types primitifs. C'est pourquoi le programme principal (la méthode main() de la classe Z) proposé à droite utilise des références sur des Element<Integer> plutôt que sur des Element<int>.

Remarquez toutefois que le paramètre passé à la construction est un primitif (un int de valeur 3) : Java, depuis la version 1.5, supporte une technique nommée le *Boxing* (empruntée de C#), ce qui fait que la JVM instanciera, si elle a besoin de références plutôt que de valeurs, la classe équivalent à un type primitif donné.

Remarquez les deux notations, équivalentes, pour instancier un Element<T> à droite. La seconde, plus légère car elle ne demande pas de répéter le type auquel s'applique T, est à privilégier.

```
class Element<T> {
   private T valeur;
   public T getValeur() {
      return valeur;
   }
   private void setValeur(T val) {
      valeur = val;
   }
   public Element(T val) {
      setValeur(val);
   }
}
public class Z {
   public static void main(String [] args) {
      Element<Integer> e0 =
         new Element<Integer>(3);
      System.out.println (e.getValeur());
      Element<Integer> e1 = new Element<>(3);
      System.out.println(e0.getValeur());
      System.out.println(e1.getValeur());
   }
}
```

Si plusieurs types s'appliquent, ils sont séparés par des virgules (par exemple <T,U> si la généricité s'applique aux types T et U).

La JVM utilise la généricité pour valider au préalable et dissimuler des conversions explicites de types. Elle utilise une véritable structure générique mais manipule, en arrière-plan, des références sur des Object.

L'infrastructure .NET supporte en partie la programmation générique, du moins depuis la version 2.0 de l'infrastructure. Le caractère hétérogène du système de types marque au fer rouge les langages .NET (comme le langage Java) : seuls les types accédés de manière indirecte sont acceptables, dans ces langages, pour la généricité. Ceci diminue beaucoup leur utilité pour la mise en place de stratégies de calcul performantes et génériques (le *Boxing* de Java

et de C# fait en sorte de limiter la complexité syntaxique, mais ne compense pas pour la performance réduite).

En C#, on pourrait définir un `Element` générique sur un type `T` de la manière décrite dans l'exemple proposé à droite.

Notez l'absence de mots clés spécifiques : la simple adjonction de `<T>` suite au nom d'une classe au moment de sa déclaration suffit à indiquer la généricité de cette classe sur `T`.

La syntaxe à l'usage rappelle celle des *templates* avec C++. Il peut (et il y a, souvent, pour des raisons historiques) deux classes de même nom en C#, soit une générique et l'autre non (donc `Element` et `Element<int>` ne réfèrent pas au même type, sauf dans le contexte spécifique de la définition de `Element<T>` où le mot `Element` désigne par défaut `Element<T>`, comme en C++.

À l'image du mot clé `auto` en C++, on peut utiliser `var` pour indiquer le type d'une variable lors de sa déclaration.

```
namespace zz
{
   class Element<T>
   {
      public T Valeur { get ; private set ; }
      public Element(T val)
      {
         Valeur = val;
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Element<int> e0 = new Element<int>(3);
         var e1 = new Element<int>(3);
         System.Console.WriteLine(e0.Valeur);
         System.Console.WriteLine(e1.Valeur);
      }
   }
}
```

Bien que C# impose beaucoup de contraintes à la programmation générique, je vous invite à explorer le sujet par vous-mêmes (il est passionnant malgré ses limites). La section ***Génériques et contraintes***, plus bas, offre un (très) bref survol de la question des contraintes dans ces langages.

Si plusieurs types s'appliquent, ils sont séparés par des virgules – par exemple `<T,U>` si la généricité s'applique aux types `T` et `U`.

Avec VB.NET, le même code s'écrit de manière générique mais avec une syntaxe bien différente.

L'apposition du suffixe (Of T) à la classe générique sur la base d'un type T fait de cette classe un type générique et du type T le paramètre sur lequel repose la généricité. Si plusieurs types s'appliquent, ils sont séparés par des virgules (par exemple Of T, U si la généricité s'applique aux types T et U).

L'instanciation (dans main()) doit, bien entendu, spécifier la nature du type T, avec une syntaxe analogue (ici, on parle d'un Element(Of Integer)).

Le CLR de .NET génère des versions distinctes des diverses classes et méthodes génériques pour chacun des types en fonction desquels elles sont appliquées. Cette génération de code chaque fois distincte est analogue à ce que fait C++.

```
Namespace zz
    Class Element(Of T)
      Private valeur_ As T
      Public Function GetValeur() As T
         Return valeur_
      End Function
      Private Sub SetValeur(ByVal val As T)
         valeur_ = val
      End Sub
      Public Sub New(ByVal val As T)
         SetValeur(val)
      End Sub
   End Class
End Namespace
Module Test
   Public Sub main()
      Dim e As zz.Element(Of Integer) = _
         New zz.Element(Of Integer)(3)
      System.Console.WriteLine(e.GetValeur())
   End Sub
End Module
```

Étant donné que .NET réalise la compilation juste à temps des méthodes invoquées et que le fruit de cette compilation se trouve logé dans une antémémoire, il faudra faire des métriques pointues pour savoir quel sera l'impact des génériques sur la bonne gestion de cette antémémoire en période de pointe.

Génériques et contraintes

Les langages .NET et Java permettent (et obligent, dans bien des cas) la spécification de contraintes particulières sur les types sur lesquels la généricité est appliquée.

En Java, les contraintes s'imposent sur la base des classes et des interfaces génériques. Il n'est pas possible, par exemple, de spécifier une seule méthode générique dans une classe qui n'est pas elle-même générique, ou de spécifier une méthode générique sur un type U dans une classe générique sur un type T, tactiques tout à fait légales en C++.

L'exemple proposé à droite met en place une clase Afficheur offrant un service capable d'invoquer la méthode afficher() de tout type étant au moins Affichable.

On remarquera que la contrainte est d'une utilité limitée ici (on aurait simplement pu passer un Affichable en paramètre à la méthode agir() de la classe Afficheur pour en arriver au même résultat). Les contraintes sont cependant plus puissantes que ce que peut illustrer ce petit exemple. On peut (et, parfois, on *doit*) utiliser des symboles de remplacement (des *Wildcards*), le symbole **?**, pour exprimer, par exemple :

- ceci s'applique sur tout type dérivant d'un type générique T (<**?** extends T>); ou
- ceci s'applique sur tout type ancêtre d'un type générique T (<**?** super T>).

Ces catégories importent avec les génériques de Java du fait que la correspondance se fait sur les types, pas sur les familles de types (une classe définie sur un type T s'applique à T seulement, pas nécessairement à ses enfants).

Dans le programme principal, aff.agir() peut recevoir des références sur des objets implémentant au moins Affichable (ce qui exclut une instance de la classe Autre).

```
interface Affichable {
   void afficher();
}
class PeuImporte implements Affichable {
   public void afficher() {
      System.out.println("Peu importe");
   }
}
class Autre {
}
class Element<T> implements Affichable {
   private T valeur;
   public void afficher () {
      System.out.println(getValeur());
   }
   public T getValeur() {
      return valeur;
   }
   private void setValeur(T val) {
      valeur = val;
   }
   public Element(T val) {
      setValeur(val);
   }
}
class Afficheur <T extends Affichable> {
   public void agir(T aff) {
      aff.afficher();
   }
}
public class Z {
   public static void main(String[] args) {
      Afficheur aff = new Afficheur();
      aff.agir(new Element<Integer>(3));
      aff.agir(new PeuImporte());
      //aff.agir(new Autre());
   }
}
```

Le langage C# permet la généricité sur la base de méthodes individuelles, en plus de la permettre sur la base de classes ou d'interfaces entières.

De même, avec C#, l'idée de contraintes sur un type générique donné s'exprime en ajoutant une clause **`where`** à la déclaration de l'unité générique (classe, interface ou méthode), ce qui suit un peu la notation que C++ vise pour les concepts[10] [POOv03].

La clause `where` spécifie l'ensemble des caractéristiques que doit supporter le type sur lequel s'appliquera la généricité. Dans notre exemple, `T` doit dériver de `Affichable` ou implémenter `Affichable` (selon la nature de `Affichable`).

Si nous avions écrit une méthode générique instanciant un `T` à l'aide de son constructeur par défaut, par exemple, alors nous aurions dû indiquer la contrainte `where new T()`.

Si plusieurs contraintes s'appliquent, alors elles doivent être séparées les unes des autres par des virgules, par exemple

```
where
   T : Affichable,
   new T()
```

dans le cas où `T` doit être `Affichable` et doit être instanciable à l'aide d'un constructeur par défaut).

Ces contraintes permettent au compilateur de valider rapidement le code sur le plan sémantique et de s'assurer de pouvoir compiler correctement chaque méthode au moment opportun.

```
namespace zz {
   interface Affichable {
      void afficher();
   }
   class PeuImporte : Affichable {
      public void afficher() {
         System.Console.WriteLine("Bof");
      }
   }
   class Autre { }
   class Element<T> : Affichable {
      public T Valeur
      {
         get; private set;
      }
      public void afficher() {
         System.Console.WriteLine(Valeur);
      }
      public Element(T val) {
         Valeur = val;
      }
   }
   class Afficheur {
      public void agir<T>(T aff)
         where T : Affichable {
         aff.afficher();
      }
   }
   class Program {
      static void Main(string[] args) {
         Afficheur aff = new Afficheur();
         aff.agir(new Element<int>(3));
         aff.agir(new PeuImporte());
         //aff.agir(new Autre());
      }
   }
}
```

[10] Cette formulation n'est pas accidentelle : la version commerciale produite la plus rapidement s'inspire ici du travail fait pour celle qui sera livrée ultérieurement. Les standards sont plus longs à concevoir que les produits.

Sans surprises, un programme VB.NET sera soumis aux mêmes règles qu'un programme C# pour ce qui est de la généricité. Les différences entre les deux langages seront d'ordre syntaxique.

La contrainte que T soit Affichable, imposée par la méthode générique agir() de la classe Afficheur sur son type générique T s'exprime ainsi :

```
(Of T As Affichable)
```

Si plusieurs contraintes sont imposées à un type générique donné, celles-ci doivent être placées entre accolades (oui, oui) et séparées par des virgules.

Par exemple, si T doit pouvoir être instancié à l'aide de New et doit aussi être Affichable, alors nous aurions les contraintes suivantes :

```
(Of T As { Affichable, New })
```

Pour une plus grande liste de contraintes possibles (et pour les nombreux détails syntaxiques et mots clés impliqués), référez-vous à la documentation du langage.

```
Namespace zz
   Interface Affichable
      Sub afficher()
   End Interface
   Class PeuImporte
      Implements Affichable
      Public Sub afficher() Implements Affichable.afficher
         System.Console.WriteLine("Peu importe")
      End Sub
   End Class
   Class Autre
   End Class
   Class Element(Of T)
      Implements Affichable
      Private valeur_ As T
      Public Sub afficher() Implements Affichable.afficher
         System.Console.WriteLine(GetValeur())
      End Sub
      Public Function GetValeur() As T
         Return valeur_
      End Function
      Private Sub SetValeur(ByVal val As T)
         valeur_ = val
      End Sub
      Public Sub New(ByVal val As T)
         SetValeur(val)
      End Sub
   End Class
   Class Afficheur
      Public Sub agir(Of T As Affichable)(ByVal aff As T)
         aff.afficher()
      End Sub
   End Class
End namespace
Module Test
   Public Sub Main(ByVal args As String())
      Dim aff As New zz.Afficheur
      aff.agir(New zz.Element(Of Integer)(3))
      aff.agir(New zz.PeuImporte())
      ' aff.agir(new zz.Autre())
   End Sub
End Module
```

Syntaxe unifiée des fonctions

Supposons que nous rédigions une classe comme la suivante :

```
#include <vector>
template <class T>
   class tapon_en_ordre {
   public:
      using value_type = T;
   private:
      using conteneur_type = std::vector<value_type>;
      conteneur_type elems;
   public:
      using iterator = typename conteneur_type::iterator;
      using const_iterator = typename conteneur_type::const_iterator;
      using size_type = typename conteneur_type::size_type;
      void ajouter(const value_type &);
      iterator begin();
      iterator end();
      const_iterator begin() const;
      const_iterator end() const;
      size_type size() const;
      bool empty() const;
      // etc.
   };
```

En soi, cela semble banal. Cela dit, vous remarquerez que le code ci-dessus se limite à des déclarations de méthodes. Comme c'est le cas pour bien des gens, on peut supposer que les programmeuses et les programmeurs aient souhaité séparer interface et implémentation (bien que, dans le cas de code générique, ces deux parties doivent demeurer visibles au compilateur lorsque celui-ci rencontre le lieu où les méthodes seront appelées).

On aurait alors, probablement dans le même fichier d'en-tête, les définitions de méthodes suivantes :

```
// ...inclusions et using...
void tapon_en_ordre::ajouter(const value_type &elem) {
   elems.push_back(elem);
   sort(elems.begin(), elems.end());
}
tapon_en_ordre::size_type tapon_en_ordre::size() const {
   return elems.size();
}
bool tapon_en_ordre::empty() const {
   return elems.empty();
}
tapon_en_ordre::iterator tapon_en_ordre::begin() {
   return elems.begin();
}
tapon_en_ordre::iterator tapon_en_ordre::end() {
   return elems.end();
}
tapon_en_ordre::const_iterator tapon_en_ordre::begin() const {
   return elems.begin();
}
tapon_en_ordre::const_iterator tapon_en_ordre::end() const {
   return elems.end();
}
// ...
```

Remarquez l'écriture un peu longue du type de retour de certaines méthodes (celles où des éléments sont indiqués en caractères gras). Le type `tapon_en_ordre` doit explicitement qualifier chaque recours à l'un de ses types internes du fait qu'au moment de spécifier le type de la valeur retournée par une méthode définie hors de la déclaration de sa classe, le compilateur ne sait pas de quel type on parle.

Un exemple éclairant est la comparaison de l'écriture du type nommé `tapon_en_ordre::value_type` lorsqu'il est utilisé en tant que paramètre passé à la méthode `tapon_en_ordre::ajouter()` avec l'écriture requise pour identifier le type `tapon_en_ordre::size_type` lorsque ce dernier est utilisé tant que type de ce que retourne `tapon_en_ordre::size()` :

En tant que paramètre	**En tant que type retourné**

`void tapon_en_ordre::ajouter`
`  (const value_type &elem) {`
`  elems.push_back(elem);`
`  sort(elems.begin(), elems.end());`
`}`
	`tapon_en_ordre::size_type`
	`  tapon_en_ordre::size() const {`
	`  return elems.size();`
	`}`

Dans le cas du paramètre, le nom `value_type` est utilisé une fois le contexte `tapon_en_ordre::` connu, et ce contexte est implicite. Dans le cas du type de la valeur retournée, le contexte en question n'est pas encore introduit (il apparaît plus loin, en préfixe au nom de la méthode) et celui-ci doit donc être exprimé explicitement.

Un raffinement possible avec C++ 11 est une formule dite « unifiée » pour l'écriture de fonctions. Cette formule permet d'exprimer le format habituel, soit :

```
type nom(params) { code }
```

par un nouveau format, qui place le type de la valeur retournée par la fonction après le nom de la fonction lui-même, comme ceci :

```
auto nom(params) -> type { code }
```

Le mot clé `auto` est obligatoire dans cette deuxième écriture. Dans bien des cas, cette nouvelle formule a un apport cosmétique, et on peut l'aimer ou non sans que cela n'ait de réel impact (le code généré étant, dans les deux cas, le même) :

Format traditionnel	Format « unifié »

`int f() {`
`  return 3;`
`}`
	`auto f() -> int {`
	`  return 3;`
	`}`

Dans le cas d'une λ (voir ***Les expressions Lambda (ou expressions λ)***), il y a parfois un intérêt à expliciter le type de retour souhaité. Dans un tel cas, le format « unifié » est clairement pertinent.

```
auto carre = [](int n) -> long { return n * n; }
```

Ne vous en faites pas trop avec la syntaxe particulière de cette expression; nous y reviendrons.

Dans un cas comme celui de `tapon_en_ordre`, cela dit, l'impact simplificateur de la nouvelle syntaxe saute aux yeux :

Format traditionnel

```
// ...inclusions et using
void tapon_en_ordre::ajouter
  (const value_type &elem) {
   elems.push_back(elem);
   sort(elems.begin(), elems.end());
}
tapon_en_ordre::size_type
   tapon_en_ordre::size() const {
   return elems.size();
}
bool tapon_en_ordre::empty() const {
   return elems.empty();
}
tapon_en_ordre::iterator
   tapon_en_ordre::begin() {
   return elems.begin();
}
tapon_en_ordre::iterator
   tapon_en_ordre::end() {
   return elems.end();
}
tapon_en_ordre::const_iterator
   tapon_en_ordre::begin() const {
   return elems.begin();
}
tapon_en_ordre::const_iterator
   tapon_en_ordre::end() const {
   return elems.end();
}
// ...
```

Format « unifié »

```
// ...inclusions et using...
void tapon_en_ordre::ajouter
   (const value_type &elem) {
   elems.push_back(elem);
   sort(elems.begin(), elems.end());
}
auto tapon_en_ordre::size() const -> size_type {
   return elems.size();
}
bool tapon_en_ordre::empty() const {
   return elems.empty();
}
auto tapon_en_ordre::begin() -> iterator {
   return elems.begin();
}
auto tapon_en_ordre::end() -> iterator {
   return elems.end();
}
auto tapon_en_ordre::begin() const -> const_iterator {
   return elems.begin();
}
auto tapon_en_ordre::end() const -> const_iterator {
   return elems.end();
}
// ...
```

Vous remarquerez que, étant placé dans le contexte du nom de la classe à laquelle appartient la méthode, la version « unifiée » est moins verbeuse, plus directe dans sa formulation. Il s'agit donc d'un outil intéressant pour alléger le travail de rédaction du code sans que cela n'entraîne quelque coût que ce soit en termes de temps d'exécution.

L'opérateur statique `decltype`

Couplé à un opérateur statique `decltype` de C++ 11, qui permet de déduire le type d'une expression, la syntaxe unifiée des fonctions permet d'exprimer élégamment des opérations complexes à exprimer en se limitant à la syntaxe traditionnelle.

Ainsi, imaginons une fonction `mult(f,g)` retournant le produit de l'exécution des fonctions nullaires `f` et `g`. En se limitant à la syntaxe traditionnelle, l'écriture de `mult()` pourrait être déplaisante, du fait que le type retourné par `mult(f,g)` dépend à la fois du type retourné par `f` et du type retourné par `g`. Toutefois, en combinant la syntaxe « unifiée » et `decltype`, cette fonction s'exprime naturellement :

```
template <class F, class G>
   auto mult(F f, G g) -> decltype(f() * g()) {
      return f() * g();
   }
```

Il reste manifestement un peu de travail à faire dans le langage pour simplifier l'écriture (ici, après tout, l'expression `f()*g()` apparaît à deux reprises) mais la possibilité de définir le type d'une fonction après son nom, combinée à celle d'exprimer des idées comme « je veux que ce type soit celui de l'expression suivante » facilite l'accès à l'expression de programmes plus généraux et plus clairs.

Depuis C++ 14, dans un cas comme celui-ci, on peut écrire simplement :

```
template <class F, class G> auto mult(F f, G g) { return f() * g(); }
```

Là où `decltype` se distingue du mot clé `auto` [POOv00] est dans ce qui touche à la précision du type déduit. Ainsi :

```
class X {
   int n;
public:
   X(int n) : n{n} {
   }
   int& valeur() {
      return n;
   }
};
// ...
int main() {
   X x{3};
   auto val = x.valeur(); // val est un int
   decltype(x.valeur()) ref = x.valeur(); // ref est un int&
}
```

Références sur des *rvalue* et sémantique de mouvement

Avec C++ 11, certains changements fondamentaux font leur entrée dans la conceptualisation OO. Les expressions constantes généralisées [POOv03] en sont un intéressant, qui joue un rôle d'optimisation et accroît l'élégance du système de types du langage, mais un changement plus profond a été introduit, soit les ***références sur les `rvalue`*** et leur conséquence, qui est l'introduction de la ***sémantique de mouvement*** dans le langage.

Les rvalue *et les* lvalue

Sur le plan historique, les noms *rvalue* et *lvalue* sont associés aux expressions pouvant se situer à gauche (*Left*, *lvalue*) ou à droite (*Right*, *rvalue*) d'une affectation.

Intuitivement, `int i = 3;` est raisonnable car `i` est un *lvalue* et `3` est un littéral, donc un *rvalue*, et il en va de même pour `float f = g();` si `g()` retourne un `float`. À l'inverse, `3 = i;` serait illégal (tentative d'affectation à un *rvalue*) et il en serait de même pour `g() = f;`.

Typiquement, les *lvalue* sont des variables ou des constantes mais, dans le cas des constantes, seulement lors de la construction (parce qu'une fois construites, elles ne sont plus modifiables). Les *rvalue* incluent les variables temporaires résultant d'expression arithmétiques ou retournées par des fonctions, les littéraux, les constantes une fois celles-ci construites, *etc*. Celles qui nous intéressent ici sont les variables « jetables », qui ne sont plus susceptibles d'être utilisées une fois utilisées à droite d'une affectation. Ces variables sont le plus souvent anonymes, ou retournées par une fonction.

Notez que `++i` est une *lvalue* puisque l'opérateur `++` en forme préfixée retourne une référence sur son opérande, alors que `i++` est une *rvalue* puisque l'opérateur `++` en forme suffixée retourne une copie de son opérande avant incrémentation.

Avec l'arrivée des références sur des *rvalue*, en fait, le système de types de C++ s'est beaucoup enrichi (ou complexifié, selon les interprétations). Pour en savoir plus, lisez [StrouTerm] qui aborde le sujet avec humour.

Surcharges sur la base de lvalue *et de* rvalue

Qu'en est-il de l'interaction entre références, références vers `const`, copies, copies constantes, références sur des *lvalue* et références sur des *rvalue*? Voici un exemple, adapté d'un article de l'équipe de développement de *Visual Studio*[11] :

```
#include <string>
#include <iostream>
// ...using...
void afficher(const string &s) {
   cout << "afficher(const string&): " << s << endl;
}
void afficher(string &&s) {
   cout << "afficher(string&&): " << s << endl;
}
string copie() {
   return "copie()";
}
const string const_copie() {
   return "const_copie()";
}
int main() {
   string var_locale{"var_locale"};
   const string const_locale{"const_locale"};
   afficher(var_locale);
   afficher(const_locale);
   afficher(copie());
   afficher(const_copie());
}
```

Ce programme affichera ce qui suit (à droite). La question est, bien sûr, pourquoi?

```
afficher(const string&): var_locale
afficher(const string&): const_locale
afficher(string&&): copie()
afficher(const string&): const_copie()
```

Voici les règles :

- la variable `var_locale` est nommée, donc pourrait être utilisée ultérieurement. Ce faisant, le compilateur privilégiera la copie au transfert dans son cas;
- la constante `const_locale` est nommée, et qualifiée `const` par-dessus le marché. Le compilateur utilisera la copie, qui ne détruit pas les états de l'objet original, dans ce cas aussi;
- la fonction `copie()` retourne une copie anonyme d'un objet. Nous savons donc que cet objet est destiné à être détruit, et que ses états peuvent être arrachés sans effets adverses; enfin
- la fonction `const_copie()` retourne un objet non-modifiable. Le compilateur obéit et en fait une copie plutôt que d'en permettre le transfert des états.

[11] http://blogs.msdn.com/b/vcblog/archive/2009/02/03/rvalue-references-c-0x-features-in-vc10-part-2.aspx

Cas d'espèce – la fonction `swap()`

S'il y a une fonction clé dans l'univers de C++, c'est bien la fonction `swap()`. Partie intégrante du standard, dans la bibliothèque `<algorithm>`, elle sert de pièce de voûte à la plupart des algorithmes de tri, à pratiquement tous les algorithmes réalisant des permutations d'éléments dans un conteneur, et sert à l'implémentation de l'idiome d'affectation sécuritaire [POOv00].

Traditionnellement, cette fonction s'implémente tel que proposé à droite. Il existe des variantes profitant des particularités des divers compilateurs, mais l'essentiel y est.

```
template <class T>
   void swap(T &a, T &b) {
      T temp = a;
      a = b;
      b = temp;
   }
```

On remarquera que le temps d'exécution de cette fonction est dicté par les opérations qui y apparaissent, soit une construction par copie et deux affectations. Essentiellement, trois copies d'instances de `T`.

Cette fonction semble optimale, et elle l'est… presque. Une intuition clé nous révèle qu'on peut améliorer même une fonction aussi simple et aussi fondamentale que `swap()`, tout comme il est possible d'améliorer l'implémentation d'un large éventail de classes, incluant presque tous les conteneurs standards.

Une optimisation fondamentale, donc. Pour comprendre l'intuition en question, examinons brièvement la fonction `swap()` en détail :

- les paramètres `a` et `b` sont tous deux des indirections vers des `T`. La raison pour cela est que nous voulons modifier les paramètres de la fonction. Notez qu'une référence non-`const` permet de modifier le référé, et doit donc mener vers un `lvalue`;
- la variable temporaire (`temp` dans ce cas-ci) est un `T` à part entière, et sert d'espace tampon pour faire en sorte que les états en cours d'échange ne soient pas perdus. Cette variable est initialisée en tant que copie de `a`, ce qui fait que suite à cette opération, il existe deux objets dans la fonction possédant les mêmes états (ceux de `a`);
- le contenu de `a` est ensuite écrasé par le contenu de `b`. Suite à cette opération, `a` et `b` ont chacun une copie des mêmes états, alors que `temp` contient toujours une copie des états de `a` (pour ne pas les perdre);
- enfin, le contenu de `b` est écrasé par le contenu de `temp`. Suite à cette opération, `b` et `temp` ont chacun une copie du même contenu, celui de a au début de l'exécution de la fonction, alors que le contenu de `a` est une copie du contenu initial de `b`;
- la variable `temp` est quant à elle détruite à la fin de sa portée.

```
template <class T>
   void swap(T &a, T &b) {

      T temp = a;

      a = b;

      b = temp;
   }
```

Pour des données de types primitifs, ces opérations sont optimales. Pour des entités complexes, par exemple un conteneur, les trois copies sont susceptibles d'être quelque peu prohibitives.

L'intuition, donc, est celle-ci : **les copies ne sont pas toujours nécessaires**. En fait, ***pour les objets longs à copier, elles sont probablement superflues***.

Voici pourquoi, à l'aide de pseudocode :

- l'objectif est d'échanger les états de `a` et de `b`. La postcondition de cette fonction est donc que les états de ces deux entités soient permutées;
- la première étape est de placer les états de `a` *quelque part* pour ne pas les perdre quand nous déposerons, éventuellement, les états de `b` dans `a`. Remarquez que **cette étape n'a pas à être une copie** : les états de `a` suite à cette opération nous importe peu puisque nous allons sous peu les écraser avec ceux de `b`. Conséquemment, nous pourrions simplement arracher à `a` ses états et les déposer dans `temp`[12] (en d'autres mots, nous pouvons déplacer les états plutôt que les copier);
- ensuite, les états de `b` sont placés dans `a`. Ici encore, peu nous importe ce qu'il advient de `b`, dans la mesure où il demeure dans un état valide suite à cette opération. Il est donc possible de transférer, ici encore, les états de `b` vers `a` plutôt que de les copier;
- enfin, le contenu de `b` est écrasé par le contenu de `temp`. Peu nous importe ce qu'il advient de `temp`, car elle sera détruite à la fin de sa portée, alors nous pouvons encore une fois réaliser un transfert plutôt qu'une copie.

```
swap(&a, &b)

    temp ← a

    a ← b

    b ← temp
```

Dans tous ces cas, on n'a pas à faire de copie à une condition : il ne faut pas que le transfert (le *mouvement*) remplaçant la copie ne lève une exception.

Constats d'ensemble : tout comme la stratégie classique qui consiste à copier les états, le transfert des états permet de réaliser la postcondition de la fonction `swap()`.

Comment implémenter `swap()` avec une sémantique de mouvement? C'est ici qu'interviennent les **références sur des *rvalue***.

[12] Ceci dans la mesure où `a` demeure dans un état légal – respectant ses invariants – suite à cette opération, car en vertu de l'encapsulation, tout objet est présumé se trouver dans un état valide du début à la fin de son existence.

Références sur des rvalue

La syntaxe des références sur des *rvalue* est somme toute assez simple. En effet, dans le code ci-dessous, `a` est un `int`, `r` est une référence usuelle sur a. Sans surprises, l'affectation de `4` à `r` est en fait une affectation vers a, donc le résultat du 1er affichage sera `4`.

```
#include <iostream>
int main() {
   using namespace std;
   int a = 3;
   int &r = a;
   r = 4;
   cout << a << endl;
   int &&rr = 4;
   cout << rr << endl;
}
```

De son côté, `rr` est une référence sur un *rvalue* (notez la syntaxe, soit le type suivi d'une double esperluette) Il aurait été illégal de la faire mener vers `a` directement car `a` est un *lvalue*, une entité modifiable qui porte un nom et est susceptible d'être utilisée ultérieurement (lui arracher son contenu serait dommageable).

L'affectation du littéral `4`, un *rvalue*, est par contre correcte (notez que la référence `r`, elle, ne pourrait référer à un *rvalue*; exprimé autrement, la définition `int &r = 4;` est illégale).

Pourquoi donc voudrait-on des références sur des *rvalue*? Imaginez un cas comme celui-ci :

```
// ... inclusions et using ...
vector<int> concatener(const vector<int> &v0, const vector<int> &v1) {
   vector<int> v;
   v.reserve(v0.size() + v1.size());
   copy(begin(v0), end(v0), back_inserter(v));
   copy(begin(v1), end(v1), back_inserter(v));
   return v;
}
int main() {
   vector<int> v0{ 2, 3, 5, 7, 11 };
   vector<int> v1{ 2, 3, 5, 7, 11 };
   auto v = concatener(v0, v1);
   copy(begin(v), end(v), ostream_iterator<int>{cout, " "});
   cout << endl;
}
```

Le constructeur de `v` dans `main()` génère-t-il une copie de `v` dans `concatener()`? Non, mais parce que le compilateur applique une optimisation nommée `RVO` (*Return Value Optimization*) qui lui permet tout simplement de construire `v` dans `main()` à partir de la valeur retournée.

Maintenant, peut-on éviter le vecteur temporaire dans `concatener()`? Supposons ici, par souci de simplicité, que `main()` n'ait plus besoin des vecteurs `v0` et `v1` suite à l'appel à la fonction `concatener()`. Nous aurons alors :

```
// ... inclusions et using ...
vector<int> concatener(vector<int> &&v0, vector<int> &&v1) {
   v0.reserve(v0.size() + v1.size());
   copy(begin(v1), end(v1), back_inserter(v));
   return v;
}
int main() {
   vector<int> v0{ 2, 3, 5, 7, 11 };
   vector<int> v1{ 2, 3, 5, 7, 11 };
   auto v = concatener(std::move(v0), std::move(v1));
   copy(begin(v), end(v), ostream_iterator<int>{cout, " "});
   cout << endl;
}
```

Remarquez les appels à `std::move()`, qui indiquent « je n'utiliserai plus cette variable ». Il se trouve que `vector` expose un constructeur de mouvement, dont la signature est la suivante (pour le paramètre `A`, voir [POOv03], section ***Gestion avancée de la mémoire***) :

```
template <class T, class A>
   vector(vector<T, A> &&autre)
   {
      // ... code ...
   }
```

Dans la plupart des cas, ce constructeur sera préféré au constructeur de copie par un compilateur C++ 11 lorsque le paramètre suppléé au constructeur est une *rvalue*. Typiquement, les *rvalue* sont des littéraux (pour lesquels le compilateur peut générer le code qui lui semble optimal) ou des variables temporaires, typiquement anonymes ou résultant d'expressions, donc qui ne sont pas susceptibles d'être utilisées par la suite – n'ayant pas de nom, on ne pourra plus leur faire référence. Bien entendu, si le paramètre est une référence sur une *rvalue*, la question ne se pose même pas et ce constructeur sera nécessairement utilisé.

Il reste à voir s'il est avantageux de transférer plutôt que de copier. Pour ce faire, nous utiliserons un exemple simple, soit un extrait du code d'un classique tableau dynamique (réduit à l'essentiel pour illustrer notre propos, question d'économie). L'idée sera la même que pour un vecteur, mais le code sera plus simple à analyser.

Supposons un tableau de `T`, dont le substrat (ce vers quoi pointe `elems`) est alloué dynamiquement, donc pour lequel l'objet est responsable du substrat en question.

Cela signifie que la Sainte-Trinité devra être programmée pour gérer les états d'un `Tableau<T>`.

Les itérateurs et les méthodes qui y ont trait sont banals.

Certaines méthodes vont aussi de soi. Remarquez la méthode `swap()`, qui peut être utilisée pour échanger les états de deux instances de `Tableau<T>`, programmée de manière typique. Nous souhaitons ici que les optimisations promises s'appliquent.

Le constructeur paramétrique sera banal.

```
// ...inclusions et using...
template <class T>
   class Tableau {
   public:
      using size_type = size_;
   private:
      T *elems;
      size_type nelems;
   public:
      using iterator = T*;
      using const_iterator = const T*;
      iterator begin() noexcept {
         return elems;
      }
      const_iterator begin() const noexcept {
         return elems;
      }
      iterator end() noexcept {
         return begin() + size();
      }
      const_iterator end() const noexcept {
         return begin() + size();
      }
      size_type size() const noexcept {
         return nelems;
      }
      bool empty() const noexcept {
         return !size();
      }
      void swap(Tableau &tab) noexcept {
         using std::swap;
         swap(elems,  tab.elems);
         swap(nelems, tab.nelems);
      }
      Tableau(size_type n)
         : nelems{n}, elems{new T[n]} {
         try {
            fill(begin(), end(), T{});
         } catch(...) {
            delete [] elems;
            throw;
         }
      }
```

Il en sera de même du constructeur de copie…

```
   Tableau(const Tableau &tab)
      : nelems{tab.size()}
        elems{new T[tab.size()]} {
      try {
         copy(tab.begin(), tab.end(), begin());
      } catch(...) {
         delete [] elems;
         throw;
      }
   }
template<class It>
   Tableau(It debut, It fin)
      : nelems{distance(debut,fin)} {
      elems = new T[size()];
      try {
         copy(tab.begin(), tab.end(), begin());
      } catch(...) {
         delete [] elems;
         throw;
      }
   }
```

…et du constructeur de séquence.

Notez toutefois le constructeur de mouvement. Ses caractéristiques sont des plus intéressantes :

- il ne lèvera *probablement* aucune exception, n'allouant presque aucune ressource (l'objet construit arrache les états d'un objet présumé correct). Une implémentation évitant totalement l'appel à `new T[0]` aurait été *no-throw*;
- il est extrêmement rapide;
- l'objet auquel l'état a été arraché sera, suite à la perte de ses états, dans un état convenable (tableau vide).

Cela dit, l'écriture à droite est nettement meilleure. Nous examinerons pourquoi en examinant la fonction `std::move()`, plus loin.

L'affectation est banale.

```
/*
      Tableau(Tableau &&tab)
         : nelems{}, elems{new T[0]} {
         swap(*this, tab);
      }
*/

      Tableau(Tableau &&tab)
         : elems{std::move(tab.elems)},
           nelems{std::move(tab.nelems)} {
         tab.elems = {};
         tab.nelems = {};
      }
   Tableau& operator=(const Tableau &tab) {
      Tableau{tab}.swap(*this);
      return *this;
   }
```

L'affectation d'une référence sur une `rvalue` est aussi très simple : elle ne lève aucune exception, ne faisant que permuter les états de l'objet à gauche de l'affectation avec ceux de l'objet à droite de celle-ci (peu importe, cette dernière étant sur le point d'être éliminée).

```
    Tableau& operator=(Tableau &&tab) noexcept {
        delete [] elems;
        elems = tab.elems;
        nelems = tab.nelems;
        tab.elems = {};
        tab.nelems = {};
        return *this;
    }
    // ...
};
```

Dans chaque cas, les mouvements sont plus rapides (et moins sujets à exceptions!) que ne le sont les copies. Maintenant, réexaminons `swap()` sous cette lueur : sans en changer la signature, nous souhaitons l'implémenter sur la base de mouvements (de transferts) d'états plutôt que sur la base de copies, ce qui peut entraîner des optimisations considérables.

La fonction `std::move()`

Pour y arriver, nous souhaitons en fait implémenter une fonction capable de transformer un paramètre en `rvalue`, sur demande du code client. Essentiellement, nous souhaitons que `std::swap()` soit implémenté comme suit.

```
template <class T>
   void swap(T &a, T &b) {
      T temp = std::move(a);
      a = std::move(b);
      b = std::move(temp);
   }
```

Remarquez le très léger changement à la fonction, c'est-à-dire le remplacement des copies par quelque chose qui, nous le souhaitons du moins, provoquera des transferts d'état (et optimisera de manière presque indécente notre programme).

Comment la fonction `std::move()` transformera-t-elle une référence ordinaire en référence sur un `rvalue`?

Le truc est tout simple (et extrêmement performant). Le voici, tel qu'implémenté dans la bibliothèque standard :

- la fonction prend une instance d'un type `T`, quel qu'il soit (ce qui inclut les copies, les références – `const` ou non – et les références vers des `const`), sous forme de référence sur un `rvalue`;
- elle retire les qualifications de références du type `T` (voir [POOv03], section ***Traits, polymorphisme statique et bases de métaprogrammation***); et
- retourne une référence sur un *rvalue* vers l'objet résultant.

Ce qui suit montre une implémentation typique (ne réécrivez pas cette fonction; utilisez la version standard) :

```
template <class T>
   typename remove_reference<T>::type&& move(T&& a) {
      return static_cast<typename std::remove_reference<T>::type&&>(t);
   }
```

Cette fonction n'entraîne en fait aucun coût à l'exécution. Elle est, essentiellement, *parfaite* : elle s'optimise sans effort, et permet une sémantique de mouvement (de transfert d'états) chaque fois que cela s'avère possible.

Impact des références sur des `rvalue`[13]

Les références sur des *rvalue* constituent un changement fondamental au système de types de C++. Plusieurs optimisations sont possibles à même le standard, et ce sans effort de la part des programmeurs (comme le démontre l'exemple de `std::swap()`), ce qui est en soi un avantage inestimable.

Quels sont les impacts sur les programmeuses et les programmeurs de ce changement dans le langage? En voici quelques-uns :

- lorsque nous construisons un type, il est maintenant possible de spécifier la sémantique de mouvement associée à ce type en codant le constructeur de mouvement (*Move Constructor*) et l'affectation de mouvement (*Move Assignment*), comme nous l'avons fait avec `Tableau` un peu plus haut. Sans que ces opérations soient nécessaires, elles nous permettent d'exprimer des manières pour le compilateur d'optimiser nos types encore plus qu'auparavant;
- l'écriture de fonctions retournant des copies ou d'expressions générant des variables temporaires anonymes devient très avantageuse, puisque ces copies et ces objets anonymes peuvent voir leurs états transférés à loisir. Pour cette raison, il devient pertinent d'utiliser les opérateurs binaires (à deux opérandes) même s'ils génèrent des copies (par exemple `operator+()`), alors qu'il était auparavant préférable de se limiter aux versions qui opéraient sur l'opérande de gauche (par exemple `operator+=()`);
- les copies et les temporaires sont des outils précieux en situation de multiprogrammation. `C++ 11` est un langage bien différent de `C++ 03`, et la multiprogrammation est l'une des clés de la programmation contemporaine;
- certains types sont transférables mais pas copiables. Le cas patent est `std::unique_ptr` (voir ***La bibliothèque `<memory>`***). Plusieurs classes incopiables peuvent en fait être déplaçables. Les conteneurs standards de C++ préfèrent le mouvement à la copie chaque fois que cela s'avère possible, ce qui signifie entre autres qu'il est possible de placer dans un vecteur des objets déplaçables sans être copiables. En ceci, `C++ 11` ouvre d'importants champs d'application qui n'étaient pas envisageables avec `C++ 03`.

[13] Voir [BeckMov] pour en savoir plus sur le sujet.

Impact sur la Sainte-Trinité

Pour l'essentiel, la sémantique de mouvement est une optimisation. Outre les objets incopiables, pour lesquels elle ouvre de nouvelles avenues, les objets supportant la sémantique de mouvement fonctionnent comme ceux qui ne supportent que la copie… mais sont plus faciles à utiliser dans certains cas, tout en étant plus rapides à l'exécution.

Il se trouve par contre que le compilateur ne peut pas générer automatiquement un constructeur de mouvement ou un opérateur d'affectation par mouvement comme il le fait avec les opérations de copie, car **les références sur des *rvalue* brisent certains programmes mal conçus**. Il faut en prendre conscience : les compilateurs généreront automatiquement des opérations de mouvement dans certaines circonstances, et il est possible qu'une classe ayant des invariants cachés soit brisée par le code résultant[14].

Pour cette raison :

- si vous implémentez un constructeur de mouvement ou un opérateur d'affectation par mouvement, le compilateur ne générera pas le constructeur de copie et l'affectation de manière automatique comme il le ferait normalement;
- de même, si vous implémentez un constructeur de copie ou un opérateur d'affectation, le compilateur ne générera pas le constructeur de mouvement et l'affectation par mouvement de manière automatique non plus;
- conséquemment, prendre conscience de la sémantique de mouvement implique chez nous une réflexion plus poussée sur la Sainte-Trinité et son impact dans notre programme.

[14] Voir [ImplMovGo] de ***Dave Abrahams*** sur le sujet, de même que [StrouMov] par ***Bjarne Stroustrup*** suite à son analyse de la situation. Il est rare qu'on introduise en C++ des changements au langage qui soient susceptibles de briser du code existant; ici, le retour sur l'investissement est immense et le comité de standardisation est allé de l'avant malgré tout.

Bibliothèque standard de C++[15]

Le langage C++, comme tous les langages, est porteur de certains choix philosophiques. Parmi ceux-ci, on retrouve une détermination de limiter les ajouts au langage lui-même[16], et de motiver plutôt le développement d'extensions au langage à travers des bibliothèques.

Avec la norme `ISO`, donc, certains des plus grands changements au langage C++ furent des changements aux bibliothèques standards du langage. De nouveaux fichiers d'en-tête[17], des implantations tenant compte des espaces nommés et tirant profit des *templates* (voir ***Programmation générique et templates***, plus haut), une approche nouvelle à des objets d'usage courant, et ainsi de suite.

Cette section offre un bref survol de certaines des bonnes idées apparaissant dans ces bibliothèques. Comprendre les bibliothèques standards et la philosophie sous-jacente permet un développement informatique plus intelligent, plus efficace, plus sécuritaire. Certains des points exposés ici sont couverts plus en détail ailleurs dans ce document.

L'intention derrière cette section est de simplifier votre travail. Les outils de la bibliothèque standard sont remarquablement souples, performants et génériques. Les comprendre à fond demande temps et pratique, et ce survol ne suffira pas à vous amener à ce point. Au mieux, il vous donnera le goût de creuser le sujet…

Dans ce qui suit, puisque nous discutons d'éléments de la bibliothèque standard, vous pouvez présumer que la plupart des types et des fonctions se situent dans l'espace nommé `std` (p. ex. : le vrai nom de `vector` est `std::vector`). Nous omettrons généralement la qualification par l'espace nommé dans un souci de concision.

Pour un chapitre de [CppPL] par ***Bjarne Stroustrup***, faisant un (bref) tour d'horizon des bibliothèques standards de C++, voir http://www.research.att.com/~bs/3rd_tour2.pdf. Vous pouvez aussi consulter http://www.research.att.com/~bs/tour2.pdf.

[15] Voir [hdStl] pour des détails sur la partie `STL`, qui est d'une importance capitale.

[16] L'un des exemples les plus frappants est la prudence et la parcimonie avec laquelle on y introduit des mots clés, privilégiant presque systématiquement l'introduction de bibliothèques à une modification au langage.

[17] Je fais un écart de langage en parlant de *fichiers d'en-tête standards* ici. Je devrais plutôt parler d'***en-têtes standards*** tout simplement. En effet, la norme `ISO` de C++ permet aux gens offrant des compilateurs conformes de ne pas représenter ces en-têtes par des fichiers; il est possible que les en-têtes standards soient tout simplement des entités toujours placées en mémoire vive et qui apparaissent comme des fichiers pour les programmes.

Les bibliothèques survolées ici sont les suivantes (la liste n'est *vraiment* pas exhaustive[18]). Toutes les classes et toutes les fonctions sont dans l'espace nommé `std`.

Nom	Bref descriptif
<algorithm>	Offre plusieurs algorithmes standards applicables à des séquences d'objets (copier une séquence, générer des valeurs pour une séquence, mélanger les éléments d'une séquence, *etc*.).
<fstream>	Représente les flux d'entrée/ sortie sur des fichiers sous forme d'objets et d'opérateurs sur ces objets. Exemples typiques : `ifstream`, `ofstream`.
<iosfwd>	Déclare les types des flux d'entrée/ sortie comme `istream` et `ostream`.
<iostream>	Définit les flux d'entrée/ sortie sous forme d'objets et d'opérateurs sur ces objets. Exemples typiques : `istream` (dont `cin` est une instance), `ostream` (dont `cout` est une instance), `endl`, `cerr`, `clog`.
<memory>	Offre des outils de gestion de la mémoire (par exemple des pointeurs intelligents et des outils d'initialisation primitive de blocs de mémoire).
<sstream>	Représente un flux d'entrée/ sortie sur une chaîne de caractère, très utile pour réaliser des opérations de transtypage maison.
<string>	Représente les séquences de caractères (ou d'objets se comportant comme des caractères) sous forme d'objets et d'opérations sur ces objets. Exemples typiques : `string`, `wstring`.
<vector>	Représente une séquence d'un certain type sous la forme d'un conteneur (ici, un tableau dynamique) nommé `vector`. Cette bibliothèque est un exemple de conteneur standard, mais il en existe plusieurs autres (que vous trouverez dans `<map>`, `<list>`, `<queue>`, *etc*.).

Dans chaque cas, il y aurait beaucoup à dire pour montrer comment exploiter à fond les classes, fonctions et concepts proposés. Pour offrir une couverture convenable et complète, il aurait par contre fallu présenter les bibliothèques beaucoup plus loin dans ce document, ce qui n'est pas une manière pragmatique de procéder.

La bibliothèque standard de C++ est un exemple d'approche *multiparadigme*, reposant à la fois sur des techniques de programmation fonctionnelle, de programmation générique et de `POO`.

[18] Quelques autres, par exemple `<functional>`, `<numeric>`, `<utility>`, `<new>`, `<map>` ou `<deque>`, seront couvertes dans le cadre de thématiques plus sophistiquées. Leur absence ici signifie seulement que nous n'en aurons pas besoin pour faire un bref tour d'horizon. S'ajoutent, avec `C++ 11`, de nouvelles bibliothèques sophistiquées pour la représentation du temps, des expressions régulières, des *threads*, des opérations atomiques, de la génération de nombres pseudo-aléatoires, *etc*.

Concepts fondamentaux

Sans entrer dans les détails, certains concepts sont fondamentaux pour comprendre ce qui suit :

- une **séquence** est une suite d'objets de même type, qu'il est possible de parcourir, donc à travers lesquels on peut itérer (exemple primitif : les éléments d'un tableau ou d'une liste);
- un **itérateur** est un objet permettant de parcourir les éléments d'une séquence, donc au minimum d'accéder à un élément de la séquence et d'aller à l'élément suivant dans cette séquence (p. ex. : un pointeur sur un élément d'un tableau). Un itérateur sur une liste ne sera donc pas le même qu'un itérateur sur un tableau, mais les deux joueront un rôle analogue et exposeront un sous-ensemble commun d'opérations;
- un **flux** est une abstraction sur laquelle on peut écrire (s'il s'agit d'un flux en sortie) et de laquelle on peut lire des données (s'il s'agit d'un flux en entrée). Quelques exemples : un fichier, la console, une chaîne de caractères, …;
- l'usage C++ est d'accéder à un flux en lecture et en écriture à l'aide d'opérateurs binaires (à deux opérandes) tels que `>>` et `<<`, ce qui est *largement* préférable, pour la sécurité, que les stratégies offertes par C (`printf()`, `scanf()`) ou par les divers langages `.NET` (`Console.Write()`) du fait que C++ peut détecter la majorité des erreurs et prendre la majorité des décisions à la compilation;
- un **conteneur** permet d'entreposer des éléments d'un type donné et de leur accéder. Dans la plupart des cas (pas tous), un conteneur peut être traversé comme une séquence et encapsule des itérateurs permettant un parcours efficace (p. ex. : `vector`, `string`).

Pour aller plus loin dans les descriptifs, il nous faudra avoir couvert un certain nombre de concepts plus avancés (foncteurs, *templates*, itérateurs, *etc*.) plus en détail.

Rôle de la sémantique de valeur

Un objet destiné à être un citoyen à part entière d'un programme C++ devrait offrir ce qu'on nomme parfois une sémantique de valeur [POOv00]; informellement, cela signifie que la Sainte-Trinité [POOv00] doit y être correctement définie.

L'essentiel des conteneurs standards, des algorithmes standards et des stratégies de programmation générique reposent sur l'idée d'**équivalence opérationnelle des types**. En C++, tout type est (au moins potentiellement) un citoyen de première classe du langage. Les conteneurs et les algorithmes standards, en ce sens, présument que les objets sous leur gouverne sont conçus pour être, eux-aussi, des citoyens de première classe des programmes.

Pour exploiter pleinement la bibliothèque standard à l'aide de vos propres objets, prenez soin d'assurer pour chacun une sémantique de valeur complète, ou de les encapsuler dans des objets offrant cette sémantique. Vérifiez que la Sainte-Trinité y soit convenablement supportée, de manière implicite ou explicite. Faites en sorte que votre code ne dépende pas du nombre de copies qui sera fait de vos objets.

La bibliothèque <iostream>

Le langage C++, comme la plupart des langages de programmation, offre des outils standards d'entrée/ sortie à la console à travers les objets globaux `cin`, `cout`, `cerr` et `clog`. Chacun de ces objets représente un flux et permet des entrées/ sorties pour les types sur lesquels sont définis des opérateurs d'insertion dans un flux et d'extraction d'un flux.

L'objet `cin` est une instance de `stream` (un flux en entrée), alors que `cout` est une instance de `ostream`.

Si vous êtes familière ou familier avec d'autres langages, en particulier le langage C, sachez que `cout` correspond à `stdout`, `cin` correspond à `stdin`, et `cerr` correspond à `stderr`.

Les entrées/ sorties standards sont extensibles, s'adaptant à divers types de données, comme nous l'avons vu en explorant la surcharge d'opérateurs [POOv00]. La lecture (ou l'écriture) d'une donnée avec ces outils se fait de manière uniforme au regarde du code client, peu importe le type de la donnée lue ou écrite.

La possibilité de surcharger les opérateurs sous forme de fonctions globales permet d'ajouter de manière homogène tout type maison aux types supportés de façon standard.

L'uniformité avec laquelle procèdent les entrées/ sorties proposées par `iostream` est étendue à l'ensemble des flux d'entrée/ sortie (incluant les fichiers – voir `fstream`)[19].

Les usages typiques des flux d'entrée/ sortie se rapportent, sur le plan opératoire, à l'utilisation de `cin` et de `cout`. Certains usages avancés apparaîtront plus loin, en particulier dans la section sur la bibliothèque `<algorithm>`.

Les flux standards sont très polyvalents. Des opérateurs d'écriture et de lecture génériques sont définis pour tous les types primitifs du langage (`bool`, `int`, `const char*`, *etc.*) et peuvent être créés pour des objets arbitraires (par exemple `string` ou `Rectangle`).

L'en-tête `<iostream>` est très volumineux. L'inclure force le compilateur à charger en mémoire une masse impressionnante d'information culturelle et linguistique, ce qui alourdit fortement le processus de compilation des programmes. Si un fichier d'en-tête n'a besoin que des déclarations des classes comme `istream` et `ostream`, alors privilégiez l'inclusion de l'en-tête **`<iosfwd>`** qui ne fait que décrire les types des flux ; n'incluez `<iostream>` dans un fichier d'en-tête qu'en tout dernier recours[20].

[19] Il y aura toujours de petites différences, quand bien même ce ne serait qu'une question des particularités des systèmes de fichiers sur une plateforme donnée, mais il y a une limite technique à ce qu'il est possible d'uniformiser dans un appareillage d'entrée/ sortie portable.

[20] Il peut arriver, surtout avec des *templates*, que vous n'ayez pas le choix…

La bibliothèque <string>

On prend parfois certaines choses pour acquis. Par exemple, notre apparente familiarité avec le texte nous fait penser qu'il s'agit là d'un élément de programmation primitif, fondamental. En pratique, manipuler les chaînes de caractères est quelque chose de fondamental mais qui n'est *vraiment* pas primitif.

Permettre à la fois flexibilité et sécurité quant au nombre de caractères dans une chaîne de caractères donnée, tout en minimisant l'espace requis pour entreposer cette chaîne, le tout sans que les opérations fondamentales sur les chaînes de caractères[21] ne deviennent si lentes qu'on ne veuille pas les utiliser, exige de faire des choix à la fois technologiques et conceptuels. Aucune des solutions qu'on puisse proposer n'est parfaite à la fois dans sa gestion de l'espace mémoire et dans la vitesse de ses opérations. Tous les choix viables sont des compromis entre l'un et l'autre.

La bonne gestion des chaînes de caractères se prête bien à une réflexion OO, et a été abordée par plusieurs, et de plusieurs manières. Rédiger une classe pour représenter les chaînes de caractères est un problème académique classique, auquel on ne se prête plus vraiment maintenant en C++ parce que les alternatives aux simples tableaux de caractères existent et sont agréables à utiliser.

Celles et ceux parmi vous qui ont une expérience de travail sur plateforme *Microsoft Windows* avec la bibliothèque MFC ont probablement développé une habitude avec la classe CString, qui y est un standard propriétaire. Celles et ceux qui utilisent C++/ CLI, qui est un dialecte .NET de C++ (et un langage très différent de C++ à plusieurs égards), ont sans doute rencontré la classe System::String.

CString accompagne la bibliothèque MFC de manière presque indissociable. Cette classe offre les services de base pour la représentation d'une chaîne de caractères, en plus d'offrir un support spécialisé pour transiger avec d'autres types propres aux environnements *Win32* (par exemple _bstr_t).

System::String accompagne la plateforme .NET et simplifie l'interopérabilité avec les autres outils sur cette plateforme, mais évitez-la hors de ce contexte pointu : la classe est coûteuse à utiliser en temps et en espace, étant immuable, et y avoir recours implique utiliser massivement l'allocation dynamique de mémoire.

La classe de la bibliothèque standard C++ pour représenter les chaînes de caractères est **string**[22], de <string>, et appartient à l'espace nommé std. C'est à cette classe que nous nous intéresserons[23].

[21] Évaluer la taille d'une chaîne, concaténer deux chaînes, comparer le contenu de deux chaînes, chercher la première occurrence d'une chaîne dans une autre chaîne, *etc*.

[22] Pas string.h, maintenant nommée cstring (ne pas confondre avec la classe CString de MFC), qui est une bibliothèque du langage C. Le contenu des deux fichiers est bien différent.

[23] Voir [hdStrConv] pour quelques techniques et outils permettant de passer de std::string ou de std::wstring à System::String et inversement.

Structure de `std::string`

Dans ce qui suit, les types `basic_string`, `string` et `wstring` sont tous logés dans l'espace nommé `std`.

La classe `string` de la bibliothèque standard de C++ repose sur un modèle générique, `basic_string`, appliqué au type `char`. Ce n'est qu'une spécialisation d'un *template*.

En effet, `basic_string` encode la mécanique d'une chaîne délimitée d'éléments d'un certain type `C`, dans la mesure où `C` se comporte comme un `char`. La classe `basic_string` est un conteneur : elle expose des itérateurs, peut être parcourue, et offre une large gamme de services.

La classe `basic_string` est une chaîne générique applicable à un type `C` qui se comporte comme un caractère. La classe `string` est une spécialisation de `basic_string` pour le type char. La classe `wstring` est une spécialisation de `basic_string` pour le type `wchar_t`.

L'approche générique fait en sorte que le type `string` soit très efficace : ses méthodes sont des algorithmes *purs*, pensés comme tels. Les opérations de base sur des chaînes, comme en évaluer la longueur, les concaténer ou les comparer, sont aussi efficaces (parfois même *plus* efficaces) que les équivalents fonctionnels des bibliothèques du langage C.

L'inconvénient de cette stratégie est que `basic_string` n'est pas pensé strictement en termes de chaînes de *caractères*, se prêtant à des chaînes de n'importe quel type.

Ainsi, le type `string` n'offre pas certaines fonctionnalités qui pourraient tirer profit d'une connaissance *a priori* du type utilisé dans la représentation, comme par exemple un constructeur qui prendrait un nombre en paramètre et construirait une représentation textuelle de ce nombre, ou des opérations de conversion d'un texte représentant un entier (ou un réel) dans le type représenté.

Des classes comme `CString`, qui n'ont pas été pensées pour des cas généraux et focalisent sur le cas du texte en soi, peuvent de leur côté se permettre ces opérations supplémentaires[24].

En C++, ces opérations peuvent être écrites (nous le verrons) mais sont définies à l'extérieur de `string`, pas à l'intérieur.

On obtient, au besoin, un `const char*` menant vers une représentation d'une `string` de bas niveau, compatible (entre autres) avec le langage C, en appelant sa méthode `c_str()`. Ceci fonctionne aussi avec les applications de `basic_string` sur d'autres types que `char`, évidemment[25].

Le pointeur retourné par `c_str()` ne restera valide que jusqu'à la prochaine modification de l'instance dont il a été tiré ou jusqu'à la destruction de cet objet. Évitez donc du code tel que :

```
const char *f()
{
   string temp{"allo"};
   return temp.c str();
} // temp est détruit
// le pointeur retourné est illégal
```

[24] Cela dit, voir la bibliothèque `sstream`, plus bas.

[25] Évidemment, la méthode `c_str()` d'un `basic_string<T>` retournera un `const T*`, pour tout type `T`.

Exemples d'utilisation de `string`

Allons-y maintenant, pour illustrer le propos, de quelques exemples simples d'utilisation du type `string` et des outils qui l'accompagnent.

Entrées/ sorties par mot ou par ligne

Avec la bibliothèque `<string>`, on trouve le type `string`, soit, mais aussi une gamme d'opérations d'entrée/ sortie standardisées qui, bien qu'externes au type, font partie de son interface. L'exemple ci-dessous utilise `cin` et `cout`, mais s'applique à tout flux d'entrée ou de sortie standard[26].

La lecture d'un `std::string` avec l'opérateur `>>` sur un flux d'entrée se complète lorsqu'un caractère d'espacement est rencontré. On peut donc lire un seul mot[27] à la console comme dans l'exemple à droite. Rien n'empêche, bien entendu, de lire plusieurs mots à l'aide d'une répétitive.

```
#include<iostream>
#include<string>
int main() {
   using namespace std;
   cout << "Mot :";
   string s;
   if (cin >> s)
      cout << s << endl;

   cout << "Texte : ";
   if (getline(cin, s))
      cout << s << endl;
}
```

Remarquez que la longueur du mot lu n'est pas une considération ici. L'opération de lecture d'un flux telle que définie pour une instance de `string` implante déjà la totalité des opérations pour assurer un espace suffisant, quel que soit le nombre de caractères à lire[28]. Utiliser `string` est un excellent moyen de sécuriser les programmes et d'éviter les débordements de zone tampon; en arriver au même résultat sans outils `OO` serait… pénible.

La lecture d'une ligne entière, allant jusqu'à un retour de chariot plutôt que jusqu'au premier caractère d'espacement trouvé, se fait à l'aide de la fonction `getline()`, livrée elle aussi avec le type `string`.

Notez le premier paramètre, qui est le flux d'entrée servant à la lecture : on peut utiliser la fonction avec un flux d'entrée sur un fichier comme avec un flux d'entrée sur le clavier.

Réflexion `02.0` : il y a une différence fondamentale entre lire une ligne et lire tous les mots d'une ligne. Lirait-on le même contenu au total dans un cas comme dans l'autre? Pourquoi? Réponse dans ***Réflexion `02.0` : des mots et des lignes***.

Notez que depuis C++ `17`, le code ci-dessus peut être amélioré en réduisant la portée de certaines variables. Ainsi, ci-dessous, à droite, la portée de `s` est le `if` où elle est déclarée :

Avant C++ `17`

```
string s;
if (cin >> s)
   cout << s << endl
```

Depuis C++ `17`

```
if (string s; cin >> s)
   cout << s << endl
```

[26] Voir `fstream` et `sstream`, plus bas, pour d'autres exemples de flux d'entrée/ sortie standards.

[27] On entend ici par *mot* une séquence d'au moins un caractère et délimitée à la fin par un caractère d'espacement. Les outils d'entrée/ sortie standard à la console imposent (pour des raisons de portabilité) qu'on complète l'entrée de données par un retour de chariot, ce qui fait que si l'usager entre le texte `Bonjour cher prof`, la première lecture d'un `string` lira `Bonjour`, mais les mots `cher` et `prof` resteront dans le tampon en mémoire pour lectures subséquentes.

[28] La structure interne à un `string`, servant à emmagasiner les caractères, est redimensionnée au besoin par les opérations de lecture sur un flux d'entrée. Et vive l'encapsulation!

Rechercher une sous-chaîne dans une chaîne

Pour vérifier la présence d'une chaîne dans une autre, le type `string` offre la méthode `find()`, de même plusieurs déclinaisons de méthodes similaires.

L'exemple suivant lit une ligne, puis un mot, et cherche la première occurrence du mot en question dans la ligne. La constante de classe `string::npos` sera retournée dans le cas où la recherche est un échec.

Vous noterez que pour l'essentiel, les métaphores de la bibliothèque standard de C++ s'expriment sur la base d'*itérateurs*, mais que les services de `string` tendent à opérer sur la base de *positions*. C'est un irritant de `string`, qui fait un peu office de vilain petit canard en comparaison avec ses comparses.

```
#include <string>
#include <iostream>
int main() {
   using namespace std;
   cout << "Ligne : ";
   if (string ligne; getline(cin, ligne)) {
      cout << endl << "Mot à chercher : ";
      if (string mot; cin >> mot) {
         auto pos = ligne.find(mot); // pos est un string::size_type
         if (pos != string::npos)
            cout << endl << mot << ", position " << pos << endl;
         else
            cout << endl mot << " introuvable dans \"" << ligne << "\"" << endl;
      }
   }
}
```

Note : si vous êtes familière ou familier avec les populaires outils `grep` sous `UNIX` ou Linux, vous avez peut-être déjà réalisé qu'en combinant ceci avec les entrées/ sorties sur un fichier (plus loin dans ce document), on peut facilement (!) implanter les bases d'un tel outil[29].

[29] La bibliothèque standard de C++ 11 inclut une bibliothèque d'expressions régulières, `<regex>`, prévue spécifiquement à cet effet.

Extraire une sous-chaîne d'une chaîne

Pour extraire une partie d'une chaîne, le type `string` offre la fonction `substr()` sous plusieurs déclinaisons. L'exemple suivant lit une ligne, puis un entier `n`, et extrait et affiche les `n` derniers caractères de la chaîne (ou affiche un message d'erreur si `n` est invalide).

Remarquez en particulier l'utilisation de la méthode `size()`, qui retourne la longueur d'une instance de `string` (de type `string::size_type`), exprimée en nombre de caractères. On aurait pu utiliser la méthode `length()`, qui fait la même chose, mais `size()` est défini sur tous les conteneurs standards et le code reposant sur cette méthode est plus facile à réutiliser.

```
#include <string>
#include <iostream>
int main() {
   using namespace std;
   cout << "Ligne: ";
   if (string ligne; getline(cin, ligne)) {
      cout << endl << "Nombre de caractères : ";
      if(int n; cin >> n) {
         if (n < 1 || static_cast<string::size_type>(n) > ligne.size())
            cerr << "Nombre de caractères " << n << " invalide";
         else {
            string s = ligne.substr(ligne.size() - n);
            cout << "Extrait : " << s << " (" << s.size() << " caractères)";
        }
      }
   }
}
```

Opérations naturelles sur une chaîne

Les opérations auxquelles on pourrait s'attendre d'une chaîne de caractères sont implantées. On pense notamment :

- à l'affectation;
- aux comparaisons (== et !=), qui font une distinction entre les majuscules et les minuscules, considérant les chaînes "Allo" et "allo" comme différentes;
- à l'ordonnancement (<, <=, > et >=, appliquées à l'ordre lexicographique);
- à la concaténation (+ et +=); et
- à des méthodes à peu près essentielles, par exemple insert() pour insérer une chaîne dans une autre, erase() pour effacer une partie d'une chaîne, clear() pour vider la chaîne, replace() pour remplacer un segment d'une chaîne, *etc*.

De manière générale, plusieurs versions existent pour la plupart de ces méthodes, et celles qui opèrent en bloc sont habituellement plus rapides que celles y allant un caractère à la fois. Prenez soin de consulter la documentation avant de programmer.

Avec +=, on pourrait d'ailleurs écrire assez simplement notre propre version de la fonction getline(), c'est-à-dire quelque chose comme ceci (à droite).

L'algorithme est simple :

- vider la chaîne dans laquelle nous écrirons;
- indiquer qu'on ne souhaite pas escamoter les blancs;
- consommer les caractères un à un jusqu'à une erreur de lecture (ce qui inclut la fin du flux) ou jusqu'à ce que le délimiteur souhaité soit atteint;
- replacer dans le flux le comportement standard qui est d'escamoter les blancs; et
- retourner le flux suite à la lecture.

```
#include <string>
#include <iostream>
// ...using...
istream& lire_ligne
   (istream &is, string &s, char delim = '\n')
{
   if (!is) return is;
   s.clear();
   is >> noskipws;
   for(char c; is.get(c) && c != delim; )
      s+= c;
   is >> skipws;
   return is;
}
```

Il se trouve que getline() est plus complexe (et beaucoup plus rapide!) que lire_ligne(), mais l'idée est la même. Du point de vue du code client, les deux fonctions ont le même comportement.

La bibliothèque <fstream>

L'abstraction des flux d'entrée/ sortie faite par C++ fait en sorte que les flux définis sur des fichiers et ceux définis sur les périphériques d'entrée/ sortie habituels (clavier, écran console) présentent précisément la même interface, le même comportement.

Manipuler des fichiers en C++

Parmi les flux possibles, les fichiers occupent une place particulière.

On les utilise à plusieurs sauces, en particulier lorsque le souci d'automatisation des paramètres à un programme se fait sentir ou lorsqu'on a recours à une forme de support permanent sans nécessairement vouloir avoir recours à une base de données.

Note sur l'homogénéité

La beauté du modèle OO vient en grande partie de son homogénéité. Cela transparaît entre autres dans le fait que les flux définis sur des fichiers se comportent de la même manière que les flux standard associés à la console et au clavier.

De manière générale, tout sous-programme écrit pour opérer sur un `istream` saura opérer sur n'importe quel flux d'entrée standard, peu importe ce à quoi correspond, physiquement, ce flux (fichier, chaîne de caractères, périphérique, *etc.*), et il en ira de même pour tout sous-programme opérant sur un `ostream`.

En situation de développement 3D ou avec contraintes de performance serrées, l'emploi d'un débogueur n'est pas toujours une option. Ainsi, pouvoir laisser rapidement et efficacement une trace dans un fichier des variables pertinentes à l'analyse de l'exécution d'un processus peut alors constituer l'une des meilleures pratiques envisageables.

Copier un fichier texte

L'exemple à droite montre comment on peut copier le contenu d'un fichier texte (nommé `src.txt`) dans un autre (nommé `dest.txt`).

```
#include <fstream>
#include <string>
#include <iostream>
int main() {
   using namespace std;
   const string
      SRC  = "src.txt",
      DEST = "dest.txt";
   ifstream in{SRC};
   ofstream out{DEST};
   for (string s; getline(in,s); )
      out << s << endl;
}
```

Ceci fonctionne, mais on peut faire *beaucoup* plus rapide.

Dans C++ 03, les classes `ifstream` et `ofstream` ne connaissaient pas la classe `string`, ce qui forçait les programmeuses et les programmeurs à utiliser la méthode `c_str()` d'une string pour passer un `const char*` en tant que nom de fichier aux classes représentant des flux sur des fichiers. Cet irritant est corrigé avec C++ 11.

Copier un fichier binaire

La même manœuvre est possible avec des fichiers de données brutes, comme ceux contenant des images ou du son. L'exemple à droite montre d'ailleurs comment il est possible de copier le fichier `src.jpeg` dans le fichier `dest.jpeg`.

```
#include <fstream>
#include <string>
int main() {
   using namespace std;
   const string SRC  = "src.jpeg",
                DEST = "dest.jpeg";
   ifstream in{SRC, ios::binary};
   ofstream out{DEST, ios::binary};
   for (char c; in.get(c); )
      out.put(c);
}
```

Ceci fonctionne, mais on peut faire *beaucoup* plus rapide.

Par défaut, les flux d'entrée et de sortie lisent du texte, ce qui signifie qu'ils accordent un traitement spécial à certains symboles (les retours de chariots, les changements de ligne, *etc*.). On utilise des entrées/ sorties brutes ici en ajoutant aux constructeurs de nos flux le paramètre `std::ios::binary`.

Plutôt que de lire les `char` d'un fichier binaire avec l'opérateur `>>`, qui accorderait à certaines valeurs un traitement spécial, on privilégiera la méthode `get()` pour un flux brut. Elle ne fait de faveur à personne.

L'écriture dans un `ostream` binaire avec l'opérateur `<<` fonctionne telle qu'on l'espère, quelle que soit la valeur de la donnée, mais cet exemple utilise `put()` pour que l'ensemble apparaisse plus homogène (par souci d'esthétisme).

Copier tous les mots d'un fichier texte à un autre

Un programme copiant tous les mots d'un fichier texte nommé `"in.txt"` dans un autre fichier texte nommé `"out.txt"` serait celui proposé à droite.

```
#include <fstream>
#include <string>
int main() {
   using namespace std;
   ifstream ifs{"in.txt"};
   ofstream ofs{"out.txt"};
   for (string s; ifs >> s; )
      ofs << s << ' ';
}
```

Ce programme remplacera le contenu du fichier `"out.txt"` si ce fichier existait déjà. Il s'arrêtera lorsque la lecture dans le fichier source aura échoué, ce qui se produira normalement lorsqu'on aura rencontré la fin du fichier.

Il ne s'agit pas d'un bon programme si on veut conserver le format complet du texte original puisque tous les caractères d'espacement originaux disparaissent en chemin, remplacés en sortie par des espaces génériques. Il est donc probable que les tailles des fichiers en entrée et en sortie soient différentes suite à l'exécution du programme.

Pour être orthodoxe, en milieu de production, on pourrait vérifier si l'ouverture du fichier en écriture a fonctionné en insérant un test sur l'échec du flux en écriture. Les deux écritures ci-dessous sont essentiellement équivalentes.

```
#include <fstream>
#include <string>
int main() {
   using namespace std;
   ifstream ifs{"in.txt"};
   ofstream ofs{"out.txt"};
   if (ofs.good())
      for (string s; ifs >> s; )
         ofs << s << ' ';
}
```

```
#include <fstream>
#include <string>
int main() {
   using namespace std;
   ifstream ifs{"in.txt"};
   ofstream ofs{"out.txt"};
   if (ofs)
      for (string s; ifs >> s; )
         ofs << s << ' ';
}
```

Autre solution, en apparence plus rigoureuse encore mais un peu plus lente car elle teste l'état du flux en sortie à chaque itération de la répétitive (ce qui peut être une bonne chose si on suspecte le média derrière le flux de ne pas être stable) :

```
#include <fstream>
#include <string>
int main() {
   using namespace stdl
   ifstream ifs{"in.txt"};
   ofstream ofs{"out.txt"};
   for (string s; ofs.good() && ifs >> s; )
      ofs << s << ' ';
}
```

```
#include <fstream>
#include <string>
int main() {
   using namespace std;
   ifstream ifs{"in.txt"};
   ofstream ofs{"out.txt"};
   for (string s; ofs && ifs >> s; )
      ofs << s << ' ';
}
```

En pratique, la version ne procédant qu'à un seul test initial est meilleure que celle testant le flux à chaque itération, du fait qu'une tentative de lecture sur un flux incorrect met le flux dans un état d'invalidité et fait en sorte que les tests sur ce flux échouent (le flux se comporte comme un booléen de valeur `false`).

Cela dit, on ne teste en pratique que rarement les flux en sortie, alors que les flux en entrée méritent d'être testés fréquemment.

Copier tout le texte (ligne par ligne) d'un fichier texte à un autre

Un programme copiant le texte tout entier, en conservant tous les caractères d'espacement, d'un fichier texte nommé `"in.txt"` dans un autre fichier texte nommé `"out.txt"` serait celui proposé ci-dessous.

Présumant que la source soit un fichier texte, la seule différence possible entre le fichier en entrée et le fichier en sortie se produira si la dernière ligne du fichier en entrée ne se termine pas par un changement de ligne (car notre version insère un changement de ligne à la fin de chaque ligne lue, et ce sans discrimination aucune).

Un correctif à ce léger irritant est possible mais est laissé en exercice (la documentation sur `getline()` explique comment il est possible de résoudre ce problème élégamment).

```
#include <fstream>
#include <string>
#include <iostream>
int main() {
   using namespace std;
   ifstream ifs{"in.txt"};
   ofstream ofs{"out.txt"};
   for (string s; getline(ifs, s); )
      ofs << s << endl;
}
```

Copier un fichier arbitraire

Tel que vu précédemment, un programme copiant le contenu tout entier d'un fichier arbitraire (image, musique, exécutable, peu importe) nommé `"in.xyz"` dans un autre fichier arbitraire nommé `"out.xyz"` serait celui à droite.

```
#include <fstream>
int main() {
   using namespace std;
   ifstream ifs{"in.xyz",  ios::binary};
   ofstream ofs{"out.xyz", ios::binary};
   char c;
   while (ifs.get(c))
      ofs.put(c);
}
```

Les amatrices et les amateurs de forme compacte privilégieront peut-être la forme à droite. On peut insérer une validation du flux en sortie, encore une fois.

Une approche plus sophistiquée et moins manuelle sera présentée dans la section ***Algorithmes standards et itérateurs sur un flux***, un peu plus loin.

```
#include <fstream>
int main() {
   using namespace std;
   ifstream ifs{"in.xyz",  ios::binary};
   ofstream ofs{"out.xyz", ios::binary};
   for (char c; ifs.get(c); ofs.put(c))
      ;
}
```

Plus vite encore

Si vous ne voulez qu'une copie *brute* du contenu d'un fichier, sans que le moindre traitement ne soit fait sur son contenu, ceci est encore plus efficace :

```
#include <fstream>
int main() {
   using namespace std;
   ifstream ifs{"in.txt",  ios::binary};
   ofstream ofs{"out.txt", ios::binary};
   ofs << ifs.rdbuf();
}
```

Une écriture plus compacte encore (mais moins lisible, alors privilégiez celle ci-dessus plutôt que celle ci-dessous) :

```
#include <fstream>
int main() {
   using namespace std;
   ofstream{"out.txt", ios::binary} <<
      ifstream{"in.txt", ios::binary}.rdbuf();
}
```

Ici, le flux en sortie (`ofs` dans le premier exemple) aspire tel un drain de lavabo la totalité des données du flux en lecture (`ifs` dans le premier exemple). La première version crée des variables locales nommées, détruites en fin de portée, alors que la seconde crée des variables anonymes pour la durée d'une expression seulement.

Journal d'exécution d'un programme

On peut garder une trace d'un programme dans un fichier en ouvrant ce fichier en écriture au début de l'exécution du programme, puis en y ajoutant des données (formatées pour être lues par des humains) alors que le programme s'exécute.

Une manière de faire une telle trace (ici, dans le but de valider l'exécution de la très sophistiquée procédure `permuter()`) serait celle proposé dans le programme à droite.

L'ouverture en mode `ios_base::app` permet d'insérer des données à la fin du fichier plutôt que de remplacer le contenu du fichier existant. Ainsi, un programme peut, pendant son exécution, laisser une trace lisible par laquelle on peut ensuite procéder à une analyse détaillée de son comportement.

Notre exemple à droite repose sur l'utilisation de destructeurs déterministes. Lors de la création (ou du remplacement) du journal d'exécution, dans `main()`, le flux est créé en remplacement dans une variable anonyme, ce qui la crée puis la détruit.

Par la suite, dans `permuter()`, le fichier est ouvert (mode d'ajout à la fin) par le constructeur et est fermé par le destructeur.

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
using namespace std;
const auto JOURNAL = "sortie.txt";
void permuter(int &a, int &b);
int main() {
   // Création ou remplacement du journal
   // (flux anonyme, ouvert puis fermé)
   ofstream{JOURNAL};
   int x, y;
   while (cin >> x >> y) {
      permuter(x, y);
      cout << x << ' ' << y << endl;
   }
}
void permuter(int &x, int &y) {
   ofstream ofs{JOURNAL, ios_base::app};
   ofs << "permuter(), début, x = "
       << x << ", y = " << y << endl;
   swap(x, y);
   ofs << "permuter(), fin, x = "
       << x << ", y = " << y << endl;
}
```

Une approche semblable, plus conforme au standard, est d'utiliser le flux standard `clog`. Ce flux en sortie est une entité globale, au comportement conforme à celui de `cout`, mais pour lequel l'usage veut que les écritures soient dirigées vers un fichier de journalisation (choisi par le programme) plutôt qu'à la console.

Selon cette approche, un flux en sortie est ouvert tôt dans le programme (ici, dans `main()`) et n'est fermé que quand le programme en a vraiment terminé (ici, par le destructeur de `journal` dans `main()`). Pendant que ce flux est ouvert, il est bien sûr possible d'y écrire.

L'association entre le flux en sortie journal, local à `main()`, et le flux global `clog` qui sera utilisé dans le programme se fait par partage direct des tampons en écriture. La méthode `rdbuf()` de `clog` reçoit en paramètre le tampon à utiliser, soit celui de journal, au début de `main()`. À la fin de `main()`, `clog()` se voit offrir un pointeur nul, ce qui fait en sorte que les écritures subséquentes disparaissent dans une espèce de trou noir.

Utiliser `clog` pour les journaux d'exécution d'un programme est une approche simple et accessible à la majorité des programmeuses et des programmeurs.

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
using namespace;
const auto JOURNAL = "sortie.txt";
void permuter(int &a, int &b);
int main() {
   // Création ou remplacement du journal
   ofstream journal{JOURNAL};
   clog.rdbuf{journal.rdbuf()};
   int x, y;
   while (cin >> x >> y) {
      permuter(x, y);
      cout << x << ' ' << y << endl;
   }
   clog.rdbuf(nullptr);
}
void permuter(int &x, int &y) {
   clog << "permuter(), début, x = "
        << x << ", y = " << y << endl;
   swap(x, y);
   clog << "permuter(), fin, x = "
        << x << ", y = " << y << endl;
}
```

Connaître la taille d'un fichier

Une question classique, à la fois simple et peu documentée, est celle-ci : comment connaître la taille (en *bytes*) d'un fichier? Doit-on en lire le contenu tout entier? Doit-on dépendre de fonctions propres à la plateforme?

Le truc est tout simple, pourtant. Il suffit :

- d'ouvrir le fichier à l'aide d'un flux en lecture;
- d'aller à la fin du flux; et
- de retourner la position courante une fois à cet endroit.

Un exemple de code faisant ce travail est proposé à droite. Une taille négative est possible si le fichier n'existe pas.

Détail : il est possible d'avoir le même résultat en écrivant tout simplement :

```
auto taille_fichier(const string &nom) {
   return ifstream{nom,ios::ate}.tellg();
}
```

où `ate` signifie *At End*.

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
auto taille_fichier(const string &nom) {
   ifstream fich{nom};
   fich.seekg(ios::end);
   return fich.tellg();
}
int main() {
   const string NOM = "test.dat";
   cout << "Fichier " << NOM << ", taille : "
        << taille_fichier(NOM) << " bytes."
        << endl;
}
```

Les méthodes clés ici sont `seekg()` et `tellg()` (ou, pour un flux en sortie : `seekp()` et `tellp()`). Elles permettent respectivement de déplacer le point de lecture (ou d'écriture) dans un flux et de connaître la position courante du point de lecture (ou d'écriture) relativement au début du flux. Le *g* de `seekg()` et de `tellg()` signifie *get* alors que le *p* de `seekp()` et de `tellp()` signifie *put*.

Vérifier l'existence d'un fichier

Une autre question classique dont la solution est toute simple est : comment vérifier l'existence d'un fichier? Ici encore, la technique est toute simple. Il suffit :

- d'ouvrir le fichier à l'aide d'un flux en lecture; et
- de vérifier s'il y a problème; ce qui signifie en C++
- tester ce fichier pour voir s'il se comporte comme un booléen vrai (valide) ou faux (invalide).

Un petit exemple de code permettant de réaliser cette tâche est proposé à droite.

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
bool fichier_existe(const string &nom) {
   return ifstream{nom};
}
int main() {
   const auto NOM = "Test.dat";
   cout << Le fichier " << NOM;
   if (fichier_existe(NOM))
      cout << " existe." << endl;
   else
      cout << " n'existe pas." << endl;
}
```

Conteneurs standards : la bibliothèque `<vector>` et autres[30]

Le conteneur standard le plus commun et le plus rapide (pour les opérations les plus fréquemment réalisées) de la bibliothèque standard de C++ est `vector`. Nous utiliserons ce conteneur pour la plupart des exemples généraux ci-dessous, mais nous discuterons de conteneurs standards et d'itérateurs de manière générale.

⇒ La classe **`vector`** (en-tête `<vector>`) est un conteneur générique analogue à un tableau de taille dynamique.
⇒ Cette classe est un bon candidat à remplacer les tableaux bruts pour la plupart des opérations. Lorsqu'elle est utilisée correctement, cette classe offre d'ailleurs un seuil de performance analogue à celui des tableaux – et parfois meilleur![31]

Le dynamisme de la taille d'un `vector` libère l'équipe de développement de la tâche souvent irritante de rédiger elle-même le code requis pour gérer des tableaux. L'ajout d'un élément[32] à une instance de `vector` ajuste automatiquement sa taille s'il y a lieu.

La classe `vector` est un digne représentant des conteneurs standards de C++.

Chaque conteneur standard montre des particularités structurelles qui lui sont propres. Certains font en sorte que l'insertion d'un élément soit rapide, d'autres facilitent l'acte d'itérer sur les éléments qui y sont contenus, d'autres encore sont plus longs à construire et consomment plus de mémoire mais sont, par la suite, très rapides d'accès. Dans certains cas, c'est l'ordre selon lequel les éléments insérés peuvent être extraits qui changera.

Ces particularités sont des détails d'implémentation. Avec C++, on choisira les conteneurs en fonction des besoins, mais les opérations comme insérer, extraire et traverser la séquence seront standardisées pour tous les conteneurs. Sachant cela, *presque* tout ce qui sera dit ici sur `vector` pourra être appliqué à *presque* tout conteneur standard.

Les méthodes permettant de redimensionner une instance `vector` sont publiques. Il est donc possible de dimensionner au préalable un tel objet si des besoins de performance spécifiques se présentent, ou d'appliquer un algorithme différent de celui utilisé par défaut pour déterminer sa nouvelle taille.

Les conteneurs, les itérateurs, les algorithmes et les foncteurs sont plus que des outils techniques : ce sont les bases d'une philosophie de programmation complète, amenée par un sous-ensemble précieux de la bibliothèque standard : la `STL`[33].

Dans les sections qui suivent, nous ferons un survol des conteneurs et des itérateurs pour eux-mêmes. Ensuite, nous irons plus loin en les combinant aux algorithmes standards.

---

[30] Pour des conteneurs similaires, vous pouvez aussi explorer `list`, `map`, `forward_list` et `deque` (quelques exemples parmi tant d'autres). Vous aurez pour chacun des variantes de performance en fonction des opérations à réaliser.

[31] Cela peut sembler suspect, puisque les vecteurs utilisent des tableaux bruts de manière sous-jacente, mais `vector` est une œuvre d'art, extrêmement optimisée, et écrire du code de cette qualité avec des tableaux bruts n'est vraiment, *vraiment* pas à la portée de tous. Voir [hdVecTab] pour des détails.

[32] … entendu qu'il faut procéder à un ajout à l'aide des méthodes prévues à cet effet pour bénéficier de ces mécanismes. L'encapsulation est directement responsable des gains que nous obtiendrons à l'utilisation de conteneurs génériques standard comme `std::vector`.

[33] Selon les sources, cet acronyme peut signifier *Standard Template Library* ou *Stepanov and Lee*, du nom des principaux initiateurs de ce projet.

Généralités sur les conteneurs standards

Les conteneurs standards, dont fait partie `vector`, offrent une interface de base semblable, de même qu'un ensemble de fonctionnalités pointues. Pour à peu près tous les conteneurs, il est raisonnable de s'attendre entre autres aux caractéristiques suivantes :

- par défaut, un conteneur nouvellement créé sera vide;
- un conteneur sera un type valeur, supportant pleinement la Sainte-Trinité [POOv00];
- les éléments d'un conteneur doivent être des types valeurs. Chaque conteneur est en droit de copier comme bon lui semble ses éléments;
- un conteneur exposera un méthode booléenne `empty()` qui retournera `true` seulement si ce conteneur est vide. Cette méthode s'exécutera en temps constant (complexité $O(1)$);
- un conteneur exposera un méthode `clear()` qui permettra de le vider (ce qui ne signifie pas que l'espace sera récupéré, par contre);
- un conteneur exposera souvent des méthodes `erase()` et `insert()` pour éliminer ou ajouter efficacement une séquence d'éléments;
- un conteneur exposera une méthode `size()` qui retournera le nombre d'éléments qu'il contient (connaître le nombre d'éléments d'un conteneur n'est pas la même chose que connaître sa capacité). Pour vérifier si le conteneur `c` est vide, il est préférable d'invoquer `c.empty()` plutôt que de tester si `size()==0` puisque `empty()` s'exécute toujours en temps constant alors que la performance de `size()` varie selon les conteneurs;
- un conteneurs dont les éléments peuvent être traversés exposera les méthodes `begin()` et `end()`, offrant des itérateur sur le début et sur la fin de la séquence traversable;
- pour les conteneurs dont les éléments peuvent être traversés, les méthodes `front()` et `back()` retournent une référence sur le 1er et sur le dernier élément du conteneur;
- ajouter un élément à la fin d'un conteneur se fait typiquement avec sa méthode `push_back()`.

La gamme de services des conteneurs est plus vaste que celle listée ci-dessus, mais il nous faudra discuter de philosophie pour bien comprendre la puissance et la polyvalence de ces outils.

Voici, au préalable, un exemple d'utilisation de conteneur standard avec `vector`. Notez que ce conteneur permet d'accéder directement, en temps constant, à un de ses éléments avec l'opérateur `[]`, mais que ce ne sont pas tous les conteneurs qui font de même.

Un vecteur standard appliqué à un type `T` peut, pour l'essentiel des tâches courantes, remplacer un tableau brut du type `T`.

```
#include <vector>
#include <iostream>
int main() {
   using namespace std;
   int a = 2, b = -5;
   // déclaration d'un vecteur d'entiers
   vector<int> v;
   // ajout de 3 int, par copie
   v.push_back(a);
   v.push_back(9);
   v.push_back(b);
   // v contient 2, 9, -5. Ceci les
   // affichera dans l'ordre
   for (vector<int>::size_type i = 0;
        i < v.size(); i++)
      cout << v[i] << endl;
}
```

Initialisation uniforme : `std::initializer_list`

Vous aurez peut-être remarqué une tare syntaxique agaçante dans l'initialisation d'un vecteur standard, à partir de l'exemple de la page précédente : le recours à une séquence de `push_back()` pour y insérer des valeurs, même si celles-ci sont connues *a priori*, alors qu'un tableau brut aurait pu être initialisé directement en insérant les valeurs entre accolades.

```
// C++03
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
int tab[] = { 1, 2, 3 };
```

Cette tare s'applique d'ailleurs aux classes en général sous C++ 03, qui n'ont pas (pour leur initialisaton) le même support que les agrégats du langage C (les `struct` et les tableaux bruts, par exemple). Un palliatif existe depuis longtemps, à travers ce qu'on nomme les constructeurs de séquence (voir ***Construction de séquence***, plus loin), mais une tare demeure une tare. C++ 11 corrige cet irritant avec des `initializer_list`, de l'en-tête standard `<initializer_list>`.

Désormais, pour les compilateurs suffisamment à jour, la syntaxe proposée à droite est légale, et puisque les valeurs entre accolades sont des `int`, le constructeur de `v` recevra une instance de `initializer_list<int>` en paramètre. Ceci permettra la rédaction de code client beaucoup plus homogène.

```
// C++11
vector<int> v = { 1, 2, 3 };
int tab[] = { 1, 2, 3 };
```

Notez que le recours à `=` avant la liste de valeurs est optionnel ; l'écriture à droite est toute aussi légale.

```
// C++11
vector<int> v { 1, 2, 3 };
int tab[] { 1, 2, 3 };
```

Conteneurs et itérateurs

Cette section repose en bonne partie sur des stratégies de programmation générique.

⇒ Un **itérateur** représente la position d'un élément dans un conteneur `STL`. *Le concept d'itérateur est une abstraction du concept de séquence*. Il s'agit d'un des schémas de conception les plus répandus.

⇒ En C++, l'analogue opératoire traditionnel d'un itérateur pour un tableau brut est un pointeur sur un élément d'un tableau.

Le conteneur organise les éléments, alors que l'itérateur permet de traverser une séquence d'éléments. On comprendra que chaque conteneur susceptible d'être travers offrira ses propres itérateurs, faits sur mesure pour opérer à partir de sa structure interne :

- un itérateur sur une liste chaînée suivra les nœuds de la liste un à un à partir de ce qui, dans un nœud, indique son successeur;
- un itérateur défini sur un tableau ou sur un vecteur trouvera le prochain élément à partir de simples opérations arithmétiques;
- un itérateur sur un arbre aura évidemment un algorithme de navigation plus riche et plus complexe; *etc*.

Un itérateur permet entre autres de traverser un conteneur sans avoir à se soucier de la structure interne de ce dernier. Syntaxiquement, les itérateurs ressemblent aux pointeurs ; en particulier, l'opérateur permettant d'aller au prochain élément d'un itérateur est `++` et l'opérateur permettant de connaître l'élément auquel réfère un itérateur est `*`. Cette proximité syntaxique permet d'écrire des algorithmes qui fonctionnent, de manière homogène, sur des conteneurs complexes comme sur des tableaux bruts.

Introduction aux itérateurs

Pour comprendre ce qu'est un itérateur, il est utile de se remettre en tête comment est structuré un tableau en langage C/ C++ : avec ces deux langages, un tableau est un pointeur sur le premier élément d'une séquence contiguë en mémoire d'éléments du même type.

Par définition, tous les éléments d'un même tableau sont consécutifs en mémoire. Ainsi, pour un type `T` quelconque et une constante entière `N` strictement positive, si `tab` est un tableau de `N` éléments de type `T` déclaré ainsi (à droite), alors :

```
T tab[N];
```

- la variable `tab` est l'adresse du premier élément du tableau;
- conséquemment, `tab + 0 == &tab[0];`
- de même, `*tab == tab[0]` puisque l'entier pointé par `tab` est celui qui se trouve à l'indice zéro du tableau.

On remarquera que `tab[0]` est l'élément trouvé en mémoire si on ajoute `0*sizeof(T)` *bytes* à l'adresse `tab`. En termes arithmétiques, `tab == tab+0` (mais certains compilateurs sont pointilleux alors `tab + 0` est plus portable que `tab`).

De même, `tab[1]` est l'élément trouvé en mémoire si on ajoute `1*sizeof(T)` *bytes* à l'adresse `tab` et, de manière générale, `tab[i]` est l'élément trouvé en mémoire si on ajoute `i*sizeof(T)` *bytes* à `tab`.

Clairement, pour tout entier `i` :

- `&tab[i] == tab+i`, donc
- `tab[i] == *(tab+i)`.

Manifestement, donc, les trois répétitives à droite devraient être absolument équivalentes[34]… et c'est effectivement le cas. Vous pouvez le vérifier pour vous en convaincre.

Il se trouve que, règle générale, on préférera la troisième forme, puisqu'il s'agit de la plus riche et de la plus expressive : elle fait la même chose que les précédentes, mais le fait *avec moins*.

```
int tab[N];
for (int i = 0; i < N; i++)
   tab[i] = -1;
for (int *p = &tab[0]; p != &tab[N]; p++)
   *p = -1;
for (int *p = tab+0; p != tab+N; p++)
   *p = -1;
```

Voir la section ***La bibliothèque <algorithm>*** pour en savoir plus sur les fonctions globales `std::begin()` et `std::end()` qui allègent beaucoup la syntaxe des itérateurs.

[34] … dans la mesure où le compilateur peut nous garantir que `&tab[N]`, qui est par définition hors du tableau `T`, soit une adresse se trouvant dans l'espace adressable du processus dans lequel loge `tab`. Heureusement, c'est bel et bien le cas: tout compilateur C++ garantit que le premier élément après un tableau soit dans l'espace adressable du processus dans lequel se trouve le tableau.

Si on comprend bien la syntaxe des pointeurs en C++ (et en C), on conviendra que :

- si `p` pointe sur un élément d'un tableau `tab`, alors `p++` fera pointer `p` sur le prochain élément de ce tableau (et il en ira de même pour `++p`);
- si `p` et `q` pointent tous deux sur des éléments d'un même tableau `tab`, alors on peut vérifier si `p` et `q` pointent au même endroit en utilisant les opérateurs `==` et `!=`; et
- si `p` pointe sur un élément d'un tableau `tab`, alors `*p` représente cet élément.

⇒ On dit que `p` est un **itérateur** pour le tableau `tab` puisqu'il permet de passer d'un élément à l'autre du tableau `tab` à l'aide de l'opérateur `++` et parce qu'on peut accéder, à travers lui, à l'élément pointé dans le tableau `tab` (une indirection utilisant l'opérateur `*`).

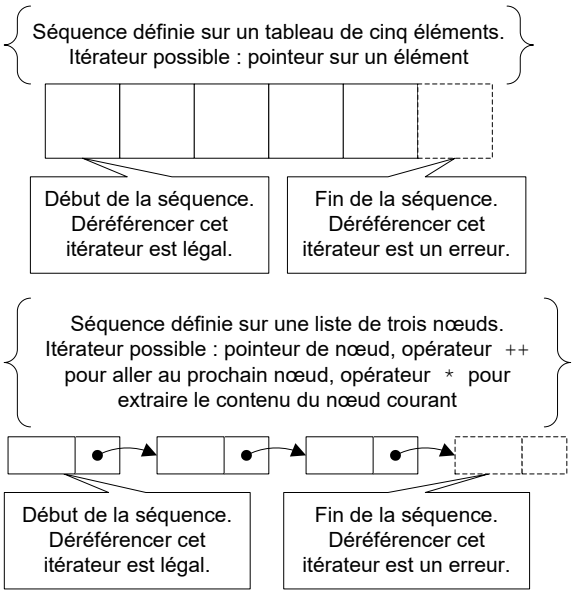
Sachant cela, on a généralisé le concept d'itérateur dans un tableau à celui d'*objet permettant d'abstraire le concept de séquence dans un conteneur*. Informellement, un itérateur pour une séquence `S` est un objet qui permet de traverser cette séquence (++), de vérifier si on a atteint un certain point dans la séquence (`==` ou `!=`), et d'accéder sur un élément de la séquence (`*`).

Une séquence aura nécessairement un début et une fin, qui seront tous deux des itérateurs. Avec la plupart des conteneurs standards, le début de la séquence sera retourné par un appel à la méthode `begin()` du conteneur et la fin de la séquence sera retournée par un appel à la méthode `end()` du même conteneur. *Dans un tableau `tab` de `N` éléments, le début est `tab+0` et la fin est `tab+N`.*[35]

Les conteneurs standards travaillent tous à partir de séquences définies dans un **intervalle à demi ouvert** (un *Half-Open Range*) :

- le début d'une séquence correspond à un itérateur sur le premier élément valide de cette séquence; et
- la fin d'une séquence correspond à un itérateur sur l'élément *tout juste après le dernier* élément valide de la séquence. On peut examiner cet itérateur mais *pas ce vers quoi il pointe*;
- déréférencer un itérateur de fin de séquence est une faute de logique.

L'exemple à droite montre deux exemples de séquences sur des conteneurs, définies par des itérateurs.



Une fois le concept d'itérateur assimilé, on obtient un outil de programmation *extrêmement* puissant, permettant de traiter sur un même pied les listes, les piles, les tableaux, les vecteurs et, de manière générale, tous les conteneurs.

[35] On peut aussi écrire `&tab[0]` et `&tab[N]` mais c'est plus lourd…

En effet, puisque tous les itérateurs suivent la même syntaxe, on peut écrire des *templates* qui exploitent leur similitude opératoire. Par exemple :

```
#include <iostream>
#include <vector>
using namespace std;
template <class T>
   void afficher(const T &v, ostream &os) { // projette un T sur le flux os
      os << v;
   }
template <class It>
   void afficher(It debut, It fin, ostream &os) { // projette (debut..fin( sur le flux os
      for (auto p = debut; p != fin; ++p)
         afficher(*p, os);
      os << endl;
   }
int main() {
   const int N = 10;
   int tab[N] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   vector<int> v;
   for (auto n : tab)
      v.push_back(n);
   afficher(&tab[0], &tab[N], cout);
   afficher(tab+0, tab+N, cout);
   afficher(v.begin(), v.end(), cout);
}
```

Remarquez que le traitement réservé à `tab` et à `v` est identique. Ce sont deux séquences, pour lesquelles il existe des itérateurs. Dans le cas de `tab`, le type de l'itérateur est `int*`, alors que dans le cas de `v`, le type de l'intérateur est `std::vector<int>::iterator` (itérateur de vecteur de `int`). La beauté est que *dans la mesure où ces types d'itérateurs se comportent de la même façon, l'algorithme générique `afficher()` fonctionnera pour les deux*.

Une conséquence de cette réalité est que tout algorithme capable d'opérer sur une séquence définie à partir d'une paire d'itérateurs fonctionnera aussi pour toutes les séquences standards, incluant les tableaux bruts[36].

[36] Nuance : un algorithme ne pourra aller à l'encontre des limites des itérateurs utilisés. Les itérateurs sont classés par catégorie (nous y reviendrons) et n'offrent pas tous les mêmes opérations (p. ex. : implémenter le recul d'un élément sur une liste simplement chaînée serait si lent qu'il serait contre-productif d'offrir l'opérateur `--` sur ses itérateurs). De même, inutile d'offrir un algorithme tentant d'accroître la taille d'un tableau brut.

Saveurs d'itérateurs

Il existe des itérateurs constants, qui ne permettent que de consulter les éléments, et des itérateurs non constants, qui permettent de lire et de modifier l'élément courant d'une séquence. La plupart des conteneurs offrent au moins ces deux saveurs.

```
#include <vector>
using std::vector;
// ...
vector<double> v;
// lecture et écriture
vector<double>::iterator iter_lecture_ecriture;
// lecture seulement
vector<double>::const_iterator iter_lecture_seule;
```

Sur un conteneur `const`, seuls les itérateurs constants (en lecture seule) seront accessibles. Dans l'exemple à droite, si nous avions utilisé `vector<T>::iterator` à titre de type pour `i`, le code n'aurait pas compilé. Dans quelques cas pathologiques, il est important d'assurer que les itérateurs obtenus sont `const`; avec C++ 11, les conteneurs offriront cette garantie avec les méthodes `cbegin()` et `cend()`.

Ici, le mot clé `auto` nous éviter d'avoir à écrire (sans erreurs) `vector<T>::const_iterator`…

```
// ...
template <class T>
   void afficher(const vector<T> &v, ostream &os) {
      for(auto i = v.begin(); i != v.end(); ++i)
         os << *i << ' ';
   }
```

Un itérateur donné ne peut, en même temps, itérer à travers deux conteneurs distincts : cela équivaudrait à faire parcourir à un pointeur l'espace entre deux tableaux distincts, chose illégale (le problème sera détecté à l'exécution… Boum!)

L'exemple à droite en témoigne : l'appel transformant en majuscules toutes les lettres de `s0` est correct mais l'appel utilisant une extrémité de `s0` et une extrémité de `s1` échouera à l'exécution.

La plupart des itérateurs permettent au moins d'aller au prochain élément d'une séquence (opérateur `++`). Certains permettent aussi de reculer (opérateur `--`) ; les conteneurs dont les itérateurs permettent d'avancer et de reculer exposent habituellement des méthodes `rbegin()` et `rend()` pour permettre de traverser aisément la séquence du dernier élément au premier.

Quelques types d'itérateurs, dont ceux définis pour `std::vector` et `std::string`, permettent d'aller à un élément arbitraire d'une séquence (par exemple à l'élément pointé par `begin()+4`), un peu comme on le ferait avec un pointeur sur un élément d'un tableau.

```
#include <string>
#include <locale>
using namespace std;
// met c en majuscules
void majuscule(char &c) {
   c = toupper(c, locale{""});
}
template <class It>
   void majuscules(It debut, It fin) {
      for (; debut!= fin; ++debut)
         majuscule(*debut);
   }
int main() {
   string s0 = "Allo les amis",
          s1 = "Coucou les potes";
   // Légal - une seule séquence
   majuscules(s0.begin(), s0.end());
   // Indéfini -- pas la même séquence
   majuscules(s0.begin(), s1.end());
}
```

Un itérateur **vers l'avant** (*Forward Iterator*) permet d'aller au prochain élément. Un iterateur **bidirectionnel** (*Bidirectional*) permet d'aller à l'élément précédent et à l'élément suivant de l'élément pointé. Un itérateur à **accès direct** (*Random Access*) offre la gamme complète des opérations arithmétiques qu'on connaît avec les pointeurs.

Les opérations disponibles sur un itérateur donné dépendent du type d'itérateur. Il faut d'ailleurs distinguer *opération sur l'itérateur* d'*opération sur l'élément vers lequel pointe l'itérateur dans le conteneur* : un itérateur est une abstraction de l'idée de position dans une séquence, pas une abstraction de l'idée de contenu.

Le tableau ci-dessous résume les principales opérations possibles sur un itérateur, constant ou non.

Opér.	Rôle	Disponibilité		
		Avant	Bidirect.	Accès direct
`==` `!=`	Compare deux itérateurs (*pas* ce vers quoi ils pointent) et retourne vrai si et seulement si ces itérateurs indiquent tous deux la même position (==) ou une position différente (!=) dans la séquence.	Oui	Oui	Oui
`*`	Retourne une référence (vers un référé constant ou non, selon le type d'itérateur) à l'élément pointé par l'itérateur. Le type de cette opération dépend du type des éléments du conteneur.	Oui	Oui	Oui
`++`	Fait pointer l'itérateur vers le prochain élément de la séquence.	Oui	Oui	Oui
`--`	Fait pointer l'itérateur vers l'élément précédent de la séquence.	*Non*	Oui	Oui
`+= -=` `+ -`	Permettent d'avancer ou de reculer d'une ou de plusieurs positions dans la séquence. Leur opérande de droite est un entier.	*Non*	*Non*	Oui
`[]`	Permettent d'avancer à un élément se trouvant un certain nombre (entier) de positions passé celle de l'itérateur dans la séquence. Pour un itérateur `itt` et un entier `n`, `itt[n]` équivaut à `*(itt+n)`.	*Non*	*Non*	Oui
`< <=` `> >=`	Comparent des itérateurs de même type au sens des positions vers lesquelles ils pointent dans le conteneur.	*Non*	*Non*	Oui

Utiliser les itérateurs

On peut utiliser des itérateurs à plusieurs fins. En voici quelques-unes. Retenez que, dans tous les exemples ci-dessous, on pourrait remplacer le mot `vector` par un autre nom de conteneur (par le mot `list`, pour donner un exemple) et on aurait quand même du code essentiellement valide.

Parcourir les éléments d'un conteneur

L'exemple ci-dessous crée un vecteur d'entiers nommé `v`, dans lequel quelques entiers arbitraires sont insérés, la plupart à la fin (`push_back()`) et un au début (méthode `insert()` avec comme point d'insertion `begin()`).

Insérer ailleurs qu'à la fin d'un vecteur est très lent puisque la structure sous-jacente est un tableau. Les insertions à des endroits arbitraires entraînent plusieurs copies d'éléments, alors qu'une insertion à la fin est une opération de complexité *constante amortie*, donc constante en général sauf quand le vecteur doit accroître sa capacité.

L'itérateur `it` procède du début du vecteur `v` à la fin de ce vecteur, s'arrêtant lorsqu'il en atteint la fin. Souvenons-nous que les intervalles standards sont à demi-ouverts, donc que la fin d'un conteneur n'est pas une position représentant un élément valide dans la séquence, alors que le début du conteneur, lui, est valide.

On passe d'un élément à l'autre à l'aide de l'opérateur `++` de l'itérateur, et on affiche le contenu pointé par l'itérateur (`*it`) à chaque étape.

```
#include <vector>
#include <iostream>
int main() {
   using namespace std;
   vector<int> v;
   v.push_back(0);
   v.push_back(4);
   v.insert(v.begin(), 6);
   for (auto it= v.begin(); it != v.end(); ++it)
      cout << *it << ' ';
   cout << endl;
}
```

Accéder à un membre à travers un itérateur

Puisque l'accès à un élément référencé par un itérateur se fait à l'aide de l'opérateur `*`, ou à travers `->` pour l'accès à un membre d'un tel élément, il faut y aller de prudence pour s'assurer que l'effet obtenu soit celui désiré.

Ainsi, prenez soin d'appliquer des parenthèses stratégiquement pour contrecarrer les effets de bord possibles résultant de la priorité des opérateurs. Dans l'exemple à droite, l'appel `(*it).size()` invoque une méthode d'une string, ce qui est raisonnable, alors que l'écriture `*it.size()` aurait signifié une tentative d'invoquer la méthode `size()` de l'itérateur (qui n'en a pas) et de déréférencer le résultat de l'appel. Cela aurait été dénué de sens.

Dans cet exemple, nous aurions aussi pu utiliser un `const_iterator`.

```
#include <vector>
#include <string>
#include <iostream>
int main() {
   using namespace std;
   vector<string> mots;
   string mot;
   for (int i = 0; cin >> mot && i < 10; ++i)
       mots.push_back(mot);
   string::size_type total = 0;
   for (auto it = mots.begin(); it != mots.end(); ++it)
      total += (*it).size();  // ou itt->size()
   cout << "Somme de la longueur des mots : "
        << total << endl;
}
```

Alléger la syntaxe

Vous avez sûrement remarqué que le type d'un itérateurs sur les éléments d'un conteneur donné tend à être long, du fait qu'il s'agit d'un type interne à une classe générique (`vector<int>::iterator`, sans compter le `std::` si on choisit de ne pas utiliser un alias à l'aide d'un `using`).

La généricité peut nous aider à alléger la syntaxe du code client, du fait que le compilateur déduit alors les types impliqués et génère silencieusement le code approprié.

La version proposée à droite fait précisément la même chose que l'exemple précédent, mais en nous épargnant l'écriture des types. Les appels aux méthodes `begin()` et `end()` retournent des itérateurs d'un type donné, et le compilateur génère la définition correcte de `somme_tailles()` à partir de ce type.

Le seul irritant de la version à droite est que le cumul des tailles y est fait sur un `int`, alors que ce type peut ne pas être approprié (le type d'une taille de `string`, après tout, est `string::size_type`, et rien ne garantit que la capacité d'un `int` suffira pour le cumul).

```
#include <vector>
#include <string>
#include <iostream>
template <class It>
   int somme_tailles(It debut, It fin) {
      int total = 0;
      for (; debut != fin; ++debut)
         total += debut->size();
      return total;
   }
int main() {
   using namespace std;
   vector<string> mots;
   string mot;
   for (int i = 0; cin >> mot && i < 10; ++i)
       mots.push_back(mot);
   cout << "Somme de la longueur des mots : "
         << somme_tailles(mots.debut(),mots.end())
         << endl;
}
```

Il existe des techniques pour résoudre cet irritant qui, pour cet exemple, est bien petit, mais qui peut devenir sérieux dans des projets de plus grande envergure ou qui impliquent des champs d'application dans les domaines militaire ou aérospatial, par exemple. Nous couvrirons ces techniques, plus avancées, dans des volumes ultérieurs.

Nous pouvons alors utiliser **`auto`** pour alléger l'écriture. Cela ne nous dispense pas de savoir ce que nous faisons, toutefois.

Ceci peut s'écrire…

```
template <class T>
   void f(vector<T> &d, const vector<T> &s) {
      vector<T>::iterator i = d.begin();
      vector<T>::const_iterator j = s.begin();
      // ...
   }
```

…comme cela

```
template <class T>
   void f(vector<T> &d, const vector<T> &s) {
      auto i = d.begin(); // iterator
      auto j = s.begin();// const_iterator
      // ...
   }
```

Constructeurs de séquences

Les conteneurs standards comme `vector` offrent tous un type de constructeur qu'on nomme le **constructeur de séquence**. Le rôle d'un constructeur de séquence est d'initialiser un conteneur à partir du contenu d'une séquence, elle-même susceptible de représenter le contenu tout entier d'un autre conteneur.

Les séquences étant définies à partir d'une paire d'itérateurs, voici une série d'exemples voulant illustrer cette mécanique et son utilité.

```
#include <vector>
#include <list>
#include <string>
int main() {
   using namespace std;
   int tab[]{ 1, 2, 3, 4, 5 };
   vector v0{tab + 0, tab + 5}; // v0 contient des copies des éléments de Tab
   vector v1{tab + 0, tab + 3}; // 1, 2, 3
   list lst{tab + 3, tab + 5}; // 4, 5
   v0 = vector(lst.begin(), lst.end()); // 4, 5
   string s0 = "coucou";
   string s1(s0.begin() + 3, s0.end()); // "cou"
}
```

L'abstraction puissante que constitue le concept d'itérateur, jointe aux constructeurs de séquences, permet de copier le contenu de n'importe quelle séquence (incluant le contenu de n'importe quel conteneur traversable) dans n'importe quel conteneur sans avoir, pour ce faire, à implémenter des constructeurs spécifiques.

Le fait d'offrir des constructeurs de séquence permet aussi d'utiliser sans peine un tableau comme source de données pour initialiser un conteneur.

La bibliothèque `<algorithm>`

La bibliothèque `<algorithm>` contient quelques algorithmes standards, efficaces et très utiles. Ces algorithmes sont trop nombreux pour énumérer ici, mais sachez qu'on retrouve :

- plusieurs algorithmes de tri bien rodés : un tri fusion avec `sort()`, un autre qui a un meilleur pire cas mais un moins bon cas moyen avec `stable_sort()`, un plus rapide qui ne trie qu'un sous-ensemble des valeurs avec `partial_sort()`, *etc.*;
- un algorithme permettant d'appliquer une même opération à tous les éléments d'un conteneur, `for_each`, qui offre un seuil de performance à la hauteur d'une répétitive rédigée de manière optimale;
- des algorithmes pour inverser l'ordre des éléments d'un conteneur, le plus simple étant sans doute `reverse()`;
- plusieurs algorithmes de recherche d'éléments, incluant des algorithmes de recherche linéaire avec `find()` et `find_if()`, de même qu'une recherche dichotomique avec `binary_search()`;
- l'algorithme `copy()` pour copier les éléments d'une séquence vers une destination, `generate()` pour initialiser une séquence, `random_shuffle()` pour mélanger les éléments d'une séquence, `fill()` pour remplir une séquence avec une valeur connue au préalable, *etc*.

Un exemple simple, qui utilise la fonction `sort()` pour trier les éléments d'un `vector`, et `for_each()` pour appliquer la procédure `afficher()` à chaque élément du `vector`.

```
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>
using namespace std;
void afficher(const string &s) {
   cout << s << ' ';
}
int main() {
   string lieux[] = {
      "Québec",
      "Ontario",
      "Saskatchewan"
   };
   vector<string> v{lieux+0 lieux+3};
   // tri appliqué au vecteur
   sort(v.begin(),v.end());
   for_each(v.begin(),v.end(),afficher);
   // ça marche aussi sur un tableau
   sort(lieux+0,lieux+3);
   for_each(lieux+0,lieux+3,afficher);
}
```

Plusieurs conteneurs offrent des versions spécialisées de certains algorithmes, et qu'il arrive que ce soit en réponse à une subtilité technique. Par exemple, `list` n'offre pas d'itérateurs convenant aux besoins de `sort()`[37] mais offre par contre une méthode `sort()` faite spécialement pour tirer profit de sa représentation interne.

Dans les cas où un algorithme et une méthode offrent le même service pour un conteneur donné, la méthode sera habituellement plus efficace que l'algorithme standard, même si ce dernier offrira souvent un seuil de performance acceptable.

[37] L'algorithme `sort()` demande des itérateurs à accès direct, mais il se trouve que `list<T>::iterator` est un itérateur bidirectionnel.

Les algorithmes standards sont en grande partie orthogonaux aux conteneurs, étant définis en termes d'itérateurs. Pour cette raison, ajouter un algorithme standard conforme aux usages de la `STL` (qui opère sur des intervalles à demi ouverts) permet d'enrichir chaque conteneur existant d'un nouvel outil. Cette idée d'***Alexander Stepanov*** et de son équipe s'est avérée précieuse et féconde depuis son intégration dans le standard de C++ en `1998`.

Prenons par exemple `std::for_each()`, qui est habituellement représenté à peu près comme ceci dans une bibliothèque standard donnée[38] :

```
template <class It, class Op>
   Op for_each(It debut, It fin, Op oper) {
      for (; debut != fin; ++debut)
         oper(*debut);
      return oper;
   }
```

Escamotons le type de retour de `for_each()` (nous y reviendrons dans la section ***Foncteurs***, plus loin) et concentrons-nous sur l'essentiel :

- cette fonction prend des itérateurs constituant un intervalle à demi ouvert, `(debut..fin(;`
- le code client est responsable de fournir des itérateurs sur une même séquence et dans le bon ordre (valider ceci entraînerait des coûts prohibitifs, et `STL` est orienté vers la très haute performance);
- la fonction applique `oper`, de type `Op`, sur chaque élément de la séquence.

Ce code est correct, il traite la boucle de manière optimale (et peut-être même mieux qu'une boucle manuelle le serait, du fait que le compilateur connaît les intentions générales du code client et est en mesure de réaliser des optimisations supplémentaires profitant de ce savoir), ne commet pas d'erreurs, et est indépendant à la fois du type de conteneur parcouru ou du type des éléments de ce conteneur.

La fonction `for_each()` a quelques exigences, rendues explicites par son écriture. En particulier, il faut que tout `Op` puisse être utilisé en lui suffixant des parenthèses (ce qui est vrai entre autres si `Op` est une fonction) et en plaçant entre ces parenthèses une entité du type pointé par `It`.

Par exemple, si `It` est un `int*` et si `Op` est une fonction prenant un `int` en paramètre, alors ce contrat est respecté. Par contre, si `Itt` est un `vector<string>::iterator` et si `Op` demeure une fonction prenant un `int` en paramètre, alors le contrat est brisé et le code ne compilera pas – il n'y aura pas de coût à l'exécution.

[38] Chaque implémentation de la `STL` peut avoir ses variantes d'implémentation, dans la mesure où certaines garanties sont respectées (préconditions, postconditions, complexité algorithmique). Vous pouvez examiner le texte du fichier `<algorithm>` si vous le souhaitez : les *templates* sont définis dans les fichiers d'en-tête et peuvent être lus par qui le souhaite.

Quelques algorithmes à connaître

L'algorithme `for_each()` est sans doute le plus simple à expliquer et le plus fréquemment utilisé des algorithmes standards, mais il n'est pas le seul (loin de là!). En voici quelques autres, pour piquer votre curiosité – une bien petite liste étant donné la taille de l'arsenal mis à notre disposition par la `STL`, mais elle vous inspirera peut-être à fouiller un peu par vous-mêmes[39].

Les algorithmes `std::begin()` et `std::end()`

Syntaxe : `begin(c)` et `end(c)`

Se trouve dans : `<algorithm>`.

Rôle : retourner respectivement un itérateur sur le début du conteneur `c` et un itérateur sur la fin du conteneur `c`. Fonctionne avec tout conteneur standard, incluant les tableaux bruts. Pour un tableau `T tab[N]`, `begin(tab)==tab+0` et `end(tab)==tab+N`. Pour un conteneur standard c muni des méthodes `begin()` et `end()`, `begin(c)==c.begin()` et `end(c)==c.end()`. Les fonctions `begin()` et `end()` respectent les qualifications `const` des conteneurs.

Complexité : $O(1)$.

Exemple :

```
#include <algorithm>
#include <ostream>
using std::ostream;
using std::endl;
template <class It>
   void afficher_sequence(It debut, It fin, ostream &os) {
      for(; debut != fin; ++debut)
         os << *debut << ' ';
   }
#include <list>
int main() {
   using std::list;
   int tab[] { 2,3,5,7,11 };
   afficher_sequence(begin(tab), end(tab));
   list<int> lst{begin(tab), end(tab)};
   afficher_sequence(begin(lst), end(lst));
}
```

[39] Pour en savoir plus sur l'arsenal en question, voir [hdStl].

Note : les répétitives `for` généralisées utilisent l'équivalent de `std::begin()` et `std::end()` pour déduire les extrémités du conteneur à parcourir. Ainsi, en pratique, l'extrait de programme suivant :

```
int tab[] { 2, 3, 5, 7, 11 };
for(const auto & val : tab)
   cout << val << endl;
```

… est équivalent à e qui suit :

```
int tab[] { 2, 3, 5, 7, 11 };
for(auto i = begin(tab); i != end(tab); ++i) {
   auto & val = *i;
   cout << val << endl;
}
```

L'algorithme `std::distance()`

Syntaxe : `distance(debut, fin)`

Se trouve dans : `<iterator>`.

Rôle : calculer et retourner le nombre de pas requis pour avancer de `debut` à `fin`. Calcule donc la taille de la séquence `(debut..fin(`

Complexité : $O(n)$ dans le pire cas, soit une séquence d'itérateurs quelconques sur une séquence de n éléments, et $O(1)$ si les itérateurs sont à accès direct.

Exemple :

```
#include <iterator>
#include <ostream>
#include <list>
using namespace std;
template <class It>
   void afficher_taille_sequence(It debut, It fin, ostream &os) {
      os << distance(debut, fin) << endl;
   }
int main() {
   int tab[] { 2,3,5,7,11 };
   afficher_taille_sequence(begin(tab),end(tab));
   list<int> lst { begin(tab), end(tab) };
   afficher_taille_sequence(begin(lst), end(lst));
}
```

L'algorithme `std::for_each()`

Syntaxe : `for_each(debut, fin, oper)`

Se trouve dans : `<algorithm>`

Rôle : appliquer une même opération `oper` aux éléments d'une séquence `(debut..fin(`

Complexité : $O\big(distance(debut, fin) \times O(oper)\big)$ pour une opération $oper$ donnée.

Exemples : voir plus haut (il y en a plusieurs).

Note : les répétitives `for` généralisées sont susceptibles de remplacer plusieurs cas d'applications de `for_each()` en C++, mais pas toutes, car `for_each()` opère sur une séquence standard alors que `for(auto...)` opère sur un conteneur duquel sont déduites les extrémités d'une telle séquence. Les cas d'utilisation des deux formes se chevauchent mais ne se subsument pas.

L'algorithme `std::transform()`

Syntaxe : `transform(debut, fin, dest, oper)`

Se trouve dans : `<algorithm>`

Rôle : appliquer une même fonction aux éléments d'une séquence source et déposer la valeur retournée par cette fonction dans une séquence destination, qui doit être au moins aussi grande que la séquence source. Il est légal que la source et la destination soient une même séquence.

Complexité : $O\big(distance(debut, fin) \times O(oper)\big)$ pour une opération $oper$ donnée.

Exemple :

```
#include <algorithm>
#include <cmath>
using namespace std;
double racine_carree(int n) {
   return sqrt(static_cast<double>(n));
}
int main() {
   int tab[]{ 2,3,5,7,11 };
   double racines[5];
   transform(begin(tab), end(tab), racines, racine_carree);
}
```

L'algorithme `std::copy()`

Syntaxe : `copy(debut, fin, dest)`

Se trouve dans : `<algorithm>`.

Rôle : copier les éléments d'une séquence source dans une séquence destination, qui doit être au moins aussi grande que la séquence source.

Complexité : $O\big(distance(debut, fin)\big)$ présumant une copie en temps constant.

Exemple :

```
#include <algorithm>
int main() {
   using std::copy;
   int tab[]{ 2,3,5,7,11 };
   double tab_d[5];
   copy(begin(tab), end(tab) tab_d);
}
```

L'algorithme `std::sort()`

Syntaxe : `sort(debut, fin); sort(debut,fin,pred)`

Se trouve dans : `<algorithm>`

Rôle : Trier les éléments d'une séquence. Les itérateurs doivent être à accès direct. Par défaut, le prédicat appliqué pour déterminer un ordre entre deux éléments est l'opérateur $<$ mais il est possible de suppléer un autre prédicat.

Complexité : $O\left(distance(debut, fin) \times log_2\left(distance(debut, fin)\right)\right)$

Exemple :

```
#include <algorithm>
bool plus_grand(int a, int b) {
   return a > b;
}
int main() {
   using std::sort;
   int tab[]{ 3,7,2,11,5 };
   sort(begin(tab), end(tab));
   // mettre en ordre decroissant
   sort(begin(tab), end(tab), plus_grand);
}
```

L'algorithme `std::accumulate()`

Syntaxe : `accumulate(debut,fin,init); accumulate(debut,fin,init,fct)`

Se trouve dans : `<numeric>`

Rôle : Cumuler les éléments d'une séquence et retourner le résultat du cumul. Par défaut, cumule avec `operator+` mais d'autres fonctions peuvent être utilisées.

Complexité : $O\big(distance(debut, fin) \times O(fct)\big)$

Exemple :

```
#include <numeric>
#include <iostream>
int produit(int cur, int val) {
   return cur * val;
}
int main()
{
   using std::accumulate;
   int tab[]{ 3,7,2,11,5 };
   std::cout << "Somme: " << accumulate(begin(tab), end(tab),0) << std::endl;
   std::cout << "Produit: " << accumulate(begin(tab), end(tab),1,produit) << std::endl;
}
```

Et il y en a beaucoup, *beaucoup* d'autres!

Algorithmes standards et itérateurs sur un flux

Les flux d'entrée/ sortie, par exemple les fichiers, sont des lieux logiques où il est possible d'écrire et d'où il est possible de lire des données. En ce sens, il serait logique de pouvoir leur appliquer des algorithmes standards.

Par exemple, écrire le contenu d'une séquence sur un fichier (ou le contenu d'un fichier dans un autre fichier) devrait pouvoir se faire à l'aide d'un algorithme standard comme `copy()`.

Cette abstraction peut être réalisée à l'aide de ce qu'on nomme des **itérateurs en entrée** (*Input Iterator*) et des **itérateurs en sortie** (*Output Iterator*). Nous effleurerons le sujet ici, pour le reprendre plus tard quand nous serons prêts, avec le programme proposé à droite. Ce programme copie à la console le contenu d'un fichier nommé `in.txt` contenant des entiers, et le fait sans itérateurs.

```
#include <iostream>
#include <fstream>
int main() {
   using namespace std;
   int tmp;
   ifstream ifs{"in.txt"};
   while (ifs >> tmp)
      cout << tmp;
}
```

Une version plus compacte, dans laquelle la portée du flux se limite à la répétitive, serait celle à droite. Cette version n'a pas non plus recours à des itérateurs en entrée ou en sortie.

```
#include <iostream>
#include <fstream>
int main() {
   using namespace std;
   int tmp;
   for (ifstream ifs{"in.txt"}; ifs >> tmp;)
      cout << tmp;
}
```

Ces deux premières versions du programme sont relativement conventionnelles. On en arrive au même résultat avec le programme ci-dessous, qui est radicalement plus compact et plus rapide.

Examinez la structure de ce code :

- la copie d'une source à une destination y est clairement exprimée (recours à l'algorithme `copy()`);
- le début de la source est un itérateur en entrée consommant des `int` d'un flux sur le fichier `"in.txt"`. Le flux ainsi créé est éphémère, n'existant que pour les fins de l'invocation et mourant immédiatement ensuite ;
- la fin de la source, par convention, est un itérateur en entrée par défaut ; et
- le début de la destination est un itérateur en sortie projetant des `int` sur la sortie standard (séparés les uns des autres par un blanc).

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>
int main() {
   using namespace std;
   copy(istream_iterator<int>{ifstream{"in.txt"}},
        istream_iterator<int>{},
        ostream_iterator<int>{cout, " "});
}
```

Cette formulation, bien que surprenante de prime abord, est à la fois concise et autodocumentée, en plus d'utiliser des outils standards, disponibles partout, et fortement optimisés.

La bibliothèque <memory>

La bibliothèque `<memory>` offre plusieurs d'outils pour effectuer des opérations de gestion appliquées sur de la mémoire vive. Pour la plupart, il s'agit d'opérations de bas niveau, sur lesquelles nous reviendrons – elles sont utiles à la conception de conteneurs sécuritaires à très haute performance.

La raison pour laquelle nous mentionnons ce fichier d'en-tête ici est qu'il contient la déclaration du type générique `std::auto_ptr`, déprécié depuis C++ 11, ainsi que de son successeur `unique_ptr` et de `shared_ptr`, soit des types de pointeurs intelligents auxquels est en partie consacrée la section ***Introduction aux pointeurs intelligents***, plus loin.

Un autre élément important de cette bibliothèque est la déclaration générique des *allocateurs*, qui constituent une partie très spécialisée de la bibliothèque standard. Une ébauche d'allocateur simple est décrite dans la section sur les ***mixin***.

Exemple d'utilité du type `unique_ptr`

À titre d'introduction aux précieux outils de `<memory>`, examinez le code proposé à droite.

```
#include "X.h"
void f();
int main() {
   X *p = new X;
   f();
   delete p;
}
```

Ici, si `f()` lève une exception, l'instruction `delete p;` ne sera jamais exécutée, menant à une fuite de mémoire (problème probablement mineur) et à une non-finalisation du `X` vers lequel pointe `p` (problème beaucoup plus grave, surtout si ce `X` gère un lien réseau, une connexion à une base de données, ou une autre ressource précieuse).

Le type `unique_ptr` est ce qu'on nomme un *pointeur intelligent*, donc un objet qui se comporte comme un pointeur mais implémente une sémantique particulière quant à son pointé. Dans le cas du type `unique_ptr`, la sémantique est celle de responsabilité exclusive : un `unique_ptr` est responsable de son pointé, et le détruira lorsque son destructeur sera sollicité.

Ainsi, ce qui suit libérera toujours le pointé, qu'une exception soit levée ou non :

Mieux

```
#include "X.h"
#include <memory>
void f();
int main() {
   auto p = std::unique_ptr<X>(new X);
   f();
}
```

Encore mieux

```
#include "X.h"
#include <memory>
void f();
int main() {
   auto p = std::make_unique<X>();
   f();
}
```

La bibliothèque <sstream>

Détail qu'on néglige souvent, mais qui existe sous une forme ou une autre depuis longtemps : un flux d'entrée/sortie est une abstraction sous laquelle peuvent se cacher plusieurs substrats. Par exemple :

- des périphériques précis (clavier, écran);
- des fichiers; et, ce qui nous intéresse ici,
- des chaînes de caractères, incluant les `string`.

Traiter une chaîne de caractères comme un flux d'entrée/sortie permet d'y écrire et d'en lire avec les opérations normalement définies sur un tel flux. Puisqu'une instance de `std::string` réside en mémoire vive plutôt que sur un média de masse, l'accès y est très rapide. Plusieurs langages ont recours à cette technique (même C le fait avec `sprintf()` et `sscanf()`).

Passer par un tel flux, de type `stringstream`, permet entre autres de traduire des données d'un type à un autre dans la mesure où les types impliqués peuvent être projetés sur un flux.

L'exemple à droite illustre comment on peut appliquer cette technique pour convertir un `int` en `string`. La stratégie se prête à une approche générique.

```
#include<iostream>
#include<string>
#include<sstream>
using namespace std;
// convertir i en string
string itos(int n) {
   // le flux à utiliser
   stringstream sstr;
   // on écrit le int sur le flux
   sstr << n;
   return sstr.str(); // Bingo!
}
int main() {
   if (int i; cin >> i) {
      auto s = "Nombre: " + itos(i);
      cout << s << endl; // voilà!
   }
}
```

Pour la conversion d'un entier en `string` et inversement, C++ 11 offre toute une gamme de fonctions telles que `std::to_string()` pour divers types de paramètres, de même que des fonctions telles que `std::stoi()` (*`string to int`*), `std::stoull()` (*`string to unsigned long long`*), … Mieux vaut les utiliser que de les réécrire.

Dans d'autres langages

Là où C++ offre une bibliothèque standard puissante mais minimaliste, en partie dû à la quasi orthogonalité des algorithmes et des conteneurs, Java et les langages .NET sont systématiquement livrés avec des bibliothèques d'outils extrêmement imposantes, et offrent fréquemment aux classes et interfaces standards un support différent de celui offert aux types que vous pourriez programmer vous-mêmes (pensez seulement au *Boxing* pour un exemple très simple de traitement préférentiel offert aux classes systèmes).

C++ est un langage, mais Java et les langages .NET ne sont pas que des langages : ce sont des plateformes à part entière. Il est hors de question de couvrir en totalité les infrastructures (*Frameworks*) de Java ou de .NET dans un document comme celui-ci. Un tel effort serait futile : l'espace et le temps requis pour y arriver seraient considérables et le tout serait périmé une fois écrit.

Nous examinerons toutefois, de manière succincte, les collections (analogues Java et .NET aux conteneurs standards de C++) et les itérateurs de chacune de ces plateformes pour obtenir un aperçu des similitudes entre chacune, et des stratégies que ces outils rendent possibles.

Nous examinerons les collections de Java 1.5+ et des langages .NET 2.0+ pour tenir compte des versions génériques des diverses collections offertes (l'introduction de la généricité dans ces plateformes est un très clair progrès en comparaison avec les versions antérieures, sur lesquelles vous pouvez lire dans Internet sir le cœur vous en dit).

Dans chaque cas, à titre d'illustration, nous écrirons l'équivalent d'un petit programme remplissant un tableau dynamique puis itérant à travers lui pour en afficher les éléments.

Java offre une classe générique `Vector` dans le paquetage `java.util`. La version non générique de ce conteneur entrepose des `Object` qu'on doit convertir explicitement pour les utiliser, alors que la version générique est fortement typée et évite ces bris d'encapsulation.

La méthode `add()` d'un `Vector` permet d'insérer un élément à la fin de ce conteneur et le redimensionne au besoin. Vous remarquerez dans l'exemple ci-dessous que nous passons à `add()` des valeurs de type primitif (des `int`), mais les collections Java ne peuvent contenir que des objets. Conséquemment, un *Boxing* (création silencieuse d'une instance d'`Integer`) entre en jeu, ce qui allège l'écriture mais ralentit l'exécution et accroît le coût en mémoire du programme. Mieux vaut en être conscient(e)s.

Les deux stratégies les plus répandues pour itérer à travers un conteneur *itérable* comme `Vector` sont :

- d'obtenir un itérateur (type générique `Iterator`, de `java.util`) puis d'invoquer les méthodes `hasNext()`, qui teste l'atteinte de la fin du conteneur, et `next()`, qui obtient l'élément courant et avancer au prochain; et
- utiliser la forme compacte de la répétitive `for` de Java qui offre un support spécial aux conteneurs *itérables* et aux tableaux.

Un conteneur itérable en Java implémente l'interface générique `Iterable`, et offre une méthode `iterator` retournant un `Iterator` permettant de le parcourir. Les tableaux Java sont de tels objets.

Ceci démontre que le langage travaille de pair avec les classes et offre un support particulier à certaines d'entre elles : tel que mentionné précédemment, *Java est une plateforme*.

```
import java.util.*;
public class Z {
   private Vector<Integer> créer() {
      Vector<Integer> v = new Vector<Integer>();
      for (int i = 0; i < 5; ++i) {
         v.add(i+1); // Boxing
      }
      return v;
   }
   private void afficher(Vector<Integer> v) {
      Iterator<Integer> itt = v.iterator();
      while (itt.hasNext()) {
         System.out.println(itt.next());
      }
      // équivalent
      for (Integer i : v) {
         System.out.println (i);
      }
   }
   public static void main(String[] args) {
      Z z = new Z();
      z.afficher(z.créer());
   }
}
```

Comme C++, Java travaille avec des intervalles à demi ouverts, mais plutôt que de tester si l'itérateur courant a dépassé le dernier élément de la séquence, le code Java examine s'il reste au moins un élément à traiter (méthode `hasNext()`) puis passe au suivant si cela s'avère (méthode `next()`, qui progresse dans la séquence et retourne le prochain élément du même coup). Pour des raisons sur lesquelles nous reviendrons, la métaphore préconisée par Java n'est sécuritaire que si un moteur de collecte d'ordures soutient l'exécution du programme.

Avec C#, le type générique `List` de l'assemblage `Generic`, lui-même dans l'assemblage `System.Collections`[40], est sans doute ce qu'on trouve de plus proche d'un tableau dynamique.

La méthode `Add()` permet d'ajouter un élément à une `List`. Un itérateur s'y nomme un `Enumerator` qui est une classe interne au conteneur et dépendante du type de ses éléments. J'ai utilisé l'interface générique `IEnumerator` dans l'exemple à droite pour expliciter le lien avec le code Java et pour rendre le code plus compact.

Le langage offre lui aussi un support spécial pour traverser un conteneur *itérable*. Avec `.NET`, un conteneur *itérable* implémente une interface nommée `IEnumerable` et offre une méthode `GetEnumerator()`. Cette méthode retourne un dérivé approprié de l'interface `IEnumerator`, et est utilisée implicitement à travers la forme répétitive `foreach`.

```
using System.Collections.Generic;
namespace zz
{
   class Z {
      public List<int> créer() {
         List<int> v = new List<int>();
         for (int i = 0; i < 5; ++i) {
            v.Add(i + 1); // Boxing
         }
         return v;
      }
      public void afficher(List<int> v) {
         IEnumerator<int> itt = v.GetEnumerator();
         while (itt.MoveNext()) {
            System.Console.WriteLine(itt.Current);
         }
         // autre version
         foreach (int i in v) {
            System.Console.WriteLine(i);
         }
      }
   }
   class Program {
      static void Main(string[] args) {
         Z z = new Z ();
         z.afficher (z.créer ());
      }
   }
}
```

[40] Cet assemblage contient aussi les collections non génériques des versions antérieures de l'infrastructure et qui, comme en Java, manipulent des `object` qu'il faut convertir explicitement avant d'utiliser.

Avec VB.NET, les types des conteneurs et des énumérateurs sont identiques à ceux de C#, du fait que l'infrastructure est celle de .NET pour chacun de ces langages.

Remarquez la forme répétitive prise en charge par le langage, qui st légèrement différente de celle proposée par C# (le mot Each est un élément lexical paramétrant le comportement de la répétitive For; il n'y a pas de répétitive foreach distincte en VB.NET).

```
Imports System.Collections.Generic
Namespace zz
   Public Class Z
      Public Function créer() As List(Of Integer)
         Dim v As New List(Of Integer)
         For i As Integer = 0 To 4
            v.Add(i + 1) ' Boxing
         Next
         Return v
      End Function
      Public Sub afficher(ByVal v As List(Of Integer))
         Dim itt As IEnumerator(Of Integer) = v.GetEnumerator()
         While (itt.MoveNext())
            System.Console.WriteLine(itt.Current)
         End While
         ' autre version
         For Each i As Integer In v
            System.Console.WriteLine(i)
         Next
      End Sub
   End Class
end namespace
Module Test
   Public Sub Main(ByVal args As String())
      Dim z As New zz.Z()
      z.afficher(z.créer())
   End Sub
End Module
```

Les langages .NET procèdent comme le fait Java : définissant une séquence à demi ouverte, leurs itérateurs débutent avant le début de la séquence, et les parcours se font typiquement tant qu'il y a encore au moins un élément à parcourir.

Programmation générique appliquée

Comment utiliser sagement et efficacement les surprenantes avenues qui s'offrent à nous de par l'avènement de la programmation générique? La question intéresse beaucoup de gens (dont votre humble serviteur) et mène à des techniques poussées comme la ***métaprogrammation***, la ***programmation par politiques*** et les ***traits***, qui sont tous des sujets couverts dans [POOv03] ou qui tombent sous la rubrique de sujets avancés de conception OO.

Dans cette section, nous examinerons quelques subtilités techniques et philosophiques de la programmation générique, puis nous explorerons quelques pratiques de base dans l'*application* de la programmation générique. Nous examinerons entre autres comment construire un type générique `Tableau`, représentant un tableau dynamique de valeurs d'un type générique. Une instance de `Tableau` sera consciente de ses bornes. Nous ferons au passage quelques brèves incursions dans les subtilités de la programmation générique.

Nous n'aurons pas l'ambition de faire compétition avec le type `vector` de la bibliothèque standard (voir ***Bibliothèque standard de C++***), mais bien celle d'explorer les ramifications de la pensée générique dans un contexte pédagogique mais crédible.

Considérations préalables

Un élément philosophique à la base de la démarche générique (et de l'informatique en général) peut s'exprimer ainsi : *lorsque le code est en partie complexe, il est préférable que la complexité soit du côté du serveur plutôt que du côté du client*. Tout serveur, tout objet, sera utilisé beaucoup plus souvent qu'il ne sera conçu, après tout.

En POO, cette philosophie se reflète entre autres à travers le principe d'encapsulation. En effet, un objet correctement encapsulé cache de ce fait les détails de sa mécanique interne et de son évolution.

Dans les systèmes répartis, le serveur est un spécialiste et cherche à exposer pour ses clients des services à la fois efficaces et simples à utiliser.

Nous l'avons entrevu dans [POOv00] : les types génériques sont parfois déclarés à travers deux fichiers d'en-tête distincts, soit un très léger qui n'expose que des déclarations *a priori* et l'autre exposant les classes et outils dans toute leur splendeur.

Pensez par exemple à `<iostream>` et à sa compagne `<iosfwd>`. Ceci permet au code client de savoir qu'un type existe et de l'utiliser indirectement, sans avoir à inclure toute la mécanique qui lui est associée.

En programmation générique, la généricité élève le seuil d'abstraction chez les conceptrices et les concepteurs des types et des algorithmes génériques, mais le code client devrait naturellement profiter du fait que les outils génériques s'appliquent *entre autres* à lui, à ses besoins.

Généricité, tableaux et les pointeurs bruts

En C++, passer des pointeurs en paramètre à une fonction ou à un *template* provoque un phénomène de décrépitude de ce pointeur (ce qu'on nomme en anglais le *Pointer Decay*).

Ainsi, présumant que `sizeof(short)==2` et présumant que `sizeof(short*)==4`, l'exemple proposé à droite affichera à la console `4 4 20` même si le paramètre passé à `g()` est clairement un tableau de `10 short`. Le type des paramètres perd en teneur sémantique, passant de tableau avant l'appel à simple pointeur dans le sous-programme appelé.

Cela peut entraîner certaines surprises pour qui souhaite définir des *templates* à partir de types qui sont, au fond, des pointeurs. Pensez par exemple, à des chaînes de caractères brutes.

```
#include <iostream>
using namespace std;
void f(short *p) {
   cout << sizeof p << ' ';
}
void g(short (&p)[10]) {
   cout << sizeof p << ' ';
}
int main() {
   short tab[10];
   f(tab);
   g(tab);
   cout << sizeof tab << endl;
}
```

Une illustration maintenant classique du phénomène est celle proposée par **David Vandevoorde** et **Nicolai M. Josuttis** dans [CppTemp]. L'exemple en question ressemble fortement à celui proposé ci-dessous.

Dans cet exemple, toutes les invocations de `min_ok()` compilent parce que cette fonction manipule des `const char *`. Les paramètres effectifs sont des tableaux de caractères de taille connue[41] alors que les paramètres formels sont de simples pointeurs grâce à la décrépitude.

La seconde invocation de `min_ow()`, toutefois, ne compile pas. En effet, `min_ow()` reçoit en paramètre des références vers une version constante du type original, or dans ce cas les littéraux d'origine que sont `"allo"` et `"yo"` ne sont pas du même type avant décrépitude, les séquences constantes pointées n'étant pas de même taille. Il se trouve que la fonction générique `min_ow()` s'attend, elle, à recevoir deux `const T &` *avec le même* `T` *dans chaque cas*.

```
#include <iostream>
using namespace std;
template <class T>
   const T& min_ow(const T &a, const T &b) {
      return a < b ? a : b;
   }
template <class T>
   T min_ok(T a, T b) {
      return a < b ? a : b;
   }
int main() {
   cout << min_ow("allo", "hola") << '\n'
        << min_ow("allo", "yo")   << '\n'
        << min_ok("allo", "hola") << '\n'
        << min_ok("allo", "yo")   << endl;
}
```

Le problème ne serait évidemment pas survenu avec des `string`. Il faut faire preuve de prudence en mêlant généricité et pointeurs bruts.

[41] Le littéral `"allo"` est de type pointeur sur une séquence constante de cinq `char` (ou `char const [5]`) et le littéral `"yo"` est de type pointeur sur une séquence constante de trois `char` (ou `char const [3]`). Dans les deux cas, le délimiteur de fin est comptabilisé dans la taille de la séquence.

Généricité par classe ou par méthode

Avec les classes et les `struct`, la généricité peut s'exprimer par type ou par méthode. Les deux approches sont valables et ont leurs applications, mais il y a une nuance entre ce que chacune représente et permet de faire.

Le `struct` nommé `X`, présenté à droite, est générique sur la base de son type. Le mot `X` en soi n'est pas une classe mais bien un modèle permettant au compilateur de générer des classes, alors que `X<int>` est une classe à part entière, distincte de `X<string>`.

Une instance de `X<T>` expose une méthode d'instance nommée `val_defaut()`, retournant un `T` par défaut, et un type interne et public `value_type`, correspondant au type `T` de ce `X<T>`.

Le programme en exemple à droite montre comment il est possible d'instancier un `X<int>` et un `X<string>` puis d'invoquer la méthode `val_defaut()` de chacune de ces instances.

```
template <class T>
   struct X {
      using value_type = T;
      static value_type val_defaut() {
         return {};
      }
   };
#include <string>
int main() {
   using std::string;
   X<int> x0;
   auto i = x0.val_defaut();
   X<string> x1;
   auto s = x1.val_defaut();
}
```

Évidemment, dans cet exemple, les types `int` et `X<int>::value_type` sont identiques, et il en va de même pour `string` et `X<string>::value_type`.

Le `struct` nommé `Y` à droite n'est pas générique mais expose une méthode générique. Ainsi, dans un programme donné, il existera une classe `Y` mais le nombre de méthodes d'instances `afficher()` de cette classe variera selon l'utilisation qui en sera faite : le compilateur générera autant de méthodes que nécessaire, une par type auquel elle sera appliquée.

Ce programme donne un exemple d'utilisation de ce `struct : Y` aura dans ce cas deux méthodes `afficher()` distinctes, l'une ayant comme premier paramètre un `int` et l'autre prenant plutôt une `std::string`, ceci *parce que c'est l'utilisation que le programme en fait*.

Les deux approches (généricité par type ou par méthode) ont leurs avantages, et peuvent être combinées.

```
#include <iostream>
#include <string>
using namespace std;
struct Y {
   ostream &os;
   Y(ostream &os) : os{os} {
   }
   template <class T>
      void afficher(const T &val) const {
         os << val << ' ';
      }
};
int main() {
   Y y { cout };
   int i = 3;
   string s = "Coucou";
   y.afficher(i);
   y.afficher(-3);
   y.afficher(s);
}
```

Généricité et méthodes polymorphiques

On pourrait être tenté d'intégrer à un objet des méthodes à la fois polymorphiques (donc sujettes à être spécialisées chez les enfants) et génériques. Concrètement, *ce n'est pas possible*. En retour, il est possible pour un type générique d'exposer des méthodes polymorphiques. Quelle phrase, n'est-ce pas? Examinons le tout avec un exemple pour nous y retrouver un peu.

La classe[42] générique `Indirecteur` proposée à droite est un exemple légal impliquant une méthode virtuelle et une méthode abstraite.

Bien qu'`Indirecteur` soit un type générique, la classe `Indirecteur<T>` pour un `T` donné est une classe comme les autres et peut exposer des méthodes virtuelles.

```
template <class T>
   struct Indirecteur {
      virtual void operer(T *) = 0;
      virtual ~Indirecteur() = default;
   };
```

Tout enfant d'`Indirecteur<T>` devra implémenter la méthode `operer(T*)` et pourra spécialiser sa mécanique de destruction. Le nombre de méthodes polymorphiques d'un cas particulier d'`Indirecteur<T>` est connu au préalable et ne dépend que du type `T` appliqué à la concrétisation cette classe.

En retour, on ne peut avoir de méthodes qui soient à la fois génériques *et* polymorphiques. Voyons un exemple pour mieux comprendre.

La classe `Utilisateur` proposée à droite expose un destructeur virtuel et une méthode virtuelle[43] nommée `operer()`, tous deux légaux. Elle offre aussi un nombre arbitrairement grand de méthodes `afficher(T)` pour divers types `T`, ce qui est absolument correct.

Toutefois, sa méthode générique `agir()`, elle, est illégale car elle cherche à être à la fois générique et polymorphique[44].

```
#include <iostream>
using std::ostream;
struct Utilisateur {
   // légal
   virtual void operer(Utilisateur *);
   // légal si un T peut être
   // projeté sur un flux
   template <class T>
      void afficher
         (const T &obj, ostream &os) {
         os << obj << ' ';
      }
   // illégal... pourquoi?
   template <class T>
      virtual void agir(T *) = 0;
   virtual ~Utilisateur() = default;
};
```

[42] J'utilise classe et `struct` de manière interchangeable puisque les nuances entre les deux n'ont pas d'impact sur notre discussion.

[43] … méthode qui pourrait être abstraite, mais ça ne changerait rien à notre propos.

[44] Qu'elle soit abstraite ou non ne change rien au propos.

Pour comprendre le problème, mettons-nous dans la peau de ce pauvre compilateur :

- les méthodes polymorphiques sont implémentées par des indirections. Une table de pointeurs de méthodes sera définie dans toute instance possédant au moins une méthode polymorphique et l'adresse de la bonne méthode à invoquer pour un objet donné y sera déposée;
- si une méthode est générique sur un type `T` quelconque, cela implique qu'une version distincte de cette méthode sera générée pour chaque type `T` en fonction de laquelle elle sera utilisée en pratique;
- ne sachant pas d'avance pour combien de types distincts cette méthode sera invoquée dans un programme donné, ni si plusieurs fichiers sources utiliseront des types distincts pour la même méthode, le compilateur ne peut tout simplement pas créer la table de pointeurs de méthodes requise pour permettre le polymorphisme sur des méthodes génériques.

Le problème est donc que la liste des méthodes polymorphiques d'une classe doit être connue à la compilation de cette classe, mais la généricité dépend de l'utilisation qu'en fait le code client, dérivés de la classe inclus. Il est impossible de générer la table de méthodes virtuelles du parent lors de sa compilation si les éléments de cette table dépendent d'éventuels enfants encore inconnus (en `POO`, répétons-le, les parents ne connaissent pas leurs enfants).

Le polymorphisme et la généricité peuvent être combinés de plusieurs manières différentes et constituent deux atouts puissants dans la boîte d'outils des informaticiennes et des informaticiens aujourd'hui, mais une méthode ne peut être à la fois générique et polymorphique.

Une classe générique, pas à pas

Entreprenons maintenant en détail le design d'un petit type générique `Tableau`, annoncé d'entrée de jeu.

Définir le type visé

Pour être efficaces, il nous faut d'abord définir clairement notre objectif. Ici, nous voulons un conteneur nommé `Tableau` qui sera capable :

- de contenir des éléments d'un certain type;
- d'ajouter, sur demande, un élément à la fin de ceux qu'il contient déjà;
- d'agir comme un véritable type valeur, supportant la Sainte-Trinité;
- d'être comparé à l'aide des opérateurs `==` et `!=`;
- d'offrir un accès direct en consultation ou en modification à l'un de ses éléments;
- de croître en taille au besoin;
- de révéler sa propre taille, calculée en nombre d'éléments;
- de nous révéler s'il est vide; et, globalement
- de se comporter, pour l'essentiel, comme un tableau.

Nous devrons, au fur et à mesure de notre développement, définir clairement quel sera le contrat applicable au type `T`. Nous devrons aussi baliser sans ambigüité ce que *doit* et ce que *ne doit pas* faire le type `Tableau`. Ces deux éléments sont aussi importants l'un que l'autre.

Certaines considérations techniques devront être examinées plus en détail au passage. Par exemple, que signifie *être vide* pour une instance du type `Tableau`? Cela peut impliquer économiser de l'espace mémoire et faire en sorte que le pointeur qui servira de représentation interne soit nul, ou faire en sorte que le pointeur soit nécessairement valide (pointe sur un espace où il est possible d'écrire) tout en travaillant à partir du compteur du nombre d'éléments valides à l'intérieur.

Les deux approches ont du bon.

Les idées de *nombre d'éléments* et de *capacité* sont des idées distinctes. L'une représente l'espace alloué pour entreposer des éléments (capacité) alors que l'autre représente le nombre d'éléments entreposés à partir du début.

Nous aurons besoin des deux pour être en mesure de savoir quand il deviendra nécessaire de réserver de l'espace[45].

Un autre détail auquel nous porterons attention est la possibilité (ou non) d'offrir des garanties de sécurité face à la levée d'exceptions. En programmation générique, dû au peu d'information connue sur les types auxquels nous appliquons nos propres types ou nos propres algorithmes, cette question n'est pas banale.

Voir [BoostExcS] pour en savoir plus à ce sujet.

[45] Si nous voulons ne garder que le nombre d'éléments et lui donner le rôle supplémentaire de capacité, alors nous devrons réallouer de l'espace à chaque ajout d'élément, stratégie très coûteuse en temps d'exécution.

Invariants à respecter

Les invariants sont ces caractéristiques d'un objet qui doivent en tout temps[46] s'avérer pour que l'objet soit considéré valide. L'encapsulation est l'outil privilégié par lequel un objet assure le respect de ses invariants. Nos invariants pour `Tableau` sont :

- la valeur de l'attribut `nelems` est précisément le nombre d'éléments entreposés dans l'objet. Cet attribut peut avoir la valeur zéro si l'objet est considéré vide, mais n'aura jamais de valeur négative;
- la valeur de l'attribut `cap` est précisément la capacité d'entreposage (exprimée en nombre d'éléments) du substrat interne de l'objet. Cet attribut peut avoir la valeur zéro si l'objet est considéré vide, mais n'aura jamais de valeur négative;
- l'attribut `elems` ne sera nul que si le `Tableau` est vide. S'il n'est pas nul, alors il contient l'adresse du 1er élément d'un tableau brut, alloué dynamiquement, de type `T`.

[46] … hormis pendant la construction ou pendant la destruction, là où l'objet est en pleine mutation.

Construction par défaut et destruction

Revenons à notre démarche de rédaction d'une classe `Tableau`. Une fois définies les abstractions de base pour les types utilisés à titre de valeur contenue et pour les questions de taille et de capacité (types internes et publics `value_type` et `size_type`), nous définirons le sens à donner aux idées de `Tableau` par défaut et de destruction d'un `Tableau`.

```
#include <algorithm>
#include <iterator>
// ...using...
```

Respectant les usages (et l'intuition), nous choisirons ici d'établir un invariant : **un `Tableau` par défaut sera vide**. Cet invariant influencera l'écriture du code client, qui comptera dessus pour ses propres opérations.

```
template <class T>
   class Tableau {
   public:
      using value_type = T;
      using size_type = size_t;
   private:
      value_type* elems {};
      size_type cap {},
                nelems {};
   public:
      Tableau() = default;
      ~Tableau() {
         delete [] elems;
      }
```

À l'interne, nous devrons par contre déterminer ce que signifie *être vide* pour un `Tableau`. Visiblement, le nombre d'éléments (`nelems`) sera nul, mais pour le reste, deux grandes options s'offrent à nous :

- faire en sorte que le pointeur d'éléments (`elems`) soit nul, donc que la capacité (`cap`) initiale soit zéro. Ne pas allouer de ressources dans ce constructeur rend cette opération *no-throw*.; ou
- allouer à l'interne un tableau d'une taille par défaut et faire en sorte que la capacité prenne initialement cette valeur.

Les deux se valent : la première est plus économique en espace et permet d'avoir un constructeur par défaut ne levant pas d'exceptions, alors que la seconde peut être économique en temps si la capacité par défaut convient à la majorité des tâches courantes.

Il faut faire un choix, alors je choisirai la première option ici, mais chaque choix a ses vertus et influence l'ensemble du code subséquent pour notre classe : déterminer que `elems` puisse être nul nous impose de tenir compte de cette réalité dans les méthodes subséquentes, alors qu'allouer un tableau initial à travers `elems` ferait en sorte que le cas du pointeur nul soit une impossibilité.

> N'oublions pas que, si construire un `T` est une opération pour laquelle nous ne pouvons *a priori* rien affirmer quant aux risques de levée d'exceptions, le type `T*` demeure un type primitif : sa construction est sans risque. Initialiser `elems` à `nullptr` ne lèvera donc jamais d'exception, peu importe le type `T`.

Il est important, que le type développé soit générique ou non, d'assurer que la construction laisse l'objet dans un état stable et connu, respectueux de ses invariante. Ici, initialiser `elems` à `nullptr` rend le destructeur banal car, en C++, `delete nullptr;` et `delete [] nullptr;` sont deux opérations inoffensives.

Un destructeur ne devrait jamais lever d'exceptions. Dans notre implémentation, cette contrainte est effectivement respectée… sauf si `T::~T()` lève lui-même une exception et contrevient à cette règle d'or!

Nous implémenterons quelques accesseurs de premier ordre, pour consulter la capacité d'un `Tableau` et pour connaître le nombre d'éléments qu'il contient.

Les noms `size()` et `empty()` ont été choisis pour fins conformité avec les conteneurs standards.

```
private:
   bool full() const noexcept {
      return capacity() == size();
   }
public:
   size_type capacity() const noexcept {
      return cap;
   }
   size_type size() const noexcept {
      return nelems;
   }
   bool empty() const noexcept {
      return size()==0;
   }
```

Toujours par souci de conformité avec les conteneurs standards, et pour simplifier grandement le code client, nous définirons les types `iterator` et `const_iterator` dans notre classe, tout comme nous définirons aussi les accesseurs usuels menant vers les extrémités de la séquence contenue, soit `begin()` et `end()`.

Pour notre type, ces méthodes sont simples, pour ne pas dire évidentes. Elles ne coûtent rien en taille ou en temps d'exécution et facilitent énormément la programmation dans son ensemble. Par ces petits gestes, ces petits services à coût nul, nous parviendrons à définir une interface homogène et conforme aux attentes du code client.

En effet, exposer un itérateur et des extrémités de séquence permet d'appliquer sur notre conteneur toute la gamme des algorithmes standards et de convertir aisément d'un conteneur standard à notre conteneur et inversement (en partie grâce aux *constructeurs de séquences*, plus loin).

```
using iterator = value_type*;
using const_iterator = const value_type*;
iterator begin() noexcept {
   return elems;
}
const_iterator begin() const noexcept {
   return elems;
}
iterator end() noexcept {
   return begin() + size();
}
const_iterator end() const noexcept {
   return begin() + size();
}
```

Construction de copie et de conversion[47]

Tel que noté dans la section précédente, notre choix d'avoir un tableau par défaut dont le pointeur `elems` est nul nous forcera à tenir compte, en tout temps, de cette possibilité dans l'implémentation des méthodes de `Tableau`. Heureusement, `elems` est un attribut pleinement encapsulé et nous sommes en mesure de cacher les ramifications de ce choix d'implémentation à l'intérieur de la classe.

Le constructeur de copie est presque évident. Par encapsulation, `tab` est présumé dans un état correct. Subtilité : j'ai fait de la capacité initiale de `*this` la taille de `tab` plutôt que sa capacité, mais c'est là un choix politique, pas une obligation technique – l'inverse aurait été défendable. J'ai supposé ce comportement plus conforme aux attentes du code client.

```
Tableau(const Tableau &tab)
   : cap{tab.size()},
     nelems{tab.size()},
     elems{new value_type[tab.size()]}
{
   copy(tab.begin(), tab.end(), begin());
}


template <class U>
   Tableau(const Tableau<U> &tab)
      : cap{tab.size()},
        nelems{tab.size()},
        elems{new value_type[tab.size()]}
   {
      copy(tab.begin(), tab.end(), begin());
   }
```

Des exceptions peuvent être levées par l'invocation de `new[]` et par la copie des éléments, opération qui repose sur l'affectation d'un `T` à un autre `T`.

L'algorithme `copy()` itère à travers une séquence source et copie les éléments un à un vers une destination en utilisant l'opérateur d'affectation[48].

Le constructeur de conversion est plus subtil : il construit une instance de `Tableau` dont les éléments sont de type `T` à partir d'une instance de `Tableau` dont les éléments sont de type `U`.

Les types `T` et `U` sont nécessairement des distincts ici, car s'il s'agissait de deux tableaux de même type, le constructeur de copie aurait été appliqué[49].

À ce stade, le **contrat de `T`** implique l'exposition d'un constructeur par défaut (création dynamique d'un tableau de `T`) et d'un opérateur d'affectation (dont se sert `copy()`).

---

[47] J'utilise le terme constructeur de conversion pour un constructeur d'un type `T` prenant en paramètre un type `U` distinct de `T`, au sens large. Notez que les implémentations proposées ici sont perfectibles (elles sont fragiles lorsqu'une exception survient), mais nous les améliorerons en temps et lieu.

[48] Il se peut que certains compilateurs (dont celui de *Microsoft*) signalent un avertissement à l'utilisation de `copy()` car cet algorithme présume que la capacité de la destination suffit à recevoir la source. On peut comprendre les compilateurs craintifs mais ceci présume des pratiques de programmation malsaines. Ici, nous agissons en adultes responsables et cet avertissement est ridicule; lisez la documentation de votre outil pour supprimer ce facteur de ralentissement dont vous n'avez assurément pas besoin.

[49] Pensez à un tableau de `int` construit à partir d'un tableau de `short` :

```
Tableau<short> tabA;
// remplir tabA
Tableau<int> tabB = tabA; // conversion covariante
```

Constructeur de conversion et bonnes pratiques

Le **constructeur de conversion** est une des clés de la bonne programmation générique. C'est là que doivent converger toutes les opérations de conversion.

À l'intérieur d'un type générique tel que `Tableau`, nous ferons en sorte de faire converger vers cette méthode toute conversion de `Tableau<U>` vers un `Tableau<T>`. L'implémentation de ce constructeur nous indique que les types `T` et `U` doivent être apparentés, au sens où *un `U` doit pouvoir être affecté à un `T`*, pour que le constructeur soit applicable. Chercher à construire un `Tableau<float>` à partir d'un `Tableau<string>` provoquerait des erreurs (justifiées) à la compilation car l'affectation d'une `string` à un `float` n'est pas définie.

La conversion par construction prend son sens avec les classes génériques, résultant en une **covariance** [POOv01] de types lors de l'initialisation des conteneurs. Constatons d'ailleurs qu'une double généricité s'applique ici, en ce sens que ce constructeur est une méthode générique sur un type `U` dans une classe générique sur un type `T`.

Pour tout conteneur générique, incluant les décorateurs et les autres classes génériques qui ne contiennent qu'un seul élément, le constructeur de type apparenté vaut la peine d'être implémenté. Il allège énormément la logique d'ensemble de la classe.

De manière générale, le constructeur de copie doit être le seul véritable lieu de duplication d'un objet, que ce constructeur soit protégé dans un type polymorphique ou qu'il soit public dans un type valeur, alors que le constructeur de conversion doit être le seul véritable lieu de conversion pour un type générique.

Retenez que **le constructeur de conversion ne remplace pas le constructeur de copie**. Il ne s'applique que pour un type `U` différent du type `T`.

Construction de séquence

Permettre de construire un conteneur à partir d'une séquence arbitraire signifie offrir un service extrêmement utile. Ceci permet, par exemple, de construire un vecteur à partir d'un tableau ou une liste à partir d'un vecteur.

Plutôt que de chercher à mettre au point un large éventail de cas particuliers, nous définirons un seul constructeur, opérant à partir d'une séquence générique[50].

La fonction `std::distance()`, appliquée à une séquence standard, trouve (de manière optimale pour les itérateurs impliqués) le nombre d'éléments de cette séquence.

```
template <class It>
   Tableau(It debut, It fin) {
      cap = nelems = distance(debut, fin);
      if (capacity()) {
         elems = new value_type[capacity()];
         copy(debut, fin, begin());
      }
   }
```

[50] Un exemple d'utilisation d'un constructeur de séquence serait :

```
short tabA[10] = { 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 };
Tableau<int> tabB(tabA + 5, tabA + 10); // constructeur de séquence
```

Note : en 2017, Jesse Emond m'a fait remarquer que le constructeur de conversion pourrait s'exprimer sans perte de généralité, et sans perte de vitesse, par une délégation vers le constructeur de séquence, comme suit :

```
template <class U>
   Tableau(const Tableau<U> &tab) : Tableau{ tab.begin(), tab.end() } {
   }
```

Permutation d'états

L'opération `swap()`, par laquelle les états de deux entités peuvent être échangés, est une opération fondamentale, en particulier lors de l'implémentation d'opérations sur un conteneur. Dans la mesure où des choix d'implémentation adéquats ont été faits pour un conteneur donné, il est toujours possible d'y implémenter une méthode `swap()` qui s'exécutera en temps constant et de manière à ne pas lever d'exception.

L'algorithme standard `std::swap()` ne lèvera pas d'exceptions si l'affectation définie sur ses paramètres n'en lève pas[51]. Notre méthode applique cet algorithme sur des types primitifs, soit des `size_type` et des `value_type*`, et peut donc se qualifier *no-throw*.

```
void swap(Tableau &tab) noexcept {
   using std::swap;
   swap(nelems, tab.nelems);
   swap(cap, tab.cap);
   swap(elems, tab.elems);
}
```

La méthode `swap()` sert dans plusieurs manœuvres en programmation générique car elle est généralement optimale en temps et s'avère être l'une des rares opérations qui peut toujours être écrite de manière à ne lever aucune exception, sans égard aux types impliqués.

La plupart des opérations d'un type `T` quelconque ne peuvent être garanties pleinement libres d'exceptions du fait que le code générique ne sait à peu près rien des détails internes au type `T` qu'il manipule. Il se trouve que `std::swap()` n'est pas appliqué à des références sur des `T` ici mais bien à des références sur des `T*`. L'affectation entre deux `T` est risquée, faute de connaître de détail de `T`, mais l'affectation entre deux `T*` est inoffensif.

[51] Et encore : il est parfois possible de l'implémenter à partir d'une sémantique de mouvement, ce qui ne lève à peu près jamais d'exceptions.

Affectation et affectation covariante

Notre objectif ici sera de localiser tout effort de conversion dans les constructeurs de conversion, pour réduire la complexité du code sans entraîner de coûts inutiles. Nous appliquerons pour ce faire l'idiome d'affectation sécuritaire [POOv00] :

- affecter un `Tableau<T>` à un autre `Tableau<T>` s'implémente sans peine par la paire construction par copie/ `swap()` et n'implique pas de conversion; et
- affecter un `Tableau<U>` à un `Tableau<T>` (`T` différent de `U`) s'implémente de la même manière et localise la conversion de `U` à `T` dans la construction par copie.

```
Tableau& operator=(const Tableau &tab) {
   Tableau{tab}.swap(*this);
   return *this;
}
template <class U>
   Tableau& operator=(const Tableau<U> &tab) {
      Tableau{tab}.swap(*this);
      return *this;
   }
```

Nous ne pouvons pas offrir de garanties quant aux exceptions dans ces méthodes du fait que nous ne pouvons pas en offrir dans les constructeurs sur lesquels elles reposent.

Respectant les saines pratiques annoncées plus haut, nous faisons converger la copie vers le constructeur de copie. Ainsi, dans l'affectation, la véritable duplication de contenu est réalisée dans le constructeur de copie qui mène à une temporaire anonyme. De même, la conversion requise par l'opérateur de conversion par affectation est réalisée à l'intérieur du constructeur de conversion. Procédant ainsi, nous évitons la duplication de fonctionnalité.

Notez au passage que nous venons, en définissant l'affectation pour `Tableau`, de compléter la Sainte-Trinité. La copie, l'affectation et la destruction sont toutes implémentées convenablement, ce qui signifie (informellement) que le type `Tableau<T>` est un type valeur, peu importe la nature de `T`.

Accès à un élément

L'accès à un élément en lecture seule (version `const`) ou en lecture/écriture (non `const`) s'écrit de la même manière, aux garanties de constance près. Nous avons choisi ici de valider les bornes à chaque utilisation des crochets `[]`; la classe `vector` ne le fait pas pour `operator[]`, mais offre une méthode `at()` qui, elle, le fait – valider les bornes, après tout, entraîne une perte de performance.

```
class HorsBornes {};
value_type& operator[](size_type n) {
   if (n >= size()) throw HorsBornes{};
   return elems[n];
}
value_type operator[](size_type n) const {
   if (n >= size()) throw HorsBornes{};
   return elems[n];
}
```

Remarquez que nous ne testons ici que la borne supérieure du tableau. Ceci tient à notre choix du type `size_t`, un type non-signé, en tant que `size_type`.

Comparaison

La comparaison de deux instances de `Tableau<T>` s'implémente bien à l'aide d'une répétitive et de l'opérateur `!=` applicable à un `T`.

Voir l'exercice `EX01`, en fin de section, pour un raffinement recommandable de cette méthode.

L'existence d'un constructeur implicite de conversion permet d'éviter d'écrire une déclinaison pour types apparentés des opérateurs `==` et `!=`. Si un besoin (de performance) se faisait sentir, implémenter des versions spécialisées de ces opérateurs serait simple.

```
bool operator==(const Tableau &tab) const {
   if (size() != tab.size())
      return false;
   for (auto iA = begin(), iB = tab.begin();
        iA != end (); ++iA, ++iB)
      if (*iA != *iB)
         return false;
   return true;
}
bool operator!=(const Tableau &tab) const {
   return !(*this == tab);
}
```

Ajout d'éléments

Deux méthodes publiques permettront d'ajouter un élément à la fin d'un `Tableau`, soit :

- l'opérateur `+=` pour accommoder les algorithmes de cumul sous une forme plus mathématique; et
- la méthode `push_back()` qui harmonisera notre conteneur avec les conteneurs standards.

La méthode privée `croitre()` sera invoquée lorsque la capacité d'un `Tableau` sera sur le point d'être dépassée. Son coût (élevé) sera amorti si elle n'est pas sollicitée souvent.

Subtilité : notre implémentation du constructeur par défaut fait qu'il soit possible que `capacity()` retourne `0`. Nous devons tenir compte de ce cas pour calculer correctement la nouvelle capacité lors d'une invocation de `croitre()`.

Réflexion 02.1 : notez l'ordre des opérations dans `croitre()`. Est-ce qu'il serait sage de déplacer l'affectation de `n` à `cap` avant `new[]` ou avant `copy()`? Réponse dans ***Réflexion 02.1 : assurer le respect des invariants***.

```
    Tableau& operator+=(const value_type &val) {
       push_back(val);
       return *this;
    }
    void push_back(const value_type &val) {
       if (full()) croitre();
       elems_[nelems] = val;
       ++nelems;
    }
private:
    void croitre() { // coûteux
       static const size_type CAP_BASE = 128;
       const auto n = capacity()?
          static_cast<size_type>(capacity()*1.5) :
          CAP_BASE;
       auto p = new value_type[n];
       try {
          copy(begin(), end(),p);
       } catch(...) {
          delete[] p;
          throw;
       }
       delete[] elems;
       cap = n;
       elems = p;
    }
```

Sémantique de mouvement

Une optimisation importante possible sous C++ 11 est celle qui permet de tenir compte des objets jetables, aussi nommés références sur des rvalue. En gros, lorsqu'une opération (souvent un constructeur ou un opérateur d'affectation) est sollicitée avec pour opérande de droite (le r de rvalue) un objet sur le point d'être détruit, il est possible d'en profiter pour remplacer l'habituelle copie des états de l'objet source dans l'objet de destination par un déplacement de contenu de l'objet source vers l'objet de destination – ceci dans la mesure où l'objet source demeure respectueux de ses invariants *a posteriori*.

Les implémentations proposées à droite sont simples et efficaces.

Il aurait aussi été possible d'implémenter le constructeur de copie avec le constructeur par défaut et la méthode swap() : le constructeur de mouvement aurait construit l'objet destination comme s'il était vide, puis aurait permuté ses états avec ceux de l'objet d'origine. Les deux auraient donc été dans un état convenable par la suite. Il est toutefois d'usage d'implémenter le mouvement pour fins d'optimisation. Pour cette raison, le code à droite, sans compromis, est préférable.

```
   public:
      Tableau(Tableau &&tab)
         : elems{ std::move(tab.elems) },
           nelems{std::move(tab.nelems) },
           cap{std::move(tab.cap) }
      {
         tab.elems = {};
         tab.nelems = {};
         tab.capacite = {}
      }
      Tableau& operator=(Tableau &&tab) {
         delete [] elems;
         elems = tab.elems;
         nelems = tab.nelems;
         cap = tab.cap;
         tab.elems = {};
         tab.nelems = {};
         tab.cap = {};
         return *this;
      }
}; // fin de la classe
```

Opérations connexes

Nous ajouterons au moins deux opérations externes à l'interface de `Tableau<T>` :

- un opérateur de projection sur un flux, qui n'aura de sens que si un `T` lui-même peut être projeté sur un flux; et
- une fonction globale `std::swap()` qui déléguera le travail aux instance de `Tableau` impliquées et fera en sorte d'appliquer le meilleur algorithme possible pour ce type.

Ceci complétera le portrait (bien que l'on puisse souhaiter encore ajouter une masse d'opérations supplémentaires).

```
#include <iosfwd>
#include <algorithm>
using namespace std;
template <class T>
   ostream& operator<<
      (ostream &os, const Tableau<T> &tab) {
      copy(begin(tab), end(tab),
           ostream_iterator<T>{os, " "});
      return os;
   }
namespace std {
   template <class T>
      void swap(Tableau<T> &a, Tableau<T> &b) {
         a.swap(b);
      }
}
```

Deux remarques importantes sur le plan technique doivent être faites à ce stade.

Fonctions génériques globales et `ODR`

Nous le savons, la définition de fonctions globales dans un fichier d'en-tête est une pratique peu recommandable en raison d'`ODR` (le *One Definition Rule*), discuté dans [POOv00]. En effet, si une définition de fonction globale est placée dans un fichier d'en-tête, alors tous les fichiers sources qui incluront cet en-tête auront leur propre définition de la fonction, en violation d'`ODR`.

Les fonction globales *génériques* (comme l'opérateur `<<` et la spécialisation de la fonction `swap()` ci-dessus) échappent à cette contrainte car elles ne seront générées qu'au moment de leur utilisation. Le compilateur doit voir les définitions pendant qu'il examine le code client pour être en mesure de générer le code attendu; conséquemment, les classes génériques et les fonctions globales génériques sont définies dans des en-têtes, visibles au code client.

Spécialiser un `template` défini dans `std`

Il est interdit, en C++, d'ajouter des éléments dans l'espace nommé `std`. Cet espace est le terrain de jeu privé des gens qui entretiennent le standard du langage; si le commun des mortels pouvait y ajouter des éléments, cet espace serait pollué et le problème qu'il vise à résoudre reviendrait en force.

Il est permis, par contre, de spécialiser des classes et des fonctions génériques définies dans `std` pour couvrir avec plus de précision des types qui nous intéressent. Ici, nous indiquons que, bien qu'il existe un `std::swap()` général pour deux instances de `T`, notre version est préférable pour deux instances de `Tableau<T>`.

Notre `swap()` demeure générique puisque `T` y est un paramètre, et peut donc être défini dans un fichier d'en-tête.

Si nous avions une spécialisation pointue pour `Tableau<int>`, il s'agirait alors d'une fonction globale normale et il faudrait placer sa définition dans un fichier source.

Contrôler la génération du code

Certains craignent que les *templates* ne consomment inutilement de l'espace dans un programme. Après tout, le risque semble clair que créer des fonctions et des classes aussi aisément soit une pratique qui mène à un gaspillage de ressources.

Pourtant, en pratique, c'est souvent l'inverse qui se produit. Par exemple, écrire ceci :

```
class C { ... };
```

…peut être moins efficace qu'écrire cela :

```
template<class T> class C { ... };
```

…car, pour la version générique, seul le code utilisé en pratique sera véritablement généré, alors que pour la version non-générique, la classe entière sera générée à la compilation.

Au besoin, il est possible de forcer la génération de certaines classes et de certaines méthodes a priori génériques. Par exemple :

```
template class C<int>; // force la génération de C<int>
template void C<int>::m(); // force la génération de la méthode m() de C<int>
```

Il est possible de réduire la taille du code objet généré en déplaçant le code redondant d'une classe à l'autre dans une classe parent commune. On parle alors en quelque sort d'hériter pour réduire les coûts.

Examinez par exemple ce qui suit, tiré de diapositives électroniques de ***Scott Meyers***. À gauche, on aura une classe entière distincte, avec toutes ses méthodes, pour chaque combinaison d'un type `T` et d'une valeur `N`, même si les services seront probablement identiques dans chaque cas, alors qu'à droite, le type `BufferBase` est indépendant de la valeur de `N`. Si les services de `Buffer<T,N>` sont bien conçus, à droite, cette classe sera essentiellement gratuite.

Design suspect

```
template<class T, std::size_t N>
   class Buffer {
      T buffer[N];
   public:
      // ...
   };
```

Meilleur design

```
template<class T>
   class BufferBase {
      // ...
   };
template<class T, std::size_t N>
   class Buffer : public BufferBase<T> {
      T buffer[N];
   public:
      // ...
   };
```

Exercices – Série 00

EX00 – Implémentez la méthode `pleine()` qui retourne `true` si et seulement si le conteneur est à pleine capacité. Assurez-vous d'avoir la meilleure implémentation possible. Pouvez-vous améliorer le reste de votre conteneur en utilisant cette méthode aux endroits opportuns?

EX01 – Pouvez-vous implémenter (au moins en partie) l'opérateur de comparaison `==` à l'aide d'un algorithme standard? Si oui, faites-le.

EX02 – Implémentez un opérateur de comparaison `==` de types apparentés. Quel est l'avantage de mettre en place cette méthode?

EX03 – Serait-il avantageux d'implémenter aussi un opérateur de comparaison `!=` de types apparentés? Expliquez votre position.

EX04 – Modifiez le design des constructeurs pour que `capacite_` ne soit jamais nulle. Modifiez la classe de manière à ce que toutes ses méthodes en tiennent compte. À votre avis, cette nouvelle implémentation est-elle meilleure que la précédente?

EX05 – Plusieurs conteneurs standards offrent une méthode `resize()` pour redimensionner le conteneur. Implémentez-la et documentez clairement son comportement, à la fois lorsque cette méthode accroît la taille du conteneur et lorsqu'elle décroit cette taille.

EX06 – Plusieurs conteneurs standards sont tels que `resize()` ne permet pas de rapetisser un conteneur. Pouvez-vous établir les avantages et les inconvénients de cette approche?

EX07 – La méthode `clear()` des conteneurs standards permet de vider un conteneur de ses valeurs. Donnez au moins deux approches différentes pour implémenter cette méthode, puis implémentez-les et comparez les résultats obtenus.

EX08 – Reprenez la classe `Cumulateur` de la section ***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.

Programmation générique et *templates*. Présentement, un `Cumulateur<int>` ne prend que des `int` en paramètre à la fonction `ajouter()`, et un `Cumulateur<string>` ne prend que des `string`. Peut-on faire en sorte qu'un `Cumulateur<`*`T`*`>` expose une méthode `ajouter(`*`U`*`)` pour tout type *`U`* en fonction duquel il serait raisonnable de cumuler (p. ex. : `Cumulateur<int>` pourrait exposer `ajouter(short)`)?

Constructeur de conversion et héritage[52]

Les constructeurs de conversion peuvent parfois jouer des tours.

Imaginons une classe `X` ayant la forme proposée à droite. Elle expose entre autres un constructeur de copie (`X::X(const X&)`) et un constructeur de type apparenté (`X::X(const T&)` pour un `T` quelconque). En soi, ce code est tout à fait légal.

```
class X {
   // ...
public:
   X(const X &x) {
      // ...
   }
   template <class T>
      X(const T &x) {
         // ...
      }
   // ...
};
```

En pratique, la plupart des classes pour lesquelles on pourrait envisager du code semblable à celui proposé à droite sont des classes terminales. Cela dit, il est légal de produire une classe dérivée de cette classe `X`.

Un tel dérivé pourrait être la classe `Y` proposée ci-dessous.

```
class Y : public X {
   // ...
public:
   Y(const Y &y) : X{y} {
   }
   // ...
};
```

La construction par copie de `Y` passe ici par la construction de sa partie `X`. Mais il y a une subtilité ici : *ce n'est pas le constructeur de copie du parent qui sera invoqué mais bien le constructeur de conversion*.

En effet, le compilateur doit choisir entre un appel de `X::X(const X&)` et un appel de `X::X(const T&)` où `T` est le type `Y`. Il se trouve en fait que `X::X(const Y&)` est considéré plus précis, dans ce cas-ci, que `X::X(const X&)` puisque le paramètre passé au constructeur est précisément de type `const Y&`.

Conséquemment, et contrairement peut-être aux attentes de la classe `Y`, le constructeur de copie de `Y` ne reposera pas ici sur la construction par copie de son parent `X`. Il faut en être conscient. Pour contourner le problème, dans le cas où le code de la classe enfant compte sur l'invocation du constructeur par copie du parent, il faut que le code de la classe enfant spécifie explicitement sa préférence pour `X::X(const X&)`.

Le code à droite donne un exemple de code efficace permettant d'expliciter cette intention.

```
Y(const Y &y)
   : X{static_cast<const X&>(y)}
{
}
```

[52] Cette section a été inspirée d'un problème soumis par le brillant ***Patrick Boutot***.

Génération de nombres pseudoaléatoires

Ce qui suit survole les principaux mécanismes permettant de générer des nombres pseudoaélatoires avec C++ depuis C++ 11, avec un bref retour sur les mécanismes (très désuets) du langage C, qui prévalaient même (tristement) avec C++ 03.

Comment fonctionne la génération de nombres pseudoaléatoires en C++ contemporain (depuis C++ 11, donc) et en quoi est-ce une amélioration sur les pratiques du passé? C'est ce que ce petit texte vise à illustrer.

Exemple concret – un dé équitable à six faces

Pour illustrer notre propos, prenons un exemple simple, soit celui d'un dé à six faces qui se voudrait équitable, donc avec la même probabilité d'obtenir chacune des six valeurs possibles. Un programme simple serait :

```
#include <random>
#include <iostream>
#include <string>
#include <numeric>
#include <algorithm>
#include <iomanip>
#include <locale>
using namespace std;
string make_line(double proportion) {
   enum { LG_LIGNE = 50 };
   return string( static_cast<string::size_type>(LG_LIGNE * proportion), '*' );
}
int main() {
   random_device rd;
   mt19937 prng{ rd() };
   enum : short { NFACES = 6 };
   uniform_int_distribution<short> d6{1,NFACES};
   enum { N = 10000 };
   short nlancers[NFACES * 2] = { 0 }; // 2..12, rien à l'entrée 0
   for (short i = 0; i < N; ++i)
      nlancers[d6(prng) + d6(prng) - 1]++;
   // on ne tient pas compte de nlancers[0]
   const auto maxval = accumulate(next(begin(nlancers)), end(nlancers), short{},
                          [](short x, short y) { return max(x, y); });
   cout << "Distribution, " << N << " lancers de deux des a " << NFACES << " faces\n\n";
   cout.imbue(locale{ "" });
   // on ne tient pas compte de nlancers[0]
   for (short i = 1; i < NFACES * 2; ++i) {
      const auto nb_lancers = static_cast<double>(nlancers[i]);
      const auto proportion_ligne = nb_lancers / maxval;
      const auto proportion_total = nb_lancers / N * 100.0;
      cout << setw(2) << i + 1 << " : " << make_line(proportion_ligne)
```

```
            << " (" << proportion_total << "%)" << endl;
    }
}
```

Quelques explications s'imposent, évidemment :

- plusieurs en-têtes de ce programme ne servent qu'à des fins auxiliaires. Ceux qui sont essentiels au propos sont `<random>` pour la génération de nombres pseudoaléatoires, et (dans une moindre mesure) `<iostream>` pour l'affichage des résultats;
- de même, une partie importante du programme sert à produire un affichage qui soit joli;
- nous utilisons un `random_device` (nommé `rd` dans le programme) pour initialiser le générateur de nombres pseudoaléatoires. Cette source d'entropie est dispendieuse à utiliser, mais permet de démarrer le processus de génération sur une base « imprévisible »;
- le moteur utilisé pour générer des nombres pseudoaléatoires dans ce programme est bien connu dans le mileu, malgré son nom étrange (type `mt19937`). Il s'agit d'un *Mersenne Twister*, que nous avons nommé `prng` (*Pseudo-Random Number Generator*);
- le dé à six faces, représenté par la variable `d6`, est une distribution uniforme sur une plage entière située inclusivement entre `1` et `6` des valeurs obtenues du moteur `prng`.

Une exécution possible de ce programme serait la suivante.

```
Distribution, 10000 lancers de deux des a 6 faces

 2 : ******** (2,79%)
 3 : ********** (5,56%)
 4 : ********** (8,22%)
 5 : ********** (10,93%)
 6 : ********** (14,1%)
 7 : ********** (17,01%)
 8 : ********** (13,56%)
 9 : ********** (10,87%)
10 : ********** (8,79%)
11 : ********** (5,66%)
12 : ******* (2,51%)
```

Visiblement, avec une distribution uniforme et un grand nombre de paires de lancers de dés, nous obtenons essentiellement une courbe normale. C'est une bonne nouvelle!

Grands principes

Ce qui suit n'est pas une explication formelle de la théorie de la génération de nombres pseudoaléatoires; ce n'est qu'un survol de quelques grands principes, dans une optique d'offrir une compréhension opératoire des mécanismes concrets offerts par le langage. Pour en savoir plus, voir [TAOCP] par ***Donald Knuth***, volume 2, un livre massif qui porte exclusivement sur ce sujet. Sans blagues.

Il est impossible de générer de vrais nombres aléatoires à partir d'un algorithme au sens classique du terme, puisque ce dernier est déterministe *a priori*. On tire habituellement de l'entropie à travers une collecte d'information sur le monde physique (vibrations d'un disque rigide, variations de température, fréquence de pressions de touches sur le clavier, etc.) mais cette collecte d'entropie doit être réalisée par le système d'exploitation, et est donc coûteuse à obtenir pour un programme.

Pour cette raison, la génération de nombres pseudoaléatoire comprend typiquement les éléments suivants.

Semis

Un semis (en anglais : *Seed*) est requis pour initier le processus de génération. Ce semis vient habituellement d'une « source d'entropie » ou du moins d'une source changeante. Le nombre de secondes depuis un moment choisi dans le passé, par exemple, a beaucoup servi en ce sens (fonction `std::time()` du langage C), et fait un travail raisonnable si on l'appelle moins d'une fois par seconde. Aujourd'hui, piger dans l'entropie accumulée par le système d'exploitation est la norme.

Le semis est utilisé une seule fois, au démarrage du processus de génération de nombres pseudoaléatoires. Utiliser plusieurs semis n'entraîne pas une meilleure génération de nombres pseudoaléatoires (en fait, c'est plutôt le contraire qui se produit).

Notez qu'en période de tests, pour que les résultats semblent aléatoires mais demeurent prévisibles, il arrive fréquemment que l'on ait recours à un semis fixe (la valeur zéro, par exemple). En procédant ainsi, nous obtiendrons toujours les mêmes valeurs « pseudoaléatoires », dans le même ordre.

Avec C, la fonction `std::srand()` permet d'initialiser le semis de l'algorithme de génération qu'est `std::rand()`.

Algorithme

Un algorithme est ensuite requis pour (a) générer un nombre pseudoaléatoire et (b) faire évoluer un état tenu à jour (et initialisé à partir du semis). L'algorithme étant déterministe, il aura une période, suite à laquelle la séquence générée se répétera.

Avec C, la fonction `std::rand()` sert d'algorithme de génération de nombres pseudoaléatoires.

Le problème de fond de cette approche, telle qu'implémentée en langage C du moins, est que l'état interne est un état global susceptible d'être accédé concurremment en écriture par plusieurs *threads*. Elle n'est donc pas utilisable en pratique dans un système multiprogrammé.

Avec C, de plus, la période de l'algorithme est `RAND_MAX`, typiquement `32768`, ce qui est un peu court pour des applications sérieuses.

Enfin, prendre des nombres pseudoaléatoires et les répartir selon une distribution est une tâche plus subtile qu'il n'y paraît. Par exemple, avec `std::rand()`, qui retourne des nombres entre `0` et `RAND_MAX`, une approche naïve pour tirer un nombre entre deux bornes serait :

```
// précondition: minval <= maxval
int lancer_de(int minval, int maxval) {
   return rand() % (maxval-minval+1) + minval;
}
```

…mais cette approche entraîne un biais, du fait qu'en général la plage de taille `(maxval-minval+1)` ne divisera pas exactement `RAND_MAX`, donc certaines valeurs apparaîtront plus souvent que d'autres en pratique. Ceci est une conséquence inacceptable pour bien des systèmes, particulièrement ceux ayant trait à la sécurité informatique. Cela explique que nos pratiques contemporaines diffèrent de celles du passé.

Avec C++ (pratiques contemporaines)

La mécanique de génération de nombres pseudoaléatoires avec C++ combine trois grands éléments :

- une source d'entropie, pour initier la génération de nombres pseudoaléatoires sur une base quelque peu imprévisible;
- un moteur pour générer des nombres pseudoaléatoires selon un certain algorithme; et
- une distribution, permettant de répartir les valeurs générées par le moteur en question selon une loi de distribution choisie.

L'exemple du dé à six faces, plus haut, montre une combinaison possible d'une source d'entropie (un `random_device`), d'un moteur (le `mt19937`) et d'une distribution (un `uniform_int_distribution<int>`). La source d'entropie est un foncteur utilisé pour générer le semis du moteur, et le moteur est utilisé par la distribution pour générer des valeurs qui seront réparties dans le respect des règles qu'elle représente.

```
// ...
random_device rd; // source d'entropie
mt19937 prng { rd() }; // moteur dont le semis est rd()
uniform_int_distribution<int> d6{1, 6}; // distribution uniforme d'entiers dans [1..6]
auto lancer = d6(prng); // lancer de dé avec probabilité égale de tirer 1, 2, 3, 4, 5 et 6
// ...
```

Introduction aux pointeurs intelligents

Un sujet à propos duquel les programmeuses et les programmeurs ont l'épiderme sensible est celui de la gestion des ressources. Un programme gérant mal les ressources qu'il s'approprie entraînera une dégradation des ressources disponibles sur l'ordinateur où il s'exécute, ce qui explique l'importance d'atteindre une saine gestion de toutes les ressources dans les systèmes informatiques, du plus simple au plus complexe.

La question de la gestion des ressources dépasse de beaucoup celle de la gestion de la mémoire. Les moteurs de collecte automatique d'ordures ne résolvent qu'une partie de cet important problème. Déterminer la meilleure stratégie de gestion des ressources, de manière générale, est une question de nature presque religieuse.

Gestion de la mémoire

Dans un langage sans collecte automatique d'ordures, l'allocation dynamique de mémoire rend les développeurs responsables de s'assurer que la mémoire allouée soit éventuellement libérée.

Dans pratiquement tous les langages, les développeurs sont responsables de s'assurer que leurs objets soient convenablement finalisés. Cette responsabilité, plus importante encore que celle de la gestion de la mémoire, est nettement plus facile à gérer en C++ qu'en C# ou en Java.

Par exemple, à droite, la mémoire associée au `X` pointé par p sera perdue puisque `f()` ne fait pas `delete p`. Pire encore : le destructeur de `*p` ne sera pas invoqué, donc la finalisation de cet objet n'aura pas lieu – son destructeur ne sera pas appelé.

```
// C++
void f() {
   X x;
   X *p = new X;
}
```

Notez que le pointeur `p` sera détruit, étant une variable locale, tout comme le sera l'objet `x`. C'est le *pointé* de `p`, qui a été créé manuellement et se trouve sous le contrôle explicite du programme, qui ne sera ni finalisé, ni réclamé.

Avec une collecte automatique d'ordures, une mécanique interne compte les références à toute instance. Lorsqu'un objet n'est plus référencé dans un processus donné, cet objet est marqué comme prêt à être éliminé de la mémoire. La collecte d'ordures passe en arrière-plan et élimine de la mémoire les débris lorsque cela lui semblera être devenu opportun.

```
// Java
public void f() {
   X px = new X();
}
```

Les *aficionados* de la collecte automatique d'ordures tendent à associer gestion des ressources et gestion de la mémoire vive, et estiment souvent que la collecte d'ordures résout, à leurs yeux, le problème de la gestion des ressources.

Cela s'explique, au moins en partie. Pour plusieurs langages `OO`, la collecte d'ordures fait intrinsèquement partie d'un modèle objet vraiment utilisable, du fait que ces langages sont souvent d'importants consommateurs d'instanciation dynamique. Dans ces langages, créer dynamiquement (parfois implicitement) de petits objets temporaires est perçu comme une technique de programmation, un idiome. Dû à la popularité de ces langages, la recherche quant aux stratégies pour ramasser les miettes dans un programme est un domaine très vivant.

Finalisation

Il se trouve que cette conception selon laquelle collecter la mémoire non référencée est une saine stratégie de gestion de ressources est fausse, sauf pour les programmes les plus banals : un fichier est une ressource, mais si un objet ouvre un fichier pour réaliser ses tâches, la collecte d'ordures n'en saura rien.

```
public void g() {
   Y y = null;
   try {
      y = new Y();
      f();
   } catch (Exception e) {
      // ...
   } finally {
      if (y != null) {
         y.close();
      }
   }
}
```

La finalisation est une question complexe. Le code à droite montre un exemple de finalisation en Java (`1.6` et moins). Comme c'est souvent le cas dans les langages munis de moteurs de collecte d'ordures, le moment de la collecte des objets n'est pas pleinement déterministe, ce qui empêche l'objet d'assurer sa finalisation convenablement (faute de contexte).

Le code client devient alors responsable d'une partie de ce qui devrait faire partie de l'encapsulation.

La finalisation, par laquelle un objet se voit offrir une dernière opportunité de libérer les ressources sous a gouverne (et qui est analogue à la destruction en C++), est un problème différent de celui la collecte de la mémoire vive inutilisée.

Quiconque a mis en place, à l'aide de l'interface `IDispose`, une stratégie homogène de finalisation dans un langage pris en charge comme le sont les langages `.NET` sait que ce n'est pas un problème banal en l'absence de destructeurs déterministes.

Ne pas libérer la mémoire associée à un objet signifie que cette mémoire sera perdue, et ce pour la durée de l'exécution du processus où l'allocation fut faite. Ne pas finaliser un objet peut impliquer des pertes de ressources *beaucoup* plus sérieuses.

```
void g() {
   Y y;
   f();
}
```

Avec C++, assurer la finalisation déterministe d'un objet est aussi simple que lui conférer une portée. Dans le code à droite, l'instance `y` de la classe `Y` sera détruite – finalisée – dès que la fin de sa portée sera atteinte, ou si le bloc déterminant sa portée se conclut d'une manière atypique (`return`, `throw`, `exit()`, etc.).

Pour établir des stratégies de finalisation, il est important de déterminer une responsabilité claire face aux ressources manipulées par les objets. Une sémantique de copie aide beaucoup en ce sens : les pointeurs en C++, tout comme les références en Java et dans les langages `.NET`, sont des indirections gérées par le langage, alors que les objets sont contrôlés par programmation, ce qui leur donne une opportunité de réagir aux copies et d'assurer une forme de contrôle sur les ressources placées sous leur responsabilité.

Cette section se concentrera sur C++, puisque la gestion des ressources en Java et dans les langages `.NET` est typiquement laissée au code client, et donc de peu d'intérêt dans un discours sur l'approche `OO`.

L'idiome `RAII`, couvert dans les volumes précédents de cette série et très fréquemment rencontré en C++, est un exemple de gestion des ressources reposant sur la responsabilité dont s'affublent eux-mêmes les objets. Cet idiome repose pleinement sur l'appel déterministe des mécanismes de finalisation tels que les destructeurs.

La possibilité de responsabiliser, par `RAII`, les objets dans la gestion de leurs ressources est l'une des raisons pour lesquelles l'invocation explicite de méthodes de nettoyage[53] sont fréquemment rencontrées et utilisées en Java et dans les langages `.NET` mais ne sont pratiquement jamais invoquées explicitement par le code client en C++.

C'est aussi la raison de l'absence de blocs `finally` dans la gestion d'exceptions en C++ : appliquer `RAII` est nettement préférable.

Cela dit, `RAII` dans son acception simple ne résout pas tous les problèmes, loin de là. Parfois, les ressources à gérer sont partagées par plusieurs objets, ce qui implique une forme de comptabilité du nombre d'objets l'utilisant. Parfois, il est préférable de dupliquer une ressource lorsque plus d'un objet en cherche la responsabilité. Il nous faut des solutions plus générales.

Pointeurs intelligents

Les pointeurs intelligents[54] impliquent habituellement généricité et surcharge d'opérateurs. Un pointeur intelligent est un objet qui encapsule un pointeur et offre les services attendus d'une telle entité (opérateur `->` pour invoquer les méthodes et opérateur `*` pour le déréférencer) mais prend en charge une stratégie de responsabilité quant au pointeur en question.

Normalement, si `p` est un pointeur intelligent vers un `T`, alors :

- le résultat de l'opération `*p` sera de type `T`;
- utiliser `p->` équivaudra à accéder à un membre d'un `T*`;
- dupliquer `p` appliquera la stratégie de duplication de ressources associée à `p`; et
- détruire `p` appliquera la stratégie de libération de ressources associée à `p`.

Nous ne pourrons pas explorer en détail la théorie des pointeurs intelligents dans ce volume, le sujet étant *très* costaud; nous y reviendrons toutefois dans des volumes subséquents. Comme il se doit, un pointeur intelligent réussi est ardu à programmer mais facile à utiliser.

[53] Des méthodes `close()`, par exemple, comme l'exigent les blocs *try-with* depuis Java `7`.

[54] Certains parleront d'*autopointeurs*, en relation avec `std::auto_ptr` de C++ `98`, mais le sujet des pointeurs intelligents est *bien* plus vaste.

Pour fins de généralité, les pointeurs intelligents sont généralement implantés de manière générique. À titre d'exemple, voici une implantation possible d'un type de pointeur intelligent simple, **insuffisante pour un rôle commercial**, pour laquelle le pointeur intelligent libérera la donnée pointée lors de sa propre destruction :

```
#include "Incopiable.h"
template <class T>
   class Autopointeur : Incopiable {
      T *ptr;
   public:
      Autopointeur(T *p = {}) : ptr{p} {
      }
      ~Autopointeur() {
         delete ptr;
      }
      T &operator*() noexcept {
         return *ptr;
      }
      T operator*() const {
         return *ptr;
      }
      T *operator->() noexcept {
         return ptr;
      }
      const T *operator->() const noexcept {
         return ptr;
      }
};
```

Ceci donne un aperçu du type de programmation qu'implique la rédaction d'un tel type, même de manière naïve. Il y aurait lieu d'enrichir le tout en ajoutant, par exemple, les opérations requises pour implémenter la sémantique de mouvement.

Un exemple valide d'utilisation du pointeur intelligent simpliste proposé plus haut suit, appliqué à une variable d'un type primitif et à une instance d'une classe :

```
class Entier { // exemple
   int val {};
public:
   Entier() = default;
   Entier(int val) noexcept : val{val} {
   }
   int valeur() const noexcept {
      return val;
   }
   bool operator==(const Entier &x) const noexcept {
      return valeur() == x.valeur();
   }
   bool operator!=(const Entier &x) const noexcept {
      return !(*this==x);
   }
   bool operator<(const Entier &x) const noexcept {
      return valeur() < x.valeur();
   }
   // etc.
};
#include "autopointeur.h"
#include <iostream>
int main() {
   using namespace std;
   Autopointeur<int> ai0;
   Autopointeur<int> ai1{new int{8}};
   Autopointeur<Entier> ax0{new Entier{7}};
   cout << *ai1 << ax0->valeur() << endl;
  // La fin du bloc appelle les destructeurs de ai0, ai1 et ax0. Les deux
  // derniers feront un delete de leur contenu, respectivement int et Entier
}
```

Rôle des pointeurs intelligents

Un pointeur intelligent implémente typiquement une sémantique de responsabilité face au pointé. L'`Autopointeur` ci-dessus définit (naïvement) un type dont chaque instance est seule responsable de son pointé, et ne peut être copiée. Étant seule responsable du pointé, elle finalise le pointé lorsqu'elle est elle-même finalisée.

Nous examinerons très brièvement les principaux pointeurs intelligents du standard de C++ au cours des prochaines pages. Une couverture plus riche de l'implémentation des pointeurs intelligents sera donnée dans des documents ultérieurs de cette série. Ces types de pointeurs intelligents sont tous définis dans `<memory>`.

Sémantique de propriété exclusive : le type (obsolète) `std::auto_ptr`

La bibliothèque standard de C++ 98 proposait un pointeur intelligent, nommé `auto_ptr`, représentant une politique de propriété exclusive sur le pointé. Bien que ce type soit aujourd'hui obsolète (préférez `unique_ptr`), il est intéressant d'y jeter un bref coup d'œil.

Ce type ressemble à celui (simplifié) proposé plus haut, mais est plus sophistiqué et couvre beaucoup plus de cas particuliers que ne le fait notre petit démonstrateur académique.

Prenons par exemple l'extrait de code proposé à droite. Si une exception se produit entre la ligne où se trouve l'opération `X *q = new X{4};` et l'accolade fermante de la fonction `f()`, alors pointé par `q` ne sera pas détruit correctement[55], mais l'objet pointé par le pointeur intelligent `p`, lui, le sera.

On appréciera l'utilisation d'un pointeur intelligent comme la classe `std::auto_ptr` dans une telle situation. Confier à un objet la responsabilité d'un pointeur brut permet d'ajouter une dose d'intelligence à des opérations primitives comme la copie, la construction et la destruction, ce trio d'opérations auquel nous référons souvent sous le vocable de Sainte-Trinité.

La classe `std::auto_ptr` ne compte pas les références vers un pointeur donné, et ne peut donc pas servir à l'autodestruction de pointeurs référencés de plusieurs endroits. Au contraire, un `std::auto_ptr` se considère comme l'*unique propriétaire* de ce vers quoi il pointe.

```
class X { // classe très simple
   int val;
public:
   X(int val) noexcept : val{val} {
   }
   X& operator++() noexcept {
      ++val;
      return *this;
   }
   X operator++(int) noexcept {
      X temp{*this};
      operator++();
      return temp;
   }
};
#include<memory>
using std::auto_ptr;
void f() {
   auto_ptr<X> p{new X{3}};
   X* q = new X{4};
   // auto_ptr s'utilise comme un pointeur
   ++(*p);
   ++(*q);
   // ... peu importe
   delete q; // destruction explicite de *q
} // destruction implicite de *p
```

En ce sens, l'affectation d'un `std::auto_ptr` à un autre réalise un transfert de responsabilité : l'opérande de droite perd son pointeur, qui est transféré vers l'opérande de gauche… le pointé de l'opérande de droite devient nul!

Ainsi, le comportement de cette classe face à l'affectation diffère de celle de la plupart des classes. L'opérande de droite de l'affectation n'y est pas constant, et l'affectation est plus un déplacement de contenu qu'une copie de contenu.

Le principal facteur ayant motivé l'intégration d'`auto_ptr` à la bibliothèque standard de C++ est la sécurité des programmes lorsque survient une situation exceptionnelle.

[55] …à moins bien sûr d'avoir invoqué explicitement `delete q;` entre-temps!

En effet, imaginons la classe `X` proposée à droite. Présumant que `y_` et `z_` soient initialisés dans l'ordre, que se produira-t-il si une exception survient à l'initialisation de `z_`? Cette question trouve un analogue dans la question de réflexion `02.3` (voir ***Réflexion 02.3 : aléas des pointeurs bruts***).

```
#include "Y.h"
#include "Z.h"
class X {
   Y *y;
   Z *z;
public:
   X() : y{new Y}, Z{new Z} {
   }
   ~X() {
      delete z;
      delete y;
   }
};
```

Une chose est certaine : puisque le constructeur de `X` n'a pas complété son travail, le destructeur de `X` ne sera pas invoqué (l'instance de `X` n'étant pas encore officiellement construite).

Le code de `X` proposé à droite n'est donc pas sécuritaire face à la levée d'exceptions. Cette situation est exacerbée si nous projetons le problème à un contexte générique où (ou presque) rien n'est connu *a priori* quant aux types auxquels la stratégie générique sera éventuellement appliquée.

Il y a une solution à ce problème. Nous savons que, si la construction d'un attribut d'une instance donnée lève une exception, l'instance ne sera pas construite mais il se peut que certains de ses attributs, eux, aient bel et bien été construits au moment où l'exception est levée.

Le destructeur de l'instance qui n'est pas encore construite ne sera donc pas appelé, mais les destructeurs des attributs déjà construits, eux, seront appelés. C'est là qu'entre en jeu `auto_ptr`.

En effet, si un pointeur brut n'a pas de destructeur, un objet comme une instance d'`auto_ptr`, lui, en possède un, et ce destructeur détruira effectivement l'objet pointé.

De ce fait, si une exception est levée à la construction de l'un des attributs de `X` dans le cas exposé à droite, les attributs déjà construits seront eux-mêmes détruits, garantissant[56] une protection contre ces fuites de mémoire.

```
#include "Y.h"
#include "Z.h"
#include "Incopiable.h"
#include <memory>
using std::auto_ptr;
class X : Incopiable {
   auto_ptr<Y> y;
   auto_ptr<Z> z;
public:
   X() : y{new Y}, Z{new Z} {
   }
};
```

Notez par contre que nous avons, ici, fait de `X` un objet `Incopiable` puisque la copie par défaut d'un `X` serait une copie attribut par attribut; voir ***Le problème des copies***, ci-dessous, pour une explication plus détaillée de cette situation.

Notez aussi que nous aurions pu implémenter un constructeur de copie prudent sur une base manuelle. Faire de `X` un `Incopiable` est *une* option, pas *la seule* option possible.

[56] … presque; il reste un cas pathologique à couvrir, mais nous le ferons dans ***Fonctions génératrices `make_shared()` et `make_unique()`***, plus loin.

Le problème des copies

Tel que mentionné plusieurs fois déjà, la sémantique de responsabilité de `auto_ptr` en est une de possession unique, mais transférable.

Cette classe offre un constructeur de copie, qui est en fait un constructeur *destructif*. Ceci permet entre autres de passer une instance de `std::auto_ptr` par valeur à un sous-programme, mais l'impact de ce passage peut surprendre.

```
#include <memory>
#include <iostream>
using namespace std;
void f(auto_ptr<int> p) {
   cout << *p << endl;
}
int main() {
   auto_ptr<int> p{new int{3}};
   f(p);
   cout << *p << endl; // BOUM!
}
```

Prenons l'exemple à droite. Dès sa déclaration, l'objet `p` est responsable d'un pointeur vers un `int` de valeur `3`. L'appel de la fonction `f()` passe `p` par valeur, ce qui transfère la responsabilité vers le paramètre formel `p` de la fonction `f()`, et arrache (littéralement) la responsabilité sur le pointeur du paramètre effectif `p` de `main()`.

L'affichage dans la fonction `f()` ci-dessus fonctionnera et indiquera `3`, mais celui fait dans `main()` plantera car `p` aura perdu la responsabilité sur le pointeur lors de l'appel de `f()`. Au moment où l'affichage est fait dans `main()`, `p` mène vers un pointeur nul et `*p` est une erreur.

```
#include <memory>
#include <iostream>
using namespace std;
void f(const auto_ptr<int> &p) {
   cout << *p << endl;
}
int main() {
   auto_ptr<int> p{new int{3}};
   f(p);
   cout << *p << endl; // OK
}
```

Par contre, l'exemple proposé à droite est très valide puisque `f()` ne copie pas `p`. L'affichage dans `main()` y est aussi légal que celui dans `f()` puisque `p`, dans `main()`, n'a pas perdu le pointeur dont il était responsable en chemin.

`std::auto_ptr` et construction explicite

Le constructeur paramétrique d'un `auto_ptr` est qualifié `explicit`. Ceci n'est pas accidentel.

Examinons par exemple le programme proposé à droite. Sans un constructeur paramétrique explicite dans `auto_ptr`, l'appel à `f()` y serait légal et détruirait ce vers quoi pointe `p`. Ceci rendrait illégal (de manière silencieuse) l'affichage fait dans `main()` : en effet, le paramètre de `f()`, en prenant la responsabilité du pointeur reçu, détruirait ce vers quoi il pointe à la fin de l'exécution la fonction `f()`.

```
#include <memory>
#include <iostream>
using namespace std;
void f(auto_ptr<int> p) {
   cout << *p << endl;
}
int main() {
   int *p = new int{4}
   f(p); // illégal!
   cout << *p << endl;
}
```

En exigeant la construction explicite d'une instance de `auto_ptr`, les conceptrices et les concepteurs de cette classe obligent les gens cherchant à faire une manœuvre aussi dangereuse à y penser deux fois et à chercher une alternative sans danger telle que celle-ci (qui réalise explicitement une duplication de `*p`).

Notez qu'une conséquence de ce choix est que les deux invocations de construction paramétrique proposées ci-dessous n'auront pas le même résultat. En effet, la première implique une conversion explicite alors que la deuxième repose sur une conversion implicite.

```
#include <memory>
#include <iostream>
using namespace std;
void f(auto_ptr<int> p) {
   cout << *p << endl;
}
int main() {
   int *p = new int(4)
   f(auto_ptr<int>{new int{*p}});
   cout << *p << endl;
} // fuite de *p...
```

Seule la première est légale. Référez-vous à la section ***Constructeurs explicites*** de [POOv00] pour plus de détails à ce sujet.

```
auto_ptr<int> p{new int{4}}; // ok
auto_ptr<int> q = new int{4}; // illégal
```

De manière générale, pour les pointeurs dont la durée de vie est restreinte ou pour ceux créés spécifiquement pour un seul appel de fonction, il est préférable d'utiliser des pointeurs intelligents (donc l'idiome `RAII`) pour assurer la bonne gestion de la libération de la mémoire. N'oubliez toutefois jamais les aléas de la copie des objets `RAII`; aucun outil ne remplace l'esprit critique et l'analyse.

Sémantique de propriété exclusive : le type `std::unique_ptr`

Avec C++ 11, le type `auto_ptr` est considéré déprécié, dû aux difficultés qu'ont les gens à maîtriser sa sémantique de copie destructrice. La bibliothèque standard de C++ 11 propose en lieu et place un pointeur intelligent, nommé `unique_ptr`, représentant lui aussi une politique de propriété unique.

Le type `unique_ptr` a plusieurs avantages sur `auto_ptr`, entre autres :

- il offre une pleine sémantique de mouvement (abordée dans ***Sémantique de mouvement***, plus haut), et peut donc être utilisé dans un conteneur standard tel que `vector`;
- il ne peut être copié, ce qui résout le problème de la copie destructrice de son prédécesseur; et
- il permet de spécifier une action autre que `delete` lors de la libération du pointé, ce qui permet de l'utiliser avec des tableaux, par exemple.

Outre ces détails, `unique_ptr` se manipule comme `auto_ptr`.

Sémantique de propriété partagée : le type `std::shared_ptr`

La bibliothèque standard de C++ 11 propose aussi un autre pointeur intelligent, nommé `shared_ptr`, inspiré de *Boost*[57] et représentant une politique de propriété partagée.

Une instance de `shared_ptr` peut être copiée à loisir, et se comporte donc comme un type valeur. Chaque copie partage aussi un compteur de références sur le pointé, et constitue en quelque sorte un client de ce pointé. Lorsque le dernier client d'un pointé donné meurt, donc lorsque le nombre de références sur ce pointé devient nul, alors le pointé est détruit.

Le type `shared_ptr` est particulièrement utile en situation de multiprogrammation, lorsque plusieurs *threads* partagent une même donnée et lorsque l'ordre dans lequel les *threads* se termineront est inconnu *a priori* (ou simplement indéterministe).

[57] Voir http://www.boost.org/doc/libs/1_47_0/libs/smart_ptr/shared_ptr.htm pour des détails.

Fonctions génératrices `make_shared()` et `make_unique()`

Dans la bibliothèque `<memory>` de C++ 11, le type `shared_ptr` s'accompagne d'une fonction génératrice nommée `make_shared()`.

Il existe donc deux manières d'instancier un `shared_ptr<T>` :

- définir le `shared_ptr` explicitement à partir d'un objet nouvellement instancié; et
- appeler `make_shared()` en lui indiquant le type d'objet à partager et en lui suppléant les valeurs qui devront être passées au constructeur de ce type.

Vous trouverez un exemple de chaque sorte dans le petit programme proposé à droite.

Bien que les deux donnent des résultats analogues, **privilégiez la fonction `make_shared()` chaque fois que cela s'avère possible**. Pour certains types, par exemple ceux dont les constructeurs sont privés, vous ne le pourrez malheureusement pas, mais pour les autres…

```
struct Point {
   int x, y;
   Point(int x, int y)
      : x{x}, y{y}
   {
   }
};
#include <memory>
int main() {
   using namespace std;
   shared_ptr<Point> p0{new Point{3,4}};
   auto p1 = make_shared<Point>(3,4);
}
```

La raison soutenant cette recommandation est due à la structure interne d'un `shared_ptr<T>`. En effet, ce type doit essentiellement allouer deux ressources, soit le `T` vers lequel il pointe et un compteur entier du nombre de `shared_ptr<T>` menant vers ce pointé. Lorsque le partage d'un `T` se fait entre deux instances de `shared_ptr<T>` distinctes, le compteur est aussi partagé (c'est ce qui fait en sorte que la mécanique fonctionne).

Lorsque l'on instancie le `T` avant de créer le `shared_ptr<T>`, deux appels à new sont nécessaires (un pour le `T`, un autre pour le compteur). Lorsque la fonction `make_shared()` est utilisée, celle-ci est en mesure de ne faire qu'un seul new, pour un bloc de mémoire où se trouveront à la fois le compteur et le `T`, puis de gérer intelligemment le partage subséquent. Agir ainsi mène à du code plus rapide (l'allocation dynamique de mémoire est une opération dispendieuse, alors mieux vaut n'y avoir recours que lorsque c'est vraiment pertinent) et qui fragmente moins rapidement la mémoire vive.

Le standard C++ 11 était officiel depuis peu lorsque des gens se sont aperçus d'un oubli : ce standard n'offre pas de `make_unique()` pour servir de contrepartie à `make_shared()`. Ceci peut sembler inutile, puisqu'un `unique_ptr<T>` n'a pas, par définition, à partager de compteur du nombre de références sur le pointé, mais il s'avère que ces fonctions ont aussi un autre gros avantage : elles sont plus sécuritaires que l'alternative.

En effet, comme le souligne ***Herb Sutter*** dans [SutterMkUn], supposons le code suivant :

```
#include <memory>
using std::unique_ptr;
class A {
public:
   A(int); // peut lever une exception dans certaines circonstances
   // ...
};
class B {
   unique_ptr<A> p0;
   unique_ptr<A> p1;
public:
   B(int v0, int v1) : p0{new A{v0}}, p1{new A{v1}} {
   }
   // ...
};
```

On serait tenté de supposer que `B::B(int,int)` ne fuira pas si `A::A(int)` lève une exception. Cette supposition repose sans doute sur l'impression que l'ordre des événements sera l'un des deux cas ci-dessous (exprimés en pseudocode proche de C++) :

1er cas	2e cas
`operator new(sizeof(A))`	`operator new(sizeof(A))`
`A::A(v0)`	`A::A(v1)`
`unique_ptr<A>::unique_ptr(A*) // pour p0`	`unique_ptr<A>::unique_ptr(A*) // pour p1`
`// suivi de`	`// suivi de`
`operator new(sizeof(A))`	`operator new(sizeof(A))`
`A:: A(v1)`	`A::A(v0)`
`unique_ptr<A>::unique_ptr(A*) // pour p1`	`unique_ptr<A>::unique_ptr(A*) // pour p0`

Si cela s'avérait, il n'y aurait effectivement pas de risque si `A::A(int)` levait une exception, car jusqu'au moment où un `unique_ptr<A>` reçoit un `A*`, notre programme n'est pas responsable du pointeur, puis une fois le `unique_ptr<A>` créé, le programme est à la fois responsable et sécuritaire.

Le problème est que le compilateur est en droit, pour des raisons d'optimisation par exemple, de réaliser les deux `new` d'abord (peut-être même en parallèle), puis le reste des opérations. Dans un tel cas, si l'un des constructeurs de `A` échoue, l'autre `A` peut avoir été construit et fuir, faute d'avoir trouvé un responsable.

La solution la plus élégante à ce problème est d'utiliser `make_unique()`… mais voilà, tel que mentionné plus haut, C++ 11 n'offre pas ce service (il fait toutefois partie de ceux qui sont officiellement acceptés pour C++ 14). En attendant, heureusement, il est simple de suppléer notre propre version pour combler le manque et corriger cet irritant :

```
#include <memory>
using std::unique_ptr;
template <class T, class A>
   unique_ptr<T> make_unique(A arg) { // implémentation simpliste à un seul paramètre
      return unique_ptr<T>{new T{arg}};
   }
class A
{
public:
   A(int); // peut lever une exception dans certaines circonstances
   // ...
};
class B
{
   unique_ptr<A> p0;
   unique_ptr<A> p1;
public:
   B(int v0, int v1)
      : p0{make_unique<A>(v0)}, p1{make_unique<A>(v1)}
   {
   }
   // ...
};
```

Évidemment, mieux vaut avoir plusieurs versions de `make_unique()` pour divers nombres de paramètres (ou bien mieux encore : utiliser des *templates* variadiques [POOv03]).

Prise en charge des tableaux

Le type `unique_ptr` prend en charge les tableaux autant que les scalaires; dans ce cas, le pointé sera détruit avec l'opérateur `delete[]` plutôt qu'avec l'opérateur `delete`.

Un exemple simple pour illustrer le tout :

```
#include <memory>
using namespace std;
int main() {
   unique_ptr<int []> p{new int[10]};
   // ...
}
```

La manière de tirer profit de cette fonctionnalité est simplement d'indiquer dans le type du `unique_ptr` qu'il pointe vers un tableau plutôt que vers un scalaire.

Libération au choix du client

Le type `unique_ptr` permet de libérer les objets qu'il prend en charge par d'autres mécanismes que `delete` ou `delete[]`. Un exemple simple pour illustrer le tout :

```
#include <memory>
using namespace std;
class X {
   ~X() = default;
   friend void meurs(const X *p) { delete p; }
};
int main() {
   auto p = unique_ptr<X, void(*)(const X*)>{new X, meurs};
   // ...
}
```

La manière de tirer profit de cette fonctionnalité est simplement d'indiquer dans le type du `unique_ptr` la signature de la fonction qui servira à libérer le pointé, et à passer en paramètre au constructeur de cet objet la fonction en question.

Disparition des pointeurs bruts

En C++, les pointeurs bruts (comme les tableaux bruts) sont devenus, avec le temps, des outils spécialisés. Ce ne sont plus des composants de l'arsenal quotidien des programmeurs, qui ont entre les mains des vecteurs, des pointeurs intelligents et d'autres outils beaucoup plus puissants et flexibles.

Une application concrète du type `unique_ptr` est la rédaction de fabriques. Imaginez en effet le code suivant :

```
// ...
class X {
   // ...
public:
   X(double);
   // ...
};
X* fabriquerX(double init) {
   X *p = new X{init};
   // utiliser p
   return p;
}
```

Ce code est intrinsèquement fragile, car il dépend du code client pour ne pas laisser fuir de ressources. Un programme comme celui-ci :

```
// ...
int main() {
   fabriquerX();
}
```

…entraîne en effet une fuite de ressources : le `X*` retourné n'est pas pris en charge par le code client, l'objet pointé ne sera pas finalisé et sa mémoire ne sera pas libérée. On serait tentés de documenter `fabriquerX()` en indiquant que le code client est responsable de libérer le pointé, mais il s'agit d'une pratique qui repose sur la discipline humaine, et qui est par conséquent douteuse.

En retour, si nous écrivons `fabriquerX()` comme proposé à droite, le problème disparaît de lui-même : il devient impossible de laisser fuir le pointé, et celui-ci sera nécessairement finalisé.

Notez que le retour se fait ici par mouvement, le type `unique_ptr` étant incopiable.

```
unique_ptr<X> fabriquerX(double init) {
   auto p = unique_ptr<X>{new X{init}};
   // utiliser p
   return p;
}
```

À retenir à propos des pointeurs intelligents

Quelques éléments clés à retenir à propos des pointeurs intelligents :

- il est rare en C++ que nous ayons véritablement besoin de manipuler des pointeurs bruts;
- un pointeur intelligent détermine une sémantique de responsabilité quant au pointé. En ceci, il sécurise le code et le clarifie;
- deux pointeurs intelligents sont livrés avec le standard C++ 11, soit `unique_ptr` (responsabilité unique face au pointé) et `shared_ptr` (partage du pointé, qui est détruit lorsque son dernier client est détruit). Il est bien sûr possible d'en écrire d'autres;
- il est préférable de créer un pointeur intelligent à l'aide d'une fonction de fabrication (`make_unique()`, `make_shared()`) plutôt que manuellement. Le code est alors plus sécuritaire, et tend à occuper moins d'espace en mémoire;
- dans le doute, préférez `unique_ptr` qui est extrêmement léger[58], et n'utilisez `shared_ptr` que pour les cas où cela s'avère nécessaire. Un `shared_ptr` est plus lourd et plus lent à l'utilisation qu'un `unique_ptr`;
- un `unique_ptr` est capable de gérer un tableau alloué dynamiquement, tout comme il est capable de gérer un scalaire;
- un `unique_ptr` est capable de libérer son pointé par d'autres moyens que l'opérateur `delete` (ou `delete[]`, dans le cas d'un tableau);
- ces fonctionnalités (prise en charge d'un tableau, libération par fonctions au choix par le code client) seront aussi fournies pour `shared_ptr` à partir de C++ 14.

Comme me l'a fait remarquer **Manuel Parent**, on pourrait dire qu'un `unique_ptr<T>` se prête à la représentation d'une relation de composition, alors qu'un `shared_ptr<T>` se prête quant à lui à la représentation d'une relation d'agrégation.

[58] Dans bien des cas, `sizeof(unique_ptr<T>)==sizeof(T*)`!

Objets opaques et fabriques

Une question légitime qu'il est possible de se poser, après quelques séances de programmation à l'aide du langage C++ (ou des langages Java et C#, pour d'autres raisons[59]), est la suivante :

⇒ S'il faut inclure la déclaration d'une classe pour s'en servir, et si cette déclaration contient la liste des attributs de la classe de même que leur type, ***cela ne constitue-t-il pas un bris direct d'encapsulation***, du fait qu'un détail important d'implémentation (les attributs) s'avère visible à qui veut bien l'examiner?

La question est subtile, mais doit être considérée. En effet, un programme (d'autres objets, des sous-programmes) ne peut utiliser d'un objet que ce qui fait partie de son interface : ce qui en est public, ou protégé si on parle de classes dérivées.

Par contre, en C++, la déclaration entière d'une classe doit être connue pour qu'on puisse l'instancier. Ceci fait par exemple en sorte qu'une classe telle `CompteBancaire` (à droite) doive exposer aux humains (pas aux programmes) des détails comme le fait qu'elle conserve en elle plutôt que dans une base de données le solde courant du compte. Des tiers hostiles pourraient exploiter cette information.

```
class CompteBancaire {
public:
   using value_type = float;
private:
   value_type solde_;
public:
   // ...
   void deposer(value_type);
   void retirer(value_type);
   value_type solde() const;
};
```

Un programme n'utilisant que des méthodes publiques d'une classe ne devrait pas être affecté de manière adverse par des changements apportés à son implémentation, mais force est d'admettre qu'un changement au texte de la déclaration d'une classe sera requis même pour modifier un de ses membres privés, entraînant la compilation de tous les fichiers sources ayant inclus cette déclaration, directement ou indirectement. Ce *couplage physique* ralentit le développement en accroissant le temps de compilation même pour des modifications indépendantes du *couplage logique*[60].

Il est heureusement possible de réduire ce couplage physique en combinant classes, classes internes, interfaces, schéma de conception Fabrique et polymorphisme.

Cette section constituera donc à la fois une mise en application de plusieurs idées couvertes jusqu'ici et une ouverture sur de nouvelles options de design.

[59] Voir ***Obscurcissement des binaires (Code Obfuscation)***, plus loin.

[60] Dans le cas de gros systèmes, une compilation massive peut être longue. Un système bancaire peut nécessiter des jours de compilation si un nœud fondamental est modifié : un fichier d'en-tête inclus par beaucoup d'autres fichiers d'en-tête, par exemple. Un simulateur de vol industriel peut demander plusieurs heures de compilation si on doit le compiler en entier. Réduire le couplage physique peut signifier de grands gains en temps de développement, en plus d'entraîner des gains évidents sur le plan de la qualité de l'encapsulation résultante.

Séparer l'interface de l'implémentation

Comme c'est souvent le cas en informatique, l'une des clés du succès est de séparer l'interface de l'implémentation. Dans notre cas, les services à offrir doivent être offerts au code client; le client doit donc être en contact avec la déclaration de ces services, et doit être en mesure d'accéder, au moins indirectement, à l'entité qui les implémente.

Tel qu'indiqué plus haut, notre compte bancaire fortement simplifié offrira les services suivants :

- un service permettant de connaître le solde courant;
- un service permettant de déposer un montant; et
- un service permettant de retirer un montant.

Pour simplifier le propos, nos méthodes utiliseront des nombres à virgule flottante pour représenter les montants transigés. Au besoin, nous modifierons le type interne et public `value_type` pour qu'il devienne un *alias* vers une classe plus appropriée.

La technique `OO` consacrée pour exposer des services de manière abstraite est le recours aux interfaces. Ainsi, nous allons spécifier une interface (au sens technique du terme : classe strictement abstraite) exposant ces trois méthodes.

Notre interface sera nommée **`ICompteBancaire`**, le préfixe `I` étant (par convention) une indication aux programmeurs que nous ne voulons pas ici procéder à une généralisation complète et élégante de la notion de type, comme dans l'approche objet habituelle, mais bien à une opération menant à l'opacité maximale de notre implantation.

Réflexion `02.2` : la maxime *Do as the ints Do* de ***Scott Meyers***, mentionnée à quelques reprises dans [POOv00], devrait-elle être appliquée pour tous les types? Une brève discussion de ce sujet apparaît dans ***Réflexion `02.2` : des types valeurs et des autres***.

Une implémentation possible serait celle à droite[61]; elle n'expose aucun détail d'implantation. Notre objectif est de faire en sorte que les programmes n'accèdent aux comptes bancaires qu'à travers des instances dérivées d'`ICompteBancaire`.

Puisque les méthodes `deposer()` et `retirer()` sont toutes deux `void`, il faudra que l'objet derrière l'interface signale tout cas problème en levant une exception.

```
#ifndef ICOMPTEBANCAIRE_H
#define ICOMPTEBANCAIRE_H
struct ICompteBancaire {
   using value_type = float;
   virtual value_type solde() const = 0;
   virtual void deposer(value_type) = 0;
   virtual void retirer(value_type) = 0;
   virtual ~ICompteBancaire() = default;
};
#endif
```

Le fichier d'en-tête contenant `ICompteBancaire`, nommons-le `ICompteBancaire.h`, serait livré au client, car le client doit connaître l'interface pour solliciter ses services.

[61] Petite remarque sur la qualification `const` de la méthode `solde()` : il s'agit d'une contrainte forte puisque tout dérivé de `ICompteBancaire` se voit, fondamentalement, forcé de surcharger cette méthode tout en maintenant la même contrainte de constance. Ce choix de design est lourd de conséquences; il faut s'assurer que la contrainte de développement résultante est compensée par un meilleur respect des attentes du code client.

L'implémentation du compte bancaire, qui se situera derrière l'interface, se fera bien entendu à l'aide d'une classe dérivant de l'interface `ICompteBancaire`. Nous nommerons cette classe `CompteBancaireImpl`. Les fichiers sources déclarant et implémentant cette classe ne seront pas livrés aux clients[62] (les binaires, eux, pourront l'être).

Une implantation possible (et extrêmement simple) serait la suivante.

CompteBancaire.h

```
#ifndef COMPTE_BANCAIRE_H
#define COMPTE_BANCAIRE_H
#include "ICompteBancaire.h"
class MontantIllegal {};
class CompteBancaire
   : public ICompteBancaire {
   value_type solde_;
public:
   CompteBancaire();
   value_type solde() const;
   void deposer(value_type);
   void retirer(value_type);
};
#endif
```

CompteBancaire.cpp

```
#include "CompteBancaire.h"
CompteBancaire::CompteBancaire() : solde_{} {
}
auto CompteBancaire::solde() const -> value_type {
   return solde_;
}
void CompteBancaire::deposer(value_type mt) {
   if (mt <= 0) throw MontantIllegal{};
   solde_ += mt;
}
void CompteBancaire::retirer(value_type mt) {
   if (mt > solde()) throw MontantIllegal{};
   solde_ -= mt;
}
```

[62] C'est un élément important de notre démarche : la classe `CompteBancaireImpl`, déclarée dans `CompteBancaireImpl.h` et définie dans `CompteBancaireImpl.cpp`, sert pour usage interne seulement. Ces fichiers de doivent pas être dévoilés au client.

Fabriquer une abstraction

Exposer les services d'un objet de manière purement abstraite, à travers une interface, est une technique largement répandue dans le monde du développement informatique, en particulier quand le développement suit une approche OO. On retrouve entre autres cette façon de faire dans les pratiques d'à peu près tous les moteurs d'interopérabilité sophistiqués[63].

Les divers programmes souhaitant transiger avec un compte bancaire n'ont pas à en connaître l'implémentation. Les services seuls devraient suffire. Cependant, il est interdit d'instancier une interface comme `ICompteBancaire` puisque cette classe est abstraite, et si le code client devait instancier directement `CompteBancaire`, nous ne serions pas en meilleur position qu'au début de notre démarche : le code client serait, par définition, en contact avec la définition de la classe elle-même, dans tous ses détails, plutôt qu'avec ses seuls services purs.

Nous allons donc, pour parvenir à nos fins, ajouter à notre modèle un **service de fabrication**, souvent joint à un service de libération, qui ne laissera rien transparaître de ce qui est fabriqué (outre les services attendus, évidemment) et qui retirera des mains du code client la responsabilité d'instancier l'objet offrant les services.

Il existe au moins trois manières distinctes d'offrir un service de fabrication pour l'implémentation d'une abstraction donnée :

- des fonctions globales (parfois amies), dans les langages qui supportent ce concept;
- des services de l'interface, dans les langages qui le permettent; et
- une classe auxiliaire (parfois amie).

Nous examinerons tout d'abord ces deux approches, puis nous poserons notre regard sur un idiome de programmation qui raffinera encore plus le modèle. Nous examinerons au passage quelques ramifications de cet ajout d'une strate d'indirection dans le modèle de construction et de destruction d'objets.

S'il est clair qu'un service de fabrication soit nécessaire, l'interface étant abstraite et l'implémentation devant demeurer cachée, il est moins clair qu'un service de libération soit requis.

En fait, rien ne nous force de créer véritablement ou de détruire véritablement des objets dans nos services de fabrication et de libération. Si nous connaissons au préalable le nombre maximal de comptes bancaires requis, par exemple, il nous serait possible de les créer au démarrage et de ne jamais les libérer réellement.

Ce serait une approche risquée, convenons-en, si le code client pouvait invoquer `delete` à travers un pointeur d'interface.

C'est pourquoi encadrer à la fois la fabrication et la libération est une bonne idée. Nous pouvons implémenter ces fonctions avec une paire `new`/`delete`, mais nous pouvons aussi mettre en place des schèmes plus sophistiqués. Si nous n'encadrons que la fabrication, nous perdons cette liberté.

Ne soyez donc pas surpris de voir apparaître des destructeurs protégés et des qualifications `friend` dans les pages qui suivent. Nous allons encadrer la construction et la destruction, et ces façons de faire feront partie de notre arsenal. Pour en savoir plus sur la qualification `friend`, voir la section ***Amitié***, plus loin dans ce document.

[63] Pensez à `COM`, à `CORBA`, à `.NET`, aux services Web, *etc*.

Fabrication par fonction globale

Procéder par fonctions globales signifie ajouter à l'interface une fonction servant à encapsuler la fabrication de l'objet qui, derrière l'interface, offre le service, de même qu'une fonction servant à encapsuler la libération de cet objet.

Notre fonction de fabrication, ici, se nommera **`CreerCompteBancaire()`**, alors que notre fonction de libération se nommera **`LibererCompteBancaire()`**. Ces fonctions, bien que globales, font en fait partie du visage public de l'interface et seront livrées à partir du même fichier d'en-tête qu'elle.

En appliquant cette approche, nous obtiendrons les fichiers suivants. Le code client n'aura accès qu'au fichier `ICompteBancaire.h`, qui n'inclut que l'interface (opaque) et les services de fabrication et de libération. Les détails d'implémentation lui seront inconnus.

Remarquez le destructeur protégé et la fonction globale qualifiée `friend`. Cette manière de procéder mérite des explications :

- le destructeur protégé tient du fait que nous voulons empêcher le code client d'invoquer `delete` sur un `ICompteBancaire*`, donc un destructeur public est exclu, mais nous voulons aussi qu'il soit possible de définir un destructeur chez les dérivés de cette interface, ce qui exclut que le destructeur soit privé; alors que
- la qualification `friend` vient du fait que nous voulons que le service de libération, lui, puisse invoquer le destructeur d'un `ICompteBancaire`. Ce service étant offert ici sous la forme d'une fonction globale, il nous faut lui accorder des privilèges d'accès. Les amis d'une classe ont accès à ses membres privés et protégés.

Le fichier source ne sera livré aux clients que sous forme binaire. Il fera le lien entre l'interface et son implémentation, et implantera les services de fabrication et de libération, qui sont tous deux banals dans ce cas-ci.

ICompteBancaire.h

```
#ifndef ICOMPTE_BANCAIRE_H
#define ICOMPTE_BANCAIRE_H
struct ICompteBancaire {
   using value_type = float;
   virtual value_type solde() const = 0;
   virtual void deposer(value_type) = 0;
   virtual void retirer(value_type) = 0;
   friend void LibererCompteBancaire
      (const ICompteBancaire*) noexcept;
protected:
   virtual ~ICompteBancaire() = default;
};
ICompteBancaire* CreerCompteBancaire();
#endif
```

ICompteBancaire.cpp

```
#include "ICompteBancaire.h"
#include "CompteBancaire.h"
ICompteBancaire* CreerCompteBancaire() {
   return new CompteBancaire;
}
void LibererCompteBancaire
   (const ICompteBancaire *p) noexcept {
   delete p;
}
```

Un exemple de client pour notre compte bancaire muni d'une interface opaque serait celui proposé à droite. Aucun détail d'implémentation n'y transparaît.

```
#include "ICompteBancaire.h"
#include <iostream>
int main() {
   using std::cout;
   auto p = CreerCompteBancaire();
   p->deposer(10.25f);
   cout << p->solde();
   LibererCompteBancaire(p);
}
```

Par un service de l'interface

Si le langage choisi le supporte, il est possible d'exposer les services de fabrication et de libération en tant que méthodes de classe de l'interface elle-même.

Ceci évite la dépendance induite par `friend`, ce qui peut éviter des guerres de religions dans les (nombreux) milieux où les préjugés face à cette qualification foisonnent. En effet, l'interface a accès à ses propres membres, peu importe leur qualification de sécurité.

L'un des plus gros irritants de cette approche est qu'elle n'est pas universelle : seuls les langages où l'idée d'interface n'est pas limitée à un ensemble de méthodes abstraites permettent de la mettre en application (en Java ou en C#, par exemple, il est impossible de procéder ainsi).

Un exemple de client pour cette version serait :

```
#include "ICompteBancaire.h"
#include <iostream>
int main() {
   using std::cout;
   auto p = ICompteBancaire::creer();
   p->Deposer(10.25f);
   cout << p->solde();
   ICompteBancaire::liberer(p);
}
```

ICompteBancaire.h

```
#ifndef ICOMPTE_BANCAIRE_H
#define ICOMPTE_BANCAIRE_H
struct ICompteBancaire {
   using value_type = float;
   virtual value_type solde() const = 0;
   virtual void deposer(value_type) = 0;
   virtual void retirer(value_type) = 0;
   static ICompteBancaire* creer();
   static void liberer
      (const ICompteBancaire *) noexcept;
protected:
   virtual ~ICompteBancaire() = default;
};
#endif
```

ICompteBancaire.cpp

```
#include "ICompteBancaire.h"
#include "CompteBancaire.h"
ICompteBancaire* ICompteBancaire::creer() {
   return new CompteBancaire;
}
void ICompteBancaire::liberer
   (const ICompteBancaire *p) noexcept {
   delete p;
}
```

Du point de vue du code client, la nuance entre l'approche par fonctions globales et l'approche par services de l'interface est une nuance syntaxique, de surface.

Par une classe auxiliaire

Dans la littérature, l'implémentation la plus fréquemment rencontrée utilise une classe auxiliaire, une fabrique, qui offrira des services de création (et, parfois, de libération) des objets fabriqués.

Ici, la fabrique `FabriqueComptes` sera amie de l'interface `ICompteBancaire` pour être en mesure d'exploiter son destructeur sans le laisser visible au code client.

Selon les implémentations, les services de fabrication ou de libération seront parfois des méthodes d'instance, parfois des méthodes de classe. Ici, la fabrique n'ayant pas d'états en propre, des méthodes de classe suffiront.

La classe auxiliaire est, en quelque sorte, un élément constitutif du visage public de l'interface, ce qui explique que les deux classes soient livrées dans un seul et même fichier.

Un exemple de client pour cette version serait :

```
#include "ICompteBancaire.h"
#include <iostream>
int main() {
   using std::cout;
   auto p = FabriqueComptes::creer();
   p->deposer(10.25f);
   cout << p->solde();
   FabriqueComptes::liberer(p);
}
```

ICompteBancaire.h

```
#ifndef ICOMPTE_BANCAIRE_H
#define ICOMPTE_BANCAIRE_H
struct ICompteBancaire {
   using value_type = float;
   virtual value_type solde() const = 0;
   virtual void deposer(value_type) = 0;
   virtual void retirer(value_type) = 0;
   friend class FabriqueComptes;
protected:
   virtual ~ICompteBancaire() = default;
};
struct FabriqueComptes {
   static ICompteBancaire* creer();
   static void liberer
      (const ICompteBancaire *) noexcept;
};
#endif
```

ICompteBancaire.cpp

```
#include "ICompteBancaire.h"
#include "CompteBancaire.h"
ICompteBancaire * FabriqueComptes::creer() {
   return new CompteBancaire;
}
void FabriqueComptes::liberer
   (const ICompteBancaire *p) noexcept {
   delete p;
}
```

Vous remarquerez peut-être quelques irritants communs aux trois approches vues jusqu'ici :

- chacune repose sur un mécanisme manuel de libération des ressources;
- chacune demande au code client de manipuler des pointeurs;
- chacune rend évidente la stratégie de fabrication et de distanciation entre interface et implémentation mise en application. Le tout fonctionne, visiblement, mais ça manque franchement d'élégance.

Nous allons maintenant examiner comment utiliser ces techniques comme levier pour faire les choses avec flair et élégance.

Implémentation privée

Le recours aux interfaces et aux fabriques réduit le couplage :

- en séparant l'interface d'une classe, ses services, de leur implémentation;
- en dissimulant l'implémentation dans un fichier source loin des yeux du code client; et
- en offrant un service de fabrication et un service de libération de l'implémentation des services, permettant au code client de ne s'exprimer qu'en fonction de ses besoins et de son utilisation des services en question.

Nous avons notés que l'un des défauts de notre démarche jusqu'à présent est qu'elle est incomplète, en ce sens que le code client en vient à instancier manuellement (à travers les mécanismes de fabrication choisis) les objets dont il se sert, et (surtout) qu'il est forcé de libérer manuellement des objets. Il est possible de résoudre cet irritant en utilisant des mécanismes `RAII` maisons dans le code client, mais l'automatisme (et l'effort) devrait normalement se situer dans le code serveur.

Déclarations a priori

Pour pousser plus loin la démarche et automatiser pleinement la mécanique, nous aurons recours aux déclarations *a priori* [POOv00]. Allons-y donc d'un bref rappel.

Les déclarations *a priori* sont une technique pour éviter de révéler inutilement des éléments d'implémentation ou pour briser des références circulaires. Par exemple, dans le cas d'une relation biunivoque où une poule pond des œufs et où un œuf provient d'une poule, on ne pourrait pas écrire ceci en C++ :

Poule.h

```
#ifndef POULE_H
#define POULE_H
#include "Oeuf.h"
class Poule {
   enum { MAX_POUSSINS = 256 };
   Oeuf poussins_[MAX_POUSSINS];
   // ...
};
#endif
```

Oeuf.h

```
#ifndef OEUF_H
#define OEUF_H
#include "Poule.h"
class Oeuf {
   Poule maman_;
   // ...
};
#endif
```

La raison nous empêchant d'agir ainsi est que pour générer `Poule`, il faut connaître `Oeuf` et que pour générer `Oeuf`, il faut connaître `Poule`. Ce raisonnement nous met aussi face à la triste réalité que cette structure, si le compilateur devait l'accepter, générerait des classes s'incluant mutuellement de manière récursive ce qui mènerait à une explosion de la consommation d'espace mémoire (une poule contiendrait des œufs, chacun contenant une maman poule… voyez-vous l'erreur sémantique ici?).

Les déclarations *a priori* permettent d'indiquer qu'une classe existe, d'introduire son nom dans le programme, ce qui permet en retour d'utiliser son nom dans la mesure où on ne dépend en rien de sa structure (ni de sa taille, ni de ses méthodes).

Dans ce cas, en appliquant la technique dans les deux cas (bien qu'un seul cas aurait suffi; selon vous, lequel aurions-nous dû choisir?), on obtient ceci :

Poule.h

```
#ifndef POULE_H
#define POULE_H
// déclaration a priori
class Oeuf;
class Poule {
   enum { MAX_POUSSINS = 256 };
   Oeuf *poussins_[MAX_POUSSINS];
   // ...
};
#endif
```

Oeuf.h

```
#ifndef OEUF_H
#define OEUF_H
// déclaration a priori
class Poule;
class Oeuf {
   Poule *maman_;
   // ...
};
#endif
```

Visiblement, `Poule.h` n'a plus à inclure `Oeuf.h` et `Oeuf.h` n'a plus à inclure `Poule.h`. La récurrence infinie est, dirait-on, *brisée dans l'œuf*.

Cette technique fonctionne du fait que `Poule` utilise des indirections vers des instances de la classe `Oeuf` et qu'`Oeuf` utilise un pointeur sur une instance de `Poule`. Il est nécessaire, pour profiter de déclarations *a priori*, d'utiliser les objets ainsi déclarés de manière indirecte. Les pointeurs et les références ont en effet tous la même taille, ce qui permet au compilateur de générer le code requis.

Évidemment, le fichier source `Poule.cpp` inclura à la fois `Poule.h` et `Oeuf.h` de manière à pouvoir utiliser les deux classes (il en va de même pour `Oeuf.cpp`). Ceci n'entraînera aucun problème du fait que personne n'inclura les fichiers sources.

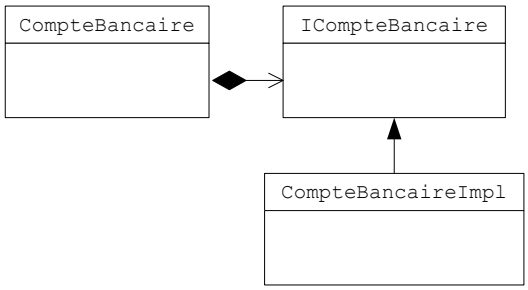
Appliquer la technique aux objets opaques : idiome `pImpl`

Nous souhaitons que le code client manipule des comptes bancaires, alors offrons-lui des comptes bancaires. Nous ne souhaitons par contre pas que le code client soit conscient des détails d'implémentation d'un compte bancaire, alors nous souhaitons séparer l'interface d'un compte bancaire de son implémentation.

Faisons le compte (!), nous avons besoin de trois classes :

- le compte bancaire. Cette classe sera un type valeur, mais qui n'aura aucune implémentation visible (outre un pointeur d'interface complètement opaque). Ce type portera le nom le plus convivial du groupe, puisque nous souhaiterons que le code client s'en serve;
- l'interface opaque, qui sera un type interne au type valeur ci-dessus. Le code client saura qu'il existe, sans plus; et
- l'implémentation de l'interface opaque, qui sera dissimulée dans un fichier source.

Le schéma UML proposé à droite résume la structure souhaitée, mais ne met pas en relief un important détail : `ICompteBancaire` est une classe interne à `CompteBancaire`, mais seule une déclaration *a priori* en sera visible dans le fichier d'en-tête livré au client.



Le code client verra `CompteBancaire`, un type valeur. La relation opératoire entre `CompteBancaire` et `ICompteBancaire` suivra une approche d'enrobage et de délégation, telle que présentée dans [POOv01].

Une invocation d'une méthode de `CompteBancaire` entraînera une délégation vers la méthode équivalente de l'interface `ICompteBancaire`, ce qui résultera en une invocation polymorphique vers le dérivé (caché) qu'est `CompteBancaireImpl`, où se situera le véritable code des méthodes.

⇒ L'idiome **`pImpl`** (pour *Private Implementation*), aussi nommé mur coupe-feu à la compilation (*Compiler Firewall*) est une technique par laquelle l'implémentation d'une classe est complètement cachée du code client, qui n'a plus accès qu'à des éléments opératoires, et encapsulée dans un type valeur. On obtient ainsi à la fois l'aisance à l'utilisation des types valeurs et le découplage de l'interface et de l'implémentation.

Vous trouverez plusieurs variantes de cet idiome sur [hdIdiom].

L'idiome `pImpl` s'exprime tel qu'indiqué plus haut. L'interface est une classe interne au type destiné à être utilisé directement et est déclarée *a priori*. Ceci ne permet de l'utiliser que pour des indirections, ce qui nous conviendra.

Les méthodes d'instance de `CompteBancaire` se limiteront, nous le verrons plus bas, à des délégations vers celles de l'interface interne.

Il aurait été possible de placer la déclaration entière de l'interface à même le fichier d'en-tête, plutôt que de se limiter à une déclaration *a priori* : nous aurions perdu en opacité mais nous aurions en retour pu gagner en vitesse en plaçant le code de délégation à même la déclaration de `CompteBancaire`. La déclaration *a priori* est publique pour qu'on puisse en dériver d'autres classes.

Certaines implémentations voudront encadrer les délégations par du code plus sophistiqué et préféreront l'opacité, comme nous le faisons ici, alors que d'autres préféreront perdre en opacité et gagner un peu de vitesse lors des appels de méthodes.

Nous avons défini `CompteBancaire` comme étant `Incopiable` [POOv01] pour faciliter la gestion du pointeur interne. Il existe des techniques plus sophistiquées qui permettraient de cloner ou (mieux, sans doute, pour un compte bancaire) de partager le pointeur interne, mais nous réserverons ces discussions pour plus tard.

La complexité est déplacée dans l'objet, comme il se doit, pas du côté du code client.

```
#ifndef COMPTEBANCAIRE_H
#define COMPTEBANCAIRE_H
#include "Incopiable.h"
class MontantInvalide {};
class CompteBancaire : Incopiable {
public:
   struct ICompteBancaire;
private:
   ICompteBancaire *compte;
   static ICompteBancaire *creer();
   static void liberer
      (const ICompteBancaire *) noexcept;
public:
   using value_type = float;
   CompteBancaire() : compte{creer()} {
   }
   value_type solde() const;
   void deposer(value_type);
   void retirer(value_type);
   ~CompteBancaire() {
      liberer(compte);
   }
};
#endif
```

Nous pouvons invoquer les méthodes de classe que sont `CompteBancaire::creer()` et `CompteBancaire::liberer()` à même le fichier d'en-tête puisque ces méthodes ne dévoilent strictement rien sur l'implémentation ou sur la structure de l'interface.

Le programme de test tout simple proposé à droite témoigne immédiatement d'une amélioration importante du point de vue de la convivialité et de la stabilité du code client. En effet, plus besoin d'opérations de fabrication et de libération devant être invoquées manuellement, plus de fuites lors d'une exception, plus de recours à des pointeurs ou à l'opérateur `->` pour invoquer des méthodes sur le compte bancaire créé, et un nom de classe significatif.

Comparez avec les versions antérieures pour constater les gains, même dans le cas d'un si petit programme.

```
#include "CompteBancaire.h"
#include <iostream>
int main() {
   using std::cout;
   CompteBancaire c;
   c.deposer(10.25f);
   cout << c.solde();
}
```

Examinons maintenant le code placé à l'écart, dans le fichier source `CompteBancaire.cpp`. Ce code comprend plusieurs éléments et constitue une forme de système client/ serveur à lui seul.

L'interface est d'abord déclarée, étant indépendante des autres éléments. Elle ressemble fortement à celles des approches précédentes, à ceci près qu'elle est maintenant déclarée de manière interne à `CompteBancaire`.

Le `using` est un *alias* dont le rôle est d'alléger l'écriture. De son côté, la classe `CompteBancaire` est qualifiée d'amie pour lui donner accès au destructeur protégé de l'interface.

Les méthodes de `CompteBancaire` sont telles qu'annoncées : elles relaient les demandes à l'implémentation cachée. Nous pourrions ajouter une forme d'encadrement à ces délégations si cela semblait pertinent.

```
#include "CompteBancaire.h"
struct CompteBancaire::ICompteBancaire {
   using value_type = CompteBancaire::value_type;
   virtual value_type solde() const = 0;
   virtual void deposer(value_type) = 0;
   virtual void retirer(value_type) = 0;
   friend class CompteBancaire;
protected:
   virtual ~ICompteBancaire() = default;
};


auto CompteBancaire::solde() const -> value_type {
   return compte->solde();
}
void CompteBancaire::deposer(value_type mt) {
   compte->deposer(mt);
}
void CompteBancaire::retirer(value_type mt) {
   compte->retirer(mt);
}
```

L'implémentation des services du compte bancaire est aussi banale que dans les versions précédentes, mais ces services pourraient évidemment être aussi complexes que nécessaire.

Remarquez que `CompteBancaireImpl` est une classe dérivée d'`ICompteBancaire`, elle-même interne à `CompteBancaire`. Ceci explique la qualification publique de cette interface dans la classe `CompteBancaire` : si elle avait été déclarée privée, il nous aurait été impossible d'en dériver par la suite.

Remarquez aussi que le type `value_type` du parent `ICompteBancaire`, est hérité par ses enfants, ce qui explique son utilisation implicite par la classe `CompteBancaireImpl`.

```
class CompteBancaireImpl
   : public CompteBancaire::ICompteBancaire
{
   value_type solde_ {};
public:
   CompteBancaireImpl() = default;
   value_type solde() const {
      return solde_;
   }
   void deposer(value_type mt) {
      if (mt <= 0) throw MontantInvalide{};
      solde_+= mt;
   }
   void retirer(value_type mt) {
      if (mt <= 0 || mt > solde())
         throw MontantInvalide{};
      solde_-= mt;
   }
};
```

Enfin, les services de fabrication et de libération sont tout ce qu'il y a de plus banals. La beauté ici est que leur utilisation est encapsulée dans la classe `CompteBancaire`; le code client ne les verra pas.

```
auto CompteBancaire::creer() -> ICompteBancaire* {
   return new CompteBancaireImpl;
}
void CompteBancaire::liberer
   (const CompteBancaire::ICompteBancaire *p) noexcept {
   delete p;
}
```

Implémentation plus idiomatique

Une implémentation C++ plus idiomatique d'un `CompteBancaire` respectant `pImpl` serait la suivante. Pour l'interface :

`CompteBancaire.h`

```
#ifndef COMPTE_BANCAIRE_H
#define COMPTE_BANCAIRE_H
#include <memory>
class MontantInvalide {};
class CompteBancaire {
   class Impl;
   std::unique_ptr<Impl> p;
public:
   CompteBancaire();
   ~CompteBancaire() noexcept;
   CompteBancaire(CompteBancaire&&);
   CompteBancaire& operator=(CompteBancaire&&);
   using value_type = float;
   void deposer(value_type);
   void retirer(value_type);
   value_type solde() const;
};
#endif
```

Remarquez que nous utilisons une déclaration *a priori* (voir ***Déclarations a priori***) pour la classe `CompteBancaire::Impl`, qui implémentera les services que nous souhaitons isoler, et un `unique_ptr` (voir ***Introduction aux pointeurs intelligents***) pour en gérer convenablement la durée de vie. Un `unique_ptr` est incopiable mais déplaçable, ce qui explique que nous nous soyons permis d'implémenter la sémantique de mouvement ici.

Il est essentiel de déclarer ici le destructeur de `CompteBancaire`, même si son implémentation sera triviale. La raison est qu'à ce stade, `CompteBancaire::Impl` est une classe incomplète, donc dont on ne connaît pas la définition; nous voulons éviter que le compilateur ne génère tout de suite un destructeur pour `CompteBancaire`, car celui-ci appellerait le destructeur de `p` qui, avec l'information disponible jusqu'ici, serait incorrect (il appellerait le destructeur d'`Impl`, et nous ne savons même pas si cette méthode sera accessible!).

Pour l'implémentation, nous avons :

CompteBancaire.cpp

```
#include "CompteBancaire.h"
class CompteBancaire::Impl {
   friend class CompteBancaire;
   friend class std::unique_ptr<Impl>;
   using value_type = CompteBancaire::value_type;
   value_type solde_ {};
   Impl() = default;
   value_type solde() const {
      return solde_;
   }
   void deposer(value_type mt) {
      if (mt <= 0) throw MontantInvalide{};
      solde_ += mt;
   }
   void retirer(value_type mt) {
      if (mt <= 0 || mt < solde()) throw MontantInvalide{};
      solde_ -= mt;
   }
};
CompteBancaire::CompteBancaire() : p{new Impl} {
}
CompteBancaire::~CompteBancaire() = default;
CompteBancaire::CompteBancaire(CompteBancaire&&) = default;
CompteBancaire& CompteBancaire::operator=(CompteBancaire&&) = default;
auto CompteBancaire::solde() -> value_type const {
   return p->solde();
}
void CompteBancaire::deposer(value_type mt) {
   p->deposer(mt);
}
void CompteBancaire::retirer(value_type mt) {
   p->retirer(mt);
}
```

Nous avons ici une implémentation complète :

- elle est sécuritaire face aux exceptions, car l'attribut `p` libérera automatiquement le pointé;
- elle est implicitement incopiable, grâce à l'attribut `p` qui est lui-même incopiable;
- elle est déplaçable, implémentant les opérations clés de la sémantique de mouvement;
- elle encapsule totalement les détails de son implémentation; et
- elle n'a même pas besoin de polymorphisme pour réaliser sa tâche.

Dans d'autres langages

Les langages .NET et Java ne font pas de distinction entre fichier source et fichier d'en-tête. Les méthodes sont systématiquement définies à même leur déclaration. Ainsi, une approche comme l'idiome PIMPL n'y trouve pas vraiment sa place.

Il est à noter que les langages .NET permettent de définir une classe par morceaux, ce qu'on nomme des classes partielles [POOv01]. Cette approche ne peut toutefois être appliquée qu'à l'intérieur d'un même assemblage, et ne sert donc pas à introduire une forme d'opacité dans les programmes.

Les interfaces sont des concepts plutôt que des techniques en Java et dans les langages .NET, nous l'avons aussi vu dans [POOv01]. Comme dans tous les langages, elles servent à faciliter le polymorphisme à travers des abstractions pures.

Les fabriques sont un schéma de conception, et font partie des pratiques répandues dans tous les langages OO. L'idée d'offrir une fonction de libération et de l'encapsuler dans un type valeur n'a que peu de sens sans destructeurs déterministes, ce qui explique que cette partie de la technique proposée ici soit connue en C++ mais pas en Java ou dans les langages .NET.

*Obscurcissement des binaires (*Code Obfuscation*)*

Dans les langages où les classes sont déclarées et définies d'un seul tenant, surtout ceux où les binaires ont un format fixe destiné à une machine virtuelle, la décompilation des binaires pour fins de *rétroingénierie* (ou de piratage) est une pratique fréquente[64]. C'est pourquoi Java et les langages .NET ont souvent recours à des techniques pour transformer les binaires dans le but de les obscurcir (pour les rendre difficiles à décompiler ou, si elles sont décompilées, difficiles à comprendre) sans toutefois altérer la sémantique d'exécution.

Il existe donc plusieurs outils commerciaux d'obscurcissement du code (en anglais : des *Code Obfuscators*) auxquels il est possible d'avoir recours avant de livrer des binaires à des utilisateurs. Les techniques d'obscurcissement du code sont un sujet de recherche universitaire pour le moment, alors nous ne les couvrirons pas ici, mais une recherche dans Internet vous donnera accès à beaucoup d'information à ce propos.

Pour éviter les ennuis à l'édition des liens, les techniques d'obscurcissement du code ne peuvent pas vraiment modifier les volets publics et protégés des classes; raison de plus pour mettre l'accent sur les éléments privés, d'ailleurs. Retenons que tout code compilé, même obscurci, peut être brisé. Le code vraiment opaque place les interfaces sur un ordinateur (celui du client) et l'implémentation sur un autre, mais il faut alors un protocole de communication entre les deux, ce qui crée de nouveaux problèmes.

Certains diront d'ailleurs que le combat pour dissimuler les sources est perdu d'avance; selon eux, mieux vaut y aller à code ouvert et offrir des services pour la personnalisation, l'entretien et la mise à jour des sources. À vous de voir.

---

[64] Le *Java Development Kit*, ou JDK, vient d'ailleurs avec un *décompilateur* standard gratuit. Pour les langages .NET, les outils ildasm.exe et ilasm.exe permettent respectivement de désassembler et de réassembler un binaire .NET pour en examiner – et en modifier, si le cœur nous en dit – les entrailles.

Exercices – Série 01

EX00 – Revenons à l'exemple des classes `Poule` et `Oeuf` proposé plus haut. Implémentez ces classes manière à ce qu'une instance de `Poule` contienne (par composition) des instances d'`Oeuf` mais qu'une instance d'`Oeuf` connaisse indirectement (à l'aide d'un pointeur) la `Poule` qui est sa maman. Rédigez (ou tentez de rédiger) le constructeur par défaut, le constructeur paramétrique, le constructeur par copie, l'opérateur d'affectation et le destructeur des classes `Poule` et `Oeuf`. Écrivez un programme de test pour ces deux classes. Rencontrez-vous des problèmes? Si oui, quels sont-ils et lors de l'implémentation de quelles opérations surviennent-ils? Quelle approche recommandez-vous?

EX01 – Revenons à l'exemple des classes `Poule` et `Oeuf` proposé plus haut. Implémentez ces classes manière à ce qu'une instance de `Poule` contienne (par composition) des instances d'`Oeuf` mais qu'une instance d'`Oeuf` connaisse indirectement (à l'aide d'une référence) la `Poule` qui est sa maman. Rédigez (ou tentez de rédiger) le constructeur par défaut, le constructeur paramétrique, le constructeur par copie, l'opérateur d'affectation et le destructeur des classes `Poule` et `Oeuf`. Écrivez un programme de test pour ces deux classes. Rencontrez-vous des problèmes? Si oui, quels sont-ils et lors de l'implémentation de quelles opérations surviennent-ils? Quelle approche recommandez-vous?

EX02 – Prenez la classe d'exception `MontantIllegal` dans l'exemple de `CompteBancaire`, et adaptez-la de manière à ce qu'elle contienne la valeur du montant problématique et à ce qu'elle offre les services pertinents pour consulter ce montant (accesseur(s), constructeur(s) et autres). Assurez-vous aussi que le type du montant soit, en tout temps, nécessairement le même que le type `CompteBancaire::value_type`.

EX03 – Considérez les changements apportés par `EX02`. Est-ce que ce changement entraîne des risques lors d'une levée d'une exception de ce type? Expliquez votre réponse.

Foncteurs

Une technique intéressante, mêlant approche OO et équivalence opérationnelle des types par la surcharge d'opérateurs, est celle qui permet de concevoir un ***foncteur***, c'est-à-dire un objet qui peut être utilisé *syntaxiquement* comme l'est une fonction.

⇒ On nomme **foncteur** un objet capable de se comporter comme une fonction, ou une fonction capable de se comporter comme un objet. Ce terme a aussi des acceptions plus mathématiques, sans lien direct avec notre propos.

⇒ Un foncteur est un **foncteur pur** s'il n'a aucun état et si le fait de l'utiliser comme une fonction n'entraîne aucun effet secondaire[65].

La terminologie anglaise typiquement rencontrée dans le monde C++ est *Function Objects*, du fait que certains *aficionados* de la programmation fonctionnelle réagissent négativement à l'utilisation du mot foncteur au sens d'objet se comportant comme une fonction. Pourtant, en programmation fonctionnelle, un foncteur est une fonction qui se comporte comme un objet et transporte ses propres états avec elle. On comprendra l'analogie entre cette idée et celle d'un objet qui se comporte comme une fonction. Le terme foncteur nous conviendra pleinement.

L'idée de foncteur peut être vue, entre autres, comme un élargissement de l'idée de fonction[66]. En effet, un foncteur possède à la fois un volet fonctionnalité, des états (c'est un objet, il peut donc avoir des attributs) et une gamme de services (c'est un objet, il peut donc avoir des méthodes).

L'extrait de code à droite définit la fonction `f()` et le foncteur `G`. On reconnaît `G` comme un foncteur parce qu'il expose une méthode `operator()` qui permet au code client d'utiliser une instance de `G` comme un utiliserait une fonction.

Le programme de test montre quelques invocations possibles de la fonction `f()` et de notre petit foncteur `g`. Certaines de ces invocations sont directes, d'autres non.

```
int f() noexcept {
   return 3;
}
struct G {
   int operator() const noexcept {
      return 3;
   }
};
template <class F>
   auto h(F fct) -> decltype(fct()) {
      return fct();
   }
int main() {
   auto i0 = f();
   G g; // construction
   auto i1 = g(); // appel
   auto i2 = h(f); // appel de f à travers h()
   // appel de G::operator() à travers h()
   // à partir d'une instance anonyme de G
   auto i3 = h(G{});
};
```

[65] Un effet secondaire typique serait de modifier d'une variable globale. Un foncteur pur se comportera plutôt comme une fonction au sens classique.

[66] À noter qu'on parle ici d'un concept de fonction un peu différent (certains diront plus large) que celui de la fonction au sens traditionnel des mathématiques, où l'on joint un et un seul élément de l'image de la fonction à tout élément de son domaine. Ceci parce que les fonctions en C++ peuvent avoir des états internes persistants, et que les objets fonctions étendent forcément cette idée de par leur structure d'objet propre. Il y a d'ailleurs des cas pour lesquels on préférera avoir des foncteurs purs, sans états outre des constantes.

Remarquez les initialisations des variables `i3` et `i4` :

- pour `i3`, l'appel à `h()` s'écrit `h((G()))` (notez les parenthèses autour de `G()`) pour distinguer l'instanciation d'un `G` anonyme de la signature `G()`, qui peut (dans certains contextes) signifier « fonction sans paramètres et retournant un `G` ». Nous y reviendrons dans la section ***Délégués en C++***;
- pour `i4`, nous utilisons la syntaxe unifiée de l'initialisation que permet C++ `11` et qui, elle, ne souffre pas d'ambiguïtés syntaxiques.

Un exemple de foncteur possible serait `Sommation`, comme celui-ci (exemple faisant la somme des éléments d'un vecteur d'entiers de deux manières différentes) :

```
class Sommation {
   int val_  {};
public:
   Sommation() = default;
   Sommation(int init) noexcept : val_{init} {
   }
   operator int() const noexcept {
      return val_;   // pour extraire la valeur
   }
   void operator()(int i) noexcept {
      val_+= i; // pour l'opération de sommation
   }
};
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
void f(const vector<int> &v) {
   Sommation s;
   s = for_each(begin(v), end(v), s);
   cout << "Somme: " << s << endl;
}
void g(const vector<int> &v) { // alternative, même résultat
   cout << "Somme: " << for_each(begin(v), end(v), Sommation{}) << endl;
}
```

Cet exemple est à la fois élégant, compact et très efficace. Un exemple équivalent mais utilisant une répétitive de forme classique aurait eu l'air de ceci :

```
void h(const vector<int> &v) {
   Sommation s;
   for (auto & val : v)
      s(val); // on utilise clairement s comme une fonction accumulant des valeurs
   cout << "Somme: " << s << endl;
}
```

La bibliothèque standard fait fréquemment usage de foncteurs, entre autres parce qu'en conjonction avec les conteneurs standards et les itérateurs, ces objets offrent un mécanisme extrêmement puissant de rédaction de code efficace et sécuritaire.

Une version plus simple et plus flexible (grâce à un foncteur générique) du même programme serait celle proposée à droite[67].

De par son caractère utilitaire et l'utilisation très ponctuelle qui en est faite, elle aurait pu être une simple `struct`, avec un attribut public par-dessus le marché, bien que j'aie décidé d'encapsuler son attribut `val_` et d'y permettre l'accès par un accesseur au nom analogue (`valeur()`) pour rester conforme à l'ensemble de ce document.

Ici, `Somme` est plus un service intelligent qu'une unité d'encapsulation.

Notre code profite du fait que l'algorithme `for_each()` retourne (par copie) l'opération qu'il aura appliquée sur tous les éléments de la séquence cible. Cela permet d'invoquer `valeur()` directement à travers la valeur ainsi retournée et nous laisse avec une forme à la fois expressive et très compacte.

L'idée de foncteur remonte à loin. Dans les langages strictement fonctionnels comme Lisp ou ses dialectes, l'emploi de foncteur permet d'implanter des concepts comme des ensembles infinis mais dénombrables, et gardant à même le foncteur une sorte d'état des opérations en cours[68].

```
template <class T>
   class Somme {
      T val_ {};
   public:
      Somme() = default;
      Somme(T init) : val_{init} {
      }
      void operator()(const T &val) {
         val_ += val;
      }
      T valeur() const {
         return val_;
      }
   };
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
   vector<int> v;
   // remplir v;
   cout << for_each(
               begin(v), end(v), Somme<int>{}
           ).valeur() << endl;
}
```

[67] On peut faire encore mieux en utilisant l'algorithme `accumulate()`. Afficher la somme des éléments d'un vecteur d'entiers `v` peut s'écrire `cout<<accumulate(begin(v),end(v),0);` (le troisième paramètre est la valeur initiale du cumul). L'algorithme `accumulate()` est d'ailleurs plus polyvalent que ce petit exemple ne le laisse paraître…

[68] Par exemple, si on veut énumérer les nombres naturels au sens de ***Giuseppe Peano*** (pour celles et ceux qui sont familières ou familiers avec cette célèbre définition), un foncteur peut permettre d'obtenir le successeur du plus récent naturel produit… dans la mesure où le plus récent naturel produit est connu par le foncteur en question. Le maintien d'un état, même minimal, est nécessaire à l'implémentation d'une telle opération.

Foncteurs et qualité du code

La plupart des algorithmes STL préfèrent les foncteurs aux fonctions pour représenter les opérations dans les algorithmes génériques. Cela se comprend : *le compilateur génère typiquement du code **plus « performant »** pour un foncteur que pour une fonction*.

Examinez la fonction générique `f()`, à droite, qui reçoit en paramètre quelque chose qu'elle utilise comme une fonction. Quand on lui passe `fA` en paramètre, le compilateur génère une version de `f()` capable de recevoir *ce qu'est* `fA`, soit l'adresse d'une fonction sans paramètres retournant un `int`[69].. La même fonction `f()` sera appelée pour tous les appels prenant en paramètre une fonction ayant cette signature.

```
int fA() {
   return 3;
}
struct fB {
   int operator() const {
      return 3;
   }
};
template <class Op>
   auto f(Op oper) -> decltype(oper()) {
      return oper();
   }
#include <iostream>
int main() {
   using std::cout;
   cout << f(fA) << ' ' << f(fB{});
}
```

Quand le compilateur voit une adresse de fonction, il ne peut présumer de rien. C'est alors un pointeur, susceptible de changer selon les invocations de l'algorithme. Quand on lui passe une instance anonyme de `fb`, par contre, le type du paramètre est clairement `fB`, et le compilateur peut réaliser du *Method Inlining* sur sa méthode `operator()`.

Ainsi, en général, le code suivant sera plus rapide dans la version du centre et de droite que dans la version de gauche, du fait que le compilateur aura de meilleures opportunités d'optimisation :

```
#include <algorithm>
template <class T>
   T Carre(T x) {
      return x * x;
   }
int main() {
   using namespace std;
   int tab[]{ 2,3,5,7,11 };
   transform(
      begin(tab),end(tab),
      begin(tab),Carre<int>
   );
   // ...
}
```

```
#include <algorithm>
template <class T>
   struct Carre {
      T operator()(T x) const {
         return x * x;
      }
   };
int main() {
   using namespace std;
   int tab[]{ 2,3,5,7,11 };
   transform(
      begin(tab),end(tab),
      begin(tab),Carre<int>{}
   );
   // ...
}
```

```
#include <algorithm>
struct Carre {
   template <class T>
      T operator()(T x) const {
         return x * x;
      }
};
int main() {
   using namespace std;
   int tab[]{ 2,3,5,7,11 };
   transform(
      begin(tab),end(tab),
      begin(tab),Carre{}
   );
   // ...
}
```

[69] Le type est `int(*)()` ce qui, vous en conviendrez, n'est pas très lisible.

À fonctionnalité équivalente, un foncteur permettra la génération de meilleur code qu'un pointeur de fonction. La POO permet aujourd'hui du code plus rapide que ne le fait la programmation procédurale.

Notez que la programmation générique n'est utilisée dans ces exemples que par souci de généralité. On aurait pu écrire le même code sans y avoir recours.

Cela est tellement vrai qu'il est en général préférable d'enrober une fonction par un foncteur lui délégant le travail dans un programme où l'invocation répétée de cette fonction par un algorithme standard constitue l'une des principales tâches réalisées par le programme.

En effet, si votre programme exploite une fonction `CalculComplexe()` opérant sur un double et retournant un double, alors dans le programme suivant la deuxième invocation de `std::transform()` sera, en général, plus rapide que la première :

```
#include <algorithm>
#include <vector>
using namespace std;
double calcul_complexe(double);
struct FctCalculComplexe {
   double operator()(double val) const {
      return calcul_complexe(val);
   }
};
int main() {
   vector<double> v;
   // ...remplir le vecteur...
   transform(begin(v), end(v), // intervalle à modifier
             begin(v),         // début de la destination (v lui-même)
             calcul_complexe); // bien; utilise la fonction
   // afficher son contenu
   v.clear();
   // remplir le vecteur
   transform(begin(v), end(v),     // intervalle à modifier
             begin(v),             // début de la destination (v lui-même)
             FctCalculComplexe{}); // mieux: utilise le foncteur
   // afficher son contenu
}
```

Mêler efficacité et élégance

Dans [EffStl], item 7, le bien connu ***Scott Meyers*** suggère une variante à l'écriture sous forme de foncteur d'une classe comme `Carre` ci-dessus. La technique qu'il propose permet d'obtenir, dans une même classe, autant de méthodes d'une forme donnée que nécessaire, incluant l'opérateur `()`.

Nous avons entrevu cette approche dans la section ***Généricité par classe ou par méthode***, plus haut dans ce document. En déplaçant la généricité du type d'un foncteur vers ses méthodes, il devient possible d'avoir à la fois :

- la vitesse et les états d'un foncteur;
- la flexibilité de la programmation générique; et
- d'effacer du code client l'exigence de spécifier explicitement les types sur lesquels il opère.

En effet, si nous remplaçons l'écriture proposée à droite…

… par celle-ci, cela a pour effet de transformer la nature de `Carre`, qui passe de *classe générique à une seule méthode* à *classe non générique munie d'une méthode générique*, donc pour laquelle le compilateur générera autant de versions d'une même méthode que nécessaire.

```
template <class T>
   struct Carre    {
      T operator()(T x) const {
         return x * x;
      }
   };
```

```
struct Carre {
   template <class T>
      T operator()(T x) const {
         return x * x;
      }
};
```

L'utilisation de la version non générique de `Carre` dans le code client peut se faire comme suit. Plus besoin de spécifier le type appliqué à `Carre` puisque cette classe n'est plus générique; c'est le compilateur qui générera des méthodes au besoin :

```
int main() {
   vector<int> v;
   // remplir v
   transform(begin(v), end(v), begin(v), Carre{}); // pas besoin de spécifier le type!
   // afficher les éléments copiés
}
```

En invoquant l'opérateur `()` de l'instance de `Carre` avec un paramètre d'un type `T` donné, l'algorithme `transform` (ou tout autre algorithme) provoquera la génération d'une méthode `operator()(T)`. Une classe, plusieurs méthodes, tout aussi rapide mais de manière transparente pour le client!

Intégrer foncteurs, fonctions et algorithmes

Les foncteurs servent de complément au système de types de C++ et permettent d'utiliser conteneurs standards et algorithmes standards de manière homogène pour tous les types.

Imaginons par exemple un vecteur d'instances de `Joueur` dans un pool de hockey, un peu comme dans [POOv00], section ***Exercices – Série 10***. L'opérateur `<` défini sur un `Joueur` pourrait définir une relation d'ordre entre deux instances de `Joueur`, relation qui serait basée sur les points de chacun.

Supposons toutefois qu'on souhaite trier les instances de `Joueur` dans un vecteur sur la base de leur nom de famille, ou encore du nom puis (de manière secondaire) du prénom. Tristement, l'algorithme `std::sort()` repose sur l'application de l'opérateur `<`… n'est-ce pas?

En fait, pas vraiment. L'algorithme `std::sort()`, comme bien des algorithmes standard, prend deux *ou trois* paramètres. Dans la version à trois paramètres, les paramètres sont :

- un itérateur sur le début de la séquence à trier;
- un itérateur sur la fin de la séquence à trier; et
- un prédicat binaire, donc quelque chose se comportant comme une fonction booléenne à deux opérandes. Pour obtenir le comportement de `std::sort()` à deux opérandes, ce prédicat doit être le foncteur `std::less()`, une opération générique qui utilise l'opérateur `<` pour déterminer l'ordre entre ses opérandes.

Imaginons donc un programme comme celui-ci. Nous présumons :

- l'existence d'un opérateur `<` entre deux instances de `Joueur` retournant vrai seulement si l'opérande de gauche a moins de points que l'opérande de droite;
- l'existence d'un opérateur de projection sur un flux permettant d'afficher un Joueur à la console.

C'est simple, rapide, clair et compact. Reste à voir quels sont les ajustements requis pour amener ce programme à trier les instances de `Joueur` selon d'autres critères que leurs points, et ce sans dénaturer la classe `Joueur` telle qu'elle existe.

```
#include "Joueur.h"
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
int main() {
   using namespace std;
   vector<Joueur> v;
   // ...remplir le vecteur (omis)...
   sort(begin(v), end(v));
   copy(begin(v), end(v),
        ostream_iterator<Joueur>{cout, " "});
}
```

On souhaite donc ici pouvoir suppléer une politique auxiliaire d'ordonnancement entre deux instances de `Joueur` à l'algorithme `std::sort()`, et faire en sorte que par défaut cette politique soit d'avoir recours à l'opérateur `<`.

Si nous présumons que les méthodes d'instance publiques `nom()` et `prenom()`, toutes deux qualifiées `const`, sont exposées par chaque `Joueur` et retournent respectivement son nom et son prénom, alors on pourrait définir une fonction prédicat telle que `ordre_lexico()` (à droite). Vous pouvez aussi envisager un foncteur pour faire le même travail.

```
// comparaison en ordre lexicographique
bool ordre_lexico
   (const Joueur &j0, const Joueur &j1) {
   return
      j0.nom() < j1.nom() ||
      j0.nom() == j1.nom() &&
      j0.prenom() < j1.prenom();
}
```

Muni de ce prédicat, offrir le même programme mais en changeant la stratégie de tri n'implique qu'un très petit changement :

```
// inclusions, définition d'ordre_lexico()
int main() {
   vector<Joueur> v;
   // ...remplir le vecteur (omis)...
   sort(begin(v), end(v), ordre_lexico);
   copy(begin(v), end(v), ostream_iterator<Joueur>{cout, " "});
}
```

Cela suffit. Joli, simple, efficace (presque optimal), extensible, générique. Cette stratégie, cette façon de réaliser un design logiciel a beaucoup d'avantages. En particulier, elle permet de découpler le développement des politiques de tri et des algorithmes de tri en tant que tels, de manière à permettre de réaliser le couplage des stratégies au moment opportun et en fonction des besoins du code client.

Imaginons par exemple que nous n'ayons pas accès à Si faire d'`ordre_lexico()` une fonction ou un foncteur est principalement une question d'ordre esthétique, certaines opérations doivent être des foncteurs : celle possédant des états.

`ostream_iterator` et que nous souhaitions projeter rapidement sur un flux en sortie une séquence d'éléments de même type.

Une approche possible serait celle à droite, qui repose sur un algorithme standard, `std::for_each()`, et sur une fonction générique, `Afficher()`.

Cette approche a quelques défauts.

```
#include <iostream>
#include <algorithm>
using namespace std;
template <class T>
   void afficher(const T &val) {
      cout << val << ' ';
   }
int main() {
   int tab[]{ 1, 2, 3, 4, 5 };
   for_each(
      begin(tab), end(tab), afficher<int>
   );
}
```

Le premier de ces irritants est syntaxique : il impose au code client d'indiquer explicitement le type `T` de la fonction `Afficher()` à générer, ne sachant pas lors de la mention de son nom (lors de l'invocation de `std::for_each()`) à quel type d'élément elle sera appliquée. Cet irritant se corrige en passant d'une fonction à un foncteur, puis en déplaçant le focus de la généricité du type vers l'opération.

Un autre défaut, plus important cette fois, est que notre implémentation actuelle permet d'écrire à la console et nulle part ailleurs.

En effet, l'algorithme standard `std::for_each()` applique à une opération qui ne prend qu'un seul paramètre tous les éléments d'une même séquence.

```
#include <iostream>
#include <algorithm>
using namespace std;
struct Afficher {
   template <class T>
      void operator()(const T &val) {
         cout << val << ' ';
      }
};
int main() {
   int tab[]{ 1, 2, 3, 4, 5 };
   for_each(
      begin(tab), end(tab), Afficher{}
   );
}
```

Ici, la méthode `operator()(int)` d'un `Afficher` est générée dû au type des éléments du tableau `tab`, puis chaque élément de `tab` est passé (par valeur) à cette fonction.

Pour être en mesure de spécifier un flux pour fins d'affichage, il faudrait pouvoir suppléer un deuxième paramètre à `Afficher()`, ce que `for_each()` ne permet pas.

Remarquez toutefois qu'il est probable que tous les éléments doivent être projetés sur le même flux. Sachant cela, si nous pouvions fixer une seule fois le flux en sortie à utiliser pour toute la séquence, *si le flux à utiliser était un état de l'opération*, alors nous pourrions utiliser cette information par la suite pour chaque affichage.

Il se trouve qu'un foncteur, étant un objet, a des états et possède au moins un constructeur :

- nous pouvons utiliser un attribut pour entreposer une référence sur un flux;
- nous pouvons fixer la valeur de cet attribut à la construction du foncteur; et
- nous pouvons y avoir recours dans l'opérateur `()` du foncteur.

```
#include <iosfwd>
#include <iostream>
#include <algorithm>
using namespace std;
class Afficher {
   ostream &os;
public:
   Afficher(ostream &os) noexcept : os{os} {
   }
   template <class T>
      void operator()(const T &val) {
         os << val << ' ';
      }
};
int main() {
   int tab[]{ 1, 2, 3, 4, 5 };
   for_each(
      begin(tab), end(tab), Afficher{cout}
   );
}
```

Il est évidemment possible de raffiner un peu plus le modèle, par exemple en suppléant au moment de la construction un délimiteur autre que `' '` pour séparer les éléments affichés, quitte à utiliser `' '` par défaut, et d'obtenir ainsi quelque chose de fonctionnellement semblable (bien que moins raffiné) à un `ostream_iterator`.

Revenons à l'exemple du tri standard proposé plus haut, auquel il est possible d'injecter un prédicat distinct de `<` pour déterminer l'ordre des éléments comparés. Cette flexibilité n'est pas réservée aux algorithmes standards; nous pouvons y arriver par nous-mêmes.

Pour que l'illustration soit simple et mette en valeur la force des algorithmes standards, des foncteurs et de la programmation générique, imaginons un programme qui :

- remplisse un conteneur, disons avec des entiers (un simple tableau fera l'affaire);
- affiche le contenu de ce tableau;
- mélange les éléments du tableau;
- affiche le contenu du tableau mélangé;
- trie le tableau; et
- affiche le contenu du tableau trié.

Nous écrirons ce programme de manière à ce qu'il repose sur l'opérateur `<` pour réaliser l'ordonnancement des éléments, puis nous ajouterons de la généricité.

Tout d'abord, générer une séquence se représente bien à l'aide d'un foncteur du fait qu'un état (la prochaine valeur à générer) doit être tenu à jour.

Le foncteur `Sequence<T>` proposé à droite fait ce travail pour un type `T` quelconque, dans la mesure où ce type supporte la construction par copie et l'autoincrémentation. Par défaut, la séquence débutera à `0` pour les types primitifs.

Les algorithmes standards que sont `generate()` et `generate_n()` permettent d'initialiser les éléments d'une séquence à partir d'une opération (ici, si la curiosité vous pique, sachez que le plus proche parent de notre foncteur serait sans doute `iota()`). Nous les utiliserons avec notre foncteur pour initialiser le tableau.

```
template <class T>
   class Sequence {
      T cur {};
   public:
      Sequence() = default;
      Sequence(T init) : cur{init} {
      }
      T operator()() {
         return cur++;
      }
   };
```

Nous appliquerons notre foncteur au type `int` avec la valeur initiale `1` dans le but d'initialiser le tableau de `MAX` éléments avec des valeurs de `1` à `MAX` inclusivement.

Le recours à `Sequence<T>` est un exemple ici; il existe un algorithme nommé `std::iota()` dans `<numeric>` qui fait très bien le même travail.

Notre tri sera un simple tri à bulles (nous voulons montrer la généricité, en effet, et non pas réécrire `std::sort()`). Évidemment, outre sur de très petites séquences, nous ne recommanderons pas d'avoir recours à ce tri en pratique.

C'est cet algorithme de tri à bulles que nous allons adapter par la suite pour qu'il permette de prendre des critères d'ordonnancement spécifiés sur demande tout en conservant l'opérateur `<` comme critère par défaut.

On préférera traditionnellement utiliser `<` du fait que c'est l'opérateur à partir duquel les gens, traditionnellement, expriment leurs algorithmes.

Étant donné ces quelques outils, le programme sera simple est explicite :

- au début, les valeurs requises seront générées dans le tableau, la séquence commençant à `1`. Ceci met en application la combinaison de notre foncteur générique `Sequence` et de l'algorithme standard `std::generate()` appliqué à un simple tableau;
- ensuite, nous utiliserons l'algorithme standard `shuffle()` pour « brasser » notre tableau pour en mélanger les éléments; et
- nous appliquerons notre tri à bulles générique sur le tableau pour le remettre en ordre.

Ces opérations seront suivies d'affichages pour faciliter le suivi des opérations.

Réflexion 02.6 : nous avons utilisé la généricité sur la base du type pour notre foncteur `Sequence`. Aurait-il été possible d'utiliser la généricité sur une base méthode? Réponse dans ***Réflexion 02.6 : choix de généricité***.

Voyons maintenant comment ajouter de la généricité à `tri_bulles`.

```
#include <utility> // std::swap()
#include <algorithm>
// ...using...
template <class It>
   void tri_bulles(It debut, It fin) {
      for (auto i = debut; i != fin - 1; ++i)
         for (auto j = next(i); j != fin; ++j)
            if (*j < *i)
               swap(*i, *j);
   }
```

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <random>
// ... Sequence, tri_bulles() ...
int main() {
   // divers using...
   const int MAX = 10;
   int tab[MAX];
   // générer des valeurs de 1 à MAX
   generate(
      begin(tab), end(tab), Sequence<int>{1}
   );
   for(auto n : tab) cout << n << ' ';
   cout << endl;
   // mélanger le tout
   random_device rd;
   mt19937 prng { rd() };
   random_shuffle(begin(tab), end(tab), prng);
   for(auto n : tab) cout << n << ' ';
   cout << endl;
   // remettre en ordre
   tri_bulles(begin(tab), end(tab));
   for(auto n : tab) cout << n << ' ';
   cout << endl;
}
```

L'idée est d'ajouter une version du *template* nommé `tri_bulles()` paramétrique sur la base de deux types génériques plutôt que sur la base d'un seul.

Ce nouveau type sera celui de l'opération qui servira à déterminer dans quel ordre placer deux éléments du type à trier. Notez que nous utilisons une syntaxe très large, considérant `Op` comme un type quelconque, plutôt qu'une syntaxe pointue qui imposerait à `Op` d'être une fonction booléenne prenant deux paramètres de même type. Ceci ouvrira la porte à l'utilisation de fonctions comme à celle de foncteurs.

Le compilateur déduira les types impliqués et générera le code approprié, dans la mesure où l'opération porte une signature compatible à l'usage qui en est fait dans `tri_bulles()`.

```
#include <utility> // std::swap()
#include <algorithm>
// ...using...
template <class It>
   void tri_bulles(It debut, It fin) {
      for (auto i = debut; i != fin - 1; ++i)
         for (auto j = next(i); j != fin; ++j)
            if (*j < *i)
               swap(*i, *j);
   }
template <class It, class Op>
   void tri_bulles
      (It debut, It fin, Op oper) {
      for (auto i = debut; i != fin - 1; ++i)
         for (auto j = next(i); j != fin; ++j)
            if (oper(*j, *i))
               swap(*i, *j);
   }
```

On peut alors reprendre l'exemple d'utilisation de `tri_bulles()` vu précédemment en utilisant à la fois la version à deux paramètres et la version à trois paramètres de notre petit algorithme de tri :

- la première invocation utilise la version à deux paramètres;
- la seconde invocation utilise la version à trois paramètres mais avec un critère d'ordonnancement différent (elle place les éléments en ordre décroissant plutôt qu'en ordre croissant); et
- la troisième invocation réalise le même travail que la première mais en explicitant son critère d'ordonnancement.

Le foncteur `std::less` compare deux éléments de même type à l'aide de l'opérateur `<` alors que le foncteur `std::greater` procède à l'aide de l'opérateur `>`.

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <functional>
// ... Sequence, tri_bulles() ...
int main() {
   // ...divers using...
   const int MAX = 10;
   int tab[MAX];
   // ...générer, mélanger...
   // remettre en ordre
   tri_bulles(begin(tab), end(tab));
   copy(
      begin(tab), end(tab),
      ostream_iterator<int>{cout," "}
   );
   cout << endl;
   tri_bulles(
      begin(tab), end(tab), greater<int>{}
   );
   copy(
      begin(tab), end(tab),
      ostream_iterator<int>{cout," "}
   );
   cout << endl;
   tri_bulles(
      begin(tab), end(tab), less<int>{}
   );
   copy(
      begin(tab), end(tab),
      ostream_iterator<int>{cout," "}
   );
   cout << endl;
}
```

Fonctions génératrices

Vous aurez peut-être remarqué un côté irritant dans la syntaxe proposée à droite. En effet, du fait que le littéral `1` est de type `int`, il paraît agaçant de devoir spécifier explicitement `Sequence<int>` comme type du foncteur.

On souhaiterait en effet que le compilateur puisse déduire le `int` du type de la valeur passée à la construction, or ce n'est pas possible.

```
// ...
generate(
   begin(tab), end(tab), Sequence<int>{1}
);
// ...
```

Par contre, il est possible de pallier cet irritant avec une technique sur laquelle nous reviendrons plus en détail dans [POOv03], soit celle des **fonctions génératrices**. D'ici là, pour vous simplifier l'existence, voici la recette :

- le problème tient du fait que nous appelions un constructeur, donc que nous devions expliciter le type complet de l'objet construit (`Sequence<int>`);
- nous pouvons toutefois écrire une fonction générique retournant une instance de ce type. Le compilateur découvre les types des paramètres et génère les cas particuliers de fonctions génériques pour nous;
- ce faisait, le compilateur écrit le code à notre place!

```
template <class T>
   class SequenceImpl {
      // ... bla bla ...
   };
template <class T>
   SequenceImpl<T> Sequence(T init) {
      return SequenceImpl<T>{init};
   }
// ...
generate(
   begin(tab), end(tab), Sequence(1)
);
// ...
```

- avec un compilateur récent, l'écriture de la fonction génératrice `Sequence()` devient si simple qu'il devient difficile de s'en passer ☺

```
// ...
template <class T>
   SequenceImpl<T> Sequence(T init) {
      return { init };
   }
// ...
```

Autre alternative idiomatique de C++, du moins depuis C++ `11` : implémenter `Sequence` sous la forme d'une expression lambda. Nous y reviendrons (section ***Les expressions Lambda (ou expressions λ)***).

Délégués

Certains langages, dont C# et Delphi[70], moussent l'usage d'une entité cousine du foncteur et des interfaces : le ***délégué***, qui permet une forme de polymorphisme sur la base des signatures.

⇒ Un **délégué** (***delegate***) permet, en langage C#, de représenter une méthode comme l'équivalent d'une fonction globale de même signature.

L'utilité d'un délégué est de permettre une généricité analogue au polymorphisme, les délégués de même signature étant interchangeables entre eux. Un délégué est un type en C#, ce qui fait qu'on peut passer un délégué en paramètre à un sous-programme ou utiliser un délégué comme attribut d'un objet.

Opérationnellement, un délégué est un analogue proche du pointeur de fonction en langage C ou C++, mais avec une syntaxe beaucoup moins lourde (la syntaxe des pointeurs de fonctions en C et en C++ est lourde, c'est le moins qu'on puisse dire).

Comme avec des pointeurs de fonctions, et un peu comme avec des *templates* en C++, les délégués sont polymorphiques sur le plan de leur signature plutôt que sur la base d'un ancêtre commun.

Les pointeurs de fonction typiques du langage C ne s'appliquent pas directement aux méthodes d'instance dans un langage `OO`, du fait que chaque méthode d'instance comporte toujours un paramètre caché, `this`, qui réfère à l'instance propriétaire de la méthode. Cette nuance ajoute à l'intérêt envers les délégués en C#, langage où (comme en Java) les fonctions globales n'existent pas.

Un délégué n'est pas une macro. Il s'agit d'une entité typée, bénéficiant du même support de la part du compilateur que les méthodes ou les fonctions globales. Syntaxiquement, les délégués se combinent et se composent comme des fonctions normales. Les délégués sont applicables aux méthodes de classe comme aux méthodes d'instance, quoique la distinction avec les méthodes polymorphiques habituelles soit plus marquante dans le cas de méthodes d'instance.

C# offre un support spécial aux délégués. Ces entités sont des objets dérivés de la classe `Delegate`. Ils ont donc des méthodes, par exemple pour retirer un délégué d'une composition ou pour vérifier si un délégué réfère à une méthode d'instance ou à une méthode de classe.

[70] La conception des langages Delphi et C# a été faite en partie par la même personne, ***Anders Hejlsberg***.

Exemple de délégué en C#

Vous pourrez trouver des exemples d'utilisation à profusion dans Internet. Celui-ci, tout simple, se veut seulement une illustration du concept.

```
namespace ExempleDeDélégués
{
   // Déclaration d'un type délégué. Remarquez le mot clé delegate
   public delegate int TiDélégué(int val); // déclaration d'un délégué
   class DemoDélégué
   {
      private TiDélégué exemple_;
      // deux méthodes respectant la signature de notre délégué
      private int OperationUn(int i)
      {
         return -i;
      }
      private int OperationDeux(int i)
      {
         return 2 * i;
      }
      public void Démo()
      {
         Console.WriteLine("Exemples avec des délégués");
         // Instanciation d'un délégué. Remarquez le paramètre passé à la construction.
         exemple_ = new TiDélégué(OperationUn);
         // Utilisation du délégué (Resultat vaudra -8)
         int résultat = exemple_(8);
         // Exemple de combinaison de délégués. Passe par la méthode de classe Combine()
         // de la classe Delegate. Le transtypage est nécessaire du fait que Combine()
         // retourne un object, dont dérivent de toutes les classes .NET
         TiDélégué d0 = new TiDélégué(OperationUn);
         TiDelegue d1 = new TiDelegue(OperationDeux);
         TiDelegue PetitComposé = (TiDélégué)Delegate.Combine(d0, d1);
         résultat = PetitComposé(3); // résultat vaudra -6
      }
      // etc.
   }
   // etc.
}
```

Les combinaisons de délégués forment une *collection*, semblable aux conteneurs `std::map` et `std::vector` de C++. Faute de généricité, ces combinaisons forcent le code client à faire des conversions explicites de types à l'occasion.

Depuis C# `2.0`, la généricité est supportée en partie par le langage, ce qui implique qu'il puisse y avoir eu des changements ici.

Délégués en C++ – `std::function`

C++ ne cache que très peu de détails techniques aux programmeurs; pour rédiger un délégué, il faut prendre conscience des nuances entre fonctions globales, foncteurs, méthodes de classe et méthodes d'instance.

Il est possible, en C++, d'exprimer des délégués, mais les manœuvres de programmation requises ne sont pas banales. Heureusement, depuis C++ `11`, l'en-tête `<functional>` propose la classe `std::function` qui joue précisément ce rôle.

Un exemple de code donnant essentiellement les mêmes résultats que celui proposé pour C# un peu plus haut serait :

```
#include <iostream>
#include <functional>
using namespace std;
int operation_un(int i) {
   return -i;
}
int operation_deux(int i) {
   return 2 * i;
}
struct X {
   int val;
   X(int val) noxecept : val{val} {
   }
   int f(int i) const noxecept {
      return i + val;
   }
};
int main() {
   function<int(int)> ti_delegue = operation_un;
   cout << ti_delegue(3) << endl;
   ti_delegue = operation_deux;
   cout << ti_delegue(3) << endl;
   X x{4};
   function <int(const X&, int)> aut_delegue = &X::f;
   cout << aut_delegue(x,3) << endl;
}
```

Remarquez la syntaxe :

- tout d'abord, l'écriture `int(int)` est la signature d'un *Callable Object*, donc de quelque chose pouvant être appelé en passant en paramètre un `int` et qui retournera un `int`;
- un `function<int(int)>` encapsule toute entité respectant la signature d'appel `int(int)`, incluant fonctions, foncteurs et lambdas (voir ***Les expressions Lambda (ou expressions λ)***);
- dans le cas où nous utilisons une méthode d'instance d'un `X`, il faut que le `function` sache de quel `X` nous parlons (tout `X` étant susceptible d'avoir un attribut d'instance val distinct de celui des autres instances de la même classe). Ceci explique la signature particulière dans ce cas.

Dans la plupart des cas, en C++, on préférera les lambdas aux `function`, mais il existe quelques cas pour lesquels un `function` est nécessaire. Nous y reviendrons.

Foncteurs et continuations

Une fonction possédant à la fois sa propre pile d'exécution et ses propres états, sujets à changements, est une fonction permettant ce qu'on nomme des **continuations**.

Une opération supportant les continuations est une opération munie d'une forme de mémoire, qui peut être arrêtée en cours d'exécution et qui peut reprendre son exécution suite au point d'arrêt lors d'une invocation subséquente.

Le programme C# proposé à droite implémente, à l'aide d'une continuation (reposant sur le mot clé `yield` de ce langage), la méthode d'instance `Prochain()` qui retourne, à chaque invocation, le prochain entier naturel (jusqu'à concurrence du plus grand `System.Integer` supporté).

Le langage impose beaucoup de contraintes au mot clé `yield` :

- la méthode doit retourner un itérateur du langage (un `IEnumerable`);
- le mot `yield` doit être suivi d'un `break` ou d'un `return`, selon le type de la méthode (`void` ou autre);
- chaque invocation subséquente de la méthode reprendra là où l'opération `yield` précédente aura été exécutée.

```
using System.Collections.Generic;
namespace zz
{
   class Entier
   {
      public int Valeur { get; set; }
      public Entier()
      {
         Valeur = 0;
      }
      public IEnumerable<int> Prochain()
      {
         while (true)
            yield return ++Valeur;
      }
   }
   class Program
   {
      public static void Main(string[] args)
      {
         Entier e = new Entier();
         foreach (int i in e.Prochain())
            System.Console.WriteLine(i);
      }
   }
}
```

En langage C, on pourrait partiellement simuler une continuation en utilisant une variable locale qualifiée `static` dans une fonction, mais l'effet serait limité du fait qu'une seule instance de la fonction pourrait être utilisée en tout temps (une variable `static` dans une fonction est, au fond, une variable globale à portée locale). C'est pourquoi les continuations sont surtout associées aux langages qui permettent le maintien à jour d'états subjectifs; aujourd'hui, ces langages sont principalement des langages `OO`.

En C++, on ne trouve pas de mots clés ou de mécanique spéciale pour encadrer les continuations. En effet, les foncteurs permettent les continuations (et même plus!) du fait que les objets sont en mesure de tenir à jour leurs propres états.

En C++, une classe `Entier` comme celle proposée dans le cadre de l'exemple C# plus haut s'écrirait sous la forme proposée à droite. Aucun mot clé spécial n'y est nécessaire.

```
class Entier {
   int val {};
public:
   Entier() = defaultl
   int operator()() {
      return ++val;
   }
};
#include <iostream>
int main() {
   using namespace std;
   Entier e;
   for (;;)
      cout << e() << endl;
}
```

Le terme continuation [hdCont] prend un sens technique avec C++ `17`, où elles feront leur entrée dans certaines pratiques de multiprogrammation associées avec des futures [hdFut], mais ce sujet, si intéressant soit-il, n'est pas au menu du présent document.

Fonctions surchargées sur la base du type retourné[71]

En C++, il est aussi possible d'utiliser des opérateurs de conversion de types pour mettre en place une mécanique de surcharge de fonctions sur la base de leur type plutôt que sur la base de leur signature. Voici comment :

- imaginons la fonction bidon `Trois()` dont le rôle est de retourner la valeur `3` exprimée sous un type ou l'autre, selon les besoins;
- affecter `Trois()` à une `std::string` devra lui déposer la valeur `"Trois"`. Affecter `Trois()` à un `int` devra y déposer `3`. Enfin, affecter `Trois()` à un `float` devra y déposer `3.0f`;
- avec des fonctions, ce problème est insoluble en C++, car l'invocation d'une fonction se fait avant l'utilisation de son résultat, ce qui empêche de distinguer deux fonctions sur la base de leur type.

```
#include <string>
using std::string;
struct Trois {
   operator string() const {
      return "Trois";
   }
   operator int() const noexcept {
      return 3;
   }
   operator float() const noexcept {
      return 3.0f;
    }
};
int main() {
   string s = Trois();
   int i = Trois();
   float f = Trois();
}
```

Avec une classe, ce problème trouve une solution puisqu'il est possible d'exposer des opérateurs de conversion dans divers types. Le constructeur permet de passer des paramètres à la fonction, et les opérateurs de conversion implicite font le reste du travail.

L'exemple ci-dessus n'utilise que des constructeurs par défaut. L'écriture `Trois()` instancie un `Trois` par défaut et anonyme. La syntaxe est identique à celle d'un appel de fonction. Cette technique fonctionnerait aussi si nous écrivions `Trois{}`, mais cela briserait quelque peu l'illusion d'avoir une surcharge sur la base du type de retour.

En effet, écrire `int i = Trois();` instancie un `Trois` par défaut et temporaire. Cette instance est alors utilisée à droite d'une affectation, ce qui amène le moteur d'inférence de types à examiner les diverses conversions possibles pour permettre l'opération d'affectation d'un `Trois` à un `int`. La conversion par `operator int() const` est précisément ce dont le moteur d'inférence de types a besoin… et voilà!

[71] Cette idée n'est pas de moi. Je l'ai prise dans un article du site *The Code Project* à l'adresse http://www.codeproject.com/useritems/returnoverload.asp

Exercices – Série 02

EX00 – Exprimez un foncteur permettant de réaliser la somme des carrés de valeurs dans une séquence l'aide de l'algorithme standard `accumulate()`.

EX01 – Écrivez un programme qui déclare un tableau de `double` et l'initialise avec des valeurs pseudo-aléatoires situées entre `0` et `1` inclusivement. Utilisez l'algorithme standard `generate()` et un foncteur de votre cru pour y arriver. Notez que la fonction `rand()` de `<cstdlib>` retourne des entiers entre `0` et `RAND_MAX` inclusivement.

EX02 – Reprenez le programme demandé à `EX01`. À l'aide des algorithmes `sort()` et `partition()` et d'un foncteur de votre cru, trouvez la position du plus grand élément du tableau qui soit inférieur à `0.4` et affichez à la fois sa valeur et sa position.

Les lieurs

Un **lieur** (traduction libre de *Binder*) est un objet ayant pour rôle de faciliter l'application d'une fonction à une séquence d'objets à l'aide d'un algorithme standard. On pourrait dire qu'un lieur a pour rôle de construire un foncteur permettant de transformer une fonction en foncteur (ouf!), mais à l'intérieur de certaines contraintes.

Cette section présente d'abord le principe des lieurs de même que quelques exemples à partir des outils de C++ 03. Cependant, notez tout de suite qu'avec C++ 11, la plupart d'entre eux ont été remplacés par un lieur général qui sera présenté un peu plus loin dans la section nommée ***Le lieur général `std::bind()`***, lui-même rendu quelque peu désuet par l'avènement des expressions λ (voir ***Les expressions Lambda (ou expressions λ)***). Ainsi, prenez cette section comme un bref voyage dans le passé.

Pour illustrer le rôle des lieurs, imaginons qu'on souhaite vérifier lesquels des objets dans un conteneur donné sont négatifs. Présumons aussi qu'on ait accès à une fonction déjà écrite et de signature `plus_petit_que(a,b)` qui retourne vrai si `a` est plus petit que `b`.

Des versions standards (espace nommé `std`) de la plupart des opérateurs relationnels et logiques sous forme de fonctions existent déjà dans `<functional>` : il existe des foncteurs tels que `logical_and`, `greater_equal`, `less`, `unary_minus`, *etc.*

Étant donné que *être négatif* signifie *être plus petit que zéro*, et présumant (pour simplifier le propos) que le conteneur soit un vecteur standard de `int`, on pourrait rédiger un foncteur `EstPlusPetitQue` de la manière suivante :

```
template <class T>
   bool plus_petit_que(const T &a, const T &b) {
      return a < b;
   }
template <class T>
   class EstPlusPetitQue {
      T seuil;
   public:
      EstPlusPetitQue(T seuil) : seuil{seuil} {
      }
      bool operator()(const T &val) const {
         return plus_petit_que(val, seuil);
      }
   };
```

Un exemple de programme utilisant ce foncteur serait :

```
// ... inclusions...
int main() {
   // ... using ...
   vector<int> v;
   for (int val; cin >> val; )
      v.push_back(val);
   auto it = find_if(begin(v), end(v), EstPlusPetitQue<int>{0});
   if (it != end(v))
      cout << *it;
}
```

Ce programme trouve le premier élément du vecteur standard satisfaisant le critère *est plus petit que zéro* et, si un tel élément existe, en affiche la valeur.

La création d'un foncteur simple pour enrober l'appel à une fonction à deux paramètres dont on fixe *a priori* le deuxième paramètre est une tactique répandue. Il en va d'ailleurs de même pour la conception d'un foncteur à deux paramètres dont on fixe *a priori* le premier paramètre.

Ces tactiques sont si répandues, en fait, qu'on a conçu autour d'elles une technique de programmation complète. L'idée va comme suit: un lieur est un foncteur capable de lier une fonction et l'un de ses paramètres pour devenir utilisable en tant que foncteur applicable à l'autre paramètre. Présenté grossièrement, *un lieur transforme une fonction en foncteur*.

Les deux lieurs standards les plus répandus avant C++ 11 étaient **binder1st()**, qui lie le premier paramètre d'une fonction à deux paramètres, et **binder2nd()**, qui lie le second paramètre d'une fonction à deux paramètres. Dans les deux cas, on peut créer un foncteur à l'aide d'une fonction de fabrication (`bind1st()` et `bind2nd()`, respectivement).

Un exemple d'utilisation du lieur `binder2nd()` reposant sur une invocation de la fabrique `bind2nd()` serait celui proposé à droite.

```
template <class T>
   bool plus_petit_que(const T &a, const T &b) {
      return a < b;
   }
// ... exemple d'utilisation
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>
int main() {
   // ...using...
   vector<int> v;
   for (int val; cin >> val; )
      v.push_back(val);
   auto it = find_if(
      begin(v), end(v),
      bind2nd(ptr_fun(plus_petit_que<int>),0)
   );
   if (it != end(v))
      cout << *it;
}
```

Ce qu'il faut lire ici est que :

- la fonction `plus_petit_que()` prend deux paramètres;
- la fonction **ptr_fun()**, qui crée un foncteur en cachette, normalise l'obtention d'un pointeur sur cette fonction et clarifie à la fois le type de ses paramètres et le type de la fonction en tant que telle (ce qui allège de beaucoup la syntaxe);
- l'expression **bind2nd(ptr_fun(plus_petit_que),0)** fait en sorte de lier la valeur `0` au 2e paramètre de `plus_petit_que()`;
- le résultat de l'invocation de `bind2nd()` est un foncteur à un seul paramètre, qui passera à chaque invocation ce paramètre comme premier paramètre à `plus_petit_que()` tout en passant `0` comme second paramètre à cet appel. En fait, le foncteur généré par `bind2nd()` est précisément ce que nous aurions pu écrire manuellement (voir `EstPlusPetitQue`, plus haut).

Si notre souhait avait plutôt été de repérer le premier élément de la séquence qui soit strictement plus grand que `0`, alors nous aurions simplement pu remplacer l'expression `bind2nd(ptr_fun(plus_petit_que),0)` par une autre expression, très proche, soit **bind1st(ptr_fun(plus_petit_que),0)**… Pensez-y!

On aurait aussi pu transformer un foncteur à deux paramètres en foncteur à un seul paramètre à l'aide de la même stratégie.

Idéalement, on dérivera (à coût zéro) le foncteur de la classe standard `binary_function`, pour alléger la syntaxe lors de l'utilisation d'un lieur.

La classe `binary_function` est un *template* à trois paramètres:

- le type du premier opérande, nommé `first_argument_type`;
- le type du second opérande, nommé `second_argument_type`; et
- le type de la fonction, nommé `result_type`.

Une classe comme `binary_function` n'offre aucune fonctionnalité particulière, mais documente (par des `typedef`) à la fois les types des paramètres du foncteur et le type de son opérateur `()`. C'est cet enrobage qu'offre `ptr_fun` pour les fonctions.

L'exemple qui suit illustre comment il serait possible d'utiliser un lieur pour une version de `EstPlusPetitQue` définie sous cette forme.

```
#include <functional>
using namespace std;
template <class T>
   struct EstPlusPetitQue : binary_function<T, T, bool> {
      result_type operator()(first_argument_type a, second_argument_type b) const {
         return a < b;
      }
   };
// ... using et inclusions omis par souci d'économie
int main() {
   vector<int> v;
   for (int val; cin >> val; )
      v.push_back(val);
   auto it = find_if(begin(v), end(v),bind2nd(EstPlusPetitQue<int>{},0));
   if (it != end(v))
      cout << *it;
}
```

À titre de rappel, les lieurs traditionnels présentés ci-dessus, de même que les classes comme `unary_function` ou `binary_function`, sont des outils dépréciés qui n'ont été présentés ici qu'à titre de contexte historique.

Le lieur général `std::bind()`

Depuis C++ 11, et dans la foulée des travaux faits sur Boost au fil des ans, il existe maintenant dans `<functional>` un lieur général qui remplace tous les autres présentés ci-dessus.

Nommé simplement `std::bind()`, ce lieur permet de prendre une opération f et retourner un objet qui, lorsqu'il sera appelé, de fixer la valeur de certains de ses paramètres. Des marqueurs de position sont utilisés pour indiquer quels sont les paramètres visés; ces marqueurs sont logés dans l'espace nommé `std::placeholders` et se nomment `_1`, `_2`, `_3`, *etc.*

À titre d'exemple, étant donné le code présenté à droite :

- l'objet `f` est un foncteur encapsulant la fonction `fct(x,y)` qui, lorsqu'on l'appellera, effectuera un appel à `fct` qui associera la valeur `0` au paramètre `x` et son propre premier paramètre au paramètre `y`; alors que
- l'objet `g` est un foncteur encapsulant la fonction `fct(x,y)` qui, lorsqu'on l'appellera, effectuera un appel à `fct` qui associera son propre paramètre au paramètre `x` et la valeur `0` au paramètre `y`.

```
#include <functional>
int fct(int x, int y);
int main() {
   using std::bind;
   using namespace std::placeholders;
   auto f = bind(fct, 0, _1);
   f(3); // équivaut à f(0, 3);
   auto g = find(fct, _1, 0);
   g(3); // équivaut à f(3, 0);
}
```

Un exemple plus complet suit :

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <random>
bool plus_petit_que(int a, int b) {
   return a < b;
}
int main() {
   using namespace std;
   using namespace std::placeholders;
   random_device rd;
   mt19937 prng { rd() };
   int vals[]{ 2, 3, 5, 7, 11 };
   shuffle(begin(vals), end(vals), prng);
   cout << "Valeur au debut: " << vals[0] << endl;
   auto it = find_if(begin(vals), end(vals), bind(plus_petit_que, vals[0], _1));
   if (it != end(vals))
      cout << "Premier element plus grand que " << vals[0] << ": "
           << *it << " a la position " << distance(begin(vals), it) << endl;
   else
      cout << "L'element " << vals[0] << " est le plus grand de la sequence" << endl;
}
```

Exercices – Série 03

EX00 – Écrivez `ptr_fun` de manière à couvrir les fonctions à un seul paramètre. À quelle solution en arrivez-vous? Êtes-vous en mesure de couvrir aussi les fonctions à deux paramètres? Êtes-vous aussi en mesure de couvrir les fonctions de type `void`? (notez que ce dernier cas est plus difficile).

EX01 – Reprenez le foncteur `EstPlusPetitQue` du début de cette section et rendez-le plus agréable d'utilisation avec une fonction génératrice.

EX02 – **Plus difficile** : écrivez votre propre version de `std::bind()` nommée `lier()` offrant un comportement semblable à celui de `std::bind()` et permettant de lier l'un de deux paramètres, sans plus, sans avoir recours aux lieurs standards.

Les expressions Lambda (ou expressions λ)

Les expressions λ font partie des nouveaux outils proposés par C++ 11, puis raffinés successivement par C++ 14 puis par C++ 17. Ils allègent en grande partie la rédaction de foncteurs pour utilisation locale. Nous présentons donc tout d'abord ces outils, que vous apprécierez sans doute beaucoup.

Avis aux curieuses et aux curieux : sous C++ 03, **Jaakko Järvi** a développé une bibliothèque *très* sophistiquée, *Boost Lambda*[72], dont vous pouvez vous inspirer pour faire travailler vos méninges à un niveau d'abstraction des plus divertissants.

Le lambda-calcul[73], souvent écrit λ-calcul, fut développé par **Alonzo Church**, un pionnier de l'informatique théorique avec **Alan Turing**, **John von Neumann** et une poignée d'autres qui mériteraient aussi d'être nommés ici. Il s'agit d'une idée à la fois d'une grande simplicité et d'une grande puissance. Avec ce calcul, on peut définir les concepts de calculabilité et de récursivité.

Certains langages, en particulier des langages fonctionnels comme LISP et ses dialectes, peuvent être considérés comme étant des applications concrètes de l'idée de λ-calcul.

La généricité du λ-calcul repose en partie sur des règles de substitution qui permettent d'exprimer une application de fonction comme étant équivalente à une autre expression.

La spécification de la version 3.0 du langage C# implémente les λ-expressions pour introduire une notation compacte de substitutions susceptible de remplacer des méthodes anonymes.

Quelques exemples en sont visibles à droite (l'opérateur de substitution y est noté =>).

```
(int x) => x + 1
(x, y) => x * y
Func<int,double> f2 = x => x * 3.14159;
```

Le type `Function<int,double>` est l'équivalent C# le plus direct du `std::function` de C++. Les types y sont dans l'ordre le 1er paramètre, le 2e paramètre, …, le type de retour. Ici, `f2` aurait simplement pu être déclaré `var`.

Dans le langage C# version 3.0, l'avènement des λ-expressions semble fortement lié à l'insertion de la technologie LINQ, dont nous reparlerons brièvement dans la section sur la ***persistance des objets*** [POOv03].

En effet, les λ-expressions permettent de simplifier certaines opérations de sélection d'éléments dans des clauses SQL générant des collections, comme dans l'exemple à droite.

L'expression C# 3.0 suivante :

```
from c in clients
where c.Ville == "Montréal"
select c.Nom
```

sera transformée en une autre expression, plus familière aux habitués de l'approche OO et exploitant le caractère synthétique des λ-expressions :

```
clients.
Where(c => c.Ville == "Montréal").
Select(c => c.Nom)
```

[72] http://www.boost.org/doc/libs/1_39_0/doc/html/lambda.html

[73] Je vous invite à consulter http://en.wikipedia.org/wiki/Lambda_calculus pour en savoir plus sur ce passionnant sujet.

Avec l'avènement d'abstractions aussi fortes que la programmation générique et le polymorphisme, toutefois, il devient possible d'exprimer des λ-expressions à même les outils du langage.

Imaginons par exemple qu'on veuille écrire un programme C++ s'appuyant sur les algorithmes standards et cherchant, par exemple, à trouver le premier élément d'une séquence d'entiers (un tableau, disons).

La solution classique avec une fonction prédicat serait :

```
bool plus_petit_que_4(int val) { // bof... Voir la section sur les lieurs, plus haut
   return val < 4;
}
#include <iostream>
#include <iterator>
#include <algorithm>
#include <random>
int main() {
   // ...using...
   int tab[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   // initialiser le générateur de nombres pseudo aléatoires
   random_device rd;
   mt19937 prng { rd() };
   // mélanger les éléments du tableau
   shuffle(begin(tab), end(tab), prng);
   // afficher le tableau mélangé
   copy(begin(tab), end(tab), ostream_iterator<int>{cout, " "});
   // trouver et afficher le premier élément respectant la contrainte du prédicat
   cout << '\n'  << *find_if(begin(tab), end(tab), PlusPetitQue4) << endl;
}
```

Cette solution n'est pas très flexible, on le remarquera; une version avec un foncteur serait un sérieux pas en avant :

```
class PlusPetitQue {
   int seuil;
public:
   PlusPetitQue(int seuil) noexcept : seuil{seuil} {
   }
   bool operator()(int n) const noexcept {
      return n < seuil;
   }
};
// ... inclusions et using ...
int main() {
   int tab[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   random_device rd;
   mt19937 prng { rd() };
   shuffle(begin(tab), end(tab), prng);
   copy(begin(tab), end(tab), ostream_iterator<int>(cout, " "));
   // trouver et afficher le premier élément respectant la contrainte du prédicat
   cout << endl << *find_if (begin(tab), end(tab), PlusPetitQue{4}) << endl;
}
```

En effet, cette version a la qualité de permettre de comparer les éléments d'une séquence avec une valeur arbitraire plutôt qu'avec une valeur connue à la compilation. On profite de la capacité qu'a le foncteur (un objet) de conserver un état pour la durée de son existence en lui confiant la borne avec laquelle comparer chaque élément de la séquence. Un gain net de généricité.

Cela dit, cette version peut encore gagner en généricité, au prix d'un petit effort supplémentaire, d'un peu d'imagination et en combinant programmation générique et polymorphisme.

Les λ de C++ 11

Avec C++ 11, les λ font leur entrée dans le langage en tant qu'outils à part entière[74]. Une λ est une expression compacte à partir de laquelle le compilateur générera un foncteur.

Par exemple, supposons que nous réduisions le problème de la page précédente à cette fonction :

```
template <class T, class It>
   void compter_sous_seuil(It debut, It fin, const T &seuil) {
      int n = 0;
      for(; debut != fin; ++debut)
         if (*debut < seuil)
            ++n;
      return n;
   }
```

Pour un vecteur d'entiers `v`, afficher à la sortie standard les éléments de valeur inférieure à `4` s'exprimerait alors :

```
cout << compter_sous_seuil(begin(v), end(v), 4);
```

Une amélioration possible de l'implémentation de la fonction serait de remplacer la répétitive par un appel à `count_if()`. Cela serait possible en passant par un foncteur intermédiaire. Supposons que ce foncteur soit local à la fonction; nous aurions alors :

```
template <class T, class It>
   int compter_sous_seuil(It debut, It fin, const T &seuil) {
      struct PlusPetitQue {
         T seuil;
         PlusPetitQue(const T &seuil) : seuil{seuil} {
         }
         bool operator()(const T &val) const {
            return val < seuil;
         }
      };
      return count_if(debut, fin, PlutPetitQue{seuil});
   }
```

Cela fonctionnerait, et serait sans doute un peu plus rapide que la version originale de la fonction, mais la quantité de code à rédiger s'est accrue de manière significative. Ceci peut décourager les gens d'utiliser les algorithmes standards et les foncteurs, malgré leurs nombreux avantages.

[74] Si vous avez de l'expérience avec d'autres langages, ne confondez pas λ et fermeture; la nuance entre les deux existe bel et bien, si mince soit-elle : http://scottmeyers.blogspot.ca/2013/05/lambdas-vs-closures.html

C'est ici que les λ entrent en scène. Examinez le code suivant :

```
template <class T, class It>
   int compter_sous_seuil(It debut, It fin, const T &seuil) {
      return count_if(debut, fin, [&](const T &val) {
         return val < seuil;
      });
   }
```

Plus concis, vous en conviendrez. Le code qu'on y retrouve va à l'essentiel, et omet presque tous les détails auxquels une implémentation manuelle d'un foncteur aurait dû porter attention. Pourtant, il s'agit de code précisément équivalent en termes de performance à la version reposant sur un foncteur.

Anatomie d'une λ

Une λ a la structure suivante :

- une liste de captures;
- une liste de paramètres;
- un type de retour (optionnel); et
- un bloc de code, correspondant à l'opérateur `()` du foncteur qui sera généré.

Examinons chacun de ces éléments constitutifs, en utilisant à titre d'exemple le λ nommé f. Remarquez le recours au type `auto` de C++ 11, qui déduit le type d'une variable du type de l'expression servant à l'initialiser. Les λ sont des types presque anonymes : nous ne voulons pas connaître le nom que le compilateur aura utilisé pour chacune d'elles.

La **liste de captures** est ce qui apparaît entre crochets au début de la λ. Cette section permet de spécifier les éléments locaux à la fonction qui seront capturés par le foncteur une fois celui-ci généré par le compilateur.

```
auto f = [&](const T &val) {
   return val < seuil;
};
```

En détail :

- lorsque `[]` est spécifié (paire de crochets vide), cela signifie qu'aucune variable de l'environnement ne sera capturée. Cela correspond à un foncteur sans attributs et muni strictement d'un constructeur par défaut banal. Notez qu'une telle λ est implicitement convertible en pointeur de fonction, ce qui permet d'utiliser des expressions λ même pour interfacer avec une bibliothèque prévue pour le langage C;
- la mention **`[&]`** signifie capturer par référence tous les éléments pertinents de l'environnement;
- la mention **`[=]`** signifie capturer par copie tous les éléments pertinents de l'environnement;
- la mention **`[x]`** signifie ne capturer que `x` et le faire par valeur;
- la mention **`[&y]`** signifie ne capturer que `y` et le faire par référence;
- une mention telle que **`[&x,y]`** signifie capturer `x` par référence et `y` par copie.

Dans notre exemple, la variable seuil est capturée par référence lors de la construction de la λ et peut donc être utilisée dans la définition de son opérateur `()`. Une capture par copie aurait aussi été correcte mais aurait pu être coûteuse si `T::T(const T&)` s'était avéré coûteux.

La **liste de paramètres** correspond aux paramètres passés à l'opérateur `()` du foncteur une fois celui-ci généré. Il peut évidemment y en avoir autant que pour n'importe quelle fonction, dans la mesure où cela correspond à l'usage qui sera fait de la λ par la suite.

```
auto f = [&](const T &val) {
   return val < seuil;
};
```

Il est possible de spécifier un **type de retour** à l'opérateur `()` qui sera généré, par exemple pour répondre à des besoins particuliers. Dans ce cas-ci, le type sera sans doute `bool` mais nous pourrions souhaiter retourner une valeur de type `int`.

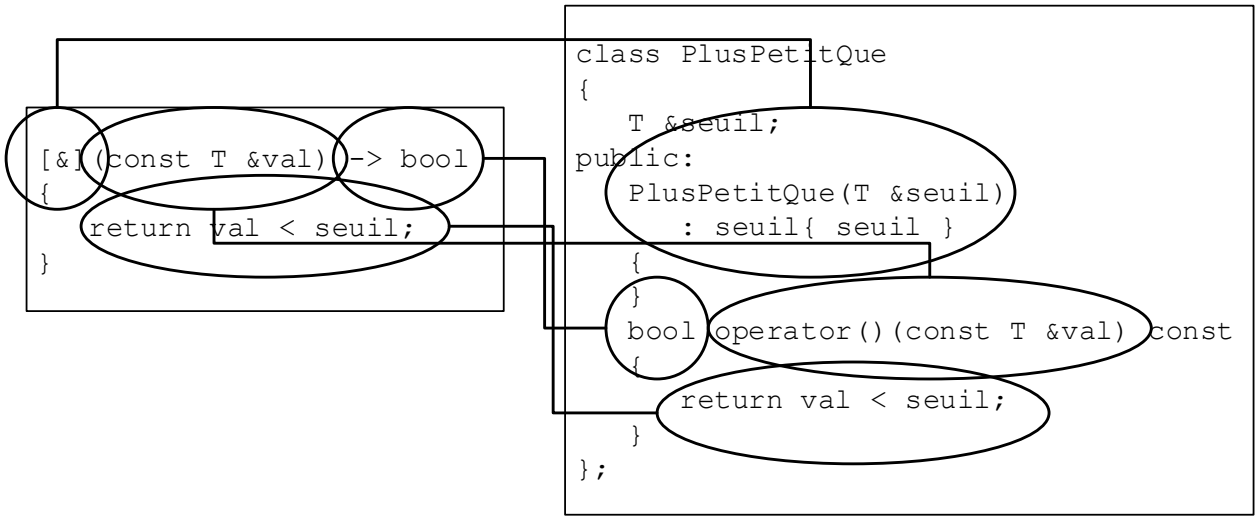
```
auto f = [&](const T &val) -> bool {
   return val < seuil;
};
```

Remarquez la syntaxe reposant sur **`->`**, nouvelle avec C++ 11, pour spécifier le type de retour d'une fonction. Cette syntaxe sert surtout aux λ mais peut aussi être utilisée pour d'autres fonctions : par exemple, **`auto`** `carre(int n)` **`->`** `long { return n*n; }` est une écriture équivalente à `long carre(int n) { return n*n; }` et peut servir dans certains cas pointus de programmation générique.

Enfin, l'implémentation de la λ suit, et est typiquement l'essence du code que nous aurions voulu générer de prime abord.

```
auto f = [&](const T &val) {
   return val < seuil;
};
```

Le schéma ci-dessous montre la relation entre les divers éléments d'une λ et du foncteur correspondant. La correspondance, comme vous pouvez le constater, est directe, et le code généré dans chaque cas offre exactement les mêmes performances.



Maintenant que les λ sont répandues dans la plupart des compilateurs, ce mécanisme se classe assurément parmi les plus populaires de C++ 11.

Expressions λ et états mutables

Par défaut, une donnée capturée par valeur est considérée `const`. Les expressions λ visent à maximiser l'immuabilité (une vertu dans plusieurs contextes subtils, en particulier la multiprogrammation).

Si votre objectif est de permettre à une λ de modifier ses états internes d'un appel à l'autre, alors il vous faudra la qualifier `mutable` :

```
int i = 0;
auto f = [i]() mutable {
   return i++; // légal parce que la lambda est mutable, illégal sinon
};
cout << f() << endl; // affichera 0
cout << f() << endl; // affichera 1
cout << f() << endl; // affichera 2
```

Cet exemple est trop banal pour montrer la pertinence de cette fonctionnalité. Passons donc à un exemple plus riche.

Exemple – utiliser une λ comme continuation

Supposons que nous souhaitions obtenir sur demande une génératrice de valeurs séquentielles à partir d'une valeur initiale choisie. Nous voudrions utiliser pour cela une expression λ, la tâche étant somme toute simple, or nous ne connaissons pas le type d'une λ, ce qui nous empêche de rédiger une fonction la retournant directement.

C'est ici qu'entre en cause le type `function`. En collaboration avec une λ, ce type nous permet de résoudre notre problème avec élégance :

- notre fonction `generer(val)` détermine le type des valeurs à générer et la valeur initiale;
- la λ qui y est créée capture `val` par valeur pour éviter de tenir en elle une référence sur une variable temporaire;
- la λ étant `mutable`, elle peut modifier sa copie interne de `val` au besoin;
- nous utilisons une `function` de signature adéquate pour que la fonction puisse retourner quelque chose d'intelligible; et
- tous ces détails sont masqués du code client grâce au mot clé `auto`.

```
#include <functional>
#include <iostream>
// ...using...
template <class T>
   function<T()> generer(T val) {
      return [val]() mutable {
         return val++;
      };
   }
int main() {
   auto gen = generer(3);
   cout << gen() << endl; // 3
   cout << gen() << endl; // 4
   cout << gen() << endl; // 5
}
```

Surcharge de λ

Il est *a priori* impossible de surcharger une λ en C++.

En effet, une seule version de l'opérateur `()` est générée pour chaque expression telle que celle proposée à droite, au sens où dans un cas comme celui-ci, le foncteur associé à la λ `f` n'expose qu'un seul opérateur `()`, soit `int f::operator(int)`.

```
auto f = [](int i) {
   return i + 1;
};
```

Cependant, il est possible de contourner cette contrainte avec un peu d'ingéniosité, comme le montre cette technique attribuée (à ma connaissance, du moins) à **Mathias Gaunard**, et avec laquelle je suis tombé en contact à la lecture de [UnGenFct] de **Dave Abrahams**.

La technique en question va comme suit :

- définir un foncteur – disons `overload_set` – générique sur la base de deux foncteurs, disons `F0` et `F1`, quelles que soient leurs signatures;
- dériver `overload_set<F0,F1>` de `F0` et `F1` par héritage multiple et public, pour qu'un `overload_set<F0,F1>` soit à la fois un `F0` et un `F1`;
- indiquer par des instructions `using` que `overload_set<F0,F1>` expose `F0::operator()` et `F1::operator()`. À titre de rappel, C++ expose les fonctions *par nom*, pas par signature. Ceci permet d'utiliser `operator()` tel que défini par `F0` et par `F1` directement à partir d'un `overload_set<F0,F1>`, faisant d'un `overload_set<F0,F1>` un foncteur combinant les opérateurs `()` de ses deux parents;
- une fois cette étape franchie, ne reste plus qu'à utiliser une fonction génératrice (ici : `overload(F0,F1)`) pour générer un `overload_set<F0,F1>` à partir de deux expressions λ.

À titre d'exemple :

```
template <class F0, class F1>
   struct overload_set : F0, F1 {
      overload_set(F1 x1, F2 x2) : F1(x1), F2(x2) {
      }
      using F1::operator();
      using F2::operator();
   };
template <class F0, class F1>
   overload_set<F0,F1> overload(F1 f0, F2 f1) {
      return { f0,f1 };
   }
auto f = overload(
   [](){ return 1; },
   [](int x){ return x+1; }
);
int main() {
   int x = f();
   int y = f(2);
}
```

Ce faisant, il devient possible de combiner (par composition d'appels à `overload()`) autant de foncteurs et de λ que souhaité.

Enrichissement avec C++ 14 – captures plus complètes

Avec C++ 11, les λ pouvaient capturer par valeur (par copie) ou par référence. Depuis C++ 14, il est possible de capturer par mouvement, et il est possible de renommer des variables lors d'une copie. Ce qui suit donne un exemple (académique et d'utilité limitée) ce que qu'il est possible d'exprimer dans un bloc de capture avec les λ maintenant :

```
#include <memory>
#include <iostream>
using namespace std;
struct X {
   int valeur = 3;
};
int main() {
   int n = 10;
   unique_ptr<X> p{ new X };
   auto f = [p = std::move(p), niter = n, inc = -1]() mutable {
      cout << p->valeur << ' ';
      return niter += inc;
   }();
}
```

J'ai délibérément limité les actions dans le bloc de capture à une utilisation de nouveaux mécanismes, les autres ayant déjà été expliqués précédemment. Ainsi :

- l'écriture `p = std::move(p)` déplace la responsabilité sur le pointé de p dans la fonction appelante vers un autre p local à la λ. Ce passage par mouvement n'était pas possible, du moins sans heurts, avec C++ 11;
- l'écriture `niter = n` copie le n de la fonction appelante vers une variable niter locale à la λ. Cette capacité de « renommer » une variable capturée est une autre innovation de C++ 14;
- enfin, `inc = -1` crée et initialise une variable locale à la λ sans que nous n'ayons à en faire une contrepartie locale à la fonction appelante.

```
auto f = [p = std::move(p),
          niter = n, inc = -1]() mutable {
      cout << p->valeur << ' ';
      return niter += inc;
   }();
```

```
auto f = [p = std::move(p),
          niter = n, inc = -1]() mutable {
      cout << p->valeur << ' ';
      return niter += inc;
   }();
```

```
auto f = [p = std::move(p),
          niter = n, inc = -1]() mutable {
      cout << p->valeur << ' ';
      return niter += inc;
   }();
```

Cet élargissement des capacités des expressions λ les rend encore plus polyvalentes. Notez au passage que j'ai omis le type de retour (qui aurait été `-> int` dans ce cas) même si la λ ne se limite pas à un `return`; c'est un autre progrès dans l'écriture des λ avec C++ 14.

Enrichissement avec C++ 14 – λ génériques

L'un des principaux atouts de C++ 14 sur le très innovateur C++ 11 est l'avènement de λ génériques. En effet, s'il est possible depuis C++ 11 d'exprimer par des λ des foncteurs de manière concise, comme :

Expressions λ	Foncteurs équivalents

`auto addint = [](int a, int b) {`
`  return a + b;`
`};`
`auto addX = [](X a, X b) {`
`  return a + b;`
`};`
	`struct addint {`
	`  auto operator()(int a, int b) const -> decltype(a+b) {`
	`  return a + b;`
	`  }`
	`};`
	`struct addX {`
	`  auto operator()(X a, X b) const -> decltype(a+b) {`
	`  return a + b;`
	`  }`
	`};`

… il devient possible avec C++ 14 d'exprimer une λ générique comme suit (notez que C++ 14 permet même à des fonctions « ordinaires » d'utiliser simplement `auto` à titre de type de retour) :

Expression λ	Foncteur équivalent

`auto add = [](auto a, auto b) {`
`  return a + b;`
`};`
	`struct add {`
	`  template <class T>`
	`  auto operator()(T a, T b) const {`
	`  return a + b;`
	`  }`
	`};`

Ce faisant, la généricité s'adjoint à la simplicité d'écriture. Cette nouvelle façon de programmer change vraiment la pratique de la programmation de manière considérable.

Dans d'autres langages

Les λ sont un mécanisme qu'offrent la plupart des principaux langages de programmation. Couplés à des mécanismes génériques comme les algorithmes standards de C++, elles permettent un degré remarquable de composition et de réutilisation du code.

Avec C#, une λ prend la forme `(par)=>{expr}`.

Par exemple, dans le code à droite, la méthode `FindAll()` accepte toute prédicat applicable à un élément du conteneur (ici, un `int`) et retourne un objet capable d'énumérer les éléments respectant de prédicat.

L'écriture `x=>` indique que la λ accepte `x` en paramètre; on aurait pu expliciter `(x)` ou `(int x)`; les parenthèses auraient été obligatoires s'il y avait eu plus d'un paramètre.

L'écriture `x>=3&&x<=10` est l'expression évaluée par la λ et `à partir de laquelle sera déterminée sa valeur de retour.

Le code de la λ n'est pas entre accolades parce qu'elle est simple et se limite à une seule instruction; si la λ avait été plus complexe, des accolades auraient été nécessaires.

```
using System;
namespace z13cs
{
    class Program
    {
        static void Main(string[] args)
        {
           List<int> lst = new List<int>(
              new int []{2,3,5,7,11}
           );
           var valeurs = lst.FindAll(
              x => x >= 3 && x <= 10
           );
           foreach (int val in valeurs)
              Console.Write("{0} ", val);
           Console.WriteLine();
        }
    }
}
```

Les λ de C# ne peuvent capturer de valeurs à la construction comme celles de C++.

Avec `VB.NET`, les règles sont semblable à celles que l'on trouve avec C#, et sur le plan syntaxique, une λ est une `Function` (ou une `Sub`) anonyme.

```
Module z
   Sub Main()
      Dim lst As New List(Of Integer)(New Integer() {2, 3, 5, 7, 11})
      Dim valeurs = lst.FindAll(Function(x As Integer)
                                   Return x >= 3 And x <= 10
                                End Function)
      For Each Val As Integer In valeurs
         Console.Write("{0} ", Val)
      Next
      Console.WriteLine()
   End Sub
End Module
```

Les λ sont l'un des ajouts les plus importants de Java 8, dictant une modernisation des outils et un changement philosophique à bien des égards.

La syntaxe et les règles choisies pour les λ de Java ressemblent à celles choisies pour celles de C#, à ceci près que le => de C# devient -> avec Java.

```
import java.util.*;
class X {
  public static void main(String [] args) {
     List<Integer> lst = new ArrayList<Integer>();
     lst.add(2);
     lst.add(3);
     lst.add(5);
     lst.add(7);
     lst.add(11);
     lst.stream().filter(
        x -> x.intValue() < 3 || x.intValue() > 10
     ).forEach(
       x -> System.out.print(x + " ")
     );
     System.out.println();
  }
}
```

Exercices – Série 04

EX00 – À l'aide d'une expression λ, utilisez `transform()` pour remplacer les éléments d'une séquence d'entiers par leur carré.

EX01 – À l'aide d'une expression λ, utiliser `find_if()` pour trouver le premier blanc dans une chaîne de caractères. Faites en sorte d'utiliser la version à deux paramètres de `std::isspace()` pour accomplir cette tâche.

EX02 – Écrivez une fonction `semble_inoffensif(f,arg)` qui reçoit une opération `f` (fonction, foncteur, λ) et un paramètre `arg`, et retourne `true` seulement si `f(arg)` ne lève pas d'exception (ne faites qu'un seul appel à `f(arg)`; on ne peut garantir si cette situation perdurera pour toujours, en général). Appelez cette à l'aide d'une expression λ. Assurez-vous que le tout fonctionne dans le cas où `f(arg)` lève une exception comme dans le cas contraire.

Objets partageables

Il est idiomatique de C++ d'offrir des solutions non-intrusives au partage des objets. Le type `shared_ptr` est emblématique de cette façon de faire, et permet de partager une instance de tout type, qu'il s'agisse d'une classe ou non.

Par contre, que ce soit avec C++ 03, dans lequel `shared_ptr` n'est pas offert de manière standard, ou avec des bibliothèques commerciales (COM, par exemple) qui font d'autres choix philosophiques et technologiques, il arrive que nous ayons à mettre en place des solutions intrusives au partage d'objets.

La responsabilité unique sur un pointeur, politique implémentée par `unique_ptr`, ne permet pas de partager un pointeur entre plusieurs unités de code distinctes, ou entre plusieurs objets distincts.

L'exemple à droite illustre ce fait. Un même pointeur (nommé `p`, dans `main()`) est confié à deux instances de `X`, chacune entreposant le pointeur reçu dans un attribut d'instance de type `unique_ptr`.

Tout va bien, jusqu'au moment où `main()` se termine. À ce stade, les instances `x0` et `x1` sont détruites (l'ordre de leur destruction importe peu ici), ce qui entraîne d'abord pour chacune la destruction de ses attributs d'instance.

Puisque chaque `unique_ptr` s'estime responsable du pointeur placé sous sa gouverne, chacun détruira son pointé, ce qui entraînera deux applications de `delete` sur la même adresse. La première détruira le pointé, et la seconde échouera avec fracas.

Pas de fuite de mémoire ici, ce qui est bien, mais un programme qui plante, ce qui n'est pas joli du tout.

```
#include <memory>
using std::unique_ptr;
class X {
   unique_ptr<int> ptr_;
public:
   X(int *p) : ptr_{p} {
   }
};
int main() {
   int *p = new int{3};
   X x0{p}, x1{p};
} // BOUM!
```

Notons que l'exemple ci-dessus est floué à la base, chacun des `X` n'ayant aucun moyen de savoir que son pointeur appartient aussi à un autre `X`. Nous avons d'abord et avant tout un problème de qualité d'interface dans ce cas-ci.

Les pointeurs bruts, quant à eux, permettent de partager des objets pointés entre plusieurs entités, mais ne permettent pas de définir une politique claire permettant de libérer le pointé quand son dernier utilisateur cessera d'y référer.

Ainsi, à droite, le programme ne plantera pas, mais puisque les instances de `X` ne s'estiment pas responsables du pointeur placé sous leur gouverne, la mémoire associée à `p` dans `main()` ne sera jamais libérée. En fait, en passant du recours au pointeur intelligent `unique_ptr` à des pointeurs bruts, nous sommes passés d'une politique possessive (un seul responsable du point de vue de chaque objet) à aucune politique (aucun responsable).

```
class X {
   int *ptr;
public:
   X(int *p) : ptr{p} {
   }
};
int main() {
   int *p = new int{3};
   X x0{p}, x1{p};
} // FUITE!
```

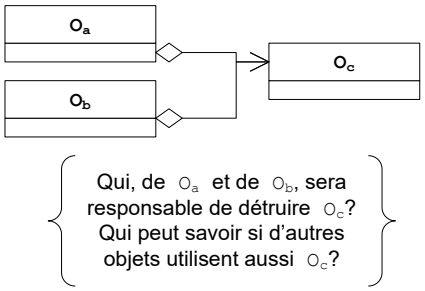
Dans les langages munis d'un moteur de collecte automatique des ordures, la problématique de bien gérer la durée de vie des objets partagés ne se pose *presque* jamais, le système étant responsable de constater vers quels objets un programme contient véritablement des références et de les détruire au moment opportun. On pense ici à un programme local (pas un système réparti) rédigé entièrement en Java ou dans un langage `.NET`, pour ne donner que deux exemples.

Dans les systèmes répartis, comme le sont la plupart des systèmes client/ serveur, ou dans les systèmes locaux construits à l'aide de plus d'un langage de programmation (outre l'environnement particulier du `CLR` de `.NET`), le problème de la bonne gestion des objets partagés se pose réellement. Il en va de même pour les systèmes multiprogrammés, ou plusieurs choses se produisent concurremment.

Utiliser C++ entraîne à la fois une réflexion sur la nature de la responsabilité quant à la vie des objets et sur la nature de leur naissance, de leur duplication et de leur destruction. Tout partage explicite d'objets mène éventuellement à une telle réflexion, même avec d'autres langages de programmation que C++[75].

Allons-y d'une illustration plus formelle. Imaginons que nous sommes en C++, ou encore dans un autre langage supportant la destruction déterministe. Présumons aussi que deux objets, O_a et O_b, utilisent un seul et même objet nommé O_c.

Pour les fins de notre illustration, présumons que l'objet O_a possède un pointeur sur O_c, et qu'il en va de même pour O_b. Présumons aussi que l'objet O_c a été alloué dynamiquement, évitant ainsi que la portée délimite son existence. Voilà visiblement une situation d'agrégation tout à fait typique.



La question clé est : *qui détruira l'objet partagé* O_c? En effet :

- si O_a prend sur lui de le faire, alors il lui faut être certain au préalable que plus personne d'autre (par exemple O_b) n'utilise encore O_c;
- de même, si O_b prend sur lui de détruire O_c, il lui faut être convaincu que ni O_a, ni aucun autre objet n'utilise encore O_c avant de procéder;
- pour un client donné, déterminer que plus aucun autre client n'utilise un objet donné est une tâche difficile, faute d'information en ce sens.

Tenir à jour dans chaque objet une comptabilité du nombre de clients de tous les autres objets est déraisonnable, en temps ou en espace. Pour cette raison, il existe deux grandes familles de stratégies pour déterminer le nombre de clients d'un objet, et pour savoir si un objet peut être collecté ou finalisé.

La première, bien entendu, est la **collecte automatique d'ordures**, intégrée aux moteurs de certains langages.

[75] L'une des raisons pour lesquelles ce qui est proposé ici demeurera intéressant dans un contexte réparti est que, par exemple lorsqu'on fait affaire avec un objet sur un autre ordinateur, on ne peut pas libérer soi-même l'objet utilisé (avec `delete`, disons), la mémoire ne nous appartenant pas.

Ces mécanismes sont offerts en plusieurs saveurs, et sont habituellement capables d'interrompre la marche normale d'un programme pour explorer l'état des relations entre les objets en mémoire.

Les interruptions introduites par une collecte automatique d'ordures compliquent les prévisions quant au temps d'exécution des programmes, mais donnent de bonnes performances en moyenne puisque la collecte d'ordures et la compaction de mémoire qui l'accompagne sont faits par lots plutôt que d'être faits à la pièce.

Une manière parmi plusieurs de collecter les ordures est de partir de références racines, soit celles menant aux objets alloués par le sous-programme principal de chaque *thread* d'un programme, puis d'explorer, récursivement, le graphe des références que possèdent ces objets et de marquer les objets visités.

Une fois que toutes les références accessibles à partir des références racine ont été identifiées, il ne reste plus qu'à éliminer les autres.

Tel que vu à plusieurs reprises déjà, la collecte automatique d'ordures réduit les risques de perte de mémoire mais, de par son imprévisibilité (on ne sait ni quand elle se fera, ni dans quel ordre), empêche les objets de gérer eux-mêmes leur finalisation.

L'autre solution répandue est le **comptage de références**, qui implique de laisser chaque objet partageable comptabiliser lui-même les références dans sa direction, et de faire en sorte que cet objet se fasse *hara-kiri* lorsque la dernière référence menant vers lui s'en sera déconnectée.

Contrairement à la collecte d'ordures, qui doit être intégrée à même le moteur d'un langage puisqu'elle implique une suspension occasionnelle de toutes les opérations, le comptage de références peut être implémenté par programmation, dans la mesure où des mécanismes suffisants existent (incluant une destruction déterministe et, pour en arriver à une solution élégante, la surcharge d'opérateurs).

Nous verrons, ci-dessous, deux approches distinctes pour en arriver à un comptage de références : une approche *collaborative*, bien connue mais un peu fastidieuse, et une approche *générique*, plus élégante à bien des égards mais plus subtile à implémenter, se rapprochant quelque peu des pointeurs intelligents. Nous avons sommairement abordé cette question un peu plus haut.

Notez que toute solution automatisée au problème de la responsabilité des objets partagés a des forces et des faiblesses. Les moteurs de collecte automatisée d'ordures donnent de bons résultats d'ensemble, du fait qu'ils ont une vue globale de la consommation de mémoire et de la localité des objets; ils sont en mesure de compacter la mémoire lors des collectes, et peuvent ainsi réduire les risques de fragmentation de la mémoire, ce que ne peuvent pas faire les pointeurs intelligents[76]. En retour, cette approche introduit une imprévisibilité dans les programmes : la collecte interrompt le fonctionnement du programme à des moments pouvant être inopportuns, ce qui empêche par exemple la rédaction de programmes soumis à des contraintes de temps réel.

[76] En fait, pour réaliser une compaction de la mémoire lors de collecte d'ordures, il faut être capable de déplacer concrètement les objets en mémoire à l'insu des programmes. Pour y arriver, il est nécessaire de faire en sorte que les programmes ne voient pas les véritables adresses de leurs objets (sinon, les déplacements de ces objets détruiraient silencieusement la sémantique du programme). Toute solution révélant les véritables adresses des objets ne peut que compacter la mémoire inutilisée par le programme.

Les approches par pointeurs intelligents définissent des responsabilités quant à la gestion des ressources, mais n'ont pas la vision globale d'un moteur de collecte d'ordures. Ces approches peuvent être plus prévisibles dans leur gestion des ressources : avec un `unique_ptr`, par exemple, il est clair que la portée du pointeur intelligent délimite cette du pointeur dont il a la charge. Toutefois, les performances moyennes, à long terme, d'un moteur de collecte automatisée d'ordures en comparaison avec celles de pointeurs intelligents mettront typiquement en valeur le moteur, de par sa compréhension plus globale des enjeux.

En tout temps, l'automatisation de libération des ressources se heurte à certains problèmes inextricables, par exemple celui des références circulaires, où un objet `A` réfère à un objet `B`, qui réfère à `A`), qui peut compliquer la libération de ces objets (dans quel ordre deux objets se référant mutuellement l'un et l'autre devraient-ils être libérés?).

Partage par comptage collaboratif de références

Un objet pouvant être partagé voudra tenir à jour, à l'interne, un compteur de références, donc un entier dont la valeur correspond au nombre de pointeurs distincts vers l'objet partagé. À la construction, un objet qu'on peut partager initialisera ce compteur de références à `1` (pas à `0`). Après tout, par définition, si un objet partageable vient d'être créé, c'est que quelqu'un, quelque part, pointe déjà vers lui.

Par exemple, dans l'ébauche de classe `Partageable` proposée à droite, l'attribut `cptref` est un compteur de références[77]. Tout objet pointant sur un objet partageable devra immédiatement lui ajouter une référence, pour s'assurer que personne d'autre ne détruise cet objet entre-temps. Il faut donc qu'un objet partageable offre une méthode pour qu'on puisse lui ajouter une référence. Dans l'exemple à droite, cette méthode est `AddRef()`.

```
class Partageable {
   long cptref;
   // ...
public:
   Partageable() noexcept : cptref{1} {
   }
   void AddRef() noexcept {
      ++cptref;
   }
   // ...
```

S'il apparaît utile de connaître le nombre de références sur un objet partageable, on peut bien sûr ajouter au modèle un accesseur qui retourne cette valeur. Si le système est très dynamique, par contre, cette valeur risque de changer sans avertissement et ne sera que d'une utilité limitée.

Évidemment, il faut offrir au code client une manière d'indiquer qu'il n'utilisera plus l'objet partageable. On ajoutera donc une méthode à l'objet partageable qui aura un double rôle :

- décrémenter le compteur de références, bien sûr; et
- détruire l'objet partageable dès que ce compteur tombe à zéro.

Ce deuxième rôle est essentiel, puisque la définition de la mécanique de partage proposée établit qu'à partir de ce moment, l'objet partageable n'a plus aucun client; plus personne ne pourra le détruire suite à l'exécution de cette méthode.

[77] Une gestion solide du compteur de références partagé demanderait que celui-ci soit atomique, une thématique avancée que nous couvrirons ultérieurement.

La méthode `Release()` dans l'extrait un peu plus raffiné de `Partageable`, à droite, fait ce travail. L'expression `delete this;` peut sembler étrange, mais c'est bien la chose à faire. Si l'objet ne décède pas là, alors il ne sera jamais détruit.

```
   // ...
   void Release() noexcept {
      if (--cptref == 0)
         delete this;
   }
protected:
   virtual ~Partageable() = default;
};
```

La qualification `protected` du destructeur empêche le code client de détruire un tel objet manuellement. Le code client ne peut savoir s'il est raisonnable de détruire un objet partagé; seul l'objet lui-même peut le savoir.

Notez que, du fait que `Partageable::Release()` invoque `delete this;` et qu'il est hautement probable que l'objet à détruire soit un dérivé de `Partageable`, il faut que le destructeur de `Partageable` soit polymorphique. C'est un coût de cette approche.

Classes anonymes

Ce qui suit est d'un intérêt surtout historique, la proéminence des expressions λ dans la plupart des langages de programmation rendant ces pratiques un peu caduques.

L'opacité des interfaces, couplée au polymorphisme, fait en sorte qu'à travers un pointeur d'interface, un programme peut utiliser des services sans connaître ni la nature exacte de l'objet derrière, ni ses détails d'implémentation, ni même son nom.

Certaines classes n'existent que pour servir d'implémentation à une interface; pensez à `CompteBancaireImpl`, dans la section ***Objets opaques et fabriques***, plus haut. D'autres encore n'existent que pour servir une cause précise et ponctuelle. Dans de tels cas, il y a lieu de se demander pourquoi créer une classe toute entière, et pourquoi lui assigner un nom qui ne sera plus utilisable pour d'autres fins dans le programme.

Il existe une technique très utilisée en Java, et maintenant possible en C# (depuis la version `2.0`) et, de manière un peu moins élégante, en C++, de générer des classes sans consommer (en quelque sorte) des noms : avoir recours à des *classes anonymes*.

⇒ Une **classe anonyme** est une classe créée au moment où on a besoin de l'instancier. Il s'agit nécessairement d'une classe dérivée d'un parent ayant au moins une méthode virtuelle, et qui raffine au moins l'une des méthodes virtuelles de son parent.

⇒ *Une classe anonyme ne peut être abstraite*, devant nécessairement être instanciée dans l'immédiat, et ne pourra pas servir de parent à une autre classe, n'ayant elle-même pas de nom.

Les classes anonymes sont fréquemment des dérivés de classes polymorphiques, souvent même des dérivés de classes abstraites). Le code client d'une classe anonyme opère donc typiquement à travers une indirection sur le parent et ne connaît rien de l'enfant; pour ce dernier, l'anonymat est tout indiqué.

Par exemple, en Java, un `WindowAdapter` (dérivée de `WindowListener`) sert à écouter des événements de fermeture de fenêtre, et offre des méthodes bidon pour chaque événement possible. Ceci permet au code clie t de n'implémenter que certains comportements (pas tous) de l'interface `WindowListener` et allège le code client.

Nous examinerons un exemple plus académique de classe anonyme en Java dans la section ***Dans d'autres langages***, plus loin.

Pour ajouter une instance capable d'écouter des événements sur une fenêtre, on appelle la méthode `addWindowListener()` de cette fenêtre et on lui passe une référence à un `WindowListener` en paramètre.

Si nous souhaitons fermer un programme lorsque la fenêtre `f` est fermée, nous pouvons invoquer `System.exit()` au moment où cet événement se produit. L'événement de fermeture résulte en un appel de la méthode `windowClosing()` des instances de `WindowListener` associés à la fenêtre se faisant fermer.

Pour ajouter à la fenêtre `f` un dérivé de `WindowAdapter` anonyme qui ne fait que réagir correctement à un événement de fermeture, on procède alors comme à droite

```
// ...
f.addWindowListener (
   // dérivé anonyme de WindowAdapter
   new WindowAdapter() {
      // on surcharge cette méthode bien précise
      public void windowClosing(WindowEvent w) {
         System.exit(0);
      }
   }
);
// ...
```

Pas besoin de nommer la classe temporaire générée : la classe dérivée de `WindowAdapter` utilisée dans cet exemple a pour seule raison d'être celle de fermer le programme au moment opportun. Le parent de la classe anonyme est ici `WindowAdapter`, et le dérivé qui a la méthode `windowClosing()` visible dans notre exemple est une classe qui n'a aucun nom, mais qu'on instancie quand même à l'aide de `new`. La classe anonyme sera utilisée par polymorphisme comme le serait n'importe quel `WindowAdapter`.

Créer une classe entière et lui donner un nom aurait été possible ici. Ce serait l'approche à privilégier si nous avions souhaité en créer plusieurs instances ou en faire des classes dérivées.

Classes locales aux fonctions

En C++, les *vraies* classes anonymes, le mot anonyme étant pris ici au sens strict de *sans nom*, n'existent pas[78]. Par contre, on peut, à l'aide d'une classe déclarée *localement à une fonction*, contourner ce problème et en arriver presque au même résultat.

Un exemple simple (et académique) serait celui proposé à droite. En C++, deux classes du même nom ne peuvent être déclarées dans le même espace nommé. Par contre, deux fonctions distinctes, comme `f()` et `g()`, peuvent chacune contenir une classe interne, donc locale, de même nom. À droite, les classes `X` de `f()` et de `g()` n'entrent pas en conflit car leurs noms respectifs n'ont qu'une portée locale.

```
int f() {
   struct X {
      int f() const {
         return 3;
      }
   };
   X x;
   return x.f();
}
int g() {
   struct X {
      int f() const {
         return 4;
      }
   };
   X x;
   return x.f();
}
int main() {
   int x = f() + g();
}
```

Évidemment, plusieurs manœuvres sont interdites sur une classe de portée locale. Entre autres :

- on ne peut en dériver, puisque son nom n'existe pas hors de la fonction où elle est déclarée (mais une classe locale peut dériver d'une classe qui ne l'est pas);
- elle ne peut être générique; et
- elle ne peut avoir de membres qualifiés `static`, pour des raisons très techniques.

Les langages C et C++ ne permettent pas de définir des fonctions locales à d'autres fonctions comme le font par exemple Pascal ou Modula. En mêlant classes locales et foncteurs (voir section ***Foncteurs***, plus loin) ou λ-expressions (voir ***Les expressions Lambda***, aussi plus loin), il devient possible en C++ de définir *à toutes fins pratiques* des fonctions locales à d'autres fonctions, ce qui mène à du code à la fois propre, élégant et très performant.

Examinons maintenant comment une classe locale en C++ peut jouer un rôle analogue à celui des classes anonymes en Java. Supposons qu'on ait une classe abstraite `Minuterie` dont la méthode virtuelle `agir()` est appelée de manière récurrente et régulière, la fréquence de l'appel à cette méthode étant fixée à la construction d'une instance (en millisecondes, disons). Présumons ensuite qu'on désire un dérivé de `Minuterie` qui affiche un `'.'` à l'écran à chaque seconde, pour montrer le temps qui passe.

Si on pense générer plusieurs minuteries vouées à afficher des `'.'` à la console (ou réutiliser cette classe ailleurs), il devient intéressant d'en faire une classe nommée à part entière (nommée par exemple `TiPoints`). Toutefois, si on juge que la classe réutilisable est `Minuterie` et que l'idée d'afficher des `'.'` est une fonctionnalité à usage unique, il se peut qu'on choisisse plutôt d'utiliser une classe anonyme.

---

[78] En fait, il existe quelque chose que les gens nomment *classe anonyme*, mais il s'agit plus un dérivé syntaxique des structures C des années `'70` que d'un sujet vraiment intéressant pour nous (en fait, ce sont des `struct` sans nom dont on déclare spontanément des instances, une technique illégale en C++ strict), par exemple :

```
struct { int x, y; } point; // en C, point est une variable dont le type n'a pas de nom
```

L'idée de classe anonyme au sens où nous l'entendons dans ce document est plus près de ce que présente Java.

Voici comment on implanterait le tout. Présumant que la classe `Minuterie` ressemble à l'extrait exposé à droite, on créera une fonction `CreerTiPoints()` qui retournera un `Minuterie*`, puisque c'est à travers cette interface que le code client connaîtra notre classe anonyme.

```
class Minuterie {
   // ...
public:
   // Frequence en millisecondes
   using freq_type = int;
   Minuterie(freq_type);
   virtual ~Minuterie() = default;
   virtual void agir() = 0;
};
```

Cette fonction apparaîtra comme suit :

```
#include <iostream>
#include <memory>
// ... using ...
unique_ptr<Minuterie> CreerTiPoints(ostream &os) {
   // création de la classe anonyme (le nom n'importe pas)
   class PetitsPoints : public Minuterie {
      ostream &os;
   public:
      PetitsPoints(ostream &os) : Minuterie{1'000}, os{os} { // agir à chaque seconde
      }
      void agir() { // la tâche à réaliser à chaque seconde
         os << '.' << flush;
      }
   };
   // instanciation. Le pointeur retourné au sous-programme utilisateur est un
   // Minuterie*; le code client ne connaîtra jamais le type réel de l'objet créé
   return make_unique<Minuterie>(new PetitsPoints{os});
}
```

On voit que la classe n'est pas vraiment anonyme au sens strict, mais que son nom est local à la (courte) fonction qui la déclare. Ainsi, on pourrait utiliser un nom générique de classe (p. ex. : `XYZ`) et utiliser ce même nom dans toutes les fonctions du genre sans créer le moindre conflit.

Un client désireux d'utiliser cette fonctionnalité pourrait ressembler à celui proposé à droite. Évidemment, il manque des détails à la classe `Minuterie`, mais nous devrions couvrir des éléments de multiprogrammation pour les présenter alors nous nous en abstiendrons ici.

```
// ...
int main() {
   auto p = CreerTiPoints(cout);
   char c;
   cin >> c;
}
```

Dans d'autres langages

L'anonymat peut prendre diverses formes d'un langage à l'autre.

En Java, une classe anonyme est un dérivé d'une abstraction construite sur-le-champ, en fonction des besoins, et qui a accès aux données avoisinantes, dans la mesure où ce sont des entités de valeur fixe (pour des objets, les références doivent être qualifiées `final`).

Dans l'exemple à droite, `X` est une abstraction pure (une interface), et `main()` crée de manière anonyme un dérivé de `X` qui saura afficher le contenu d'une référence constante locale. Remarquez la syntaxe : on y crée un nouveau `X` puis on place ses spécificités dans un bloc anonyme. Il s'agit donc bien d'un cas de spécialisation au sens habituel du terme.

```
public class Z {
   public interface X {
      void f();
   }
   public static void g(X x) {
      x.f();
   }
   public static void main(String [] args) {
      final String s = "Coucou";
      g (
         new X() {
            private String msg_ = s;
            public void f() {
               System.out.println(msg_);
            }
         }
      );
   }
}
```

En C#, deux formes d'anonymat sont possibles, soit l'anonymat opérationnel, à l'aide de délégués, qui sont des fonctions polymorphiques sur la base de leur signature, et l'anonymat structurel, à partir d'enregistrements sans noms. L'anonymat structurel existe entre autres pour faciliter les accès aux bases de données avec le moteur *Language Integrated Query* (`LINQ`) de C#.

L'anonymat opérationnel ressemble aux classes anonymes de Java mais, reposant essentiellement sur des fonctions plutôt que sur des objets, ne permet pas de tenir à jour des états.

L'anonymat structurel permet de créer des `struct` (de C#) anonymes mais dont il est toute de même possible d'accéder aux attributs. Le mot clé `var` tient alors la place du type dans la déclaration de la variable.

```
namespace Z
{
   class Test
   {
      delegate void f();
      private static void g(f fct)
      {
         fct();
      }
      public static void Main(string[] args)
      {
         string mot = "Coucou";
         g (
            delegate() {
               System.Console.WriteLine(mot);
            }
         );
         var X = new { Nom = "Patate", Age = 3 };
         System.Console.WriteLine
            ("{0} {1}", X.Nom, X.Age);
      }
   }
}
```

En VB.NET, l'anonymat opérationnel n'est pas supporté, même si les délégués le sont.

L'anonymat structurel est possible à l'aide du mot clé With. L'exemple à droite en montre une application possible.

```
Module Z
   Sub Main()
      Dim X = New With _
      { _
         .Nom = "Patate", _
         .Age = 3 _
      }
      System.Console.WriteLine _
         ("{0} {1}", X.Nom, X.Age)
   End Sub
End Module
```

Réflexion

Remplacez l'implémentation de `Minuterie` dont il faut dériver par une version générique telle que ceci :

```
unique_ptr<Minuterie> CreerTiPoints(ostream &os) {
   // création de la classe anonyme (le nom n'importe pas)
   class PetitsPoints : public Minuterie {
      ostream &os;
   public:
      PetitsPoints(ostream &os) : Minuterie{1'000}, os{os} { // agir à chaque seconde
      }
      void agir() { // la tâche à réaliser à chaque seconde
         os << '.' << flush;
      }
   };
   // instanciation. Le pointeur retourné au sous-programme utilisateur est un
   // Minuterie*; le code client ne connaîtra jamais le type réel de l'objet créé
   return make_unique<Minuterie>{new PetitsPoints{os}};
}
```

…deviendra cela :

```
Minuterie CreerTiPoints(ostream & os) {
   return Minuterie{[&os]() { os << '.'; }, 1'000};
}
```

Vous aurez probablement besoin pour y arriver d'utiliser `std::function`, de la bibliothèque `<functional>`. Comparez les deux implémentations (polymorphique et générique) pour faire ressortir les forces et faiblesses de chacune.

Les *mixins*[79]

Il arrive qu'on rencontre, dans la littérature informatique contemporaine, le terme un peu particulier *mixin*. On trouve au moins deux dénominations alternatives pour cette technique, soit celle de *classes constructeurs* et celle de *programmation par théorème*.

Ce terme dénote une technique répandue dans certains langages, par exemple `Modula-3` ou `CLOS`, peu connue en C++, et impossible *a priori* (du moins au sens strict) dans les langages qui ne supportent pas héritage multiple d'implémentation tels que Java et C#, quoiqu'on puisse contourner le problème en travaillant fort[80] ou en utilisant des extensions au langage.

Le terme *mixin* provient du monde Lisp, plus particulièrement du monde de ce dialecte `OO` de Lisp qu'est `CLOS`, et correspond à un schéma de conception développé dans cette communauté pour encadrer l'héritage multiple.

Caractéristiques des mixins

Les *mixins* sont une application directe et ingénieuse mêlant fabriques, interfaces, classes anonymes, héritage multiple et virtuel. On n'a habituellement pas recours à des *mixins* pour réaliser des petites classes à usage unique; cette technique est surtout utilisée pour concevoir du code susceptible d'être utilisé souvent et longtemps, par exemple du code de bibliothèque.

Les *mixins* permettent de découper une classe en deux blocs disjoints et de penser les algorithmes généraux et les détails d'implémentation. En ceci, ils représentent une alternative à la programmation par politiques, connue des *aficionados* de programmation générique. À partir d'une classe racine donnée, l'approche par mixins définit les algorithmes d'un côté et les éléments d'implémentation de l'autre côté, puis joint les deux par un enfant anonyme.

L'algorithme sollicite des définitions qu'il ne connaît pas *a priori*, ce qui provoque ce qu'on nomme des ***appels frères***, ou *Sibling Calls*[81], c'est-à-dire l'idée d'appels entre descendants d'un ancêtre commun, une application subtile du polymorphisme avec héritage multiple.

---

[79] Inspiré à l'origine de http://cpptips.hyperformix.com/cpptips/mixins

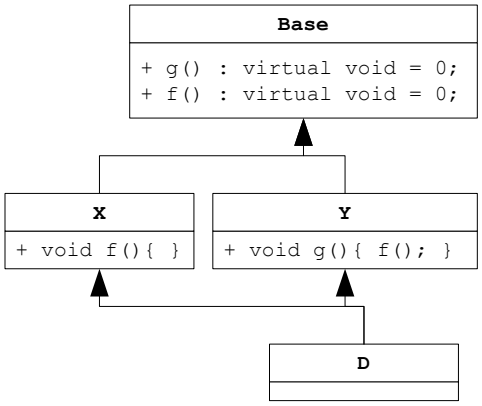
[80] Voir [POOv03], section ***Concepts ou techniques***, pour voir comment simuler l'héritage multiple dans un langage à héritage simple seulement comme le sont Java et C#; de manière alternative, avec Java, voir [JavaMulH]. Pour des extensions permettant d'appliquer l'héritage multiple (centré sur les *mixins* en Java), voir [JAM].

[81] Je préfère le terme anglais, plus neutre (*Sibling* réfère à frère ou à sœur de manière indifférenciée).

Appels frères

Examinons brièvement ce qu'est un appel frère. Pour ce faire, imaginons qu'une classe parent donnée (`Base`) dévoile une méthode abstraite (ici, la méthode `Base::f()`). Imaginons aussi les dérivés virtuels `X` et `Y`, donc susceptibles de partager une même partie `Base` fusionnelle si un éventuel enfant les unifie.

Dans notre exemple, la méthode `g()` est abstraite dans `Base`, par analogie avec l'approche par *mixins*. Ce n'est toutefois pas nécessaire pour décrire les appels frères (le polymorphisme seul suffit).



L'un des dérivés de ce parent commun (ici, `Y`) se sert, dans l'une de ses méthodes, de la méthode abstraite `f()` sans lui-même l'implanter. Ne définissant pas `f()`, `Y` demeure une classe abstraite.

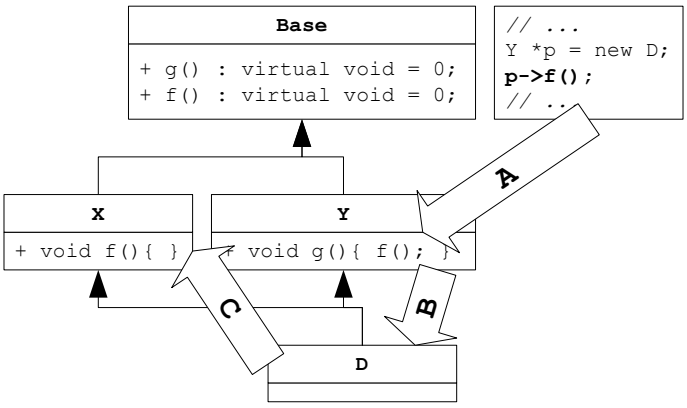
Un autre dérivé de ce parent, `X`, implémente la méthode `f()` comme bon lui semble, mais est abstrait puisqu'il ne définit pas la méthode `g()`.

Enfin, un dérivé indirect de `Base`, la classe `D`, dérive à la fois de la classe `Y`, qui utilise `f()` sans la définir, et de la classe `X`, qui implémente `f()`. La classe `D` n'est pas abstraite puisque toutes ses méthodes abstraites ont été implémentées – *une combinaison de deux classes abstraites peut ne pas être abstraite*.

Imaginons qu'un client obtienne un pointeur de `Y` pointant en fait sur un `D` (car ici, un `D` est clairement un `Y`).

Si le client invoque la méthode `f()` à travers ce pointeur, alors l'appel passera de la partie `Y` vers la partie enfant `D` vers la partie `X`, *résultant en un appel tout à fait valide de méthode entre deux classes qui ne sont pas directement liées l'une à l'autre*.

Voilà ce qu'on nomme un **appel frère**.



Dans notre schéma, la méthode abstraite `void Base::f() = 0;` est ce qu'on appelle un ***méthode hameçon*** (*Hook*). Certains diront aussi une *méthode vecteur*, quoiqu'il faille être prudent et faire attention de ne pas confondre le terme vecteur avec l'une ou l'autre de ses multiples autres significations.

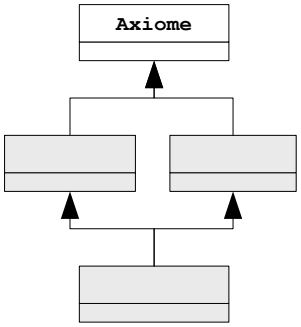
Dénominations des éléments d'un mixin

Construire des *mixins* est une technique légèrement complexe qui demande qu'on définisse au préalable un peu de vocabulaire. Voici donc les principaux éléments constitutifs de cette technique et les noms qu'on leur donne habituellement.

Classes axiomes

Les **classes axiomes** d'un mixin sont celles servant à titre de base de la hiérarchie.

Les classes axiomes tendent à être petites, indépendantes les unes des autres, et abstraites. En général, les méthodes des classes axiomes sont peu nombreuses, indépendantes les unes des autres, et abstraites. Certains qualifieront aussi ces méthodes d'*axiomes*.



Les axiomes servent de colle à la technique mais ne sont pas des abstractions destinées à être utilisées par le code client. En pratique, il est sage (sans être nécessaire) de faire de ses enfants des dérivés privés (plutôt que publics) et virtuels; la technique des *mixins* **dépend** de l'héritage virtuel.

Fonctions théorèmes

Certains préfèrent toutefois utiliser des *mixins* à partir des axiomes, question de style. Il n'y a pas de raison d'être dogmatique ici. Les classes axiomes se retrouveront souvent dans une bibliothèque, à laquelle on ajoutera (si les axiomes sont des parents publics) des fonctions globales capables d'opérer sur ces abstractions de manière polymorphique.

Si l'axiome expose un constructeur autre que le constructeur par défaut, alors il est préférable que ses dérivés soient protégés ou publics. En effet, l'héritage virtuel par lequel les enfants de la classe axiome dériveront cette dernière impose aux classes les plus dérivées de la hiérarchie (aux *mixins* eux-mêmes) d'invoquer directement les constructeurs des bases fusionnelles. Si le *mixin* ne connaît pas ses grands-parents, alors il ne sera pas en mesure de les construire correctement.

Ces fonctions globales ressemblent alors à des théorèmes (voir plus bas), du fait qu'elles sont définies en fonction des axiomes purs, pour lesquels il n'existe pas encore de définition. On les nomme donc souvent *fonctions théorèmes*.

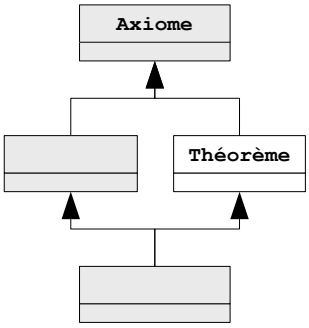
```
g(Ax1&); g(Ax2&); g(Ax3&);
h(Ax1*); h(Ax2*); h(Ax3*);
k(Ax1*,Ax2*);
```

Remarquez que ces fonctions, tout aussi globales qu'elles soient, ne polluent pas vraiment l'espace nommé global[82], étant qualifiées par les classes sur lesquelles elles opèrent. Elles sont aussi liées à ces classes axiomes que le sont les méthodes des classes axiomes elles-mêmes, mais ne savent en rien comment les axiomes seront implémentés. On pourrait aussi contraindre les fonctions théorèmes en les plaçant dans un espace nommé.

On peut, comme le font les mathématiciennes et les mathématiciens, combiner les classes axiomes avec d'autres classes axiomes à l'aide d'héritage virtuel (public, bien sûr) pour générer des ensembles axiomatiques plus étendus.

[82] Au sens où il ne s'agit pas de fonctions tout usage, mais bien de fonctions qui doivent être utilisées avec au moins une classe axiome (ou l'une de ses classes dérivées).

Classes théorèmes



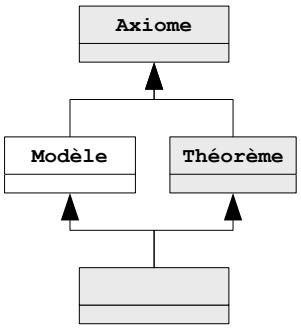
Les **classes théorèmes** sont celles qu'on appelle aussi les *classes d'interfaces*[83]. Au sens ○○ strict, évidemment, les classes axiomes sont souvent déjà, en elles-mêmes, des interfaces.

Habituellement, les classes théorèmes implémentent des algorithmes généraux de manière efficace et élégante, mais faisant abstraction de détails techniques pour lesquels plusieurs approches distinctes seraient judicieuses selon le contexte.

À moins d'utiliser des fonctions théorèmes, c'est par une classe théorème, prise comme une interface abstraite, que le code client opérera habituellement sur un *mixin*.

Classes modèles

Une **classe modèle** implémente au moins une méthode (souvent, un groupe de méthodes) d'une classe axiome dans un contexte de *mixin*. À la différence des classes théorèmes, qui implémentent des algorithmes généraux, les classes modèles implémentent des variantes stratégiques d'implémentation.



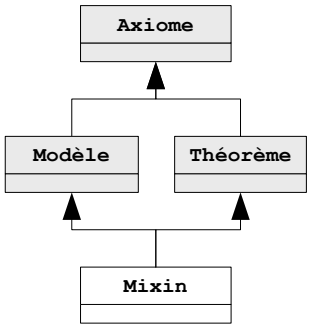
Dans une bibliothèque pensée pour des *mixin*, on trouvera, pour chaque axiome, plusieurs implémentations possibles, chacune avec ses caractéristiques propres.

À chaque implémentation jugée pertinente, on pourra faire correspondre une classe modèle distincte. Par exemple, pour un algorithme de calcul donné dans une bibliothèque de *mixins*, on retrouvera normalement une implémentation optimisée à des fins de vitesse et une autre minimisant les besoins en mémoire.

L'objectif derrière une classe modèle n'est pas d'être instanciée directement, mais bien d'implanter une (ou plusieurs) méthodes abstraites d'un des axiomes dont elle dérive. Dans un *mixin*, les algorithmes implémentés par les classes théorèmes invoquent (par appels frères) les implémentations des classes modèles avec lesquelles elles sont intégrées.

[83] La raison de ce choix de nom tient du fait que les méthodes dévoilées par les classes théorèmes sont généralement faites pour être appelées par des programmes utilisateurs, alors que celles des classes axiomes servent surtout aux fins des classes théorèmes. Ainsi, les programmes feront plus affaire avec les théorèmes qu'avec les axiomes, ce qui signifie que les théorèmes tendront à servir d'interfaces pour les axiomes (ou pour des théorèmes plus abstraits).

Classes constructeurs : les mixin



Un *mixin* intègre par héritage public des classes théorèmes et des classes modèles, permettant ainsi aux théorèmes d'utiliser, à travers des appels frères, les modèles.

Les classes théorèmes sont abstraites, par définition, puisqu'il leur manque les méthodes qu'implémentent les classes modèles. Les classes modèles sont abstraites, aussi par définition, puisqu'il leur manque les méthodes implémentées par les classes théorèmes.

Un *mixin*, lui, n'est pas abstrait puisqu'il intègre une implémentation légale de chaque méthode des axiomes qui lui servent d'ancêtres.

Un *mixin* n'est jamais utilisé pour lui-même. Son seul rôle est de mettre en commun des théorèmes et des modèles. Pour cette raison, l'implémentation typique d'un *mixin* est définie de manière locale à une fonction de fabrication : **les *mixins* sont des classes anonymes**.

Une fonction dans laquelle est défini un *mixin* et qui est (par définition) la seule à pouvoir instancier ce *mixin* sera la **fonction génératrice** du *mixin*.

Le nom *mixin* provient du mot anglais pour l'acte de mélanger (*mixin'!*) et représente une mise en commun de l'implémentation désirée pour chaque axiome avec les interfaces dévoilées par chaque théorème. On fait ce mélange en dérivant virtuellement la classe qui sera utilisée du/ des théorèmes voulus et de l'implémentation choisie pour chacun de ses axiomes.

Construire un mixin

Clarifions le propos par un exemple concret mais un peu académique[84]. Imaginons que nous voulions un outil capable d'afficher un message sur le flux de sortie standard (`std::cout`) avec un certain formatage, mais que plusieurs formatages distincts soient possibles.

La classe axiome pourrait se présenter de la manière proposée à droite. Elle se décomposera en deux méthodes (en plus du destructeur polymorphique) :

- la méthode `afficher()`, qui affichera une ligne formatée et sera implémentée par le théorème; et
- la méthode `formater()`, qui prendra un message et retournera une version formatée de ce message. Puisque plusieurs stratégies de formatage seront envisageables, cette méthode sera implémentée par des modèles.

```
#include <string>
// ... using ...
struct AxiomeAffichage {
   virtual void afficher(const string&) = 0;
   virtual string formater(const string&) = 0;
   virtual ~AxiomeAffichage() = default;
};
```

Notez que, comme toute classe exposant au moins une méthode virtuelle devrait le faire, la classe axiome proposée ici exposera un destructeur virtuel. Ainsi, dans le cas où un de ses dérivés aurait recours à des ressources particulières (p. ex. : des règles de formatage prises dans une base de données ou dans un document `XML`), la libération correcte de ces ressources sera possible.

La classe théorème, `Afficheur`, sera celle qui sera éventuellement utilisée par le code client. Ceci explique que son nom soit plus convivial que celui attribué à la classe axiome.

Nous y implémenterons l'algorithme de la méthode `afficher()` en faisant abstraction de la manière dont sera éventuellement implémentée la méthode `formater()`.

Le recours à l'héritage privé tient du fait que le lien de parenté entre axiome et théorème, ici, est un détail d'implémentation.

```
// ...
#include <iostream> // bof
#include <string>
// ... using ...
struct Afficheur
   : private virtual AxiomeAffichage {
   void afficher(const string &s) {
      cout << formater(s) << endl;
   }
};
```

Nous utilisons le parent fusionnel pour unifier les implémentations partielles des théorèmes et des implémentations dans un *mixin*, mais notre interface pour le code client sera `Afficheur`.

[84] Un exemple plus réaliste (mais plus complexe) sera proposé plus loin.

La technique des *mixins* se comprend mieux s'il existe plusieurs modèles pour un théorème donné. Ainsi, imaginons trois modèles distincts pour implémenter `formater()`, soit :

- le modèle que nous qualifierons de *vanille*, qui ne fera aucun formatage (un *Pass-Through*);
- le modèle que nous qualifierons de *décoratif*, qui insérera un peu de flafla autour du message; et
- le modèle *paragraphe HTML*, qui insérera les balises `HTML` de paragraphe `<p>` et `</p>` autour du message.

Ces trois stratégies de formatage sont toutes aussi convenables les unes que les autres, dans la mesure où elles sont utilisées au moment opportun. Il n'y a pas de bon ou de mauvais formatage, que des formatages utiles selon la situation. Tout est question de besoins, de contexte pour le code client.

Comme dans le cas du théorème, l'héritage sera privé mais virtuel. Dans une bibliothèque commerciale, les classes modèles ne seraient pas placées dans un fichier visible pour le code client mais bien dans un fichier source, livré sous forme binaire.

Les prototypes des fonctions génératrices seront visibles du code client, mais leur implémentation sera habituellement dissimulée.

Le code client transigera à partir de l'abstraction servant de classe théorème, `Afficheur`, et chacune de ces méthodes créera bel et bien un `Afficheur`, donc donnera accès à une implémentation du théorème.

```
#include <string>
using std::string;
// ...
struct ModeleVanille
   : private virtual AxiomeAffichage {
   string formater(const string &s) {
      return s;
   }
};
struct ModeleDecoratif
   : private virtual AxiomeAffichage {
   string formater(const string &s) {
      static const string
         AVANT = "<< ", APRES = " >>";
      return AVANT + s + APRES;
   }
};
struct ModeleParagrapheHTML
   : private virtual AxiomeAffichage {
   string formater(const string &s) {
      static const string
         AVANT = "<p>", APRES = "</p>";
      return AVANT + s + APRES;
   }
};
```

```
// ...
unique_ptr<Afficheur> CreerAfficheurVanille();
unique_ptr<Afficheur> CreerAfficheurDecoratif();
unique_ptr<Afficheur> CreerAfficheurHTML();
```

Chaque fonction génératrice définira une classe locale (donc anonyme du point de vue du programme) nommée `Mixin` (par tradition) qui unifiera théorème et modèle.

Vous remarquerez que les *mixins* proposés à droite sont des classes vides. On nomme parfois les *mixins* **classes constructeurs** du fait que la seule méthode susceptible d'être définie dans une telle classe serait un constructeur, ce qui permet de contrôler la construction des parents si la construction requise n'est pas celle par défaut (d'où la prudence dans le choix du type d'héritage pour les diverses classes de la hiérarchie menant à un *mixin*).

```
// ...
unique_ptr<Afficheur> CreerAfficheurVanille() {
   class Mixin
      : public Afficheur, ModeleVanille {
   };
   return make_unique<Mixin>();
}
unique_ptr<Afficheur> CreerAfficheurDecoratif() {
   class Mixin
      : public Afficheur, ModeleDecoratif {
   };
   return make_unique<Mixin>();
}
unique_ptr<Afficheur> CreerAfficheurHTML() {
   class Mixin
      : public Afficheur, ModeleParagrapheHTML {
   };
   return make_unique<Mixin>();
}
```

Chaque *mixin* dérive publiquement de son théorème, pour que sa fonction génératrice puisse le traiter comme une spécialisation de son parent lors du `return`. Toutefois, comme le montre cet exemple, un *mixin* donné peut dériver de manière privée de ses modèles.

Contrairement aux apparences, un *mixin* n'est pas vide : il intègre tout le savoir opérationnel et structurel de ses parents.

Un programme de test un peu simple est proposé à droite. Il y est clair que le client n'a véritablement besoin d'y connaître que la classe théorème et le prototype des fonctions génératrices.

```
// ...
int main() {
   auto p = CreerAfficheurVanille();
   p->afficher("Coucou");
   p = CreerAfficheurDecoratif();
   p->afficher("Coucou");
   p = CreerAfficheurHTML();
   p->afficher ("Coucou");
}
```

Compiler cet exemple générera plusieurs fois un avertissement (pas une erreur) indiquant que chaque *mixin* hérite de certaines méthodes d'un parent ou de l'autre *par dominance*.

C'est normal : nous profitons ici du fait que le compilateur préférera une version implémentée par un parent d'une méthode à une version abstraite de la même méthode. Cet avertissement résulte directement de notre technique.

Mixins *et design*

Réfléchir en termes de *mixins* tend à mener les développeurs sur le chemin de la conception de bibliothèques de petits modules facilement assemblables. Plusieurs techniques associées à la programmation générique et aux foncteurs font d'ailleurs de même.

Une crainte répandue lors de la conception d'une bibliothèque de *mixins* tient de la difficulté à conceptualiser les classes requises. En général, mettre au point des objets vraiment réutilisables est plus difficile que mettre au point des objets à usage unique ou restreint; de même, concevoir une bibliothèque solide nécessite du temps et de la réflexion, surtout du point de vue du découpage et de la répartition des fonctionnalités entre les différentes classes.

L'avantage premier de la démarche, toutefois, est que le résultat est extrêmement simple à utiliser, et tend à être d'une grande efficacité (pour un objet polymorphique, du moins). Un objet bien pensé sera utilisé pendant une période de temps beaucoup plus longue que celle requise pour son développement.

Il existe peu de bibliothèques commerciales de *mixins* sur le marché, en partie pour des raisons historiques : C++, qui est un langage OO parmi les plus connus, n'offrait pas au départ l'héritage multiple. L'utilisation de *mixins* repose sur une implémentation solide du support à l'héritage multiple (et virtuel) à même le compilateur. Les bibliothèques en circulation depuis les débuts du langage n'utilisent donc pas la technique des *mixins*, celle-ci n'étant pas possible au moment de leur conception.

La force de l'habitude encourage les gens à utiliser les bibliothèques qu'ils connaissent bien, et à développer à de ce qu'ils connaissent. La réticence que les gens ont à appliquer l'héritage multiple est un obstacle du même acabit: même un langage à héritage simple permettant d'implémenter plusieurs interfaces, comme Java, ne suffit pas pour appliquer cette stratégie, puisque l'héritage simple empêche un *mixin* d'être constitué d'une combinaison de classes modèles et de classes théorèmes.

Même avec C++, beaucoup de compilateurs ont tardé à implanter l'héritage multiple et virtuel correctement et ont longtemps échoué dans leur tâche de reconnaître qu'un *mixin* n'est pas abstrait.

On peut contourner, en partie, les ennuis causés par l'absence d'héritage multiple par une combinaison d'héritage d'interfaces, permis en Java comme en C++, et d'un mélange composition/ enrobage appliqué aux attributs.

L'héritage d'interfaces sert dans les deux langages pour permettre le polymorphisme sur les interfaces (théorèmes), alors que l'encapsulation d'attributs par des méthodes d'enrobage règle le cas des classes modèles.

Le prix à payer, bien sûr, est le poids supplémentaire ajouté à la création d'un *mixin*, où la classe constructeur nécessite une composition d'attributs et la rédaction de méthodes d'enrobage.

Les développeurs désireux d'utiliser des *mixins* malgré un compilateur déficient en ce sens ont longtemps dû faire quelques acrobaties pour y arriver[85]. Ceci nuit à l'élégance manifeste de l'approche, et a représenté un obstacle à sa dissémination.

[85] La rustine à appliquer alors est de ne pas spécifier les méthodes des classes axiomes comme étant abstraites, mais simplement virtuelles (donc avec une définition vide). Ces méthodes représentent alors un *corps mort*, voué à ne jamais être utilisé, et nous prive de la protection naturelle découlant de l'usage de méthodes abstraites, au sens où le compilateur ne peut plus nous avertir dans les cas où une classe dérivée aurait négligé de surcharger une méthode d'une classe axiome, mais permet de contourner le problème et d'implanter les *mixin*.

Petit exemple concret de mixin

Ce qui suit est un petit exemple concret utilisant des *mixin*. L'exemple est proposé comme s'il était écrit dans un seul fichier source, mais vous pourrez le découper et le nettoyer à loisir si le cœur vous en dit.

L'implémentation proposée ici n'est pas sécuritaire face aux exceptions, mais peut le devenir avec un peu de soins.

À partir d'une classe axiome, on implante une stratégie de croissance pour des tableaux à taille dynamique, puis on compare deux *mixins* implantant chacun une stratégie différente d'accroissement de taille.

Vous remarquerez la facilité avec laquelle, une fois les axiomes et les théorèmes conçus, on a implanté les *mixins* utilisés dans le programme principal (voir plus bas).

Nous présumerons que l'idiome de classe incopiable, couvert dans [POOv01], est compris en assimilé. Les *mixin* étant des abstractions destinées à être utilisées par indirection, ce ne seront pas des types valeurs à part entière (voir toutefois l'exercice `EX00`, plus bas). Si le besoin de dupliquer un *mixin* se fait sentir, alors envisagez remplacer `Incopiable` par une infrastructure covariante de clonage.

La classe axiome sera `BaseTableauCar`. Cette classe propose des services pour un caractère de manière brute à un tableau dynamique, mais laisse une classe théorème implanter l'ajout avec croissance potentielle qui sera offert au code client.

Il y a un irritant d'implémentation ici du fait que l'attribut tableau_ est protégé, ce qui peut causer un cauchemar d'entretien. Voir `EX01` pour corriger ceci.

```
#include "Incopiable.h"
class Debordement { };
class BaseTableauCar : Incopiable {
public:
   using size_type = int;
   using value_type = char;
private:
   size_type cap {};
   size_type nelems {};
   int ncroiss {};
protected:
   value_type *tableau {};
   void ajouter_brut(value_type c) {
      if (!tableau || full())
         throw Debordement{};
      tableau[nelems++] = c;
   }
   void nouvelle_capacite(int capacite) {
      cap = capacite;
   }
public:
   size_type capacite() const noexcept {
      return cap;
   }
   size_type size() const noexcept {
      return nelems;
   }
   bool full() const noexcept {
      return size() == capacite();
   }
   bool empty() const noexcept {
      return size() == 0;
   }
   int nb_croissances() const noexcept {
      return ncroiss;
   }
```

Les classes modèles implanteront deux méthodes :

- la méthode `creer()`, qui a pour rôle de créer le tableau interne et de retourner sa capacité; et
- la méthode `croitre()`, qui accroît la capacité d'un tableau.

Vous remarquerez que `croitre()` met en application l'idiome `NVI`, couvert dans [POOv01]. La méthode publique, qui est concrète, encadre pour fins de statistiques une méthode polymorphique et protégée nomme `croitre_impl()`, et c'est cette dernière qui sera implémentée par les classes modèles.

Remarquez que le nombre de services publics offerts par cette classe est plutôt restreint, la majorité étant protégés ou privés (même le constructeur par défaut et le destructeur y sont protégés). La gamme de services publics est limitée à ce que le code client devrait pouvoir accéder directement, sans plus.

```
   virtual size_type creer() = 0;
   size_type croitre() {
      const auto nouv_cap = croitre_impl();
      ++ncroiss_
      return nouv_cap;
   }
   using iterator = value_type*;
   using const_iterator = const value_type*;
   iterator begin() noexcept {
      return tableau;
   }
   const_iterator begin() const noexcept {
      return tableau;
   }
   iterator end() noexcept {
      return begin() + size();
   }
   const_iterator end() const noexcept {
      return begin() + size();
   }
protected:
   virtual size_type croitre_impl() = 0;
   BaseTableauCar = default;
   virtual ~BaseTableauCar(){
      delete[] tableau;
   }
};
```

La classe théorème portera le nom convivial `TableauCar`, étant destinée à être utilisée par le code client. La classe théorème se veut l'abstraction utilisée en pratique pour invoquer les services du *mixin*.

Pour empêcher le code client de contourner cette décision de design et de la traitée comme un cas particulier de son parent, la classe théorème dérive de la classe axiome de manière privée (mais virtuelle), et les éléments du parent qu'elle souhaite exposer directement le sont à partir d'alias (à l'aide de `typedef`) et d'exposition volontaire (à l'aide de `using`).

L'algorithme général (et simpliste) d'ajout d'un élément au tableau existant et de redimensionnement de ce tableau au besoin est implémenté ici, bien que la stratégie de croissance (méthodes `Creer()` et `Croitre()`) soient implémentées dans les classes modèles. Les invocations de ces méthodes sont donc, ici, des appels frères.

```
struct TableauCar : virtual BaseTableauCar {
   using BaseTableauCar::capacite;
   using BaseTableauCar::nb_croissances;
   using value_type = BaseTableauCar::value_type;
   void ajouter(value_type c) {
      if (empty())
         nouvelle_capacite(creer());
      if (full())
         nouvelle_capacite(croitre());
      ajouter_brut(c);
   }
};
```

L'une de nos deux classes modèles, `CroissanceIncrementielle`, décrira une stratégie de croissance de la capacité du tableau par incréments de taille fixe. Ici, la taille de l'incrément de capacité sera une constante de classe, mais d'autres approches auraient été possibles (voir `EX02`).

L'approche incrémentielle est préférable à l'approche exponentielle (plus bas) si le tableau garde en général sa capacité originale, ou encore si elle ne varie que peu et par blocs de taille inférieure ou égale à la taille de l'incrément choisi.

Cette approche tend à économiser de la mémoire, car la capacité d'un tableau donné tend à être utilisée au complet.

Remarquez la dérivation protégée et virtuelle. Ceci permettra au *mixin*, la classe terminale, de profiter d'un lien avec la classe axiome.

```
#include <algorithm>
class CroissanceIncrementielle
   : protected virtual BaseTableauCar {
   static const size_type TAILLE_INITIALE,
                          TAILLE_INCREMENT;
   size_type creer() {
      tableau_ = new value_type[TAILLE_INITIALE];
      return TAILLE_INITIALE;
   }
   size_type croitre_impl() {
      using std::copy;
      const auto nouv_cap =
         capacite() + TAILLE_INCREMENT;
      auto p = new value_type[nouv_cap];
      copy(begin(), end(), p);
      delete[] tableau;
      tableau = p;
      return nouv_cap;
   }
};
const CroissanceIncrementielle::size_type
   CroissanceIncrementielle::TAILLE_INITIALE = 100;
const CroissanceIncrementielle::size_type
   CroissanceIncrementielle::TAILLE_INCREMENT = 50;
```

L'autre classe modèle de cet exemple, `CroissanceExponentielle`, décrira une stratégie de croissance de la capacité du tableau par un facteur de croissance.

Cette approche donnera de meilleurs résultats en pratique que la précédente (si le facteur de croissance est choisi avec soin) du fait que le redimensionnement, qui est une opération coûteuse, se produira moins souvent. Par contre, l'espace inoccupé dans le tableau sera en général plus grand ici que dans l'approche incrémentielle.

Le choix entre l'un ou l'autre des modèles proposés ici est un classique choix entre économie de temps d'exécution (approche exponentielle) et économie d'espace utilisé (approche incrémentielle).

Évidemment, la classe théorème est exposée au code client (la classe axiome aussi, par la force des choses) mais les classes modèles, elles, sont dissimulées dans un fichier source et livrées sous forme binaire.

```
class CroissanceExponentielle
   : protected virtual BaseTableauCar {
   static const size_type TAILLE_INITIALE,
                          FACTEUR_INCREMENT;
   size_type creer() {
      tableau_ = new value_type[TAILLE_INITIALE];
      return TAILLE_INITIALE;
   }
   size_type croitre_impl() {
      const size_type nouv_cap =
         cacacite()?
            capacite() * FACTEUR_INCREMENT :
            TAILLE_INITIALE;
      auto p = new value_type[nouv_cap];
      std::copy(begin(), end(), p);
      delete[] tableau;
      tableau = p;
      return nouv_cap;
   }
};
const CroissanceExponentielle::size_type
   CroissanceExponentielle::TAILLE_INITIALE = 100;
const CroissanceExponentielle::size_type
   CroissanceExponentielle::FACTEUR_INCREMENT = 2;
```

Les fonctions génératrices des *mixins* sont celles proposées à droite. Leurs prototypes seront exposés au code client mais leurs définitions seront cachées dans un module compilé. Ceci permettra de ne pas révéler les classes modèles (qui sont des parents privés du *mixin*, puisqu'il s'agit d'un détail d'implémentation).

```
unique_ptr<TableauCar> CreerTableauIncrementiel() {
   class Mixin
      : public TableauCar, CroissanceIncrementielle {
   };
   return make_unique<Mixin>();
}
unique_ptr<TableauCar> CreerTableauExponentiel() {
   class Mixin
      : public TableauCar, CroissanceExponentielle {
   };
   return make_unique<Mixin>();
}
```

Tel que mentionné à quelques reprises déjà, les *mixins* sont des classes anonymes. Chacune est définie localement à une fonction, par combinaison spontanée d'interface et d'implémentation(s), puis instanciée et retournée comme un cas particulier de son interface.

Le programme de test, écrit pour démontrer la facilité d'utilisation des *mixin*s, crée deux tableaux dynamiques distincts, chacun avec son propre modèle de croissance.

Un test simple est réalisé sur chacun (ajouter plusieurs fois le même caractère), puis les états de chacun sont comparés, illustrant pour chacun le nombre de redimensionnements réalisés et la capacité du tableau suite aux insertions.

Réflexion 02.3 : le code client se défait ici de la tâche de libérer les pointeurs sur les *mixins* à l'aide de pointeurs intelligents de la bibliothèque standard. Pourquoi recommande-t-on de faire en sorte que ces fonctions retournent des pointeurs intelligents plutôt que des pointeurs bruts? Réponse à l'***annexe 00***.

À la fin de l'exécution de ce programme, le tableau à stratégie incrémentielle aura une taille de 1000 caractères mais aura grossi 18 fois, alors que le *mixin* à stratégie exponentielle aura une taille de 1600 caractères mais aura grossi 4 fois seulement.

```
#include <iostream>
// ...using...
int main() {
   auto pInc = CreerTableauIncrementiel();
   auto pExp = CreerTableauExponentiel();
   const int NB_INSERTIONS = 1'000;
   for (int i = 0; i < NB_INSERTIONS; i++) {
      pInc->ajouter('A');
      pExp->ajouter('A');
   }
   cout << "Capacité finale de pInc: "
        << pInc->capacite() << '\n'
        << "Capacité finale de pExp: "
        << pExp->capacite() << "\n\n"
        << "Nb accroissements de pInc: "
        << pInc->nb_croissances() << '\n'
        << "Nb accroissements de pExp: "
        << pExp->nb_croissances() << endl;
}
```

L'emploi stratégique de conception par mixin vient, nous le constatons, de déplacer le problème de **programmation** vers un problème de **gestion** : gestion des ressources en temps et en espace, selon les besoins du système à développer.

Exercices – Série 05

EX00 – Créez un type `RAII` encapsulant l'abstraction de *mixin* `Afficheur` proposée un peu plus haut.

EX01 – Dans l'implémentation par *mixin* de `BaseTableauCar`, plus haut, l'attribut `tableau_` est considéré protégé plutôt que privé, ce qui compliquera grandement son entretien. Que devrait-on considérer pour corriger cet irritant? Établissez une démarche claire et mettez-la en application.

EX02 – Dans l'implémentation par *mixin* d'un tableau dynamique, modifiez la stratégie de croissance incrémentielle pour que la taille de l'incrément soit déterminée par le code client lors de la construction du *mixin*. Quelles sont les modifications que vous devez apporter au code existant pour y arriver? Est-il possible de faire ces modifications sans affecter le code client existant? Comment devrait-on gérer les erreurs (par exemple une demande d'une croissance par incréments négatifs ou nuls)?

EX03 – **Travail de plus longue haleine** : adaptez l'implémentation par *mixin* d'un tableau dynamique proposée dans cette section pour qu'elle devienne sécuritaire même lorsque des exceptions sont levées.

Singletons

Il peut arriver qu'on ait besoin de classes à instanciation unique. Par ***instanciation unique***, on entend ici une classe qu'il est *impossible* d'instancier plus d'une fois dans un programme donné.

Aussi étrange que cela puisse apparaître, il est relativement fréquent en pratique qu'on ait recours à des classes ne pouvant (ou ne devant) être instanciées qu'une seule fois par programme. C'est même l'un des schémas de conception les plus communs (et des plus malmenés par des implémentations déficientes).

⇒ On nomme **singleton** une classe qui ne peut être instanciée qu'une seule fois pour un programme donné. On utilisera souvent, de manière un peu abusive, le mot singleton pour désigner à la fois la classe à laquelle le schéma de conception est appliqué et l'instance unique de cette classe.

L'idée de base derrière ce schéma de conception est de responsabiliser le singleton dans la gestion de l'unicité de sa propre instanciation. Si `X` doit être un singleton, alors `X` sera responsable de s'en assurer. Nous ne ferons pas reposer sur les épaules du code client la tâche d'assurer l'instanciation unique de la classe; au contraire, nous mettrons en place des mécanismes qui empêcheront les programmes de contourner notre choix de design.

Parmi les raisons typiques entraînant un recours à cette technique, on trouve entre autres :

- la conception d'une classe représentant un composant matériel. Une telle classe peut être limitée à une seule instanciation pour éviter des conflits de gestion des ressources;
- un objet devant offrir certaines garanties pour le programme, comme par exemple un générateur de nombres entiers uniques. Le singleton trouvait son utilité du fait que la multiplicité des instances pourrait compromettre l'unicité des nombres générés;
- un objet qui servira de gestionnaire pour d'autres objets, question d'automatiser certains mécanismes d'un programme.

Implémenter un singleton est relativement simple, à tout le moins en apparence. Nous examinerons ici plusieurs manières différentes de procéder, examinant au passage les avantages et les inconvénients de chacune.

Les programmes qui font affaire avec un singleton, surtout s'il s'agit d'un gestionnaire de ressources, sont souvent faits de plusieurs fils d'exécution concurrents, ou *threads*. Les problèmes de concurrence, inhérents à la multiprogrammation, sont exacerbés du fait que le singleton, de par son unicité, devient un goulot d'étranglement. Nous ne couvrirons pas la multiprogrammation dans ce document, mais retenez cet appel à la prudence : dans un logiciel commercial multiprogrammé, les singletons doivent être sécurisés convenablement.

Quelques ébauches incorrectes

Avant de procéder avec des solutions convenables, prenons le temps d'examiner quelques stratégies simples mais naïves, qui peuvent paraître correctes à première vue mais ne le sont pas (et doivent, par conséquent, être évitées).

Dans l'ébauche proposée à droite, un attribut public de classe, nommé `X::singleton`[86], est une instance spécifique de `X`.

Il est légal de définir un attribut *de classe* de la classe `X` qui soit une instance de `X`, puisque l'attribut de classe n'est pas dans une instance. Il serait par contre illégal de définir un attribut *d'instance* de type `X` dans une instance de la classe `X` : cela provoquerait, par récurrence, un objet de taille infinie.

On pourrait, à tort, penser que le dossier est clos, or nous n'avons pas de garantie que cette instance sera la *seule* instance de `X` dans un programme. En effet, rien n'empêcherait ici un programme de se déclarer une autre instance de `X`.

On peut souhaiter que les programmeuses et les programmeurs respectent notre souhait de ne pas instancier `X` plus d'une fois mais rien ne les *contraint* à respecter ce souhait.

```
#ifndef X_H
#define X_H
struct X {
   static X singleton;
   // ...
};
#endif
```

```
// dans un fichier source
#include "X.h"
X X::singleton;
// ...
```

Cette ébauche a l'important défaut de briser l'encapsulation de `X`. Le code client, avec cette version, accéderait directement à une donnée qui est, en quelque sorte, globale au programme; cette implémentation serait un désastre.

Un premier raffinement serait d'encapsuler notre tentative de singleton dans une méthode de classe qui retournerait une référence (ou un pointeur, selon les préférences) sur lui. Cependant, rien dans cette écriture n'empêche un programme client d'instancier à volonté des `X`. Ici, nous pouvons garantir qu'il existera au moins un `X` dans le programme, mais nous ne pouvons pas garantir qu'il y aura un seul `X` dans le programme.

Réflexion `02.4` : faut-il absolument que `get()` soit une méthode de classe? Réponse à la section ***Réflexion `02.4` : instances et méthodes de classe***.

```
class X {
   static X singleton;
public:
   static X& get() noexcept {
      return singleton;
   }
   // ...
};
```

```
// dans un fichier source
#include "X.h"
X X::singleton;
```

Notez que j'ai nommé `get()` la méthode de classe donnant accès au singleton en tant que tel; j'aime bien cette écriture, concise et directe. Dans la littérature, les écritures `getInstance()` ou `GetInstance()` sont les plus fréquemment rencontrées. Notez aussi que `get` est un mot réservé en C# et ne serait donc pas adéquat dans ce cas.

[86] Le nom `singleton` choisi pour l'attribut est une simple convention. Ce n'est pas un mot réservé ou quoi que ce soit du genre.

Une autre approche à éviter (mais qu'on rencontre fréquemment, malgré tout) est de remplacer le singleton par une classe n'ayant que des membres de classes.

L'attrait de cette approche est sa simplicité apparente. Plutôt que d'être un singleton, la classe résultante devient un regroupement de variables globales et de fonctions globales, avec en prime un peu d'encapsulation.

Pourtant, cette approche comporte plusieurs inconvénients. L'un des principaux inconvénients, d'ailleurs, est qu'elle nous prive de ces précieux services que sont le constructeur et (surtout!) le destructeur.

```
class X { // beurk
   static int attrib;
public:
   static int f();
};
```

En effet, les moments névralgiques que sont la construction et la destruction d'un objet font partie des raisons pour lesquels un singleton peut être avantageux en comparaison avec des fonctions globales ou un simple regroupement de méthodes de classe :

- à la construction, un singleton peut réaliser des opérations d'initialisation arbitrairement complexes, s'attribuer des ressources particulières, réaliser des tests, charger des modules externes et ainsi de suite; puis
- lors de la destruction, un singleton peut s'assurer que le code sous sa gouverne libère les ressources qu'il s'est attribué au cours de son existence et faire en sorte que le système ne conserve pas de traces indésirables de son passage.

Si nous cherchons à implémenter des services globaux sans singleton, ou si nous implémentons de manière incorrecte ce schéma de conception, il sera possible de remplacer en partie le constructeur par des variables globales et des tests pour valider si celles-ci ont été initialisées ou non, mais il sera impossible de garantir[87] le nettoyage en fin de parcours.

Certains seraient tentés de débuter par de simples regroupements, puis de migrer vers des singletons à partir du moment où le besoin de code d'initialisation ou de code de nettoyage pointe son nez. Sur le plan de l'évolutivité du code, procéder ainsi est une erreur, puisque cela forcerait des modifications au code client pour l'adapter au changement d'approche. Mieux vaut planifier une version complète du schéma de conception dès le début, même lorsque le constructeur et le destructeur ne servent que peu ou pas du tout.

[87] Ceci ne veut pas dire que le code ne pourra pas être nettoyé. Une fonction ou une méthode spéciale de nettoyage pourra être définie et invoquée par le code client, mais il n'est pas possible, de manière générale, de garantir que le code client l'invoquera systématiquement, correctement ou au bon moment.

Une autre démarche possible serait de reconnaître, dans le constructeur du singleton, qu'une instance de cette classe a déjà été construite, puis de lever une exception si c'est le cas.

Bien que cette approche puisse fonctionner, elle est loin d'être idéale. En effet :

- elle ne résout pas le problème de l'accès à l'instance unique de cette classe. Si le singleton est un gestionnaire utile au programme dans son ensemble, invoquer ses services devient une tâche artisanale;
- elle repose sur un traitement d'erreur à l'exécution, ce qui complique les programmes. Les tentatives menant à plus d'une instanciation d'un singleton sont préférablement résolues à la compilation;
- la gestion des opérations de copie et de la destruction devient un problème complexe : comment gérer une séquence de constructions et de destructions successives du singleton?

```
#ifndef X_H
#define X_H
class X {
   static int ninstances;
public:
   class ExisteDeja {};
   X() {
       if (ninstances > 0)
         throw ExisteDeja{};
       ++ninstances;
      // ...
   }
};
#endif
```

```
#include "X.h"
int X::ninstances = 0;
```

Les diverses options que nous allons estimer viables pour concevoir un singleton, plus bas, évitent toutes ces écueils, chacune à sa manière.

Assurer l'instanciation unique

Plusieurs stratégies sont possibles pour empêcher une classe d'être instanciée plus d'une fois dans un programme, mais toutes ces stratégies ont en commun un certain nombre d'éléments, parmi lesquels on trouve :

- le recours à des constructeurs privés. Ceci empêche le code client d'accéder aux constructeurs et d'instancier lui-même la classe destinée à être un singleton. Utiliser des constructeurs privés empêche aussi la mise en place de classes dérivées du singleton, évitant un cas un peu pervers de duplication auquel trop de développeurs ne pensent pas[88];
- des mécanismes pour empêcher la duplication du singleton. Souvenons-nous que C++ offre une implémentation par défaut des mécanismes de copie que son constructeur de copie et l'opérateur d'affectation. Nous voudrons bloquer ce comportement; pour ce faire, nous appliquerons l'idiome de classe incopiable [POOv01]);
- un destructeur public, pour des raisons techniques;
- un mécanisme interne à la classe pour définir le singleton. Ce mécanisme reposera souvent, mais pas toujours, sur un attribut de classe; et
- une stratégie d'encapsulation du singleton, pour contrôler les accès à cet objet et éviter les accidents en situation de multiprogrammation.

[88] Si `D0` et `D1` dérivent de `B`, alors un `D0` est un `B` et un `D1` est aussi un `B`. Si `B` doit être un singleton et si on trouve un `D0` et un `D1` dans un programme, alors l'objectif d'unicité n'a pas été atteint.

Approche 0 – membre de classe, instanciation sur demande

Une première approche possible pour appliquer ce schéma de conception est celle d'allouer dynamiquement un singleton lors de la première tentative d'avoir recours à ses services. Avec cette approche, l'instance sera instanciée dynamiquement, seulement si elle est véritablement utilisée, et son adresse sera affectée à un attribut de classe (initialisé à zéro).

L'instanciation est faite lors d'un appel à une méthode (ici : `get()`). Seul le premier appel à cette méthode résultera en une instanciation, évidemment (dans le cas contraire, ce ne serait pas un singleton). Les appels subséquents réutiliseront tous l'instance initialement construite.

Chaque invocation de la méthode `get()` nécessite l'évaluation d'une condition : on y vérifie à chaque fois si l'instanciation a été faite ou non[89]. Si un programme sait qu'il *devra* instancier le singleton, quoiqu'il arrive, ou si le singleton est peu coûteux en ressources, le temps utilisé à valider que l'instanciation ait eu lieu (ou non) est une perte sèche, et on privilégiera souvent d'autres stratégies.

En pratique, cette solution provoque une condition de course dans un programme à plusieurs *threads*. Investiguez `std::call_once` si telle est votre situation.

```
#ifndef X_H
#define X_H
class X : Incopiable {
   static X *singleton;
   X() { // constructeur privé
      // ...
   }
public:
   ~X() { // destructeur public
      // ...
   }
   static X *get() {
      if (!singleton)
         singleton = new X;
      return singleton;
   }
   // ...
};
#endif
```

```
#include "X.h"
X *X::singleton = nullptr;
// ...
```

Sous la forme proposée ici, cette approche a un important défaut : le destructeur du singleton n'est jamais appelé, ce qui est susceptible d'entraîner des pertes de ressources. Ceci ne sera pas un problème si votre langage repose sur une collecte automatique d'ordures et si la finalisation de votre singleton est banale, mais pourra causer de sérieux problèmes si l'une de ces deux considérations n'est pas rencontrée. Nous expliquerons ce phénomène plus en détail à l'approche 1, plus bas.

[89] En situation multiprogrammée, la petite implémentation de `get()` proposée ici serait trop naïve et entraînerait une condition de course. Plusieurs techniques sont connues pour régler ce problème; voir http://www.cs.wustl.edu/~schmidt/editorial-3.html et http://en.wikipedia.org/wiki/Singleton_pattern.

Approche 1 – membre de classe, instanciation au démarrage

Cette approche ressemble à la précédente, à ceci près que l'instanciation du singleton se fait plutôt au démarrage du programme, avant la première instruction de `main()`.

Avec cette solution, chaque appel à `get()` sera plus économique (en temps) que dans l'approche `0`, mais le `singleton` sera instancié, qu'on s'en serve ou non. Ainsi, l'approche `0` sera préférable à l'approche `1` si le singleton est coûteux en ressources ou susceptible de ne pas être utilisé, mais le contraire sera vrai si le singleton est très susceptible d'être utilisé puisque chaque accès au singleton sera plus rapide.

Comme dans le cas de l'approche `0`, le destructeur du singleton ne sera jamais détruit si cette approche est choisie. La mémoire associée au singleton sera libérée à la mort du programme, bien entendu, mais le destructeur ne sera pas invoqué.

Il existe des cas où il est important d'instancier une classe dynamiquement, et où la finalisation correcte est essentielle. Heureusement, il y a une solution, reposant sur des classes imbriquées, qui permet de concilier les deux.

Nous y reviendrons un peu plus loin, dans la section ***Automatiser la destruction d'un singleton dynamique***.

```
#ifndef X_H
#define X_H
class X : Incopiable {
   static X *singleton;
   X() {
      // ...
   }
public:
   ~X() {
      // ...
   }
   static X *get() {
      return singleton;
   }
   // ...
};
#endif
```

```
#include "X.h"
X *X::singleton = new X;
// ...
```

Approche 2 – membre de classe, instanciation statique

Cette approche est identique à la précédente, à ceci près que le singleton n'y est pas alloué dynamiquement. Bien que le singleton y soit en mémoire statique, il est essentiel de ne donner vers lui qu'un accès indirect, par adresse ou par référence (voir les deux exemples à droite) : si nous retournions une copie du singleton, alors ce ne serait plus un singleton puisqu'il y en aurait au moins deux instances dans le programme.

Personnellement, je privilégie la référence lorsque cela s'avère possible. Cela simplifie l'écriture du code client sans diminuer la qualité du schéma de conception ou la vitesse du programme.

Avec cette approche, le singleton sera instancié au lancement du programme, donc avant le début de l'exécution de `main()`, et sera détruit à la fin de l'exécution du programme, suite à la fin de l'exécution de `main()`.

```
#ifndef X_H
#define X_H
class X : Incopiable {
   static X sing;
   X() {
      // ...
   }
public:
   ~X() {
      // ...
   }
   static X *get() {
      return &sing;
   }
   // ...
};
#endif
```

```
#include "X.h"
X X::sing;
// ...
```

```
#ifndef X_H
#define X_H
class X : Incopiable {
   static X sing;
   X() {
      // ...
   }
public:
   ~X() {
      // ...
   }
   static X &get() {
      return sing;
   }
   // ...
};
#endif
```

```
#include "X.h"
X X::sing;
// ...
```

Le singleton est, selon cette approche, une sorte de variable globale mais encapsulée et sécurisée. Le compilateur prend en charge sa construction et sa destruction. Ceci explique pourquoi le singleton doit exposer un destructeur public : s'il ne l'était pas, alors le code client (le programme) n'y aurait pas accès.

Approche 3 – variable statique locale à une méthode

Avec cette approche, attribuée à ***Scott Meyers***, le singleton ne vivra en quelque sorte réellement que dans la méthode qui y donne accès.

Le singleton, étant statique et local à `get()`, sera instancié lors du premier appel à cette méthode, comme dans l'approche `0` proposée plus haut. Par conséquent, si le code client n'a pas recours au singleton, alors celui-ci ne sera jamais instancié.

Encore une fois, il est possible (au choix) de retourner une référence ou un pointeur sur le singleton. Dans chaque cas, l'adresse rendue disponible restera valide tout au cours de l'exécution du programme parce que la variable est statique (au sens qu'a ce mot dans le langage C[90]).

```
class X : Incopiable {
   X() {// constructeur privé
      // ...
   }
public:
   ~X() {
      // ...
   }
   static X &get() {
      // instanciation au premier
      // appel seulement
      static X singleton;
      return singleton;
   }
   // ...
};
```

Implémentation semblable mais avec quelques nuances : une classe générique `Singleton` sur la base du type voué au rôle de singleton. La classe `Singleton` sera `Incopiable`, et il en sera de même pour ses enfants.

La méthode `get()` déclarera localement (et de manière statique) le singleton en tant que tel.

La classe qui offrira les services de singleton, avec cette approche, sera un dérivé privé de `Singleton` *appliqué à elle-même*.

Elle se dira amie de son parent pour que `Singleton` appliqué à elle puisse l'instancier, elle, et qualifiera son constructeur par défaut de privé pour que seul sont parent (et ami), en plus d'elle-même évidemment, y ait accès.

Cette stratégie a l'intérêt d'être explicite : un singleton *est*, clairement, un singleton, en vertu de la relation d'héritage. Le code client en fait foi, comme le montre notre exemple.

```
template <class T>
   class Singleton : Incopiable {
   public:
      static T& get() {
         static T singleton;
         return singleton;
      }
   };
class X : public Singleton<X> {
   friend class Singleton<X>;
   X() = default;
public:
   int f() const {
      return 3;
   }
};
int main() {
   auto i = Singleton<X>::get().f();
}
```

[90] Cette variable est localisée au même endroit en mémoire que le sont les variables globales. Elle est initialisée au premier appel du sous-programme qui la déclare, et garde sa valeur (pour un objet : son état) d'un appel à l'autre (pour un objet : n'est pas détruite, ou reconstruite, avant la fin de l'exécution du programme).

Singletons statiques et singletons dynamiques

Vous remarquerez que les approches `0` et `1` reposent sur une allocation dynamique du singleton, donc sur une gestion de la vie du singleton placée sous la responsabilité explicite du programme, alors que les approches `2` et `3` reposent quant à elles sur une mécanique d'allocation statique, sous la responsabilité implicite du compilateur.

J'abrégerai par *singletons dynamiques* et par *singletons statiques*, respectivement, ces deux petites familles d'approches. En Java comme dans les langages `.NET`, seuls les singletons dynamiques sont possibles.

Il existe des écoles de pensées appuyant chacune de ces familles. En pratique, chacune a des avantages et des inconvénients et il est préférable d'éviter une prise de position dogmatique. Un traitement détaillé de ces deux philosophies dépasse ce que nous sommes en mesure de faire pour le moment.

Singletons et interdépendances

Certaines stratégies de singletons, incluant les approches `0` et `3` ci-dessus, créent le singleton lors de la première demande faite pour ses services. D'autres, comme par exemple les approches `1` et `2`, créent le singleton avant même le démarrage du *thread* principal d'un programme (de la fonction `main()`).

Les approches `2` et `3` invoquent éventuellement le destructeur du singleton suite à la fin de l'exécution de la fonction `main()`; il est aussi possible d'adapter les approches `0` et `1` en ce sens, comme nous le verrons dans la section ***Automatiser la destruction d'un singleton dynamique***, ci-dessous.

Même si la construction et la destruction d'un singleton sont toutes deux automatisées, quelques inconnues demeurent. Prenons par exemple un cas où plus d'un singleton se trouve créé avant même le démarrage du programme, comme c'est le cas pour les classes `A`, `B` et `C` à droite. En particulier, imaginez que chacune de ces classes soit dans sa propre paire de fichiers `.h`/ `.cpp`.

Dans quel ordre seront instanciés `A::singleton`, `B::singleton` et `C::singleton`?

Dans l'ordre selon lequel ils sont déclarés? Dans l'ordre inverse de leur déclaration? Dans l'ordre selon lequel ils sont définis? Dans l'ordre inverse de leur définition? En ordre alphabétique? Dans un tout autre ordre?

Que signifierait d'ailleurs l'ordre de leur déclaration? Après tout, chaque fichier source inclut les fichiers d'en-tête dont il a besoin, alors d'un fichier source à l'autre, l'ordre de déclaration de ces singletons pourrait différer. L'ordre de définition, lui, relève de l'ordre selon lequel l'éditeur de liens intégrera ensembles les divers modules objets, fruit de la compilation des fichiers sources; cette donnée échappe complètement au programme en soi.

Il est difficile de prédire l'ordre de construction des variables globales. De même, il est imprudent (on pourrait d'ailleurs utiliser sans gêne ici le mot *dangereux*) d'écrire du code qui dépende de l'ordre de ces constructions.

```
// A.h...
class A : Incopiable {
   static A singleton;
   // ...
public:
   static A &get() noexcept {
      return singleton;
   }
   // ...
};
// A.cpp...
#include "A.h"
A A::singleton;
// ...
// B.h...
class B : Incopiable {
   static B singleton;
   // ...
public:
   static B &get() noexcept {
      return singleton;
   }
   // ...
};
// B.cpp...
#include "B.h"
B B::singleton;
// ...
// C.h...
class C : Incopiable {
   static C singleton;
   // ...
public:
   static C &get() noexcept {
      return singleton;
   }
   // ...
};
// C.cpp...
#include "C.h"
C C::singleton;
// ...
```

Pourtant, l'une des utilités potentielles des singletons est d'automatiser certaines initialisations et certains monceaux de code de terminaison ou de nettoyage. Visiblement, un problème conceptuel, qui devient vite un problème technique, survient du moment que la construction d'un singleton global donné dépend de la construction d'un autre.

Un cas simple serait celui-ci : la fonction `std::srand()` initialise le générateur de nombres pseudo-aléatoires standard du langage C, et ne devrait être invoquée qu'une seule fois par programme, avant toute génération de nombres pseudo-aléatoires. L'appel initial de `std::srand()` influence l'exécution subséquente de la fonction `std::rand()` qui, elle, génère des entiers pseudo-aléatoires (situés inclusivement entre 0 et la constante `RAND_MAX`).

Invoquer la fonction `std::srand()` dans le constructeur d'un singleton statique permet d'automatiser son invocation, et évite de faire reposer sur les épaules du code client des étapes d'initialisation qui, en vertu de l'encapsulation, sont vraiment à leur place dans le code serveur, donc du côté des objets eux-mêmes.

Imaginons toutefois que d'autres singletons d'un même programme utilisent `std::rand()` dans la réalisation de leurs tâches. Si tel est le cas, alors il est essentiel que celui invoquant `std::srand()` soit construit avant les autres, or (nous venons de le constater) il est imprudent, en général, d'écrire du code dépendant de l'ordre d'initialisation d'objets globaux.

L'approche `3`, plus haut, résout ce problème en instanciant les singletons lors de leur première utilisation, ce qui fonctionne bien dans la mesure où il n'existe pas de dépendance circulaire à la construction des singletons (et, s'il existe une telle dépendance, alors le problème en est un de design, pas de technique). L'approche `3` ne résout toutefois pas le problème de l'ordre de leur destruction, qui demeure imprévisible.

Si un singleton représente un outil de journalisation, par exemple, et si ce singleton doit se fermer après tous les autres (pour que les autres en question puissent y écrire pendant leur propre destruction), alors il existe une dépendance quant à l'ordre de destruction des singletons.

Certains spécialistes, et pas les moindres[91], se sont frottés à ce problème et sont arrivés à des solutions moins que satisfaisantes.

Que devrait-on en déduire? Voici :

Il est possible (contrairement à ce que plusieurs pensent) de contrôler la création de singletons statiques avant l'exécution du programme. Le code pour y arriver est subtil et repose fortement sur des stratégies de programmation générique[92].

- que, dans la plupart des cas, les singletons devraient être indépendants les uns des autres; et
- que dans le cas où une dépendance existe entre au moins deux singletons globaux, il faut avoir recours à d'autres mécanismes pour les coordonner. On a recours dans ce cas à des mécanismes que contrôlera le programme.

Quand une situation de dépendance à la construction ou à la destruction des singletons survient, la solution simple est souvent d'éliminer complètement les singletons et de créer un gestionnaire de démarrage, lui-même singleton, qui sera responsable de créer les objets qui auraient pu être des singletons globaux, dans un ordre respectant leurs contraintes de dépendance, et qui les détruira en ordre inverse.

---

[91] ***Andrei Alexandrescu*** consacre un chapitre de [ModCppD] à ce problème.

[92] Voir [hdSingSt] si vous en avez envie mais c'est un texte très complexe.

Une solution reposant sur un tel gestionnaire est à la fois statique (l'ordre étant déterminé à la compilation, par design) et spécifique à chaque projet. Une solution dynamique est aussi possible, à partir d'une adaptation de l'approche 3 et d'un système basé sur une pile (pensez-y, c'est un exercice intéressant).

Enfin, il faut être prudent avec l'utilisation d'objets standards à l'intérieur des constructeurs et des destructeurs des singletons. En effet, certains objets de la bibliothèque standard, comme par exemple les flux standards tels `std::cin` ou `std::cout`, peuvent ne pas être construits avant qu'un singleton donné ne soit lui-même construit, et il n'est pas non plus garanti qu'ils ne soient pas encore détruits au moment où le singleton sera détruit.

En pratique, sur l'immense majorité des compilateurs, l'ordre de destruction des objets globaux sera l'inverse de l'ordre de construction. Par contre, l'ordre de construction dépend de considérations qui dépassent le code source, par exemple l'ordre d'apparition des modules compilés lors de l'édition des liens. Il n'est pas sage d'écrire des programmes dépendant de considérations sur lesquelles on ne peut exercer aucun contrôle.

Une solution peut élégante mais relativement simple au problème de l'interdépendance entre « singletons » (car ce ne sont pas exactement des singletons au sens classique du terme) se déclinerait comme suit :

- supposons un programme dans lequel l'ordre de construction et de destruction des « singletons » `SingX` et `SingY` doit être très contrôlé avec soin (typiquement, présumons que l'un doive utiliser les services de l'autre pendant sa construction ou pendant sa destruction);
- pour les fins de notre discussion, les importants services de `SingX` et `SingY` se nomment respectivement `f()` et `g()`, et les interdépendances résideront dans notre imagination;
- remarquez l'injection d'une classe amie, nommée `Ges` (pour « *Gestionnaire de simili singletons* » dans les déclarations des deux classes susmentionnées. C'est cet élément de design qui fera fonctionner le tout : ici, ni `SingX`, ni `SingY` n'assurent leur propre construction ou leur propre destruction, et personne ne peut les instancier outre un ami…

```
#include "Incopiable.h"
class SingX : Incopiable {
   friend class Ges;
   SingX() = default;
   ~SingX() = default;
public:
   int f() const {
      return 3;
   }
};
class SingY : Incopiable
{
   friend class Ges;
   SingY() = default;
   ~SingY() = default;
public:
   int g() const {
      return 4;
   }
};
```

- nous faisons donc en sorte que les « singletons » soient en fait des classes qui ne peuvent être instanciées que par un tiers très précis, qui soit lui-même un singleton (dans l'exemple à droite, on parle du singleton `Ges`). Ceci se fait typiquement à l'aide de l'amitié (mot clé `friend`);
- nous faisons en sorte que ce tiers instancie les « singletons » dans un ordre qui nous convient (ici, la méthode privée `Ges::creer()` fait ce travail à l'aide d'instanciation dynamique);
- nous faisons en sorte que ce tiers détruise dans un ordre qui vous convient les « singletons » (ici, la méthode privée `Ges::detruire()` est utilisée à cette fin); et
- nous offrons un service permettant d'obtenir les « singletons » désirés de manière homogène à partir du tiers spécialisé (ici, il s'agit de la méthode générique `Ges::obtenir<T>()`).

Sans être très élégante ni très facile à transporter d'un projet à l'autre, cette approche a le mérite d'être simple à implémenter.

Cette implémentation est à risque pour un programme à plusieurs *threads*, et requiert de la synchronisation.

```
class Ges : Incopiable {
   SingX *pX;
   SingY *pY;
   void creer() {
      pY = new SingY;
      try {
         pX = new SingX;
      } catch (...) {
         delete pY;
         throw;
      }
   }
   void detruire() noexcept {
      delete pX;
      delete pY;
   }
   Ges() {
      creer();
   }
public:
   static Ges &get() {
      static Ges sing;
      return sing;
   }
   template <class T>
      static T & obtenir();
   template <>
      static SingX& obtenir<SingX>() {
         return *(get().pX);
      }
   template <>
      static SingY& obtenir<SingY>() {
         return *(get().pY);
      }
   ~Ges() {
      detruire();
   }
};
int main() {
   auto i = Ges::get().obtenir<SingX>().f();
}
```

Automatiser la destruction d'un singleton dynamique

Les approches `0` et `1`, prises telles quelles, ont le défaut de créer un objet (le singleton) dynamiquement (avec `new`) sans garantir sa destruction correcte (avec `delete`). Ces approches sont typiques du monde pris en charge, avec collecte automatique d'ordures, mais ne se prêtent pas, quel que soit le langage, à la mise en place de singletons pour lesquels il existe un besoin de finalisation. Dans les langages pris en charge, la finalisation gérée par l'objet à finaliser est rarement possible, ce qui explique que ce problème soit souvent oublié ou remis, de manière un peu cavalière, entre les mains du code client.

Même en C++, il arrive qu'on souhaite vraiment utiliser `new` pour créer un objet, car cet opérateur peut être surchargé et servir entre autres à placer l'objet ainsi créé à un endroit spécifique en mémoire (par exemple sur un morceau de matériel précis).

La surcharge des opérateurs `new` et de `delete` sera couverte dans [POOv03], section ***Gestion avancée de la mémoire***.

Une solution capable d'assurer à la fois l'emploi de `new` pour créer le singleton, la bonne construction du singleton et sa bonne destruction repose sur l'utilisation d'une classe interne, ou imbriquée (voir la section **Erreur ! Source du r envoi introuvable.**, plus loin).

Le singleton y est *indirect*; voyons pourquoi. Avec cette approche, on déclare une classe interne et privée dans le singleton (classe nommée ici `X::GesX`, au sens de *gestionnaire du singleton `X`*).

Toute instance de cette classe possédera un attribut de type `X*`, créé dynamiquement dans son constructeur et détruit dynamiquement dans son destructeur. La vie de cet objet interne chapeautera la mécanique de construction et de destruction dynamique du singleton.

Dans la classe `X`, l'attribut de classe deviendra un `X::GesX`, nommé `X::gX`. La méthode d'accès au singleton exposée par `X`, `X::GetInstance()`, deviendra un relais indirect vers celle de `X::gX`.

L'objet `X::gX` sera construit et détruit de manière statique, et on utilisera sa mécanique de construction/ destruction symétrique pour automatiser l'application symétrique de `new` et de `delete` sur le singleton véritablement désiré.

```
class X : Incopiable {
   class GesX {
      X *singleton;
   public:
      GesX() : singleton{new X} {
      }
      ~GesX() {
         delete singleton;
      }
      X *get() noexcept {
         return singleton;
      }
   };
   static GesX gX;
   X() { /* ... */ }
public:
   // accès indirect
   static X *get() noexcept {
      return gX.get();
   }
};
// instanciation du singleton
X::GesX X::gX;
```

Exemple d'utilisation d'un singleton

Une application simple du schéma de conception Singleton pourrait être un outil automatisant la création d'un fichier permettant d'écrire des événements pour un programme entier. Chaque écriture dans ce fichier serait ajoutée à la fin, mais le fichier serait vidé à chaque démarrage de programme, avant même le début de l'exécution de la fonction `main()`.

Notre exemple utilisera deux objets :

- un singleton statique, dont le constructeur sera appelé avant `main()` et créera le fichier; et
- une classe `Sortie` dont le constructeur ouvrira le fichier en sortie en mode ajout, dont le destructeur fermera ce fichier, qui offrira une méthode `flux()` retournant une référence au flux en sortie et qu'on instanciera localement dans les sous-programmes où on voudra générer une trace.

Un programme exploitant cette automatisation suit. On peut faire encore un peu mieux, d'ailleurs[93], mais ceci vous est laissé en exercice.

```
#include "Incopiable.h"
#include <fstream>  // std::ios_base::app
#include <string>
#include <iostream>
using namespace std;
// Le fichier dans lequel nous allons écrire
const string FICHIER_SORTIE = "Sortie.txt";
class CreateurSortie : Incopiable {
   static CreateurSortie singleton;
   // Constructeur privé. Crée et ferme le fichier en sortie
   CreateurSortie() {
      // création puis détruction par une temporaire anonyme
      ofstream{FICHIER_SORTIE};
   }
public:
   static CreateurSortie& get() noexcept {
      return singleton;
   }
   // service pour aider le code client (rendre privé si Sortie est friend...)
   void ouvrir(ofstream &os) {
      if (!os) os.open(FICHIER_SORTIE, std::ios_base::app);
   }
};
```

[93] Entre autres en implémentant l'opérateur de projection sur un flux de la classe `Sortie` et en définissant `Sortie` comme amie (`friend`) de `CreateurSortie`, ce qui permettrait de réduire la surface d'exposition de la méthode `ouvrir()`. Nous y reviendrons plus loin dans la section ***Amitié***.

```
// Définition du singleton.
CreateurSortie CreateurSortie::singleton;
class Sortie {
   ofstream sortie;
public:
   Sortie() {
      CreateurSortie::get().ouvrir(sortie);
   }
   ofstream& flux() {
      return sortie;
   }
}; // le destructeur fermera le fichier
template <class T>
   void permuter(T&, T&);
int main() {
   // Pas besoin de créer le fichier; le singleton
   // s'en est chargé au lancement du programme
   int x, y;
   while (cin >> x >> y) {
      permuter(x, y);
      cout << x << ' ' << y << endl;
   }
}
template <class T>
   void permuter(T &x, T &y) {
      // Création d'un flux en mode ajout. Pas besoin de connaître le nom du fichier
      // en sortie ou celui des constantes utilisées dans l'ouverture en mode ajout.
      Sortie s;
      auto & flux = s.flux();
      flux << "Début de l'appel à permuter(), x = " << x << ", y = " << y << endl;
      using std::swap;
      swap(x, y);
      flux << "Fin de l'appel à permuter(), x = " << x << ", y = " << y << endl;
   }
```

Dans d'autres langages

En Java, l'une des implémentations le plus souvent rencontrées du schéma de conception Singleton ressemble à l'approche 1 proposée plus haut.

Notez que Java ne permet pas de contrôler le mécanisme de copie et ne permet pas non plus de manipuler d'objets directement (le langage ne supporte de sémantique d'accès direct sur les objets, donc pas d'objets valeurs).

```
final class X {
   private static X singleton = new X();
   private X() {
      // ...
   }
   public static X getInstance() {
      return singleton;
   }
   // ...
}
```

Par conséquent, utiliser un constructeur privé et un attribut de classe privé suffit à éviter la duplication d'un singleton.

Une variante directe de cette première version reposera sur un bloc d'initialisation propre à la classe (bloc `static`, à droite) mais donne, dans ce cas-ci, précisément le même résultat. Un bloc `static` est une sorte de constructeur de classe, réalisant par un élément de langage un analogue de ce que les constructeurs des singletons statiques peuvent faire en C++.

```
final class X {
   private static X singleton;
   static {
      singleton = new X();
   }
   private X() {
      // ...
   }
   public static X getInstance() {
      return singleton;
   }
   // ...
}
```

En Java, une classe n'est chargée en mémoire que lors du premier accès qui y sera fait dans le programme. On ne peut donc pas compter sur un singleton pour réaliser, de manière transparente, des initialisations pour un programme.

Une autre implémentation typique est celle n'initialisant le singleton que s'il est utilisé au moins une fois (que la classe, elle, soit utilisée ou non). Le choix fait dans ce cas est d'économiser les ressources associées au singleton si personne n'y a recours, en échange d'un test supplémentaire lors de chaque tentative d'y accéder.

Le choix entre l'une et l'autre des options est d'abord et avant tout économique : veut-on épargner des ressources au besoin ou du temps d'exécution?

```
final class X {
   private static X singleton = null;
   private X() {
      // ...
   }
   public static X getInstance() {
      if (singleton == null) {
         singleton = new X();
      }
      return singleton;
   }
   // ...
}
```

Cette version-ci n'est pas sécuritaire en situation multiprogrammée, car elle contient une condition de course.

La collecte automatique d'ordures de Java élimine le problème de la libération du singleton. Comme à l'habitude en Java, toutefois, aucune garantie de finalisation n'est possible, alors toute finalisation sera vouée à une solution maison.

En C#, la situation est la même qu'en Java. Comme en Java, C# offre une sorte de constructeur de classe, qui sont syntaxiquement identiques aux autres constructeurs mais sont qualifiés `static`. **En VB.NET**, la situation est la même qu'en C#, à ceci près que le constructeur de classe est qualifié `Shared` plutôt que `static`.

Dans les langages `.NET` comme en Java, la collecte d'ordures libérera éventuellement le singleton, mais la finalisation demeure non déterministe; conséquemment, la code de finalisation échappera au singleton et devra être pris en charge par le code client ou par d'autres mécanismes.

Exercices – Série 06

EX00 – Éliminez la méthode `flux()` de la classe `Sortie`, plus haut, et remplacez-la par un opérateur `<<` prenant une `Sortie&` comme paramètre de gauche, et montrez, par un programme de test, que cet opérateur fonctionne pour plusieurs types. Pouvez-vous écrire cette opération de manière générique?

EX01 – Écrivez un singleton `GenerateurStochastique` dont le constructeur initialise le générateur de nombres pseudo-aléatoires avec :

```
srand(static_cast<unsigned int>(time(nullptr)));
```

Vous devrez au préalable inclure `<cstdlib>` et `<ctime>`. Les fonctions `rand()`, `srand()` et `time()` font toutes partie de l'espace nommé `std`; je vous invite à lire la documentation en ligne à leur sujet. Votre singleton devra offrir au moins les services suivants :

- une méthode d'instance `prochain()` sans paramètres qui retournera le prochain entier pseudo-aléatoire entre `0` et `RAND_MAX` inclusivement;
- une méthode d'instance `prochain()` prenant en paramètres deux bornes entières, `borne_min` et `borne_max`. Une exception doit être levée dans le cas où `borne_min > borne_max`. Une exception doit aussi être levée si l'intervalle `borne_min..borne_max` est plus grand que l'intervalle `0..RAND_MAX`. Cette méthode retournera le prochain entier pseudo-aléatoire entre `borne_min` et `borne_max` inclusivement;
- si vous en avez envie, envisagez aussi un service retournant un nombre à virgule flottante entre `0` et `1` inclusivement.

Rédigez un programme de test qui démontre que votre singleton fonctionne tel qu'attendu.

EX02 – Écrivez le singleton `GenSequentiel` offrant un service de numérotation entière à partir de zéro. Définissez-y un service générant à chaque appel le prochain entier dans la séquence (donc retournant `0` lors du premier appel, `1` lors du second, `2` lors du troisième…).

EX03 – Modifiez le singleton défini à l'exercice précédent pour qu'au lancement d'un programme, le singleton reprenne là où il en était rendu la dernière fois que le programme aura été interrompu. Utilisez un fichier local pour entreposer l'état entre deux exécutions. Si le fichier est absent au démarrage du programme, alors la numérotation doit reprendre à zéro. Déterminez vous-mêmes une politique à appliquer dans le cas où le fichier existe mais contient des données incorrectes (hors bornes ou de mauvais format), puis justifiez et documentez votre choix. Votre approche doit être essentiellement transparente pour le code client.

Classes internes comme mécanique de support

Important : cette section détaille une approche qui est désuète en situation de multiprogrammation. Dans la programmation contemporaine, privilégiez la sémantique de mouvement et les pointeurs intelligents.

Les classes internes servent parfois à titre de classes de support pour permettre une implémentation plus efficace de certains concepts.

Si vous avez consulté la section ***Bibliothèque standard de C++***, en particulier ***Conteneurs et itérateurs***, vous connaissez déjà un exemple important de classe imbriquée dans les conteneurs standards : les itérateurs.

En effet, dans une classe générique comme `vector<T>`, un itérateur correct dépendra souvent du type `T`, et sera donc idéalement défini à même le conteneur pour que le même type `T` s'applique aux deux. Un itérateur est un service offert par un conteneur sous la forme d'une classe plutôt que sous la forme d'une méthode.

L'exemple que nous esquisserons ici en est un cas classique.

Imaginons l'ébauche de programme visible à droite. Ce petit programme semble banal : copier du texte d'un endroit à un autre est une tâche des plus routinières. Présumer de son efficacité, malheureusement, signifie sous-estimer un facteur important.

En effet, nous ne connaissons pas ici la taille du texte retourné par `lire_texte()`… et nous ne voulons pas nécessairement la connaître, d'ailleurs!

```
#include <string>
using std::string;
// ...
int main() {
   string s0 = lire_texte();
   string s1 = s0;
   // ...
}
```

Si `lire_texte()` retourne une chaîne de quelques milliers de caractères à peine, l'action de copier ce texte de sa représentation dans `lire_texte()` à `s0` dans `main()` est banale, et il en va de même pour la copie dans `s1` de `s0` à la ligne suivante.

En retour, si `lire_texte()` retourne une masse de plusieurs mégaoctets de caractères, ces deux opérations de copie seront à la fois longues et potentiellement coûteuses en espace (`s1` et `s0` ayant la même portée, les deux existeront côte à côte!), du moins pour une implémentation naïve de `std::string`.

L'encapsulation et les classes internes viennent toutefois à notre secours, de manière transparente, comme il se doit quand on parle d'encapsulation. En effet, on présume sans doute que `std::string` conserve les caractères qui y sont insérés dans un tableau, comme proposé dans l'ébauche (inexacte) à droite.

Selon cette illustration, chaque chaîne posséderait sa propre copie du texte. Copier `s0` dans `s1` signifierait alors (grossièrement) créer un tableau de la bonne taille dans `s1` et y copier un à un les caractères contenus dans `s0`.

```
namespace std {
   // naïf et inexact
   class string {
      char *donnees;
      int taille;
      // ...
   };
}
```

Ce n'est toutefois pas la seule implantation possible du concept de chaîne de caractères, ni celle réellement utilisée d'ailleurs dans les implémentations efficaces. En pratique, en effet, on peut être *beaucoup* plus ingénieux. Imaginons qu'une `std::string` soit plutôt comme une enveloppe, dans laquelle on retrouverait la représentation du texte qu'elle contient. La représentation en question est une abstraction entièrement locale et interne à la chaîne, un cas clair de classe interne.

Copier `s0` dans `s1` pourrait alors être équivalent à faire de la représentation de `s1` une copie de la représentation de `s0`, bien sûr, mais on n'y gagnerait pas en efficacité. Par contre, *il pourrait s'agir de faire partager par `s0` et par `s1` une même représentation*. Ainsi, plutôt que de copier un contenu (de taille arbitraire), on copierait dans la mesure du possible un pointeur (de taille fixe et petite).

Pour que cela fonctionne bien, il faut :

- que la représentation interne soit vraiment interne et privée. Une classe interne est sans doute la meilleure approche pour y arriver;
- que la représentation interne soit allouée et libérée dynamiquement; donc
- que la durée de vie de la représentation interne soit rigoureusement gérée, probablement à l'aide d'un mécanisme pour compter les références; et
- que les références multiples vers une même représentation interne soient maintenues aussi longtemps que possible, mais *pas plus*. Toute opération modifiant la représentation en question devrait mener à une duplication de contenu pour l'objet sur lequel la représentation est effectivement modifiée.

Illustration : une chaîne maison (`Partageable`)

Illustrons le concept avec une chaîne académique, sorte de `std::string` simplifiée dans le but de démontrer l'idée présentée ici. Nous nommerons `chaine` notre chaîne simplifiée.

Nous n'implanterons pas toutes les opérations utiles qui soient possibles sur une instance de `chaine`. Certaines parmi celle que nous n'aurons pas implantées seront proposées en exercice ; bien entendu, vous pourrez en ajouter d'autres si le cœur vous en dit.

Notre programme de démonstration sera celui visible à droite. Si notre implantation est correcte, il ne devrait y avoir dans ce programme qu'une seule représentation interne du texte `"allo toi"`, du moins jusqu'à la ligne où on trouve l'opération `c2+='!'`. Cette opération modifie la représentation de `c2`, forçant `c2` à se créer une version interne propre à lui de la représentation du texte avant de procéder à l'ajout du caractère `'!'`.

L'affichage à la console, à la toute fin, devrait présenter les lignes `"allo toi"`, `"allo toi"` et `"allo toi!"`, dans l'ordre.

```
#include "chaine.h"
#include <iostream>
int main() {
   using namespace std;
   chaine c0{"allo toi"};
   chaine c1 = c0;
   chaine c2;
   c2 = c0;
   c2 += '!';
   cout << c0 << endl
        << c1 << endl
        << c2 << endl;
}
```

La technique d'optimisation reposant sur le partage aussi longtemps que possible des données d'objets, donc sur leur copie tardive, porte communément le nom de ***Copy on Write***, ou *COW* Évidemment, les classes internes peuvent servir à une variété de fins, et cette optimisation n'est qu'un exemple parmi d'autres.

L'implémentation de `chaine` offerte dans cette section implémente *COW*, mais de manière subtilement dangereuse. Examinez la réflexion `02.5` de l'***annexe OO*** pour des détails.

Pour implémenter le partage de la représentation interne d'une chaîne de caractères, nous mettrons à profit la classe `Partageable`, présentée plus haut.

Cette stratégie de partage d'objets n'est pas la plus simple qui soit pour le code client, mais puisqu'elle servira à titre de parent pour une classe interne et privée à `chaine`, le code client de `chaine` ne sera pas en contact avec elle.

La classe `Partageable` est répétée à droite, à titre de rappel. Référez-vous à la section ***Objets partageables***, plus haut, pour un rappel des détails associés à cette classe.

```
#ifndef PARTAGEABLE_H
#define PARTAGEABLE_H
class Partageable {
   long cptref = 1L;
public:
   long nb_refs() const noexcept {
      return cptref;
   }
   void AddRef() noexcept {
      ++cptref;
   }
   void Release() noexcept {
      if (--cptref == 0)
         delete this;
   }
   Partageable() = default;
protected:
   virtual ~Partageable() = default;
};
#endif
```

Le fichier `chaine.h` contiendra la déclaration de la classe `chaine`, incluant celle de sa classe interne `chaine::chaineRep`.

La classe `chaine` se présentera comme un conteneur à part entière, ce qu'elle est. Elle exposera quelques types internes et publics habituels pour une telle classe.

Tel que mentionné précédemment, une classe interne a accès aux membres et aux abstractions de la classe englobante.

Ici, les types internes et publics de `chaine` seront aussi ceux de `chaine::Rep`, du moins jusqu'à preuve du contraire. Si la classe interne définit sa propre version de ces types, alors sa version primera (pour elle) sur celle de sa classe englobante.

La classe interne tiendra à jour deux attributs, soit le nombre d'éléments qui y sont contenus (`lg`) et un pointeur sur le premier d'entre eux (`texte`).

Une chaîne plus sophistiquée tiendrait aussi compte de sa capacité et chercherait à croître plus intelligemment que celle-ci.

```
#ifndef CHAINE_H
#define CHAINE_H
#include <iosfwd>
class chaine {
public:
   using value_type = char;
   using iterator = value_type*;
   using const_iterator = const value_type*;
   using size_type = int;


private:
   class Rep : public Partageable {
      size_type lg{};
      value_type *texte = nullptr;
```

La méthode `push_back()` permettra d'ajouter un élément à la fin. Elle sera détaillée plus bas, dans le fichier source `chaine.cpp`, car son implémentation ne sera pas banale.

Les petites méthodes que sont `size()`, `empty()`, `begin()` et `end()`, sont toutes banales et semblables à celles qu'on pourrait trouver dans d'autres conteneurs.

Le constructeur par défaut et le destructeur sont banals.

Pour leur part, le constructeur paramétrique et le constructeur de copie sont plus subtils et seront définis dans le fichier source.

Une méthode `nth_elem()` est exposée sous forme constante et non constante pour offrir un accès brut aux données. La classe englobante, `chaine`, sera responsable d'en valider les intrants au préalable.

Le seul attribut d'instance d'une chaine sera un pointeur sur sa représentation interne.

Par convention, une chaine vide aura une représentation nulle.

```
public:
   void push_back(char);
   size_type size() const noexcept {
      return lg;
   }
   bool empty() const noexcept {
      return size()==0;
   }
   iterator begin() noexcept {
      return texte;
   }
   const_iterator begin() const noexcept {
      return texte;
   }
   iterator end() noexcept {
      return begin() + size();
   }
   const_iterator end() const noexcept {
      return begin() + size();
   }

   Rep() = default;
   Rep(const value_type*);
   Rep(const Rep&);
   ~Rep() {
      delete[] texte;
   }
   value_type& nth_elem(size_type n) noexcept {
      return texte[n];
   }
   value_type nth_elem(size_type n) const {
      return texte[n];
   }
};


Rep *rep = nullptr;
```

Encore une fois, pour `chaine` comme pour sa représentation, les méthodes `size()`, `empty()`, `begin()` et `end()` sont banales.

Réflexion 02.5 : il y a un piège ici. Le voyez-vous? Réponse dans ***Réflexion 02.5 : difficile de partager***.

Vous remarquerez que les versions de la classe englobante seront nécessairement un peu moins rapides que celles de la classe interne, du fait que la classe englobante doit s'assurer que la représentation est non nulle puis, si c'est le cas, déléguer le travail à la représentation en question.

Les constructeurs et le destructeur sont tous simples :

- la représentation interne d'une chaine par défaut sera nulle;
- une construction paramétrique déléguera le travail à sa représentation. Le recours à l'allocation dynamique de mémoire ici entraîne un risque de levée d'exception;
- une construction par copie ne dupliquera pas le contenu représenté mais ajoutera plutôt une référence sur la représentation nouvellement partagée; et
- la destruction retirera une référence sur la représentation partagée (s'il y en a effectivement une).

```
public:
   size_type size() const noexcept {
      return rep? rep->size() : 0;
   }
   bool empty() const noexcept {
      return size() == 0;
   }
   iterator begin() noexcept {
      return rep? rep->begin() : nullptr;
   }
   const_iterator begin() const noexcept {
      return rep? rep->begin() : nullptr;
   }
   iterator end() noexcept {
      return begin() + size();
   }
   const_iterator end() const noexcept {
      return begin() + size();
   }
```

```
   chaine() = default;
   chaine(const value_type *s) : rep{new Rep{s}} {
   }
   chaine(const chaine &s) noexcept : rep{s.rep} {
      if (rep) rep->AddRef();
   }
   ~chaine() {
      if (rep) rep->Release();
   }
```

Le constructeur paramétrique pourrait être raffiné pour détecter les pointeurs nuls et lever une exception.

Quelques opérateurs conventionnels et utiles compléteront le portrait pour cette ébauche un peu simple de chaîne de caractères.

```
   bool operator==(const chaine&) const noexcept;
   bool operator!=(const chaine &s) const noexcept {
      return !(*this == s);
   }
   class HorsBornes {};
   value_type operator[](size_type) const;
   value_type& operator[](size_type);
   chaine& operator=(const chaine&);
   chaine& operator +=(value_type c) {
      push_back(c);
      return *this;
   }
   void push_back(value_type);
   friend std::ostream& operator<<(std::ostream&, const chaine&);
};
#endif
```

Le fichier source définira les quelques méthodes et opérations qu'il n'était pas à propos de placer dans le fichier d'en-tête.

Projeter une chaine sur un flux signifiera projeter chacun de ses caractères sur un flux. J'ai utilisé une répétitive sur un intervalle par souci de simplicité.

La méthode `push_back()` de la représentation, détaillée à droite, est à la fois lente et complexe. Il y aurait lieu de la raffiner.

Son plus gros défaut est de ne pas distinguer capacité et nombre d'éléments, ce qui fait que chaque nouvel ajout implique une nouvelle allocation de mémoire et une copie de contenu.

C'est une approche vraiment très coûteuse (ce que vous constaterez si vous ajoutez, dans une répétitive, un grand nombre de caractères à une même instance de `chaine`).

L'exercice `EX07`, plus bas, amène à corriger cet irritant.

```
#include "chaine.h"
#include <ostream>
#include <algorithm>
#include <cstring>
using namespace std;
ostream & operator<<(ostream &os, const chaine &s) {
   for(auto c : c)
      os << c;
   return os;
}


void chaine::Rep::push_back(value_type c) {
   if (!texte) {
      texte = new value_type[2];
      texte[0] = c;
      texte[1] = '\0';
      lg = 1;
   } else {
      auto p = new value_type[size() + 2];
      copy(begin(), end(), p);
      delete[] texte;
      texte = p;
      texte[lg++] = c;
      texte[size()] = '\0';
   }
}
```

Le constructeur paramétrique d'une représentation calculera la longueur de la chaîne brute reçue en paramètre, allouera un tableau suffisamment grand pour contenir cette chaîne et un délimiteur de fin, puis fera une copie locale du contenu de la chaîne brute.

Le constructeur de copie réalisera une copie de contenu, pas un partage de contenu. Les opérations menant à un partage de la représentation seront placées entre les mains de `chaine`.

La comparaison avec `==` est banale.

L'accès à un élément est banal dans sa déclinaison constante mais l'est beaucoup moins dans sa déclinaison non constante.

Envisagez implémenter ceci avec `std::equal()` de la bibliothèque `<algorithm>`.

```
chaine::Rep::Rep(const value_type *s) : lg{strlen(s)} {
   texte = new value_type[size() + 1];
   copy(s, s + size(), texte);
   texte[size()] = 0;
}
chaine::Rep::Rep(const Rep &cr) : lg{cr.size()} {
   texte = new value_type[size() + 1];
   copy(cr.begin(), cr.end(), texte);
   texte[size()] = 0;
}


bool
chaine::operator==(const chaine &s) const noexcept {
   if (size() != s.size()) return false;
   for (auto p = begin(), q = s.begin();
        p != end(); ++p, ++q)
      if (*p != *q)
         return false;
   return true;
}
```

En effet, laisser filtrer une référence vers un élément d'une chaine partagée est une opération dangereuse : si le code client modifie le référé, cela affectera non pas l'instance de chaine qui lui y aura donné accès, mais bien toutes les instances de `chaine` qui se le partagent.

Conséquemment, il est important que la `chaine` propriétaire de la méthode devienne d'abord seule propriétaire de sa représentation interne. Ensuite seulement sera-t-il raisonnable de laisser filtrer une indirection vers un élément.

Réflexion `02.5` : il y a un piège ici. Le voyez-vous? Réponse dans ***Réflexion `02.5` : difficile de partager***.

```
auto chaine::operator[](size_type n) const->value_type {
   if (n >= size()) throw HorsBornes{};
   return rep->nth_elem(n);
}
auto chaine::operator[](size_type n) -> value_type& {
   if (n >= size()) throw HorsBornes{};
   if (rep-> nb_refs() > 1) {
      auto p = new Rep(*rep);
      rep->Release();
      rep = p;
   }
   return rep->nth_elem(n);
}
```

L'affectation implique passer d'une représentation interne à une autre, en ajustant le comptage de références sur les représentations impliquées.

```
chaine& chaine::operator=(const chaine &s){
   if (rep) rep->Release();
   rep = s.rep;
   if (rep) rep->AddRef();
   return *this;
}
```

L'ajout d'un élément ne doit modifier que la chaine visée. Ainsi, au besoin, une nouvelle représentation sera construite avant de procéder à l'ajout d'un élément à l'aide de `push_back()`.

```
void chaine::push_back(value_type c) {
   if (!rep)
      rep = new Rep;
   else if (rep->nb_refs() > 1) {
      auto p = new Rep{*rep};
      rep->Release();
      rep = p;
   }
   rep->push_back(c);
}
```

Nous verrons ultérieurement plusieurs autres techniques reposant sur une classe interne pour implémenter des détails techniques et faciliter le sain fonctionnement de la classe externe. Dans la bibliothèque standard de C++, rares sont les classes qui n'ont pas recours à une manœuvre reposant, en tout ou en partie, sur ce mécanisme :

- Une `std::list<T>` entrepose des `T` dans des nœuds, qui n'ont pas à être visibles du code client
- La classe `std::basic_string<C>`, de laquelle est faite la classe `std::string`, peut implémenter ce qu'on nomme le *Small String Optimization* [hdSSO] par laquelle on évitera d'allouer dynamiquement de la mémoire pour entreposer des séquences de caractères suffisamment petites. Ainsi, une `std::string` opèrera de deux manières différentes selon le contexte, à l'insu du code client
- La classe `std::function` permet d'entreposer des entités appelables de divers types (pointeur de fonction, foncteur, λ) de manière relativement transparente
- Les conteneurs standards implémentent typiquement leurs itérateurs avec des classes internes, et ainsi de suite

Exercices – Série 07

Note :	certaines paires d'exercices peuvent être réalisées de manière simple en utilisant l'une pour réaliser l'autre. Essayez d'obtenir une version simple et une autre qui soit plus efficace dans ces cas. Réfléchissez aussi à celle que vous préféreriez dans le cadre de votre travail, et aux raisons qui guident votre choix.

EX00 – Ajoutez à la classe `chaine` l'opérateur `+`, qui retourne une `chaine` faite de la concaténation des opérandes de gauche et de droite tout en garantissant la constance de ces deux opérandes.

EX01 – Ajoutez à la classe `chaine` les opérateurs `<`, `<=`, `>` et `>=`, qui réalisent les comparaisons lexicographiques usuelles.

EX02 – Ajoutez à la classe `chaine` la méthode `inverser()`, qui inverse l'ordre des caractères dans la chaîne propriétaire de la méthode.

EX03 – Ajoutez à la classe `chaine` la méthode `copieInversee()`, qui retourne une chaîne qui ait ses caractères dans l'ordre inverse de ceux de la chaîne propriétaire de la méthode, tout en garantissant que l'originale demeure intacte.

EX04 – Inspirez-vous des exercices `EX02` et `EX03` pour implanter les méthodes `majuscules()`, `minuscules()`, `copieMajuscules()` et `copieMinuscules()`.

EX05 – Ajoutez à la classe `chaine` la méthode `sousChaine()`, qui prend en paramètre les index de début (inclusif) et de fin (exclusif) et retourne la chaîne correspondant à cet intervalle dans l'instance propriétaire de cette méthode. L'instance propriétaire devrait être garantie constante au sens de cette opération.

EX06 – Essayez d'écrire une version de `chaine`, appelons-la `chaineNaive`, qui n'utilise pas la technique de classe interne pour la représentation du contenu. Comment pourriez-vous comparer les performances des deux implantations? Faites le test. Vaut-il la peine d'appliquer la technique proposée ici?

EX07 – Modifiez les classes `chaine` et `chaine::Rep` pour que la représentation tienne à jour à la fois un nombre d'éléments et une capacité, et pour que la capacité croisse de manière plus efficace (p. ex. : faites croître la capacité par un facteur de `1,5` lorsqu'une tentative d'ajout est faite sur une chaîne pleine).

EX08 – Ajoutez un constructeur de séquence à la classe chaine modifiée dans l'exercice `EX07` et cherchez à rendre ce constructeur le plus efficace possible. Les améliorations apportées ici ont-elles des ramifications ailleurs dans la classe?

EX09 – Implémentez l'opérateur `==` de chaine en partie à partir de `std::equal()`.

Réflexion

Comment pourrait-on bâtir un plan de test pour valider le bon fonctionnement d'une classe comme `chaine`? Quels sont les tests à faire? Y a-t-il des contenus particulièrement importants à valider? Comment vérifierait-on chaque méthode? Comment vérifierait-on l'efficacité des opérations? Comment s'assurerait-on qu'il n'y ait pas de pertes de mémoire?

Amitié

Reprenons brièvement une version simplifiée d'un exemple examiné dans la section ***Singletons*** un peu plus haut.

À titre de rappel, ce petit programme crée (ou vide) un fichier nommé `Sortie.txt` à chaque démarrage (dans le constructeur d'un singleton nommé `CreateurSortie`) puis permet, à l'aide d'instances d'une classe utilitaire nommée `Sortie`, d'ajouter des données à la fin de ce fichier. Chaque paire construction/ destruction d'une `Sortie` ouvre le fichier pour permettre des ajouts à la fin, et ce pour un très bref moment.

Ce programme a un défaut structurel : le service `ouvrir()` du singleton `CreateurSortie` est destiné à la seule classe `Sortie`, avec laquelle le singleton collabore étroitement; si étroitement, en fait, qu'il faudrait probablement livrer ces deux classes dans un même module.

Toutefois, pour permettre à une instance de `Sortie` d'accéder à `ouvrir()`, le singleton doit révéler cette méthode au monde entier.

Ceci étend inutilement sa surface d'exposition publique et crée un bris d'encapsulation. La solution utilisée, offrir au monde entier une méthode destinée à une classe spécifique, est un palliatif décevant.

```
#include "Incopiable.h"
#include <fstream> // std::ios_base::app
#include <string>
#include <iostream>
using namespace std;
const string FICHIER_SORTIE = "Sortie.txt";
class CreateurSortie : Incopiable {
   static CreateurSortie singleton;
   CreateurSortie() {
      ofstream(FICHIER_SORTIE);
   }
public:
   static CreateurSortie& get() {
      return singleton;
   }
   void ouvrir(ofstream &os) {
      os.open(FICHIER_SORTIE, ios_base::app);
   }
};
CreateurSortie CreateurSortie::singleton;
class Sortie {
   ofstream sortie;
public:
   Sortie() {
      CreateurSortie::get().ouvrir(sortie);
   }
   ofstream& flux() {
      return sortie;
   }
};
int main() {
   Sortie{}.flux() << "Début..." << endl;
   // peu importe...
   Sortie{}.flux() << "Fin..." << endl;
}
```

Placés devant cette situation, quelques constats s'imposent :

- la classe `Sortie` a été pensée pour travailler avec la classe `CreateurSortie`. Elles prennent leur sens lorsqu'elles sont utilisées ensemble;
- ces deux classes seront *développées ensemble* et *livrées ensemble*, dans un même module;
- la collaboration étroite attendue entre ces classes fait en sorte qu'il soit légitime d'offrir à la classe `Sortie` des privilèges sur `CreateurSortie` qui ne seront pas offerts à d'autres classes. Un de ces privilèges est l'accès à la méthode `ouvrir()`;
- ces privilèges ne seront pas sources d'abus puisque les deux classes vont de pair. Remplacer l'une impliquera remplacer l'autre.

Ici, la meilleure approche n'est pas d'offrir `ouvrir()` au monde entier, mais bien d'en faire une méthode privée et d'accepter que, pour `CreateurSortie`, la classe `Sortie` n'est pas une classe comme les autres : c'est une classe privilégiée, une *amie*.

Le mot clé `friend`

L'amitié, associée au mot clé `friend`, est souvent décriée dans les milieux `OO`. On associe souvent ce concept à l'idée de bris d'encapsulation. Pourtant, si on applique le concept avec sagesse, il peut au contraire s'agir d'une manière *d'éviter* un bris d'encapsulation.

Voyons un peu de quoi il s'agit.

⇒ Une classe peut spécifier, dans sa déclaration, qu'elle a des ***sous-programmes amis*** ou des ***classes amies*** en ajoutant la mention **`friend`** suivie du prototype de cette fonction ou du nom de cette classe.

⇒ Un **ami** d'une classe est un sous-programme ou une classe ayant droit d'accéder aux membres privés ou protégés de cette classe.

Il faut immédiatement établir, à la lecture de cette définition, que *la mention `friend` ne force pas un bris d'encapsulation dans une classe*. En effet, un sous-programme ou une autre classe ne peut pas s'imposer ami(e) d'une classe, la mention `friend` devant être incluse à même la déclaration de la classe qui s'ouvre à ses amis.

Un ami d'une classe donnée fait partie de l'interface de cette classe, au même titre que ses membres publics et que les fonctions qui sont livrées avec elle. Par conséquent, les amis d'une classe doivent l'accompagner et faire partie du même module (`.h/ .cpp`) qu'elle.

Lorsqu'on présente deux classes, l'une étant l'amie de l'autre, il est fortement préférable de fournir les deux classes dans un même module (même fichier d'en-tête, mêmes sources ou mêmes binaires). On agira ainsi pour éviter qu'un programmeur véreux remplace la classe ayant des droits en tant qu'amie par une autre, et commette par cette classe remplaçante un sérieux bris d'encapsulation. La même remarque s'applique aux sous-programmes amis, d'ailleurs.

```
class Y {
   // ...
   friend class X;
   friend void fy();
   // ...
};
class X {
   // ...
};
```

À droite, `X` est ami de `Y` et peut donc accéder aux membres privés et protégés de cette dernière. La fonction globale `fy()` a les mêmes privilèges que `Y` que la classe `X`. Notez que la déclaration d'amitié de `fy()` est un prototype à part entière (pas besoin de le répéter), et celle de `X` est une déclaration *a priori*.

Certains utilisent des stratégies OO reposant fortement sur l'amitié, comme par exemple une classe polymorphique servant d'interface et plusieurs classes dérivées dont chaque méthode est privée et qui implémentent les spécialisations des méthodes de leur parent. Dans un tel cas, on trouve, adjointe à chaque classe dont toutes les méthodes sont privées, une fonction amie servant à l'instancier.

Voir [AccCpp] d'**Andrew Koenig** et **Barbara Moo**, section ***Source code … for Visual Studio .NET***, chapitre 15 (fichiers Pic.h et Pic.cpp) pour un exemple très joli.

Le site http://www.parashift.com/c++-faq-lite/friends.html m'a aussi été recommandé par un brillant ancien étudiant du nom d'**André Louis Caron**. C'est concis et clair.

Ce que l'amitié n'est pas

Quelques remarques de fond doivent être faites quant à l'amitié dans un modèle OO.

L'amitié n'est pas héritée

Ainsi, dans notre exemple plus haut, un dérivé de X ne serait pas, en tant que ce dérivé, l'ami de Y.

De même, dans l'exemple à droite, Y est ami de X, mais Z (un enfant de Y) ne l'est pas.

Conséquemment, il faut limiter (ou mieux, éviter!) les méthodes virtuelles dans une classe étant l'amie d'une autre; sinon, on dérivera de cette classe une classe malsaine et on pourra procéder à une fraude en passant par cette méthode.

Cela permet de voir que l'amitié ne provoque par *a priori* de bris d'encapsulation (en fait, lorsqu'elle est bien utilisée, l'amitié *renforce* l'encapsulation en réduisant l'espace public d'une classe au strict nécessaire). On ne peut contourner la protection d'une classe en se créant un dérivé *ad hoc* d'un de ses amis.

```
class X {
   int val;
   friend class Y;
};
class Y {
protected:
   X &x;
   // Ok (Y est l'ami de X)
   int f() const {
      return x.val;
   }
public:
   Y(X &x) : x{x} {
   }
};
class Z : public Y {
public:
   Z(X &x) : Y{x} {
   } // Ok
   int g() const {
      return f(); // Ok
   }
   // illégal (l'amitié n'est pas héritée)
   int h() const {
      return x.val;
   }
};
```

L'amitié n'est pas transitive

En termes simples : *l'ami d'un ami n'est pas notre ami* à moins qu'on ne l'ait explicitement indiqué au préalable.

Dans l'exemple à droite, `Y` est amie de `X`, et `Z` est amie de `Y`, mais `Z` n'est pas amie de `X`.

L'amitié n'est pas symétrique

Si `X` est amie de `Y`, cela implique que `Y` a accès aux membres privés et protégés de `X`, mais cela n'implique pas en retour que `X` ait accès aux membres privés et protégés de `Y`.

C'est injuste, diront certains, mais c'est comme ça. Les relations d'amitié se pensent sur une base individuelle et dirigée.

```
class X {
   int val;
   friend class Y;
};
class Y {
protected:
   X &x;
public:
   Y(X &x) : x{x} {
   }
private:
   int f() const {
      return x.val;
   }
   friend class Z;
};
class Z {
   Y &y;
public:
   Z(Y &y) : y{y} {
   }
   // Ok (Z est amie de Y)
   int g() const {
      return y.f();
   }
   // illégal (Z n'est pas amie de X)
   int h() const {
      return y.x.val;
   }
};
```

Utiliser sainement l'amitié

L'amitié est la relation ayant le plus fort couplage (plus même que l'héritage public!) dans une approche OO. Sachant cela, pourquoi donc voudrait-on la mettre en application?

En fait, *on aura recours à l'amitié lorsque l'impact de ne pas y avoir recours serait pire encore*. Deux exemples concrets d'utilisation valable d'amis sont offerts ailleurs dans ce document, mais pour l'idée générale :

- il arrive qu'un détail d'implémentation d'une classe doive être révélé à un sous-programme ou à une autre classe, souvent pour des raisons techniques (pensons à `Sortie` qui doit être seule cliente de la méthode `ouvrir()` de `CreateurSortie`);
- dans certains cas, on utilisera une encapsulation dans une méthode pour résoudre le problème, mais il arrivera qu'on ne veuille pas même exposer cette méthode. Pensez à un cas où l'accès *direct* à un attribut soit requis, par exemple si cet attribut est un pointeur, mais où il serait dangereux de donner publiquement accès à cet attribut. Retourner une copie d'un pointeur donne accès au même objet que si on utilisait le pointeur original, après tout;
- devant révéler un tel détail à un nombre restreint de classes ou de sous-programmes, mais ne voulant pas ouvrir l'accès à ce détail au grand public, on aura recours aux amis.

Utiliser l'amitié avec modération peut être moins dommageable que l'alternative, soit offrir un accès public à un attribut ou à une méthode donnant un accès direct, bien que de manière détournée, à un attribut. De manière générale, l'amitié est préférable à une corruption du visage public d'une classe.

La même démarche intellectuelle se pose pour l'accès à une méthode devant rester privée ou protégée, mais pour laquelle un nombre restreint d'autres classes ou de sous-programmes devraient avoir un accès immédiat.

L'un des aléas de la mention `friend` est qu'un ami ainsi spécifié a accès à *tous* les membres privés ou protégés de la classe. Ce n'est pas une mention contrainte à un sous-ensemble de la classe, mais bien une ouverture complète devant l'ami en question.

L'amitié est un mécanisme à utiliser avec une grande prudence. Chaque apparition de cette spécification devrait vous amener à vous interroger sur la légitimité des choix de design qui vous ont mené jusque là, mais pas nécessairement à les rejeter.

L'amitié est la relation la plus intime que peut entretenir une classe hormis celle résultant de l'exposition publique de ses attributs. Ceci signifie, tel qu'indiqué plus haut dans cette section, que l'amitié est aussi la relation à plus fort couplage que peut entretenir une classe. Sachant cela, tout abus devient malsain.

La manière la plus convenable de percevoir l'amitié est comme extension de l'interface d'une classe. Une classe amie de la classe `X` devrait être considérée comme faisant partie logiquement de l'interface de `X`, et il en va de même pour toute fonction amie de `X`. Par conséquent, les amis d'une classe devraient être insérés dans les mêmes fichiers que cette classe. Les amis doivent être déclarés ensembles et définis ensembles, pour éviter qu'un tiers hostile ne remplace ou ne personnifie l'un sans personnifier l'autre.

Exemple concret

Reprenons l'exemple utilisé en début de section, d'abord pour voir quels changements apporter pour profiter de l'amitié et améliorer le design de notre petit programme et, ensuite, examiner en quoi l'impact de ce changement est strictement positif du point de vue de la qualité du programme résultant.

```
// inclusions et using...
const string FICHIER_SORTIE = "Sortie.txt";
class CreateurSortie : Incopiable {
   static CreateurSortie singleton;
   CreateurSortie() {
      ofstream{FICHIER_SORTIE};
   }
   friend class Sortie;
   void ouvrir(ofstream &os) { // privée!
      if (os) os.open(FICHIER_SORTIE, ios_base::app);
   }
public:
   static CreateurSortie& get() {
      return singleton;
   }
};
CreateurSortie CreateurSortie::singleton;
class Sortie {
   ofstream sortie;
public:
   Sortie() {
      CreateurSortie::get().ouvrir(sortie);
    }
   ofstream& flux() {
      return sortie;
   }
};
#include <iostream>
int main() {
   Sortie{}.flux() << "Début..." << endl;
   // peu importe...
   Sortie{}.flux() << "Fin..." << endl;
}
```

Le code client est identique à celui vu précédemment, et rien n'a changé non plus dans la classe utilitaire `Sortie`.

Par contre, ici, le risque qui existait auparavant qu'un tiers mal intentionné ou mal écrit invoque la méthode `ouvrir()` du singleton de manière à nuire au bon fonctionnement de la classe `Sortie` disparaît du fait que :

- la classe `Sortie` est seule amie de `CreateurSortie`; et que
- la méthode `ouvrir()` de `CreateurSortie` y est spécifiée privée. Conséquemment, seuls une instance de `CreateurSortie` ou un ami de cette classe (une instance de la classe `Sortie`) y auront accès.

L'introduction de la mention `friend class Sortie;` peut être intégrée dans un bloc privé, protégé ou public de la classe `CreateurSortie`, peu importe.

Amitié et injection de fonctions globales

Une autre utilité directe de la mention `friend` dans une classe est qu'elle permet d'injecter une définition de fonction globale dans un programme. Ceci est particulièrement utile dans le cas de fonctions globales génériques.

Reprenant notre exemple plus haut, si nous souhaitions surcharger l'opérateur de projection sur un flux applicable à une instance de `Sortie` pour divers types `T`, nous pourrions apporter le raffinement suivant au code proposé (notez que j'ai délibérément remplacé `std::endl` par `\n`, pour des raisons subtiles dont nous traiterons dans un autre document).

Nous ne conserverons pour cette illustration que le code (modifié) de la classe `Sortie`. Pour le reste du code, vous pourrez vous référer aux exemples précédents.

La fonction marquée `friend` dans la classe `Sortie` n'est pas une méthode mais bien une fonction globale, créée directement à même la classe `Sortie`. Cette fonction définit le sens de l'opérateur `<<` prenant une référence sur une `Sortie` comme opérande de gauche et une référence constante sur un `T` comme opérande de droite, et ce pour tout type `T`.

Que cette fonction globale soit générique n'est pas un accident. Il est aussi possible de définir une fonction plus traditionnelle à même une déclaration de classe, mais il faut alors se limiter au prototype de la fonction dans la déclaration et définir la fonction dans un fichier source.

Cette notation met en relief le fait qu'un ami soit une extension de l'interface d'un objet.

```
// ...
class Sortie {
   ofstream sortie;
public:
   // ...
   ofstream& flux() {
      return sortie;
   }
   template <class T>
      friend Sortie&
         operator<<(Sortie &s, const T &val) {
            s.flux() << val;
            return s;
         }
};
int main() {
   Sortie{} << "Début...\n";
   // peu importe...
   Sortie{} << "Fin... \n";
}
```

Questions de qualifications

Prenons le temps d'examiner quelques petits détails concernant les qualifications de sécurité et certaines de leurs applications plus subtiles sur le plan technique.

Empêcher l'instanciation automatique

Question moins académique qu'il n'y paraît : est-il possible d'empêcher l'instanciation automatique d'objets?

Pour être plus précis : dans un cas comme celui du code à droite, peut-on faire en sorte que l'instanciation de `qqch` dans `f()` soit illégale, sans empêcher l'instanciation (dynamique) à travers `p` dans `main()`?

```
class Qqch {
   // ...
};
void f() {
   Qqch qqch;
}
void g() {
   // mieux : unique_ptr
   auto p = new Qqch;
   // ...
   delete p;
}
```

Forcer l'instanciation dynamique découle habituellement d'un souci de contrôle des ressources. Les cas typiquement associés à cette technique ont en commun ces particularités :

- un constructeur allouant des ressources particulières (fils d'exécution, *sockets*, connexion à une base de données…);
- un destructeur susceptible de prendre un temps arbitraire pour à compléter son travail, dû aux contraintes particulières des ressources à libérer;
- les délais à la destruction peuvent être tels qu'il devient important de détruire les objets en parallèle, dans un autre *thread* que celui ayant utilisé l'objet. Un objet instancié automatiquement est plus simple à gérer mais sera détruit en fin de portée, ce qui aura un impact sur la vitesse d'exécution du *thread*.

Une autre situation où l'instanciation automatique n'est pas indiquée est celle où un même objet doit fournir des services à plusieurs clients de façon concurrente. Les objets pouvant servir à plusieurs clients doivent être alloués dynamiquement, et doivent aussi contrôler leur propre durée de vie (se suicider lorsqu'ils ne sont plus utilisés par personne). Ce sont de bons candidats à la technique discutée dans cette section.

De manière générale, les objets polymorphiques sont souvent susceptibles d'être alloués dynamiquement. Si vous estimez que votre classe devrait être instanciée ainsi, alors la technique proposée ici peut vous intéresser.

Nous ne mettrons pas en place ici les éléments requis pour présenter un exemple pratique qui soit un reflet réaliste des situations motivant la technique présentée ici, pour ne pas obscurcir le propos. Nous limiterons notre présentation à un exemple académique, auquel vous pourrez vous référer si vous rencontrez un jour une situation du même genre.

Examinons maintenant la technique. On aurait tendance à penser que la solution pour empêcher l'instanciation automatique serait de spécifier les constructeurs comme étant privés. Cette intuition est flouée à la base, puisqu'elle empêcherait autant l'instanciation dynamique que l'instanciation automatique[94], du moins pour le code client.

Utiliser des constructeurs privés peut être une bonne idée si on implante la génération d'instances d'une classe donnée à l'aide du schéma de conception Fabrique, examiné dans la section ***Objets opaques et fabriques***, plus haut.

Par exemple, pour la classe `X`, on pourrait écrire la méthode de classe `creer()` visible à droite, et obtenir des instances de `X` en appelant `X::creer()`. La fonction globale amie `creerX()` est une alternative menant au même résultat.

```
#ifndef X_H
#define X_H
class X {
   X(); // privé
public:
   static X* creer() {
      return new X;
   }
   friend X* creerX();
   // ...
};
#endif
// dans le .cpp
#include "X.h"
X* creerX() {
   return new X;
}
// ...
```

En pratique, on choisirait habituellement d'offrir l'une ou l'autre des approches à la création d'un `X`, et dans le cas où on choisirait d'offrir les deux, on ferait passer l'une (la fonction globale) par l'autre (la méthode) pour réduire la redondance dans le code.

En fait, la clé pour empêcher l'instanciation automatique d'une classe n'est pas tant de rendre ses constructeurs privés que de qualifier son *destructeur* de privé.

En effet, toute instanciation automatique et locale à un sous-programme nécessite une destruction en fin de portée. Pour détruire un objet local et automatique, le sous-programme doit avoir accès à son destructeur. Si le destructeur d'une classe donnée est privé, alors seule cette classe et ses amis pourront en créer des instances automatiques.

```
class X {
   ~X();
public:
   X();
};
void p() {
   X x; // illégal!
}
```

[94] Pas de constructeur public pour `X`, pas de `new X` par quiconque d'autre que `X`. Si on veut permettre l'instanciation dynamique seulement, notre problème reste alors entier.

Détruire une instance dont le destructeur est privé

Un objet instancié dynamiquement doit éventuellement être détruit, du moins si l'on veut éviter des fuites de ressources.

On se doute qu'appliquer (de l'extérieur, outre de la part d'un ami) l'opérateur `delete` sur un pointeur menant vers une instance d'une classe dont le destructeur est privé soit une opération illégale… alors comment procéder à la destruction de cet objet?

L'idée est simple : il suffit de *laisser l'objet s'autodétruire*. La destruction de l'objet doit venir de l'intérieur.

```
class X {
   ~X();
public:
   X();
};
void p() {
   X *px = new X ; // légal
   delete px; // illégal (oups!)
}
```

En effet, ce n'est pas parce qu'un sous-programme utilisateur ne peut détruire un objet lui-même qu'il ne peut demander à l'objet de s'autodétruire.

Pour être en mesure de soumettre une demande de ce genre à un objet, il faut que cet objet dévoile une méthode publique vouée à la recevoir. Nommons cette méthode `detruire()`.

La définition minimale de `detruire()` sera `delete this;` qui est une opération valide puisque (par définition) l'objet aura été instancié dynamiquement et parce que `this` est un pointeur sur cet objet. L'instance active peut appliquer l'opérateur `delete` sur elle-même, ayant accès à ses propres membres privés.

```
class X {
   ~X();
public:
   X();
   void detruire() {
      delete this;
   }
};
void p() {
   X *px = new X; // légal
   px->detruire(); // Ok
}
```

L'aspect manuel de la destruction d'un objet muni d'un destructeur privé est un irritant très visible (faire confiance au code client est rarement sage). Dans l'exemple ci-dessus, le recours à une méthode d'instance spéciale (la méthode `detruire()`) et à un destructeur privé empêche, entre autres, le recours à `std::auto_ptr` pour automatiser la libération du pointé.

Heureusement, face à cet irritant, il existe une solution élégante. Imaginons tout d'abord qu'il existe plusieurs politiques de nettoyage possibles pour un type donné. Représentons deux de ces politiques sous la forme de foncteurs génériques :

- le foncteur `Meurs` appliquera l'opérateur `delete` à l'objet à détruire (par exemple un `int*`); et
- le foncteur `Detruis` invoquera la méthode `detruire()` de l'objet à détruire (par exemple un `X*`). La syntaxe utilisée suppose un pointeur ou quelque chose de syntaxiquement équivalent.

```
struct Meurs {
   template <class T>
      void operator()(T p) noexcept {
         delete p;
      }
};
struct Detruis {
   template <class T>
      void operator()(T p) noexcept {
         p->detruire();
      }
};
```

Nous présumerons que `Meurs` (appliquer simplement `delete`) est l'approche la plus typique et nous en ferons notre cas par défaut. Ainsi, `AutoNettoyeur` à droite définira une politique de nettoyage `RAII` *très* simple sur un `T*` et, par défaut, lui appliquera l'opérateur `delete`.

En utilisant un deuxième paramètre dans ce *template*, nous offrons toutefois la possibilité de spécifier une politique de nettoyage en fonction des besoins.

```
template <class T, class Lib = Meurs>
   class AutoNettoyeur {
      T *ptr;
      Lib libfct;
   public:
      AutoNettoyeur(T *ptr, Lib libfct = {})
         : ptr{ptr}, libfct{libfct} {
      }
      T *get() noexcept {
         return ptr;
      }
      const T *get() const noexcept {
         return ptr;
      }
      ~AutoNettoyeur() {
         libfct(ptr);
      }
   };
```

L'exemple à droite montre comment il est possible de tirer profit de cette façon de faire : `p0` applique implicitement la politique de nettoyage par défaut, alors que `p1` applique une politique de nettoyage choisie par le code client.

```
int main() {
   AutoNettoyeur<int> p0{new int{3}};
   AutoNettoyeur<X, Destructeur> p1{new X};
}
```

Cette façon de faire est un premier pas vers une technique de programmation alliant programmation générique et petits objets auxiliaires, la *programmation par politiques*, sur laquelle nous reviendrons ultérieurement.

Notez que c'est aussi très similaire à l'approche préconisée pour `unique_ptr` – en pratique, utilisez ce dernier!

À titre d'exercice, pourquoi ne pas écrire un `make_cleanable<T>(arg)` générant un `AutoNettoyeur<T>` selon les mêmes modalités que `make_unique<T>(arg)` pour le type `unique_ptr<T>`?

Empêcher la dérivation

Il arrive qu'on souhaite empêcher la rédaction d'enfants d'une classe donnée, donc de faire en sorte qu'une classe soit **terminale**. Cette idée est un concept en Java et dans les langages `.NET`, mais a longtemps été une technique en C++; depuis C++ `11`, le mot clé contextuel `final`, survolé dans [POOv01] et mentionné à nouveau un peu plus loin, permet d'y arriver simplement. Ce qui suit montre comment arriver à un résultat connexe avec un compilateur C++ `03`.

En Java, une classe spécifiée `final` ne peut avoir d'enfants. En C#, le même effet est obtenu en spécifiant une classe comme étant `sealed`. En `VB.NET`, une classe `NotInheritable` ne peut avoir d'enfants.

Une tactique simple pour faire en sorte qu'une classe soit terminale est de déclarer tous ses constructeurs privés, sans négliger les constructeurs par défaut et par copie (qu'on les implémente ou non). Il sera alors impossible à une classe d'être une classe enfant de la classe terminale, faute de pouvoir construire sa partie parent.

Un inconvénient de cette approche est qu'elle impose la mise en place d'un mécanisme de construction assistée, donc d'une fabrique, dans la classe terminale. L'exemple à droite y arrive avec `Terminale::creer()` mais d'autres stratagèmes (équivalents) sont possibles.

Cette approche a malheureusement l'effet secondaire d'interdire aussi l'instanciation automatique ou statique de la classe terminale.

Pour obtenir une classe terminale mais pouvant malgré tout être instanciée de manière automatique ou statique, il faut être un peu plus pervers. L'idée pour ce faire provient de **Bjarne Stroustrup** dans [StrouNoH] et n'est pas, de son propre aveu, très jolie.

```
class Terminale {
   Terminale() {
      // ...
   }
   Terminale(const Terminale&) {
      // ...
   }
public:
   ~Terminale() {
      // ...
   }
   // mieux: retourner un unique_ptr
   static Terminale *creer() {
      return new Terminale;
   }
};
struct X : Terminale {
};
int main() {
   // X x; // illégal!
   auto p = Terminale::creer();
   // ...
   delete p;
}
```

Le truc va comme suit :

- définir une classe impossible à construire sans privilèges spéciaux (ici, cette classe est `VerrouTerminal`) et faire de la classe terminale son amie (pour lui donner les privilèges en question);
- définir la classe terminale (nommée ici `Terminale`) et en faire un dérivé virtuel de la classe impossible à construire. Parce que la classe terminale est une amie de son parent, elle a les privilèges requis pour invoquer les constructeurs de son parent et peut donc être construite par elle-même;
- l'héritage virtuel force la classe la plus dérivée de la hiérarchie à invoquer les constructeurs des bases virtuelles. Quand nous instancions `Terminale`, celle-ci est la plus dérivée et a accès aux constructeurs privés du parent, mais quand nous tentons d'instancier un enfant de `Terminale`, alors la classe la plus dérivée devient l'enfant en question qui, lui, n'a pas les privilèges requis pour accéder aux membres privés de son distant ancêtre.

```
class VerrouTerminal {
   friend class Terminale;
   VerrouTerminal() = default;
   VerrouTerminal(const VerrouTerminal&) = default;
};
class Terminale : public virtual VerrouTerminal {
   // ...
public:
   Terminale() {
      // ...
   }
   // ...
};
int main() {
   Terminale t; // ok
   struct Oups : Terminale {
   };
   Oups oups; // illégal
}
```

Conséquemment, dans l'exemple ci-dessus, il sera impossible d'instancier la classe `Oups` car ses instances sont tenues d'invoquer le constructeur de leur ancêtre `VerrouTerminal` mais cette méthode ne leur est pas accessible.

Mot-clé `final`

Depuis C++ 11, les compilateurs peuvent supporter le mot-clé contextuel `final`, signifiant « qui ne sera pas spécialisé ». Ce mot peut être placé à côté d'un nom de classe ou apposé à une méthode :

```
struct B {
   virtual void f() = 0;
   virtual ~B() = default;
};
class X final : public B { // X n'aura pas d'enfants
   void f();
   // ...
};
class Y : public B { // Y pourra avoir des enfants
   void f() final; // ... mais ils ne spécialiseront pas f()
   // ...
};
```

Ce mot-clé est dit « contextuel » car il n'est pas réservé pour le langage de manière générale. Il est donc légal (mais de très mauvais goût) d'avoir par exemple une variable nommée `final` ou une fonction portant ce nom.

Annexe 00 – 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 02.0 : des mots et des lignes

Il y a une différence fondamentale entre lire une ligne et lire tous les mots d'une ligne. Lirait-on le même contenu au total dans un cas comme dans l'autre? Pourquoi?

La question de réflexion soulevée à droite a ceci de bien qu'elle peut se répondre par une démonstration empirique.

Imaginons par exemple le fichier texte `Entree.txt` ci-dessous :

```
J'aime mon prof
Il est chouette
  Chaque  mot     sur     cette   ligne   est     precede d'une   tabluation
```

Si la réponse à la question est qu'une stratégie de lecture ligne par ligne et une stratégie de lecture mot par mot donnent les mêmes résultats, alors deux programmes réalisant l'une et l'autre des tâches devraient produire, en sortie, le même résultat. J'ai évité les caractères accentués car leur traitement est plus subtil et obscurcirait notre propos.

Un programme de lecture mot à mot serait celui proposé à droite. Si vous l'exécutez, vous verrez s'afficher tous les mots du fichier `entree.txt` mais collés les uns aux autres.

Ceci tient du fait que la lecture sur un flux de texte avec l'opérateur `>>` escamote les blancs, ce qui est en temps normal le comportement attendu d'une lecture mot à mot.

Ajouter un délimiteur arbitraire (un espace, par exemple) après chaque mot ne réglerait pas le problème du fait que les sauts de ligne et les tabulations seraient perdus, remplacés par des espaces qui n'étaient pas sur le flux original. De même, si nous avions choisis de placer plusieurs espaces consécutifs dans le flux d'origine, notre choix arbitraire d'insérer un seul espace après chaque mot en sortie altérerait la forme du texte (ce qui détruirait, par exemple, l'indentation dans le cas où la source aurait été un programme).

```
//
// Lecture mot à mot
//
#include <fstream>
#include <iostream>
#include <string>
int main() {
   using namespace std;
   ifstream ifs{"entree.txt"};
   for(string mot; ifs >> mot; )
      cout << mot << ' ';
}
```

On pourrait évidemment indiquer au flux en entrée que nous ne souhaitons pas qu'il escamote les blancs en lecture (en lui appliquant le manipulateur de flux `std ::noskipws`), mais il faudrait alors lire le flux un caractère à la fois et nous perdrions l'idée-même de lecture mot à mot.

Un exemple de lecture ligne par ligne est proposé à droite. Si vous l'exécutez, vous verrez s'afficher à la sortie le contenu exact du fichier `Entree.txt`, avec peut-être un saut de ligne supplémentaire en sortie si le flux d'origine ne se terminait pas, lui, par un saut de ligne.

En effet, `std::getline()` consomme une ligne en entier, sans escamoter les blancs, allant jusqu'à un délimiteur choisi (qui est, par défaut, le symbole `'\n'` si le flux consomme des `char`) ou jusqu'à la fin du flux, mais ne conservant pas le délimiteur trouvé (ce qui explique l'application systématique de `std::endl` lors de chaque affichage).

Il y a donc bel et bien une différence fondamentale entre la lecture réalisée mot à mot et celle réalisée ligne par ligne.

```
//
// Lecture ligne par ligne
//
#include <fstream>
#include <iostream>
#include <string>
int main() {
   using namespace std;
   ifstream ifs{"entree.txt"};
   for (string ligne; getline(ifs,ligne); )
      cout << ligne << endl;
}
```

Réflexion 02.1 : assurer le respect des invariants

La question de réflexion soulevée à droite porte sur la question du respect des invariants. Évidemment, nous présumerons que l'objet respectait ses invariants avant l'appel, n'ayant pas de contrôle localement sur cette question.

Notez l'ordre des opérations dans `croitre()`. Est-ce qu'il serait sage de déplacer l'affectation de `n` à `cap` avant `new[]` ou avant `copy()`?

Rappelons le contexte de la question. La classe générique `Tableau` représente un tableau capable d'entreposer des éléments d'un type `T` donné (l'*alias* interne et public `value_type` correspond à `T`).

Si la capacité d'entreposage d'un `Tableau` est sur le point de devenir insuffisante, alors la méthode `croitre()` est invoquée ajuste les structures internes de l'objet de manière à amener cette capacité à un seuil adéquat. Dans le présent document, cette méthode est telle que présentée à droite.

```
void croitre() {
   static const size_type CAP_BASE = 128;
   const size_type n = capacity()?
      static_cast<size_type>(capacity()*1.5) :
      CAP_BASE;
   auto p = new value_type[n];
   try {
      copy(begin(), end(), p);
   } catch (...) {
      delete[] p;
      throw;
   }
   delete[] elems;
   cap = n;
   elems = p;
}
```

Si nous omettons la constante statique, initialisée très tôt dans l'exécution du programme, les opérations normales sont :

- l'initialisation d'une constante locale, `n`, représentant la capacité du `Tableau` suite à l'opération de redimensionnement;
- l'allocation du nouveau tableau de taille `n` à travers la variable `p`;
- la copie des éléments dans `elems` vers le nouveau tableau `p`;
- la suppression de l'ancien tableau `elems`;
- l'affectation de `p` vers `elems` (une copie de pointeurs, pas une copie élément à élément); et, pour conclure
- l'affectation de `n` à `cap`.

Aucune modification n'est apportée à `nelems` puisque seule la capacité du `Tableau` est modifiée dans cette méthode. Aucun nouvel élément n'y est ajouté. Une méthode doit faire son travail, ni plus ni moins, et le faire correctement.

Vous remarquerez que certaines opération dans `croitre()` peuvent lever une exception (ce qui explique qu'elle ne soit pas qualifiée *no-throw*) :

- l'allocation dynamique du tableau `temp`, qui peut lever une exception telle qu'un `bad_alloc` suite à l'invocation de `new[]` et qui implique `n` invocations silencieuses de `T::T()`; et
- la copie élément par élément réalisée par l'algorithme `copy()`, qui appliquera `size()` fois l'affectation d'un `T` à un autre `T` (`size()<n`).

Les plus alertes auront remarqué que créer n instances par défaut de `T` puis affecter `size()` fois un `T` à un autre est inefficace : les `size()` premiers éléments de `p`, idéalement, auraient été initialisés à partir d'une construction par copie.

Il se trouve qu'il est effectivement possible, en C++, d'atteindre ce niveau d'optimalité (les conteneurs standards le font, ce qui explique en partie le niveau de performance exceptionnel qu'ils parviennent à atteindre), mais par des techniques qui ne seront couvertes que dans des volumes ultérieurs de cette série.

L'implémentation suggérée est écrite avec soin car les changements d'états de l'instance active sont tous faits après la dernière opération susceptible de lever une exception.

Si `new[]` lève une exception, alors `bad_alloc` sera levé, le tableau `p` n'aura pas été créé, et `croitre()` ne se complétera pas normalement. Les états de l'instance active n'auront pas été modifiés, et l'objet en question demeurera dans un état respectant ses invariants. L'exception levée sera relancée dans le bloc `catch`.

Si l'un des constructeurs par défaut de `T` lève une exception pendant l'initialisation du tableau tout juste alloué avec `new[]`, alors le tableau sera considéré comme n'ayant jamais été créé, et tous les `T` déjà construits de ce tableau seront détruits. L'objet en question demeurera dans un état respectant ses invariants, et l'exception levée sera relancée dans le bloc `catch`.

Notez que si l'un des destructeurs de `T` lève une exception, alors la destruction des éléments déjà créés du tableau échouera et le programme tout entier deviendra instable, avec pertes de ressources et tout le tralala.

Répétons-le : il ne faut *jamais* lever d'exception dans un destructeur.

Enfin, si l'une des affectations réalisées par `copy()` lève une exception, le bloc `catch` supprimera `p` et laissera l'exception filtrer vers le code client. Les états demeurant tels qu'ils étaient avant l'invocation de la méthode, les invariants de l'objet seront respectés.

Si nous commençons à modifier les états de l'objet avant la réalisation d'opérations susceptibles de lever une exception, alors il nous faudra aussi mettre en place des opérations qui assureront le retour de l'objet dans un état correct si une exception survient. La logique associée à ces opérations peut devenir très complexe.

Par exemple, si nous éliminons la constante `n` pour utiliser directement `cap` tout au long de l'exécution de la méthode, alors `cap` contiendra la valeur que cet attribut devrait avoir une fois la croissance dûment complétée. Si une exception survient, alors il faudra trouver le moyen de ramener cette variable à son état antérieur avant que la méthode n'ait complété son exécution.

Négliger de remettre l'attribut dans un état correct impliquerait que `cap` ne serait plus un reflet fidèle de la capacité du `Tableau`. Les calculs basés sur cette valeur (ou sur la valeur retournée par la méthode `capacity()`) seraient alors erronées, ce qui peut entraîner des conséquences arbitrairement sévères.

Réflexion 02.2 : des types valeurs et des autres

La maxime de **Scott Meyers**, rappelée dans la question proposée à droite, est une maxime d'une grande pertinence pour la conception de types valeurs.

De même, la possibilité d'atteindre l'équivalence opératoire des types, au besoin, est un atout précieux. Cela dit, il n'existe pas de panacée, de remède universel à tous les maux.

La maxime *Do as the ints Do* de **Scott Meyers**, mentionnée à quelques reprises dans [POOv00], devrait-elle être appliquée pour tous les types?

Les types valeurs, pour lesquels la copie est sans effet secondaire et qui ont un comportement aussi simple que celui des entiers ou des autres types primitifs, sont partout en C++. Ces types peuvent être entreposés dans des conteneurs, passés par valeur, retournés par des fonctions… Les chaînes de caractères standards et les conteneurs ont d'ailleurs un tel comportement.

Cela dit, les objets polymorphiques ne sont pas des types valeurs. Un pointeur peut être copié à loisir, mais la duplication polymorphique du contenu pointé est une tâche complexe, qui requiert du clonage ou qu'on peut éviter par une application de l'idiome `Incopiable`. La destruction de l'objet pointé par un pointeur pose la question de la responsabilité (qui doit détruire ce pointé parmi tous ceux qui y ont accès?) et demande des manipulations explicites (invoquer `delete` sur le pointeur).

Dans la section ***Objets opaques et fabriques***, nous avons détaillé une manière de dissimuler les détails d'implémentation derrière une interface opaque, puis d'encapsuler cette abstraction derrière un type valeur. Cette démarche met en relief trois éléments clés du design `OO` :

- l'abstraction est utile, permettant au moins de dissimuler l'implémentation;
- les types valeurs sont utiles, se manipulant avec la simplicité des entiers;
- il y a au moins deux grandes familles d'objets, soit les types valeurs, accédés directement, et les abstractions polymorphiques, accédées indirectement.

Réflexion 02.3 : aléas des pointeurs bruts

Le code client se défait ici de la tâche de libérer les pointeurs sur les *mixins* à l'aide de pointeurs intelligents de la bibliothèque standard. Pourquoi recommande-t-on de faire en sorte que ces fonctions retournent des pointeurs intelligents plutôt que des pointeurs bruts?

La question de réflexion 02.3 nous amène à cogiter sur le choix malencontreux de rédiger des fonctions retournant des pointeurs bruts sur des objets alloués dynamiquement, et recommande le recours à des pointeurs intelligents.

Ce propos est indépendant de la question des *mixins*, en fait, bien que le contexte fasse en sorte que la question ait été posée suite à l'écriture de fonctions génératrices. Examinons la problématique en des termes plus généraux, plus abstraits.

Le programme simpliste proposé à droite utilise deux classes, X et Y, sur lesquelles ont ne sait que peu de choses. Nous présumerons que la construction d'une instance de l'une ou de l'autre de ces classes peut lever une exception, que ce soit un banal cas de std::bad_alloc ou quelque chose de plus pointu.

La fonction fx() sera génératrice d'un X et la fonction fy() sera génératrice d'un Y. Ici, le choix d'implémentation qui a été fait est de retourner les pointeurs obtenus suite à l'allocation dynamique de mémoire dans chaque cas sous forme brute.

La fonction f() reçoit un X* et un Y* alloués dynamiquement, les utilise (peu importe comment puis les détruit). Enfin, main() invoque la fonction f() à partir des pointeurs retournés par les fonctions génératrices fx() et par fy().

```
class X { /* ... */ };
class Y { /* ... */ };
X *fx() { return new X; }
Y *fy() { return new Y; }
void f(X *px, Y *py) {
   // ...
   delete px;
   delete py;
}
int main() {
   f(fx(), fy());
   // ...
}
```

Ce programme risque, en pratique, d'entraîner une fuite de ressources. Pour comprendre la raison de ce risque, il nous faut tout d'abord être conscients que l'ordre d'invocation de fx() et de fy() dans l'expression f(fx(),fy()) n'est pas déterminé par le standard, ce qui implique que certaines implémentations invoqueront fx() puis fy(), alors que d'autres invoqueront fy() puis fx() (évidemment, toutes les implémentations invoqueront f() après les deux autres). Le problème que nous examinerons sera le même peu importe l'ordre d'invocation, mais il nous faut en choisir un pour illustrer clairement le problème, alors nous présumerons que fx() est invoqué avant fy().

Imaginons que new X, dans fx(), ne lève pas d'exceptions. L'instance de X est donc créée, son constructeur est invoqué, ce qui réalise un ensemble arbitrairement complexe d'opérations que son destructeur est voué à nettoyer ultérieurement.

Imaginons maintenant que new Y, dans fy(), lève une exception. Qu'advient-il alors de l'appel f(fx(),fy())?

Clairement, l'instance de `Y` n'aura pas été construite. Cependant, l'instance de `X`, elle, demeurera construite, mais `f()` ne sera jamais invoquée, l'expression l'invoquant ayant mené à une levée d'exception.

Le `X*` généré sera perdu et le destructeur de ce `X` ne sera jamais invoqué. Les pertes de ressources qui en résulteront seront arbitrairement douloureuses.

Avant de penser qu'un moteur de collecte d'ordures nous aurait sauvé la peau, réfléchissez bien à ce dont aurait l'ait la finalisation du `X` en Java ou en C# et vous constaterez que le problème demeure entier (la récupération de la mémoire associée au `X` sera réalisée s'il y a collecte d'ordures, mais la finalisation demeurera problématique).

Les pointeurs bruts ont plusieurs avantages, mais ne sont pas responsables d'eux-mêmes. Un pointeur intelligent, par exemple `unique_ptr`, offre les services d'un pointeur mais détermine aussi une responsabilité quant à la gestion du pointé.

En retournant un `unique_ptr<X>`, la fonction `fx()` rend un objet responsable de la destruction éventuelle du pointeur nouvellement créé. Si l'exécution de `fy()` mène à une levée d'exception, l'objet temporaire responsable du `X` nouvellement créé sera détruit par la mécanique du langage, comme toute temporaire gérée automatiquement, et son destructeur libérera le pointé dont il est responsable (le `X` nouvellement créé) sera libéré.

```
class X { /* ... */ };
class Y { /* ... */ };
#include <memory>
using namespace std;
unique_ptr<X> fx() {
   return make_unique<X>();
}
unique_ptr<Y> fy() {
   return make_unique<Y>();
}
void f(unique_ptr<X> &px, unique_ptr<Y> &py) {
   // ...
}
int main() {
   f(fx(), fy());
   // ...
}
```

L'une des raisons d'être des pointeurs intelligents est qu'ils permettent d'éviter des fuites de ressources. Parfois, sans eux, il serait tout simplement impossible de prévenir des pertes de ressources sans modifier drastiquement l'écriture des programmes.

Évidemment, les pointeurs intelligents ne constituent pas un remède miracle. Encore faut-il les manipuler convenablement.

Réflexion 02.4 : instances et méthodes de classe

Faut-il absolument que `get()` soit une méthode de classe?

La section sur les singletons soulève une question en apparence banale, répétée à droite. La réponse raisonnable est simple, mais la réponse complète exige des nuances.

Souvenons-nous qu'un singleton est une classe dont il n'existe qu'une seule instance dans un programme. Pour assurer l'unicité de l'instanciation, on place cette opération sous la gouverne de la classe elle-même, et on met en place des mécanismes qui assureront l'impossibilité de dupliquer l'instance en question une fois celle-ci construite.

L'ébauche de code proposée à droite, qui est incorrecte pour un singleton, résume le contexte de la question.

```
class X {
   static X singleton;
public:
   X& get() noexcept {
      return singleton;
   }
};
```

La réponse simple à la question, aussi curieux que cela puisse paraître, est un argumentaire d'ordre philosophique : il faut que la méthode `get()` de `X` soit une méthode de classe puisque son rôle est de donner accès à la *seule* instance de `X` dans le programme. S'il fallait passer par une instance de `X` pour obtenir cette instance de `X`, alors comment aurions-nous pu accéder à l'instance en question au préalable? Qu'y aurait-il à la place des points d'interrogation dans l'expression suivante?

```
X &x = ???.get(); // accès à un membre d'instance... de quelle instance?
```

Évidemment, si `get()` est une méthode de classe, alors l'écriture est simple :

```
X &x = X::get(); // accès à un membre de classe de la classe X
```

Pour un singleton, donc, le cas semble clos. En général, par contre, la situation est un peu plus complexe.

En effet, examinons le programme (peu recommandable) proposé à droite. *Les enfants, ne faites pas ça à la maison*. À votre avis, est-ce un programme légal? Surtout, l'invocation de `get()` est-elle correcte?

```
#include "X.h"
int main() {
   X *p = {};
   X &x = p->get(); // !?!?!
}
```

La réponse, en toute honnêteté, est *oui* : ce programme (si laid soit-il) est légal, du moins en C++, et l'invocation à `get()` compilera et invoquera correctement la méthode en question. Suite à l'invocation, `x` référera bel et bien au singleton.

Force est d'admettre qu'il est possible d'invoquer une méthode de classe d'une classe donnée à partir d'un pointeur, *même nul*, de ce type. Les méthodes de classe, comme les fonctions globales, ne reçoivent pas de pointeur `this` lors des invocations. Dans l'exemple ci-dessus, le passage par le pointeur `p` influencera la méthode de classe invoquée de par le *type* de `p`, mais pas du tout par le contenu pointé par `p` (dont une méthode de classe ne se servirait pas, de toute manière, alors à quoi bon).

En pratique, bien que cet exemple compile et soit légal, n'écrivez pas de programmes sous cette forme (à moins d'avoir une très, très bonne raison, et de documenter à la fois vos raisons et votre démarche). Privilégiez une démarche plus claire et accédez aux singletons par des services qui sembleront raisonnables aux yeux des autres. Ici, passer par une méthode de classe est clairement la meilleure option.

Réflexion 02.5 : difficile de partager

Il y a un piège ici. Le voyez-vous?

Le rappel de cette question, à droite, ne rend pas justice à sa subtilité. Rappelons le contexte dans lequel la question fut posée.

Nous développions une classe chaine dont la représentation interne était partagée, ce partage reposant sur une classe interne `chaine::Rep`, dérivant de `Partageable`. La question accompagne la définition des méthodes `begin()` et `end()`, constantes ou non.

L'extrait ci-dessous illustre le problème de la définition naïve de ces méthodes (pour `chaine`) dans le texte.

Il se trouve qu'exposer un itérateur menant directement vers une représentation interne partagée permet, de manière inopinée, de modifier non seulement le contenu visé, mais aussi celui des autres objets qui partagent le même contenu.

```
chaine s0 = "allo";
chaine s1 = s0;
*begin(s0) = 'z'; // oups!
```

Le correctif minimal à appliquer à la classe `chaine` est de faire en sorte que toute invocation des méthodes `begin()` et `end()` donnant accès à une représentation interne partagée de manière non constante crée d'abord une copie non partagée de la représentation, puis donne un accès direct à celle-ci. L'implémentation de l'opérateur `[]` non constant de chaine peut servir d'exemple en ce sens.

Malheureusement, cela ne suffit pas pour en arriver à du code vraiment sans risque. Le code à droite en fait la démonstration en permettant la modification de `s0` et de `s1` à partir d'un itérateur obtenu préalablement.

```
chaine s0 = "allo";
auto it = begin(s0);
chaine s1 = s0;
*it = 'z'; // oups!
```

Concrètement, l'optimisation *Copy On Write* et le partage de représentations internes ne peut être utilisé sans danger que pour les objets immuables. Si nous souhaitions utiliser la classe chaine de manière sérieuse, il faudrait en éliminer l'opérateur `[]` dans sa déclinaison non constante, tout comme il faudrait en éliminer le type `iterator` (non constant) et les méthodes donnant un accès direct et non constant à son contenu.

Réflexion 02.6 : choix de généricité

Étant donné le foncteur `Sequence` ci-dessous, capable de générer sur demande le prochain élément d'une séquence d'éléments d'un type donné, alors pourrait-on exprimer la généricité de ce foncteur sur une base méthode plutôt que sur la base de son type?

Nous avons utilisé la généricité sur la base du type pour notre foncteur `Sequence`. Aurait-il été possible d'utiliser la généricité sur une base méthode?

La réponse à cette question est simple : notre foncteur base son opération sur un état, qui représente la prochaine valeur à produire. Cet état est typé, évidemment.

Si nous avons un état du type de la séquence pour l'objet tout entier, alors la généricité sur la base de la méthode ne serait pas appropriée

```
template <class T>
   class Sequence {
      T cur {};
   public:
      Sequence() = default;
      Sequence(T init) : cur{init} {
      }
      T operator()() {
         return cur++;
      }
   };
```

La généricité sur une base méthode est appropriée aux cas où les types impliqués dans l'opération ne dépendent que de l'opération en cours. Ici, nous avons une opération qui, à partir d'appels successifs, en vient à générer une suite d'éléments *de même type*. La généricité requise transcende la durée de l'opération : ce n'est pas une séquence au sens large qui sera générée, mais bien *une séquence d'un type précis*.

Annexe 01 – Assistants en tant qu'alternative aux *mixins*

Nous avons couvert, dans ce document, une approche par mixin avec, pour illustration, l'implémentation d'un tableau dynamique. Il est évidemment possible d'en arriver au même résultat avec d'autres approches que celle des *mixin*. En particulier, nous pourrions utiliser une combinaison de classes collaborant entre elles.

Un exemple bien connu repose sur des **objets assistants**, comme le sont les allocateurs standards de STL (`std:allocator`)[95], dont le rôle est de représenter une stratégie d'allocation de mémoire. Un allocateur standard est une chose à la fois simple et complexe, étant lié intimement à des stratégies sophistiquées de gestion dynamique de la mémoire.

La classe construite ici est inspirée des allocateurs standards mais n'en est pas un; ne vous méprenez pas!

Comme dans l'implémentation par *mixin*, le code sera proposé d'un seul tenant, que vous pourrez organiser de manière convenable par vous-mêmes.

Notre implémentation alternative de tableau dynamique ira comme suit : une abstraction, que nous nommerons `AllocateurBase`, sera utilisée par la classe axiome, renommée `BaseTableauCar`, pour créer et faire croître la séquence représentant le contenu du tableau.

Nous découplerons les politiques de croissance du tableau (ceci pourrait permettre, avec quelques ajustements au code proposé, de modifier la politique de croissance pendant la vie du tableau dynamique, chose illégale si la politique est inscrite à même la classe, par héritage.

La classe `AllocateurBase` sera une abstraction dont les dérivés offriront deux services, l'un pour créer une séquence de caractères et l'autre pour faire croître la taille d'une séquence tout en y conservant le contenu original.

Puisque le pointeur menant vers le tableau brut résidera dans une classe (dans le tableau dynamique) mais sera modifié par une autre classe (`AllocateurBase`), nous devrons passer une indirection vers un pointeur.

Cette indirection aurait pu être un pointeur sur un pointeur (un `value_type**`) mais j'ai privilégié une référence sur un pointeur, ce qui allège l'écriture.

```
struct AllocateurBase {
   class TailleIllegale { };
   class SequenceIllegale { };
   using size_type = int;
   using value_type = char;
   using pointer = value_type*;
   virtual size_type
      creer_sequence(pointer&) = 0;
   virtual size_type
      croitre_sequence(pointer&, size_type) = 0;
   virtual ~AllocateurBase() = default;
};
```

Petit truc syntaxique : pour éviter les questionnements sur la priorité des opérateurs `&` et `*` dans l'écriture d'une référence sur un pointeur, notre classe définit un type interne et public `pointer`, équivalent à `value_type*`, puis manipule des `pointer&` au besoin.

[95] Voir http://www.sgi.com/tech/stl/Allocators.html pour plus de détails.

Nous utiliserons la classe `BaseTableauCar` pour remplacer la classe axiome du modèle par *mixin*.

La principale différence entre cette implémentation et la précédente tient à la délégation des invocations de création et de croissance de la séquence vers les services de l'allocateur.

Dans notre *mixin*, souvenons-nous en, nous avions plutôt appliqué l'idiome `NVI` pour déléguer, de manière encadrée, ces demandes de services vers les implémentations plus spécialisées qui seront offertes par les classes dérivées.

Remarquez que le constructeur n'admet que des allocateurs non nuls. Puisqu'il n'y a pas de services offerts dans cette implémentation pour modifier l'allocateur, l'encapsulation nous permet de prendre pour acquis que l'attribut d'instance `alloc` demeurera non nul pour l'existence entière de l'objet, ce qui explique que les délégations dans `creer()` et dans `croitre()` ne valident pas ce pointeur.

```
#include "incopiable.h"
class BaseTableauCar : Incopiable {
public:
   class AllocateurIllegal { };
   using value_type AllocateurBase::value_type;
   using size_type = AllocateurBase::size_type;
private:
   AllocateurBase *alloc;
   size_type cap {};
   size_type taille {};
protected:
   value_type *tableau = nullptr;
   void ajouter_brut(value_type c) {
      if (!tableau) throw AllocateurBase::SequenceIllegale{};
      tableau[size()] = c;
      ++taille;
   }
   void nouvelle_capacite(size_type n) {
      cap = n;
   }
public:
   size_type capacite() const noexcept {
      return cap;
   }
   size_type size() const noexcept {
      return taille;
   }
   bool plein() const noexcept {
      return size() >= capacite();
   }
   bool vide() const noexcept {
      return !size();
   }
   size_type creer() {
      return alloc->creer_sequence(tableau);
   }
   size_type croitre() {
      return alloc->croitre_sequence(tableau_,capacite());
   }
protected:
   BaseTableauCar(AllocateurBase *ab) : alloc{ab} {
      if (!ab) throw AllocateurIllegal{};
   }
   ~BaseTableauCar()  {
      delete[] tableau;
      delete alloc;
   }
};
```

L'interface visible de notre tableau dynamique demeurera une classe `TableauCar`, comme dans la version procédant par *mixin*.

Vous remarquerez d'ailleurs que les deux implémentations sont très semblables.

Une nuance qu'il importe de relever est que les invocations de `creer()` et de `croitre()` ne sont pas, ici, des appels frères, du fait qu'ils mènent plutôt (à travers le parent) vers les services de l'assistant d'allocation.

```
class TableauCar : BaseTableauCar {
   int ncroiss {};
public:
   using BaseTableauCar::capacite;
   void ajouter(value_type c) {
      if (vide())
         nouvelle_capacite(creer());
      if (plein()) {
         nouvelle_capacite(croitre());
         ++ncroiss;
      }
      ajouter_brut(c);
   }
   int nb_croissances() const noexcept {
      return ncroiss;
   }
   TableauCar(AllocateurBase *ab)
      : BaseTableauCar{ab} {
   }
};
```

Les fonctions génératrices auront la même signature que dans la version par *mixin*, ce qui signifie que le code client devrait pouvoir utiliser l'une ou l'autre des approches sans problème.

```
// ...inclusions et using...
unique_ptr<TableauCar> CreerTableauIncrementiel();
unique_ptr<TableauCar> CreerTableauExponentiel();
```

La réflexion `02.3`, à l'***annexe 00***, est tout aussi pertinente avec cette implémentation qu'avec la précédente.

L'allocateur incrémentiel sera un dérivé public d'`AllocateurBase`, puisqu'il servira à titre de spécialisation de son parent, et mettra en place une politique de croissance fortement semblable à celle de la version incrémentielle implémentée par *mixin*.

La seule véritable différence entre la version par *mixin* et celle-ci, en fait, tient au passage en paramètre de la séquence à modifier (la version par *mixin* obtenait le tableau à modifier de son parent, en vertu de l'héritage).

```
class AllocateurIncrementiel
   : public AllocateurBase {
   enum {
      INITIALE = 100, INCREMENT = 50
   };
public:
   size_type creer_sequence(pointer &seq) {
      seq = new value_type[INITIALE];
      return INITIALE;
   }
   size_type croitre_sequence
      (pointer &seq, size_type taille) {
      if (!seq) throw SequenceIllegale{};
      if (taille <= 0) throw TailleIllegale{};
      const size_type cap = taille + INCREMENT;
      auto p = new value_type[cap];
      copy(seq, seq+taille, p);
      delete[] seq;
      seq = p;
      return cap;
   }
};
```

Les remarques apposées à l'allocateur incrémentiel, plus haut, conviennent aussi dans le cas de l'allocateur exponentiel.

```
class AllocateurExponentiel
   : public AllocateurBase {
   enum {
      INITIALE = 100, FACTEUR = 2
   };
public:
   size_type creer_sequence(pointer &seq) {
      seq = new value_type[INITIALE];
      return INITIALE;
   }
   size_type croitre_sequence
      (pointer &seq, size_type taille) {
      if (!seq) throw SequenceIllegale{};
      if (taille <= 0) throw TailleIllegale{};
      const size_type cap = taille * FACTEUR;
      auto p = new value_type[cap];
      copy(seq, seq+taille, p);
      delete[] seq;
      seq = p;
      return cap;
   }
};
```

Les fonctions génératrices sont banales mais méritent d'être raffinées. Une explication détaillée est offerte dans l'***annexe 00***, réflexion 02.3.

```
unique_ptr<TableauCar> CreerTableauIncrementiel() {
   return make_unique<AllocateurIncrementiel>();
}
unique_ptr<TableauCar> CreerTableauExponentiel() {
   return make_unique<AllocateurExponentiel>();
}
```

Le programme principal, enfin, sera identique à celui utilisant une implémentation par *mixin*, et donnera les mêmes résultats.

```
// ...inclusions et using ...
int main() {
   auto pInc = CreerTableauIncrementiel();
   auto pExp CreerTableauExponentiel();
   const int NB_INSERTIONS = 1'000;
   for (int i = 0; i < NB_INSERTIONS; i++) {
      pInc->ajouter('A');
      pExp->ajouter('A');
   }
   cout << "Capacité finale de pInc: "
        << pInc->capacite() << '\n'
        << "Capacité finale de pExp: "
        << pExp->capacite() << "\n\n"
        << "Nb accroissements de pInc: "
        << pInc->nb_croissances() << '\n'
        << "Nb accroissements de pExp: "
        << pExp->nb_croissances() << endl;
}
```

Annexe 02 – Comment ne pas implémenter l'affectation

Nous avons vu, dès [POOv00], une technique saine pour implémenter l'affectation, technique nommée *idiome d'affectation sécuritaire*. Cet idiome repose sur un constructeur de copie bien écrit, sur un destructeur correct et sur une méthode `swap()` efficace. Il permet d'implémenter presque mécaniquement l'opérateur d'affectation et tend à donner des résultats près de l'optimalité quand l'implémentation sous-jacente de l'objet le permet.

Il est évidemment possible (et parfois préférable) d'implémenter l'affectation de manière plus manuelle. L'aspect fondamental de l'affectation en programmation, surtout pour des types valeurs, fait en sorte qu'il est essentiel d'éviter certains écueils en l'implémentant.

En premier lieu, *à moins d'avoir une bonne raison* (car il y en a), n'implémentez la Sainte-Trinité pour une classe donnée que si au moins l'un de ses attributs d'instance n'est pas un type valeur. En temps normal, le code généré par le compilateur est optimal pour les types valeurs.

Certains, en particulier sur Internet, préconisent la pratique suivante (à droite) pour implémenter l'affectation pour une classe `X` donnée. **C'est une très mauvaise pratique**.

L'intention derrière la pratique suggérée est noble : la programmeuse/ le programmeur cherche à réduire la redondance de code en exploitant le destructeur pour nettoyer l'instance active puis le constructeur par copie pour réinitialiser l'instance active en tant que copie du paramètre.

Cette pratique profite de deux éléments méconnus du langage C++ (et dont nous reparlerons dans [POOv03], section ***Gestion avancée de la mémoire***) :

- le destructeur ne libère pas la mémoire de l'objet détruit (ce rôle est dévolu à un autre mécanisme), mais finalise l'objet en lui donnant une dernière opportunité de laisser le programme dans un état correct. Un appel explicite au destructeur ne fait donc que laisser un objet nettoyer l'espace qu'il occupe; et
- il est possible de construire un objet à un endroit précis en mémoire à l'aide d'une syntaxe particulière de l'opérateur `new` qu'on nomme le *placement `new`*. À droite, un `X` est construit par copie à l'adresse `this`.

```
class X {
   // ...
public:
   X(const X &x) {
      // ...
   }
   ~X()  {
      // ...
   }
   X& operator=(const X &x) {
      if (this != &x) {
         // destructeur explicite
         this->~X();
         // recycler l'adresse this
         new (this) X{x};
      }
      return *this;
   }
   // ...
};
```

Cette technique cherche à économiser de l'allocation dynamique de mémoire en indiquant au compilateur de placer l'objet là où se trouve `this`, zone mémoire déjà disponible par définition.

Le problème très important de cette stratégie est qu'elle présume que l'objet à copier est une instance de la classe `X` alors que ce n'est *absolument* pas certain.

En fait, le paramètre `x` peut être une instance de `X` ou de n'importe quelle classe dérivée de `X`. Il est impossible, de l'intérieur d'un constructeur ou d'un opérateur d'affectation, de savoir si

la méthode active est appelée pour la classe terminale de l'objet ou pour un parent quelconque d'une autre classe.

Sachant cela, la destruction par un appel explicite à `this->~X()` risque d'invoquer un destructeur virtuel, détruisant ainsi non seulement la partie `X` et ses parties parentes mais aussi toutes ses parties dérivées.

En retour, l'invocation de `new (this) X(x)` ne construira qu'un `X` copié de `x` et ses parents, mais pas ses parties enfant. Les probabilités sont donc immenses que l'objet résultant soit un objet partiel, instable, et que la prochaine invocation de son destructeur (la vraie, disons) cherche à détruire aussi la partie enfant qui n'aura pas été recréée… Boum!

Retour sur l'idiome sécuritaire d'affectation

Tel que rappelé plus haut, l'idiome d'affectation sécuritaire repose sur la construction par copie, la destruction et une méthode `swap()`. De son côté, une méthode `swap()` typique invoquera `std::swap()` attribut par attribut, et de lèvera pas d'exceptions si l'affectation des attributs n'en lève pas.

Ceci explique en partie le choix de plusieurs conteneurs standards d'implémenter leurs attributs sous forme de pointeurs : puisque les pointeurs sont des primitifs, échanger deux pointeurs est une opération qui ne risque jamais de lever une exception.

De ce fait, la méthode `swap()` est typiquement *no-throw*, simple et très efficace (complexité constante, $O(1)$), même pour des types aussi complexes que des conteneurs standards.

Règle générale, l'idiome d'affectation sécuritaire mène à une implémentation de l'affectation non seulement simple et concise, mais aussi pratiquement optimale :

- il faut dupliquer les attributs de l'opérande de droite, ce qui est fait par le constructeur de copie (à droite, cette copie est `X(x)`, ce qui fait d'elle un `X` anonyme);
- il faut nettoyer l'état de l'opérande de gauche avant affectation, ce qui est fait par le destructeur de la temporaire anonyme; et
- la méthode `swap()` est typiquement faite en temps constant et sans risque de lever une exception.

```
class X {
   // ...
public:
   X(const X&);
   ~X();
   void swap(X &x) noexcept {
      // ...
   }
   X& operator=(const X &x) {
      X{x}.swap(*this);
      return *this;
   }
   // ...
};
```

Si l'affectation est exprimée manuellement, on en arrive à la même structure, mais avec redondance (nettoyage dans le destructeur **et** dans l'affectation, copie dans le constructeur de copie **et** dans l'affectation). Épargner `swap()` ne donne à peu près rien, alors que l'idiome sécuritaire d'affectation épargne le coûteux test sur l'identité de `this` et du paramètre reçu par l'opérateur d'affectation (`if(this!=&x)`…).

Annexe 03 – Documenter efficacement

Il existe plusieurs approches à la documentation efficace du code. En ces temps de méthodologies agiles, plusieurs préconiseront de mettre l'accent sur une documentation minimale, idéalement à même le code, pour éviter que le code et sa documentation ne viennent à diverger.

Les crédos usuels de la saine programmation (noms significatifs, sous-programmes courts et clairs, commentaires pour les sections de code plus complexes, *etc*.) doivent être respectés, mais quelques éléments de documentation sont très près des idées véhiculées dans ce document.

Pour chaque classe, indiquez clairement les **invariants** à maintenir. Les invariants, tel que nous l'avons vu plus haut, dans la section ***Programmation générique appliquée***, sont ces caractéristiques qui, pour un objet, doivent en tout temps (une fois l'objet construit, et jusqu'au début de sa destruction) s'avérer.

Les invariants d'un objet sont ce que l'encapsulation cherche à préserver. Définir clairement les invariants permet de définir les contraintes de validité d'un objet.

L'encapsulation permet à l'objet de garantir sa propre intégrité, donc d'assurer le respect de ses invariants. Sans encapsulation, le code client pourrait briser les conditions des invariants et les objets ne pourraient être responsables de leur propre condition.

Pour chaque élément d'un programme (classe, instance, sous-programme, attribut, variable, constante, *etc*.), identifiez clairement son rôle : ce qu'il fait, ce qu'il représente, ce à quoi il sert. Parfois, ce rôle sera très pointu, alors que dans d'autres cas, ce rôle sera très abstrait, très général. Idéalement, le rôle sera évident à partir du contexte, que ce soit dû à un nom significatif parce que la situation (la classe, le sous-programme) ne laisse planer aucune ambiguïté.

Les commentaires viendront en renforcement au contexte et aux noms utilisés dans les cas où ceux-ci ne suffisent pas, et seront synchronisés avec le code. Un commentaire faux ou inexact est dommageable à la compréhension et nuit à la productivité.

Pour chaque sous-programme, indiquez clairement la **complexité algorithmique en temps et en espace** (si cela s'avère pertinent). Cela permettra aux autres sous-programmes de construire sur la base des vôtres et d'indiquer eux-aussi leur propre complexité algorithmique. Il peut arriver que la complexité soit statique (résolue à la compilation), surtout en programmation générique.

Prenez par exemple la fonction générique `trouver_si()`, ci-dessous (étrangement semblable à `std::find_if(debut,fin,pred)`). Sa complexité algorithmique est $O\big(n \times O(pred)\big)$ où $n = distance(debut, fin)$ est le nombre d'éléments dans `(debut..fin(` du fait que `pred()` y est appelé pour chaque élément, du moins dans le pire cas.

Indiquer cette complexité facilite à la fois la mise au point de fonction utilisant `trouver_si()` et celle des prédicats qui lui seront donnés pour faire son travail.

Par exemple, si le prédicat `pred()` est $O(1)$, alors `trouver_si()` sera de complexité linéaire, alors que si `pred()` est de $O(n)$, alors `trouver_si()` sera de complexité quadratique.

```
template <class It, class Pred>
   It trouver_si(It debut, It fin, Pred pred) {
      for (; debut != fin; ++debut)
         if (pred(*debut))
            return debut;
      return fin;
   }
```

Pour chaque sous-programme, incluant les constructeurs, indiquez clairement les **préconditions**. On entend par préconditions l'état du système avant l'invocation du sous-programme, en particulier l'état de l'objet (dans le cas d'une méthode) et les valeurs des paramètres (pour les sous-programmes en général).

Un sous-programme, surtout s'il est publiquement accessible, validera le respect de ses préconditions avant de réaliser sa tâche. Une violation des préconditions d'un sous-programme mène habituellement à la levée d'une exception ou, dans le cas d'un problème grave, au plantage volontaire d'un programme (par un `assert`). Certains distingueront d'ailleurs préconditions (doit être respecté pour que le fonctionnement soit correct, peut-être sous réserve de levée d'exception) et **assertions** (programme invalide si non respecté; plantera à la compilation ou à l'exécution, selon la nature de l'assertion).

Enfin, **pour chaque sous-programme**, incluant les constructeurs, indiquez clairement les **postconditions**, incluant les effets secondaires de leur invocation. On entend par postconditions l'état du système suite à l'invocation du sous-programme, en particulier l'état de l'objet (dans le cas d'une méthode).

Pour un exemple très complet, examinez la table *Unordered associative container requirements* sur le site suivant :

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html

Cet exemple en est un parmi plusieurs qui couvre les attentes de certains conteneurs de la bibliothèque standard de C++ 11.

Annexe 04 – Quelques trucs désuets

Ce qui suit regroupe quelques fonctionnalités et techniques qui, bien que pertinentes avec C++ 03, ne le sont plus depuis C++ 11.

Foncteurs et autodocumentation

En pratique, il a longtemps été considéré sage de garnir les foncteurs d'une forme documentation quant aux types sur lesquels ils opèrent, du moins lorsqu'il est possible de le faire.

L'*autodocumentation* décrite ici est basée sur les types. Conséquemment, elle n'est applicable qu'à la généricité sur la base des types, pas à celle sur la base des méthodes.

Deux classes génériques, `unary_function` et `binary_function` (de la bibliothèque `<functional>`) définissent les types des paramètres et de la valeur retournée par un foncteur à un paramètre (fonction *unaire*) et à deux (fonction *binaire*) paramètres.

L'utilisation de ces classes est très simple, et permet aux algorithmes d'offrir un support de meilleure qualité pour les opérations auxquelles ils ont recours. De plus, utiliser (de manière convenable) ces classes n'entraîne aucun coût en taille ou en vitesse, leur apport étant strictement statique.

En bref, `unary_function` ressemble à ce qui est proposé à droite.

```
template<class Arg, class Result>
   struct unary_function {
      typedef Arg argument_type;
      typedef Result result_type;
   };
```

De son côté, `binary_function` ressemble au code proposé à droite.

```
template<class Arg1, class Arg2, class Result>
   struct binary_function {
      typedef Arg1 first_argument_type;
      typedef Arg2 second_argument_type;
      typedef Result result_type;
   };
```

Dans un cas comme dans l'autre, ces classes sont strictement documentaires. Elles offrent une manière standard de nommer les types des paramètres et de la valeur de retour.

La manière la plus simple d'utiliser ces classes documentaires est d'en dériver, tout simplement, comme dans le cas de la classe `Carre` proposée à droite.

```
struct Carre
   : std::unary_function<double,double> {
   result_type operator()(argument_type val) {
       return val * val;
   }
};
```

Les types internes et publics `argument_type` et `result_type` y sont tous deux hérités du parent documentaire.

La version à droite est probablement préférable à la précédente, du fait qu'elle offre les mêmes services et le même niveau de performance tout en couvrant étant plus générale et en couvrant une gamme de types plus vaste.

```
template <class T>
   struct Carre : std::unary_function<T,T> {
      result_type operator()(argument_type val) {
         return val * val;
      }
   };
```

Ici, sans que ses instances ne soient même un seul byte plus grosses en mémoire, grâce entre autres à l'optimisation EBCO [POOv01], la classe Carre<T> s'est garnie de deux noms internes standards la décrivant :

- le type Carre<T>::result_type, qui est le type de ce que retourne le foncteur; et
- le type Carre<T>::argument_type, qui est le type de ce que ce foncteur unaire prend en paramètre.

Les types n'ont d'existence réelle que lors de la compilation et ne laissent pas de trace à l'exécution du programme. Cependant, il est possible d'utiliser ces noms pour écrire d'autres algorithmes. Pour donner un exemple simple, sujet à *beaucoup* d'améliorations (entre autres parce qu'il ne peut être utilisé qu'avec des tableaux de T, pas avec des séquences arbitraires) :

```
template <class T>
   typename Carre<T>::result_type SommeCarres(T *debut, T *fin)
   {
      typename Carre<T>::result_type somme = 0;
      for (Carre<T> c; debut != fin; ++debut)
         somme += c(*debut);
       return somme;
   }
```

L'algorithme SommeCarres() fait la somme des carrés d'une séquence contiguë en mémoire (un tableau brut) d'éléments d'un certain type T et cumule cette somme *dans une variable du type de ce que retournera le foncteur c de type Carre<T>*.

Soyez d'ailleurs prudent(e)s : ce type n'est pas *nécessairement* T.

En effet, quelqu'un pourrait décider de spécialiser Carre<T> pour le type short et décider que cette classe s'exprime ainsi :

```
template <>
   class Carre<short> : unary function<short,int> {
      result_type operator()(argument_type val) {
         return static cast<result type>(val) * val;
      }
   };
```

sans que cela ne modifie le comportement général de Carre<T> pour d'autres types T.

Il devient donc possible de déterminer de manière standardisée, à partir des types internes au foncteur, les types des variables à utiliser pour en tirer profit. Un gain de généralité à coût zéro est un apport non négligeable à la qualité des programmes.

Pourquoi ceci est-il désuet?

Cette approche est désuète parce que C++ est trop puissant pour elle, disons-le ainsi. Examinez la classe suivante :

```
struct Cube
   int operator()(int n) const {
      return n * n * n;
   }
   double operator()(double n) const {
      return n * n * n;
   }
};
```

Si nous souhaitions la documenter par des types internes tels que `result_type` ou `argument_type`, quels types utiliserions-nous ici? Le problème est exacerbé par la généricité, comme en fait foi ce qui suit :

```
struct Cube {
   template <class T>
      T operator()(T n) const {
         return n * n * n;
      }
};
```

Clairement, dans ce cas, les types impliqués dépendent du contexte, du code client et de ses besoins. Ainsi, avec C++ 11, nous travaillons à partir des traits [POOv03] et d'opérateurs tels que `decltype` [POOv03] qui permettent par exemple d'exprimer une chose telle que « Si `c` est une instance de `Cube`, alors `result_type` est le type de ce que retourne c quand je lui passe un `float` » :

```
// ...
Cube c;
using result_type = decltype(c(float{}));
```

Le foncteur `std::mem_fun`

La fonction génératrice `mem_fun()`, pour *Member Function*, tirée de `<functional>`, permet d'utiliser comme foncteur une méthode spécifique d'une classe autre que la méthode `operator()`. Il est aussi possible de définir des délégués en C++ (la bibliothèque *Boost* le fait, la bibliothèque *Loki* le fait aussi), mais il faut alors tenir compte des divers cas possibles.

Imaginons qu'on veuille créer un vecteur de pointeurs de listes d'entiers. On remplira d'abord deux listes d'entiers, puis on créera vecteur dans lequel on insérera l'adresse de nos deux listes originales.

L'utilisation de pointeurs est importante ici dû à la manière dont `mem_fun()` est rédigée. Le foncteur généré par `mem_fun_ref()` a le même comportement mais est applicable à des objets. Les noms distincts sont une aberration historique.

L'utilisation de `mem_fun()` ici apparaîtra dans le `for_each()` à la fin du programme, pour appliquer la méthode `sort()` de `list`, en remplacement du `sort()` générique, à chaque liste du vecteur.

Ce faisant, on appliquera comme foncteur une méthode d'instance qui n'a pas été conçue pour servir de cette manière.

```
#include <functional>
// ...autres inclusions et using...
int main() {
   list<int> lst0, lst1;
   for(int val; cin >> val; ) lst0.insert(begin(lst0), val);
   for(int val; cin >> val; ) lst1.insert(begin(lst1), val);
   vector<list<int>*> v;
   v.push_back(&lst0);
   v.push_back(&lst1);
   for_each(begin(v), end(v), mem_fun(&list<int>::sort));
}
```

Il se trouve que `mem_fun()` crée un foncteur à partir de la méthode reçue en paramètre. Cette fonction exploite les opérateurs `::*`, `.*` et `->*` (que nous n'avons pas couverts jusqu'ici) dans le but de créer un type dont la méthode `operator()` déléguera son travail à un appel vers une méthode spécifique d'une instance d'une autre classe, soit la méthode passée en paramètre à `mem_fun()`.

Il n'y a donc pas de magie à l'œuvre ici : `std::mem_fun()` est rédigée en C++ tout ce qu'il y a de plus standard. La magie apparente tient simplement du fait qu'elle utilise des opérateurs peu connus du langage.

Pourquoi ceci est-il désuet?

Ceci est désuet parce que `std::bind()` et `std::function()` offrent les mêmes fonctionnalités (et plus!) de manière plus homogène et plus efficace. Voir ***Délégués en C++ – `std::function`*** et ***Le lieur général `std::bind()`*** pour plus de détails.

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!

Dave Abrahams	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#david_abrahams
Andrei Alexandrescu	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#andrei_alexandrescu
Patrick Boutot	Étudiant de la cohorte 02 du Diplôme de développement du jeu vidéo (DDJV) à l'Université de Sherbrooke
André Louis Caron	Maître en informatique de l'Université de Sherbrooke, spécialisé en imagerie.
Alonzo Church	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#alonzo_church
Jesse Emond	Étudiant au baccalauréat à l'Université de Sherbrooke à l'hiver 2017
Mathias Gaunard	Je n'ai malheureusement pas d'information à son sujet pour le moment.
Anders Hejlsberg	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#anders_hejlsberg
Jaakko Järvi	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#jaakko_jarvi
Nicolai Josuttis	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#nicolai_josuttis
Andrew Koenig	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#andrew_koenig
Donald E. Knuth	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#donald_knuth
Scott Meyers	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#scott_meyers
Barbara Moo	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#barbara_moo
Manuel Parent	Étudiant de la cohorte 09 du Diplôme de développement du jeu vidéo (DDJV) à l'Université de Sherbrooke
Giuseppe Peano	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#giuseppe_peano
Pierre Prud'homme	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#pierre_prudhomme
Alexander Stepanov	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#alexander_stepanov
Bjarne Stroustrup	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#bjarne_stroustrup
Herb Sutter	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#herb_sutter
Alan Turing	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#alan_turing
David Vandevoorde	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#david_vandevoorde
John von Neumann	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#john_von_neumann

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.

[AccCpp] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#accelerated-cpp

[BeckMov] http://thbecker.net/articles/rvalue_references/section_01.html

[BoostExcS] http://www.boost.org/more/generic_exception_safety.html

[BoostGen] http://www.boost.org/more/generic_programming.html

[CppPL] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#cpp-programming-language

[CppTemp] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#cpp-templates-complete-guide

[EffStl] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#effective-stl

[UnGenFct] http://cpp-next.com/archive/2012/09/unifying-generic-functions-and-function-objects/

[hdCont] http://h-deb.clg.qc.ca/Sujets/Parallelisme/continuation.html

[hdFut] http://h-deb.clg.qc.ca/Sujets/Parallelisme/futures.html

[hdIdiom] http://h-deb.clg.qc.ca/Sujets/Developpement/Schemas-conception.html#idiome

[hdSFINAE] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/SFINAE.html

[hdSingSt] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/CPP--Singletons-statiques.html

[hdSSO] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Implementer-SSO.html

[hdStl] http://h-deb.clg.qc.ca/Liens/Langages-programmation--Liens.html#stl

[hdStrConv] http://h-deb.clg.qc.ca/Sujets/AuSecours/Convertir-ASCIIZ-Unicode-avec-pointNET.html

[hdVecTab] http://h-deb.clg.qc.ca/Sources/comparatif_vecteur_tableau.html

[ImplMovGo] http://cpp-next.com/archive/2010/10/implicit-move-must-go/

[JAM] http://www.disi.unige.it/person/LagorioG/jam/

[JavaMulH] http://csis.pace.edu/~bergin/patterns/multipleinheritance.html

[ModCppD] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#modern-cpp-design

[POOv00] POO – Volume 00, par Patrice Roy et Pierre Prud'homme.

[POOv01] POO – Volume 01, par Patrice Roy et Pierre Prud'homme.

[POOv03] POO – Volume 03, par Patrice Roy et Pierre Prud'homme.

[StrouMov] http://www2.research.att.com/~bs/move.pdf

[StrouNoH] http://www.research.att.com/~bs/bs_faq2.html#no-derivation

[StrouTerm] http://www2.research.att.com/~bs/terminology.pdf

[SutterMkUn] http://herbsutter.com/gotw/_102/

[TAOCP] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#art_computer_programming