PROGRAMMATION ORIENTEE OBJET AVANCEE EN C++

Ce document s'adresse aux personnes qui connaissent déjà le C++ et qui désirent approfondir leurs connaissances. Des cours de tous niveaux sur ce langage et des références de livres peuvent être trouvés dans la rubrique **"Informatique"** de ce site. Je tiens à remercier ici Krishna, avec qui j'ai préparé ce cours à l'ISIMA, et l'OBC qui est à l'origine de nombreux documents que nous utilisons comme support. Le document proposé ici n'est qu'un simple complément à ces cours, dans la mesure où il présente simplement d'une manière différente des notions avancées de C++ et de programmation orientée objet.

Dans une première partie, des exemples simples sont proposés pour rappeler des concepts fondamentaux, mais également pour préciser déjà certains mécanismes plus avancés. Nous abordons notamment la surcharge d'opérateurs et les difficultés liées à l'héritage et au polymorphisme dynamique. Ensuite, nous détaillons les classes génériques, les *templates*, qui sont l'une des spécificités de C++. Nous proposons également une brève introduction à la STL (*Standard Template Library*), une bibliothèque de conteneurs et d'algorithmes très utile.

Notez que le titre de certains chapitres n'est pas toujours très évocateur de tout ce qui s'y trouve, les nombreux concepts de C++ et de l'objet y sont éparpillés, simplement parce qu'il est très difficile de fournir une présentation indépendante de toutes ces notions. Donc, n'hésitez pas à utiliser le moteur de recherche du site si vous cherchez quelque chose de précis.

- **L'approche orientée objet**
  Avant d'attaquer le langage C++ même, il est important de se remémorer les concepts fondamentaux de l'approche orientée objet: l'agrégation, la composition, l'héritage, le polymorphisme, les classes génériques... Vous trouverez tous ces éléments dans le cours **"Théorie de la programmation orientée objet"**.

- **Classe ou structure**
  Explique par un exemple simple les différences fondamentales entre une classe en C++ et une structure en C. Les notions de construction et de destruction, d'affectation et de copie sont abordées.

- **Conception d'une classe**
  Présente à travers un exemple plus sophistiqué comment concevoir correctement une classe, en proposant quelques règles d'usage. Cela permet d'aborder l'écriture de diverses méthodes (notamment des opérateurs), et de rappeler ainsi les notions fondamentales du langage.

- **Héritage**
  Rappels sur l'héritage, la virtualité et le polymorphisme dynamique. Les notions d'héritage multiple, virtuel, de classe abstraite et d'interface sont abordées. Le mécanisme RTTI (*Run-Time Type Information*) ainsi que la gestion des exceptions sont également introduits.

- **Patrons de composant (*templates*) et classes génériques**
  Introduction aux patrons de composant, ou *templates*, et plus particulièrement à la conception de classes génériques. L'instanciation partielle, qui permet notamment de spécialiser un patron, est également présentée.

- **La bibliothèque *STL***
  Présentation sommaire de la STL, introduction aux notions d'itérateur, de foncteur et d'adapteur.

- **Métaprogrammation générique**

Il peut être intéressant d'étudier la métaprogrammation générique, basée sur les patrons de composants et l'instanciation partielle. Elle est notamment utilisée pour le calcul scientifique, car elle permet des performances accrues tout en proposant des composants logiciels génériques et extensibles. Vous trouverez quelques exemples de cette approche dans le document **"Eléments de métaprogrammation générique"**.

- **C++11**
  La norme C++11 apporte de nouvelles possibilités au langage C++, notamment concernant la conception des classes et la programmation générique. Des concepts importants sont introduits, à savoir les références *rvalue* et les opérateurs de mouvement, les génériques à paramètres variables, ainsi que les expressions lambda. Vous trouverez une présentation de ces nouveaux concepts dans le document **"Programmation en C++11"**.

- **Patrons de conception - *Design Patterns***
  Il est impossible de parler de programmation avancée sans aborder les patrons de conception, ou *design patterns*, qui fournissent des solutions conceptuelles éprouvées pour répondre à des problèmes récurrents de développement logiciel. Vous trouverez la plupart des patrons de conception classiques dans le cours **"Patrons de conception - *Design Patterns*"**.

1. CLASSE OU STRUCTURE

Nous proposons ici l'étude d'une classe très simple, afin de montrer les différences fondamentales entre une classe en C++ et une structure en C. L'exemple de classe suivant nous permet de réintroduire les notions de construction et de destruction, ainsi que les opérations d'affectation et de copie.

```
class Complexe {
 double _x;
 double _y;
};
```

Vous noterez que, par défaut, les membres d'une classe sont protégés, leur accès de l'extérieur de la classe est impossible, c'est le principe fondamental d'*encapsulation*: le maximum de détails sur l'implémentation d'une classe doit être masqué, cela permet une certaine indépendance qui facilite la maintenance de la classe. L'accès (et le contrôle) aux attributs est généralement permis en lecture et/ou en écriture par des méthodes qu'on nomme des *accesseurs* (cf. chapitre suivant).

CONSTRUCTION

Pour créer un objet de la classe `Complexe`, deux solutions sont possibles. La première consiste à déclarer simplement une variable.

```
Complexe c; (1)
```

La seconde possibilité est d'allouer dynamiquement un objet.

```
Complexe * pc = new Complexe(); (2)
```

Dans les deux cas, le constructeur dit *par défaut* est appelé. Le rôle de cette méthode est de préciser comment construire un objet standard (i.e. avec des valeurs par défaut) de la classe, sans que l'utilisateur ne précise aucun paramètre. Dans notre exemple, comme nous n'avons pas écrit de constructeur, un constructeur par défaut est alors automatiquement fourni, équivalent à ce qui suit.

```
Complexe::Complexe(void) : _x(), _y() {} (3)
```

Rappelons que créer (dit également *instancier*) un objet se déroule en deux phases. La première consiste en l'allocation de l'espace mémoire nécessaire à l'élément, ce qui est effectué sur la pile et automatiquement dans le cas *(1)*, ou dynamiquement dans la mémoire centrale dans le cas *(2)*. Une fois l'allocation réalisée, un constructeur est appelé, et dans notre cas il s'agit du constructeur par défaut. Cependant, avant d'exécuter le constructeur même, la construction des attributs de l'objet doit être réalisée. C'est le propos du symbole `:` qui indique qu'on se trouve entre l'allocation de l'objet et l'appel à son constructeur. Si l'on ne précise rien, les attributs sont construits par défaut comme explicité par *(3)*, mais il est possible de préciser d'autres valeurs pour les attributs, comme dans l'exemple suivant.

```
Complexe::Complexe(void) : _x(5.0), _y(10.0) {}
```

Remarquons que si l'on change l'ordre de construction des attributs, le compilateur émet alors un avertissement et les replace dans l'ordre de déclaration. En résumé, les attributs sont toujours construits selon leur ordre de déclaration. Il faut également noter que dans le cas où les attributs sont d'un type primitif (entier, flottant, booléen...), ils ne sont pas initialisés si l'on ne précise pas de

valeur au constructeur.

Il est possible de proposer autant de constructeurs qu'on souhaite (à condition qu'ils aient une signature - i.e. la liste des types de leurs arguments - différente). Cela permet de fournir tous les paramètres désirés à la construction d'un objet.

```
Complexe::Complexe(double x, double y) : _x(x), _y(y) {}
```

Ainsi, nous pouvons créer un objet de la manière suivante.

```
Complexe c(5.0,10.0);
```

Il est à noter que, dès qu'on définit un constructeur, le constructeur par défaut n'est plus fourni automatiquement par le compilateur, et qu'il faut donc le réécrire si nécessaire.

CONSTRUCTEUR DE COPIE

Le compilateur fournit également automatiquement un constructeur dit *de copie*, qui est disponible même si on définit d'autres constructeurs (contrairement au constructeur par défaut). Son rôle est d'initialiser un nouvel objet de la classe en copiant une instance existante. Dans notre exemple, le constructeur de copie fourni par défaut est équivalent à ce qui suit.

```
Complexe::Complexe(const Complexe & c) : _x(c._x), _y(c._y) {}
```

Cela signifie que, par défaut, le constructeur de copie appelle le constructeur de copie de chaque attribut pour recopier l'objet. Voici un exemple d'utilisation du constructeur de copie.

```
Complexe c1(5.0,10.0);
Complexe c2(c1);
Complexe c3 = c1;
```

La seconde ligne appelle le constructeur de copie de la classe. Les attributs de l'instance `c1` sont alors recopiés pour initialiser `c2`. La troisième ligne est équivalente à la seconde. Même si le signe `=` apparaît ici, il ne s'agit pas d'une affectation mais bien d'une construction par copie.

OPERATEUR D'AFFECTATION

A partir de l'exemple précédent, considérons l'instruction suivante, en dehors de toute déclaration: `c2=c3`. Il s'agit cette fois-ci d'une affectation, autrement dit de la recopie d'une instance. L'opérateur qui assure cette opération, dit *d'affectation*, est fourni par défaut et est équivalent à ce qui suit.

```
Complexe & Complexe::operator=(const Complexe & c) {
 _x=c._x;
 _y=c._y;
}
```

Cet opérateur appelle automatiquement l'opérateur d'affectation de chacun des attributs afin de recopier totalement l'objet en paramètre. Dans notre exemple, l'affectation est équivalente à l'appel suivant.

```
c2.operator=(c3);
```

DESTRUCTION

Au cours d'un programme, les objets créés sont naturellement amenés à être supprimés. Cette

procédure se déroule en deux étapes (à l'image de la création d'un objet). Tout d'abord, l'appel au destructeur de l'objet et ensuite la libération de la mémoire occupée par l'objet. Dans le cas de la création d'une variable (cf. le cas *(1)* dans la section sur la construction), la suppression s'effectue à la fin du bloc où a été déclarée la variable. Dans le cas d'une allocation dynamique (cf. le cas *(2)* dans la section sur la construction), l'utilisateur doit décider du moment de la destruction par la commande suivante.

```
delete pc;
```

Il faut noter, à l'image de la création d'un objet, qu'entre l'appel au destructeur et la libération de la mémoire, les attributs sont détruits dans l'ordre inverse de leur déclaration. A l'opposé de la construction, l'utilisateur ne peut pas intervenir dans ce processus. Un destructeur est toujours fourni par défaut, mais il n'effectue aucune opération particulière.

```
Complexe::~Complexe(void) {}
```

2. CONCEPTION D'UNE CLASSE

Dans ce chapitre, nous tentons d'expliquer comment développer correctement une classe en C++ et proposons quelques règles d'usage qui permettent d'éviter certains pièges. Nous prenons l'exemple d'une classe `Chaine` dont le rôle est d'encapsuler une chaîne de caractères, afin d'en proposer une manipulation plus simple. Dans cette classe, la chaîne de caractères, un tableau au sens C, sera allouée dynamiquement, ce qui nécessite de surcharger les méthodes qui étaient fournies par défaut dans la classe `Complexe` présentée au chapitre précédent. En effet, à la construction, l'allocation dynamique du tableau est indispensable, lors de l'affectation, une réallocation peut être nécessaire et enfin, à la destruction, la mémoire allouée pour le tableau doit être rendue.

Nous en profiterons aussi pour redéfinir quelques opérateurs comme l'indexation `[]`, l'addition `+` (jouant ici un rôle de concaténation) et les flux `<<` et `>>`. Nous parlerons également du mécanisme de conversion qui repose sur la manipulation d'opérateurs. Des informations concernant certains mots-clé comme `inline`, `const`... sont également disponibles ici. L'écriture de la classe `Chaine` nous permet de conclure sur l'utilité et la manière adéquate d'utiliser les références. Voici la déclaration de la classe `Chaine`.

```
class Chaine {
  char * _t; // Tableau de caractères.
  int    _n; // Nombre de caractères.

 public:
  Chaine(void);
  Chaine(const char *);
  Chaine(int);
  Chaine(const Chaine &);
  ~Chaine(void);

  Chaine & operator=(const Chaine &);
  char     operator[](int) const;
  char &   operator[](int);
};
```

Il est conseillé, lors de la surcharge d'un opérateur, de conserver le plus possible sa sémantique initiale. La principale raison est simplement d'éviter toute confusion chez les utilisateurs de cet opérateur. Si la sémantique est trop différente, il est conseillé de passer par des méthodes classiques pour fournir les fonctionnalités souhaitées.

CONSTRUCTION

Le constructeur par défaut doit être redéfini pour initialiser les attributs.

```
Chaine::Chaine(void) : _t(0), _n(0) {}
```

Un constructeur prenant en paramètre une chaîne de caractères peut aussi être défini.

```
Chaine::Chaine(const char * s) : _t(0), _n(std::strlen(s)) {
 _t=new char[_n+1];
 std::strcpy(_t,s);
}
```

A noter que le constructeur par défaut peut être proposé simplement en fournissant un attribut par défaut au constructeur précédent.

```
Chaine::Chaine(const char * = "");
```

Le constructeur de copie fourni par défaut doit également être redéfini, la duplication simple de l'attribut `_t` (qui est un pointeur) conduirait à une incohérence dans le nouvel objet: le tableau de caractères serait partagé avec l'objet qui a été dupliqué. Voici un constructeur de copie qui évite ce problème en créant un tableau pour le nouvel objet.

```
Chaine::Chaine(const Chaine & c) : _t(0), _n(c._n) {
 _t=new char[_n+1];
 std::strcpy(_t,c._t);
}
```

Il peut être également intéressant de définir un constructeur pour créer une chaîne vide d'une taille donnée.

```
Chaine::Chaine(int n) : _t(0), _n(n) {
 int i = 0;

 _t=new char[_n+1];
 while (i<_n) _t[i++]=' ';
 _t[i]=0;
}
```

OPERATEUR D'AFFECTATION

L'opérateur d'affectation doit être redéfini pour les mêmes raisons que le constructeur de copie.

```
Chaine & Chaine::operator=(const Chaine & c) {
 if (this!=&c) {
  if (_t!=0) delete [] _t;
  _n=c._n;
  _t=new char[_n+1];
  std::strcpy(_t,c._t);
 }

 return *this;
}
```

Notons que cet opérateur renvoie une référence de l'objet sur lequel il a été appliqué, de cette manière, l'expression `a=b=c` peut être évaluée, puisqu'elle correspond à la succession d'appels `a.operator=(b.operator=(c))`. En outre, pour éviter tout problème, il est conseillé de vérifier en début de méthode que l'argument passé n'est pas l'objet lui-même. Ainsi, dans l'expression `a=a`, rien ne se passe.

DESTRUCTION

Contrairement à l'exemple du chapitre précédent, le destructeur a ici un rôle, celui de libérer la mémoire allouée par l'objet.

```
Chaine::~Chaine(void) { if (_t!=0) delete [] _t; }
```

Il faut remarquer également que la libération du tableau `_t` s'effectue ici avec l'opérateur `delete []` et non pas avec simplement `delete`. Il faut être très attentif lors de la destruction: si le pointeur référence une zone de mémoire allouée par l'intermédiaire d'un `new` simple (i.e. allocation et construction d'un seul objet) alors l'opérateur `delete` simple doit être utilisé; si le pointeur référence une zone mémoire allouée par l'intermédiaire d'un `new` collectif (i.e. utilisant les `[]`) alors l'opérateur `delete []` doit être utilisé. Toute erreur d'inversion de ces deux opérateurs de destruction conduira à des problèmes d'accès mémoire du type *segmentation fault*.

ACCESSEURS

En programmation orientée objet, l'une des règles fondamentales consiste à encapsuler les attributs d'un objet. Par défaut en C++, si l'on ne précise rien, tous les membres sont privés (`private`). Les mots-clés `public` ou `protected` peuvent être employés pour spécifier des membres publics (visibles par tout le monde) ou protégés (visibles par la classe et ses sous-classes, contrairement aux privés qui ne sont visibles que par la classe uniquement). Pour accéder à un attribut, on doit alors passer par des méthodes, appelées souvent *accesseurs*, qui peuvent contrôler l'accès en lecture et/ou en écriture des attributs.

Accès en lecture simple

Supposons par exemple l'attribut `_t` auquel on souhaite permettre l'accès en lecture simple.

```
inline int Chaine::taille(void) const { return _t; }
```

Cette simple ligne soulève plusieurs remarques. Tout d'abord, la méthode est déclarée `inline`, cela signifie que, lorsqu'elle est appelée, son code est recopié à la place de son appel (c'est ce qu'on appelle le déroulement, ou *unrolling*, de la méthode), aucun mécanisme d'appel de fonction n'est alors déclenché. Cela signifie que le code binaire généré est totalement équivalent à celui de l'accès direct à l'attribut si celui-ci avait été public. En outre, la définition de toute méthode *inline* doit impérativement se trouver dans un *header* (i.e. un fichier d'entête `.h`, `.hpp`...), de sorte que le compilateur puisse remplacer à tout moment l'appel par le corps de la méthode (dans le cas contraire, l'*unrolling* ne sera pas effectué).

La seconde remarque est que la déclaration de la méthode précédente est suivie du mot clé `const`, ce qui signifie ici que la méthode ne modifie pas les attributs de l'objet. En d'autres termes, cette méthode peut être appelée même si l'objet est constant, contrairement à une méthode qui n'a pas ce mot-clé. Par la suite, nous désignons une telle méthode comme *constante*.

A l'opposé du C, la constance d'un objet est fondamentale. Il est très important, au moment de l'écriture du prototype d'une méthode de déterminer quels sont les arguments qui seront constants, si l'objet retourné est constant ou non et enfin si la méthode est constante ou non. A cet aspect s'ajoute aussi la manière de passer les arguments et de retourner un objet: soit par copie, soit par référence (constante ou non).

Dernière remarque, dans la déclaration précédente, une copie de `_t` est retournée car on ne souhaite pas que l'attribut soit modifié. Pour que la méthode fournisse un accès à l'attribut en lecture seule, on aurait pu l'écrire également de la manière suivante.

```
inline const int & Chaine::taille(void) const { return _t; }
```

Au lieu de retourner une copie de l'attribut, on renvoie une référence constante qui pointe bien physiquement sur l'attribut, mais ne permet pas de le modifier. Cet exemple prend plus de sens dans le cas d'un attribut qui n'est pas d'un type primitif: supposons un attribut `_n` de type `Chaine` dans une classe `Personne`. Un accesseur en lecture s'écrira comme suit.

```
inline const Chaine & Personne::nom(void) const { return _n; }
```

On évite ainsi la copie de l'attribut en retournant une référence constante sur l'objet. De cette manière, l'objet est tout autant protégé contre l'écriture que dans l'exemple précédent. L'usage veut que, lorsqu'on renvoie en lecture un type primitif, on retourne une copie (car elle n'est pas plus coûteuse que le passage par référence), alors que si l'on renvoie un objet, on retourne une référence constante, cela évite une copie qui pourrait être coûteuse.

Accès en lecture/écriture

Revenons maintenant à notre exemple de la classe `Chaine`. Voici comment permettre l'accès à la fois en lecture et en écriture à l'attribut `_t`.

```
inline int & Chaine::taille(void) { return _t; }
```

Remarquez que le mot-clé `const` a disparu, ce qui signifie que cette méthode ne peut être appelée que si l'objet n'est pas constant, ce qui est tout à fait cohérent avec la sécurité d'écriture de l'objet. Il faut noter également que le mot-clé `const` qui identifie une méthode constante fait partie de sa signature. Cela signifie que le compilateur est capable de distinguer la version lecture de la méthode `taille()` de sa version lecture/écriture et qu'elles peuvent donc tout à fait cohabiter dans un même programme. La version constante sera appelée uniquement lorsque l'objet est constant. Voici un rappel des deux méthodes.

```
inline int   Chaine::taille(void) const;
inline int & Chaine::taille(void);
```

Mais il est également possible de proposer deux méthodes distinctes pour accéder aux attributs, soit en lecture, soit en écriture.

```
inline int  Chaine::getTaille(void) const { return _t; }
inline void Chaine::setTaille(int t) { _t=t; }
```

La grande différence entre ces deux possibilités est qu'avec la seconde solution, on sait quelle méthode sert à la lecture (`cout << c.getTaille()`) et quelle méthode sert à l'écriture (`c.setTaille(20)`). Alors que dans la première proposition, la version non constante peut servir sur un objet non constant aussi bien à la lecture (`cout << c.taille()`) qu'à l'écriture (`c.taille()=20`). Si un contrôle différent doit être effectué selon qu'on se trouve en lecture ou en écriture, alors il faut utiliser la seconde solution.

OPERATEUR D'INDEXATION

L'opérateur d'indexation permet d'appliquer les symboles `[]` directement à un objet. Il s'agit d'un type particulier d'accesseur. Ainsi, on peut définir une version lecture simple.

```
inline char Chaine::operator[](int i) const {
 if (i>=_n) {
  std::cerr << "Debordement !" << std::endl;
  exit(1);
 }

 return _t[i];
}
```

Et une version lecture/écriture.

```
inline char & Chaine::operator[](int i) {
 if (i>=_n) {
  std::cerr << "Debordement !" << std::endl;
  exit(1);
 }

 return _t[i];
}
```

On s'aperçoit ici de l'intérêt d'un accesseur plutôt qu'un accès direct à l'attribut, puisqu'il permet de contrôler que l'indice est valide avant de tenter l'accès aux données.

QUELQUES OPERATEURS BINAIRES

Il existe deux catégories d'opérateurs: les unaires (`*`, `++`, `--`, `[]`...) et les binaires (`+`, `-`, `*`, `/`, `<<`, `>>`...). Les opérateurs unaires n'impliquent qu'un seul objet, ils peuvent donc être définis comme des méthodes de celui-ci (e.g. la section précédente). En revanche, les opérateurs binaires mettent en relation deux objets, parfois de classes différentes. On préfère donc les définir comme des fonctions. Il reste néanmoins possible de les manipuler comme des méthodes, mais cette approche est généralement déconseillée. Elle est utile seulement si l'on souhaite exploiter la redéfinition de ces méthodes dans des sous-classes.

De retour à notre classe `Chaine`, voici l'exemple de la surcharge de l'opérateur de concaténation.

```
Chaine operator+(const Chaine & s1,const Chaine & s2) {
 Chaine s3(s1.taille()+s2.taille());
 int i = 0;
 int j = 0;

 while (i<s1.taille()) { s3[i]=s1[i]; ++i; }
 while (j<s2.taille()) s3[i++]=s2[j++];
 return s3;
}
```

L'opérateur est implémenté sous la forme d'une fonction, simplement parce que `s1` et `s2` ont des rôles symétriques, et que choisir de faire subir l'opération sous la forme d'une méthode à l'un plutôt qu'à l'autre n'a pas vraiment de sens. En outre, afin de permettre l'évaluation d'une expression du genre `c=a+b`, l'opérateur doit retourner une copie de la concaténation. Dans l'exemple, nous avons choisi d'utiliser uniquement des méthodes publiques de la classe `Chaine` pour effectuer la concaténation, mais il est possible (et parfois indispensable) que l'opérateur accède à des attributs cachés (i.e. protégés ou privés). Il faut alors que la fonction `operator+` soit amie de la classe `Chaine`. Pour cela, il suffit de rajouter la ligne suivante dans la déclaration de la classe.

```
Chaine operator=(const Chaine &,const Chaine &);

class Chaine {
 ...
 friend Chaine operator=(const Chaine &,const Chaine &);
};
```

A noter que cela fonctionne seulement si, au moment de la déclaration de l'amitié, le prototype de la fonction est connu. Nous reviendrons sur cette notion d'amitié, plus particulièrement entre deux classes, dans le chapitre sur les classes génériques.

OPERATEURS DE FLUX

Les opérateurs de flux sont binaires et impliquent des objets de classes différentes. Ils sont donc déclarés en tant que fonctions.

```
#include<iostream>

std::ostream & operator<<(std::ostream & o,const Chaine & c) {
 int i = 0;

 while (i<c.taille()) o << c[i++] << ' ';
 return o;
}

std::istream & operator>>(std::istream & i,Chaine & c) {
 char s[256];
```

```
 i >> s;
 c=Chaine(s);
 return i;
}
```

Afin de permettre le chaînage des flux, e.g. `cout << a << " " << b`, les opérateurs de flux doivent retourner le flux en question. En outre, il faut éviter de copier des objets flux, leur fonctionnement en est alors totalement altéré.

OPERATEURS D'INCREMENTATION

L'opérateur `++` (ou `--`) se décline en deux méthodes, suivant qu'on l'utilise en préfixé (e.g. `++i`) ou en postfixé (e.g. `i++`). Supposons une classe `Entier` qui agrège un attribut entier `_p` qu'on souhaite incrémenter par l'opérateur `++`. La forme préfixée se déclare de la manière suivante.

```
Entier & Entier::operator++(void) {
 ++_p;
 return *this;
}
```

Et la forme postfixée de la façon suivante.

```
Entier Entier::operator++(int) {
 Entier p(*this);

 ++_p;
 return p;
}
```

Cet opérateur doit retourner un objet de la classe en question. Pour le préfixé, l'objet même peut être retourné, alors que pour le postfixé, une copie doit être renvoyée. Le paramètre `int` passé à la version postfixée est muet: il n'est pas utilisé dans la méthode. Mais il est nécessaire au compilateur pour distinguer les deux versions par leur prototype.

CONVERSION

Déjà en C, il est possible de convertir une variable d'un type en un autre. Seulement cela s'effectue sans aucune vérification sur la validité de l'opération, et encore plus grave, sans que le programmeur ne puisse intervenir sur la manière d'effectuer la conversion.

Opérateurs de conversion

Intéressons-nous tout d'abord à la manière de décrire la conversion d'un type d'objet en un autre. Pour cela, revenons à l'exemple de la classe `Chaine`. Le constructeur suivant a été défini.

```
Chaine::Chaine(const char * s);
```

Cette méthode est un opérateur de conversion d'une chaîne de caractères de type `char *` en un objet de la classe `Chaine`. Il faut noter que cette conversion est implicite, cela signifie que, si une méthode a besoin d'un argument de type `Chaine` et que le programmeur fourni un objet de type `char *`, alors le compilateur décide seul d'appeler le constructeur qui permet la conversion. Considérons l'exemple suivant.

```
Chaine a("J'attends ");
Chaine b = a+"la suite";
```

Le compilateur va interpréter ce code comme suit.

```
Chaine a("J'attends ");
Chaine b = a+Chaine("la suite");
```

Cela peut conduire à des confusions importantes. Souvenez-vous, nous avons écrit un constructeur qui prend comme argument un entier (cf. la section sur la construction), cela nous semblait pratique pour initialiser une chaine vide. Cependant, si l'on écrit le code suivant.

```
Chaine a("Voila le nombre magique: ");
Chaine b = a+245;
```

Le compilateur va interpréter ce code comme suit.

```
Chaine a("Voila le nombre magique: ");
Chaine b = a+Chaine(245);
```

On s'aperçoit immédiatement de la confusion. La compilation est autorisée mais le résultat n'est probablement pas celui escompté par le programmeur (i.e. la conversion de l'entier `245` en une chaine `"245"`). Il vaut mieux dans cette situation que la compilation soit interdite, ce qui obligera le programmeur à vérifier que le constructeur à partir d'un entier réalise bien ce qu'il souhaite. Il pourra ensuite préciser explicitement l'appel au constructeur. Pour empêcher la conversion implicite, on utilise le mot-clé `explicit` de la manière suivante sur le constructeur souhaité.

```
class Chaine {
 ...
 explicit Chaine(const char *);
 ...
};
```

On peut également vouloir effectuer la conversion inverse, i.e. passer d'un objet `Chaine` à une chaîne de caractères. Pour cela, il est possible d'écrire un opérateur de conversion de la manière suivante dans la classe `Chaine`.

```
class Chaine {
 ...
 operator char * (void) const { return _t; }
 ...
};
```

Politiques de conversion

La conversion d'un objet d'un type en un autre peut être une opération délicate. Ainsi, différentes directives sont proposées pour préciser le type de conversion ainsi que le type de contrôle qu'on désire effectuer afin d'éviter toute erreur dans le code.

L'opérateur *(type)*

Pour convertir un objet, nous avons vu qu'il était possible d'utiliser l'opérateur `(type)` issu du langage C. En supposant une chaîne de caractères `s` et une instance `c` de la classe `Chaine`, les deux lignes suivantes sont équivalentes.

```
c=(Chaine)s;
c=Chaine(s);
```

Elles permettent la conversion en utilisant les opérateurs définis par l'utilisateur. Si aucun opérateur n'est disponible, la conversion est interdite par le compilateur. Certaines conversions sont proposées de base pour les types primitifs, comme par exemple la conversion d'un `float` en un `int`. En ce qui concerne les pointeurs, l'opérateur `(type)` autorise toute conversion, que les classes soient liées par

une relation d'héritage ou non. Supposons par exemple trois classes `A`, `B` et `C`.

```
class A { ... };
class B : public A { ... };
class C { ... };

A * a = new A();
B * b = new B();
C * c = new C();

A * pa;
B * pb;
```

Ainsi, les lignes suivantes sont autorisées.

```
pa=(A *)c; (1)
pb=(B *)a; (2)
```

Alors que la première ligne devrait être interdite, simplement parce que `C` n'est pas une sous-classe de `A`. La seconde ligne devrait également être interdite, puisque `a` n'est pas un objet de la classe `B`. Cependant, dans le second cas, on peut imaginer le code suivant.

```
a=b;
pb=(B *)a; (3)
```

La conversion de la seconde ligne devrait alors être possible. Cependant, une vérification est nécessaire au moment de l'exécution pour être sûr que `a` pointe bien sur un objet de type `B`. Nous venons de voir que l'utilisation de l'opérateur `(type)` est dangereuse dès qu'on utilise des pointeurs (avec des références le phénomène serait le même). Il est donc important de disposer d'autres opérateurs qui soient capables de vérifier la conversion. Pour cela, l'opérateur `static_cast` est introduit pour éviter le cas *(1)*. L'opérateur `dynamic_cast` est quant à lui utilisé pour vérifier au moment de son exécution les cas *(2)* et *(3)*.

L'opérateur *static_cast*

Cet opérateur est donc utilisé pour éviter de convertir un pointeur sur un type donné en un pointeur sur un type qui n'est pas lié par héritage (cela marche aussi pour des références). Ainsi la conversion suivante est autorisée (même si elle risque d'être invalide).

```
pb=static_cast<B *>(a);
```

En revanche, les conversions suivantes sont interdites (`pi` étant un pointeur sur un `int` et `pf` un pointeur sur un `float`).

```
pa=static_cast<A *>(c);
pf=static_cast<float *>(pi);
```

Les règles de conversion concernant le type de pointeur `void *` restent obscures dans les spécifications du langage. En effet, la conversion suivante est autorisée (ce qui est tout à fait logique).

```
void * pv = static_cast<void *>(a);
```

En revanche, la conversion suivante devrait être interdite (car il n'y a aucun moyen de savoir ce que référence `pv`).

```
pa=static_cast<A *>(pv);
```

Cependant, certains compilateurs autorisent cette conversion. Il faut noter que l'opérateur `dynamic_cast` chargé de régler les problèmes de conversion au moment de l'exécution interdit

cette action. Dans un souci de portabilité et pour cette situation particulière uniquement, nous vous recommandons plutôt l'utilisation de l'opérateur `reinterpret_cast` (l'opérateur `(type)` fonctionne également).

L'opérateur *dynamic_cast*

Cet opérateur ne peut être utilisé que pour des pointeurs (ou des références), et comme nous l'avons vu précédemment, il ne peut pas être employé avec le type `void *`. Néanmoins, il permet de vérifier, au moment de l'exécution, la conversion d'un pointeur d'une classe `A` vers un pointeur d'une de ses sous-classes `B`. Cette conversion est désignée par le terme *downcast*, elle est opposée à la conversion inverse, toujours possible, d'une classe vers l'une de ses super-classes et qui est nommée *upcast*. Dans l'exemple suivant, si `pa` pointe réellement sur un objet de la classe `B`, alors la conversion s'effectue. En revanche, si `pa` pointe par exemple sur un objet de la classe `A`, alors la conversion est impossible et le pointeur `NULL` est retourné.

```
pb=dynamic_cast<B *>(pa);
```

Il existe une petite différence lorsqu'on manipule des références comme dans l'exemple suivant.

```
A a;
B b;

A & ra = a;
A & rb = b;
B & ref1 = dynamic_cast<B &>(ra);
B & ref2 = dynamic_cast<B &>(rb);
```

Dans le cas de `ref2`, la conversion est possible. Par contre, dans le cas de `ref1`, la conversion est impossible, mais comme `NULL` ne peut pas être renvoyé, une exception est levée. Remarquez que la conversion par référence est importante, puisqu'elle permet d'éviter des recopies.

L'opérateur *const_cast*

Cet opérateur est utilisé pour simplement retirer l'aspect constant d'un objet. Il n'a de réel signification que s'il est appliqué sur une référence. En effet, considérons l'exemple suivant.

```
const Chaine c1;
Chaine c2 = c1;
```

La conversion ne pose aucun problème, puisqu'une copie est effectuée et que celle-ci n'hérite pas de l'aspect constant. En revanche, si l'on considère l'exemple suivant.

```
const Chaine c1;
Chaine & c2 = const_cast<Chaine &>(c1);
```

L'opérateur de conversion `const_cast` est alors indispensable. Il peut donc être utilisé lorsqu'on reçoit une référence constante pour la rendre non constante et ainsi pouvoir modifier l'objet qu'elle référence. Il est bien évident que l'usage de cet opérateur est à éviter dans la mesure où il altère totalement les règles fondamentales liées à la constance d'un objet. **Dans la majorité des cas, lorsque le programmeur se retrouve obligé d'utiliser l'opérateur `const_cast` sur une variable, c'est qu'il a commis une erreur de conception, soit en imposant à tort la constance de la variable, soit en omettant des méthodes qui permettraient un accès non constant à la variable.**

Conclusion

Il n'est pas toujours évident de déterminer le bon opérateur de conversion pour une situation

donnée. Le tableau suivant tente de résumer les différentes possibilités et d'indiquer si, dans chaque situation, les opérateurs autorisent ou non la conversion. Cela permet d'identifier l'opérateur le mieux adapté (repéré par un *) pour chaque type de conversion.

	Objet `Chaine` vers `char *`	Pointeur `B` vers pointeur `A`	Pointeur `A` vers pointeur `B`	Pointeur objet vers `void *`	`void *` vers pointeur objet
`(type)`	Oui *	Oui	Oui	Oui	Oui
`static_cast`	Oui	Oui *	Oui	Oui *	Ne devrait pas
`dynamic_cast`	Non applicable	Oui	Oui * (après vérification)	Oui	Non applicable
`reinterpret_cast`	Non applicable	Oui	Oui	Oui	Oui *

Nous rappelons que la conversion d'une référence d'un type donné en une référence d'un autre type se déroule de la même manière que la conversion entre pointeurs (i.e. la 2ème et la 3ème colonne du tableau).

REFERENCES

Il est important de préciser maintenant le rôle des références, qui est tout aussi fondamental que celui du mot-clé `const`. Il existe deux manières classiques de passer un argument à une méthode: soit par valeur, soit par référence (on oublie le passage par pointeur du C qui n'est plus très utile ici, comme nous le verrons par la suite). Le passage par valeur est souvent utilisé, notamment en C, pour éviter qu'on puisse modifier l'argument dans la méthode. En C++, il est possible d'éviter la copie (qui peut être coûteuse si l'objet est important) en fournissant une référence sur un objet constant. Pour un argument de type primitif, fournir une copie ou une référence constante a peu de différences en termes d'efficacité. Le tableau suivant résume comment passer un argument nommé `arg` en fonction de la situation qui se présente.

	Type primitif T	Classe C
Argument variable	`T & arg`	`C & arg`
Argument constant	`T arg` ou `const T & arg`	`const C & arg`

Le passage par valeur n'est donc plus très utile. Il peut seulement servir à simplifier l'écriture d'une méthode en utilisant la copie d'un argument plutôt que de définir une variable locale à l'identique de l'argument. En ce qui concerne l'objet retourné par une méthode, là aussi il peut s'agir d'un passage par valeur ou par référence (on oublie pour les mêmes raisons le passage par pointeur). Ici, la recopie de l'objet retourné peut être primordiale. En effet, il n'est pas possible de retourner une référence sur une variable locale à la méthode (le contexte de celle-ci est détruit dès que le retour s'effectue). Voici un tableau qui résume comment retourner une valeur en fonction des cas qui se présentent.

	Type primitif T	Classe C
Retour (en mode lecture) d'un attribut de la classe	`T m(...) const;` ou `const T & m(...) const;`	`const C & m(...) const;`
Retour (en mode lecture/écriture) d'un attribut de la classe	`T & m(...);`	`C & m(...);`
Retour d'un résultat produit par la méthode	`T m(...) const;`	`C m(...) const;`

Il est important de noter également que le passage par référence permet le polymorphisme (détaillé

au chapitre suivant), ce qui serait impossible avec le passage par valeur. Prenons l'exemple suivant.

```
class A {
 public:
  virtual void m(void) const { std::cout << "A::m()"; }

 ...
};

class B : public A {
 public:
  virtual void m(void) const { std::cout << "B::m()"; }

 ...
};

void f1(const A & a) { a.m(); }
void f2(A a) { a.m(); }
```

Supposons un objet `b` de classe `B`. L'appel à la fonction `f1(b)` produira le message `"B::m()"` alors que l'appel `f2(b)` produira `"A::m()"`. Dans le premier cas, il n'y a pas de recopie, donc la variable locale `a` référence bien `b`; alors que dans le second cas, l'argument est recopié (en appelant le constructeur de copie de `A`) pour produire une copie de `b` uniquement sur la partie concernant la classe `A`, en d'autres termes la variable locale est simplement un objet de classe `A`.

3. HERITAGE

L'héritage est certainement l'un des concepts les plus novateurs de la programmation orientée objet. Dans ce chapitre, nous rappelons le concept de polymorphisme dynamique lié à la virtualité et à la redéfinition de méthodes. En outre, le C++ est un langage qui autorise l'héritage multiple, il s'agit d'un concept très souvent critiqué, puisqu'il soulève quelques problèmes d'implémentation. Nous le présentons tout de même, car il peut s'avérer utile dans certaines situations. Nous expliquons alors une alternative à l'héritage multiple, à travers la notion d'interface proposée notamment dans le langage Java.

L'héritage permet de voir un objet comme appartenant à une catégorie plus ou moins précise. Il est donc fondamental de disposer d'un mécanisme, nommé RTTI (*Run-Time Type Information*) pour le langage C++, qui permet de déterminer la nature exacte d'un objet, notamment au moment de l'exécution du programme. Nous ne l'abordons que très brièvement dans ce chapitre. Enfin, bien que le sujet semble quelque peu éloigné de l'héritage, nous avons choisi de présenter ici les exceptions, un mécanisme qui offre une manière plus élégante de gérer les erreurs dans un programme.

POLYMORPHISME

Le terme *polymorphisme* est employé pour indiquer qu'un élément peut prendre plusieurs formes. Il existe deux types de polymorphisme: le polymorphisme statique et le polymorphisme dynamique. Celui qui nous intéresse ici est le dynamique et concerne plus particulièrement les méthodes. Il signifie que le contenu d'une méthode n'est définitivement établi qu'au moment de son appel. La combinaison de l'héritage, de la virtualité et la possibilité de redéfinir des méthodes permettent ce polymorphisme dynamique.

Héritage simple

Avant de s'attaquer au polymorphisme, intéressons-nous simplement à l'héritage. Une classe permet d'identifier une catégorie de variables dans un programme, en l'occurrence des objets. L'héritage est un moyen d'organiser ces catégories. Il permet de définir des catégories générales dans lesquelles seront réunies des catégories plus précises. Par exemple, la catégorie (ou classe) `Animal` peut être précisée (on dit plutôt *spécialisée*) en sous-catégories comme `Mammifere`, `Poisson`... En programmation, cette phase de spécialisation a un intérêt particulier: elle permet de factoriser le code, les parties communes aux classes `Mammifere` et `Poisson` peuvent être placées dans la classe `Animal`. Prenons l'exemple suivant où l'on souhaite positionner sur une carte des animaux identifiés par un nom (e.g. on établi les bases d'une simulation).

```
class Animal {
 protected:
  Chaine _nom; // Nom de l'animal.
  int    _x;   // Abscisse de sa position.
  int    _y;   // Ordonnée de sa position.

 public:
  Animal(char * n,int x,int y) : _nom(n), _x(x), _y(y) {}

  const Chaine & getNom(void) const { return _nom; }
  int            getX(void) const { return _x; }
  int            getY(void) const { return _y; }
  bool           estFemelle(void) const { ... }
```

```
};
```

Méthode virtuelle, redéfinition et polymorphisme

La définition précédente de la classe `Animal` est bien entendue incomplète. Supposons par exemple que `Animal` possède les méthodes `deplacer` et `engendrer`, la première déplace l'animal et la seconde, applicable seulement à une femelle, consiste à mettre bas et donc engendrer un nouvel `Animal`. Les poissons et les mammifères ne se déplacent et ne mettent pas bas de la même manière. Les méthodes sont donc déclarées dans la classe `Animal`, puisqu'elles sont communes aux deux sous-classes. En revanche, le contenu même des méthodes doit être fourni par chaque sous-classe. Voici les prototypes rajoutés à la classe `Animal`.

```
virtual void     deplacer(void);
virtual Animal * engendrer(void);
```

Notez que les deux méthodes sont virtuelles. Cela signifie qu'elles peuvent avoir un corps mais que les sous-classes sont autorisées à le remplacer (cette action étant appelée *redéfinition*). Voici la définition des classes `Poisson` et `Mammifere`.

```
class Poisson : public Animal {
 protected:
  int _profondeur; // Profondeur où vit ce genre de poisson.

 public:
  void     deplacer(void) { ... }
  Animal * engendrer(void) { if (estFemelle()) { ... } }
};

class Mammifere : public Animal {
 protected:
  int _vitesse; // Vitesse de déplacement.

 public:
  void     deplacer(void) { ... }
  Animal * engendrer(void) { if (estFemelle()) { ... } }
};
```

Les méthodes redéfinies dans `Poisson` et `Mammifere` sont bien sûr différentes. Notez qu'une fois qu'une méthode est déclarée virtuelle dans une classe, elle est automatiquement virtuelle dans les sous-classes, même si le mot-clé `virtual` n'est pas précisé. Supposons maintenant un tableau d'animaux, auxquels on applique la méthode `deplacer`.

```
int i = 0;
Animal t[] = { Poisson("Maurice",10,20,3),
               Mammifere("Rantanplan",5,9,17) };

while (i<n) {
 t[i].deplacer();
 ++i;
}
```

Si vous vous attendiez à ce que Maurice se déplace comme un poisson et Rantanplan comme un chien, et bien c'est perdu ! En effet, quand le poisson est placé dans la première case du tableau, il est converti en un animal (par le constructeur de copie de cette classe). De la même manière, le mammifère redevient un simple animal dans le tableau. En résumé, c'est la méthode `deplacer` de `Animal` qui est appelée dans les deux cas. Voici la solution qui fait que chaque animal se déplace de la bonne manière. Ce qu'on appelle *polymorphisme dynamique*, c'est le fait que, au moment de la compilation, on ne sait pas quelle méthode `deplacer` va véritablement être appelée, c'est seulement au moment de l'exécution que tout se décidera.

```
int i = 0;
Animal * t[] = { new Poisson("Maurice",10,20,3),
                 new Mammifere("Rantanplan",5,9,17) };

while (i<n) {
 t[i]->deplacer();
 ++i;
}
```

Il faut savoir que la virtualité d'une méthode l'empêche d'être déclarée `inline`, tout simplement parce que le mécanisme d'appel est différent d'une méthode simple et qu'il ne peut pas être évité (en effet, c'est à l'exécution qu'on décide réellement du contenu de la méthode appelée, il est donc impossible à la compilation de remplacer directement cet appel par un contenu). Et avec les références, qu'est-ce que ça donne ?

```
int i = 0;
Animal & t[] = { Poisson("Maurice",10,20,3),
                 Mammifere("Rantanplan",5,9,17) };

while (i<n) {
 t[i].deplacer();
 ++i;
}
```

Encore perdu ! On ne peut pas faire un tableau de références. Mais les références permettent le polymorphisme dynamique. Prenons l'exemple suivant.

```
void deplacer(Animal & a) {
 std::cout << "Je deplace " << a.getNom() << std::endl;
 a.deplacer();
}

...

int i = 0;
Animal * t[] = { new Poisson("Maurice",10,20,3),
                 new Mammifere("Rantanplan",5,9,17) };

while (i<n) {
 deplacer(*(t[i]));
 ++i;
}
```

Dans la fonction `deplacer`, l'argument reçu est une référence sur un `Animal`. Il n'y a donc pas de copie, l'animal est alors soit Maurice, soit Rantanplan. La nature précise de l'animal sera déterminée seulement à l'exécution, il y a donc bien polymorphisme dynamique.

Destructeur virtuel

Lorsqu'on déclare une méthode virtuelle dans une classe, il faut impérativement que le destructeur soit virtuel. La raison se trouve tout simplement au niveau de la destruction de l'objet. Prenons l'exemple suivant.

```
Animal * a = new Poisson("Maurice",10,20,3);
...
delete a;
```

Si le destructeur n'est pas virtuel, alors seul le destructeur de `Animal` est appelé. En revanche, s'il est virtuel, le destructeur de `Poisson` est également appelé. Mais le mécanisme est à l'image de la construction, c'est-à-dire que les destructeurs des super-classes sont appelés dans l'ordre inverse de la construction. Ainsi, pour notre exemple, le destructeur de `Poisson` est appelé avant celui de `Animal`.

Pointeur de méthode

De la même manière qu'il existe des pointeurs de fonction en C, il est possible de définir des pointeurs de méthode en C++. Supposons par exemple qu'on souhaite écrire une fonction qui parcourt tous les animaux d'un tableau et appelle une méthode donnée sur chacun des objets.

```
void parcourir(Animal * t[],int n,void (Animal::*m)(void)) {
 int i = 0;

 while (i<n) (t[i++]->*m)();
}
```

L'argument `m` représente un pointeur de méthode, la seule différence avec un pointeur de fonction est que le nom de la classe apparaît au moment de la déclaration du type. Sinon, le format général d'un pointeur de méthode est très proche de celui d'un pointeur de fonction.

type_retour `(`nom_classe`::*`nom_variable`)(`type_argument_1`,`type_argument_2...`)`

Voici maintenant comment utiliser la fonction `parcourir`, l'exemple appelle la méthode `deplacer` de tous les animaux contenus dans le tableau.

```
Poisson maurice("Maurice",10,20,3);
Mammifere rantanplan("Rantanplan",5,9,17);
Animal * t[] = { &maurice, &rantanplan };

parcourir(t,2,&Animal::deplacer);
```

HERITAGE MULTIPLE

Est-ce qu'on aurait pas oublié Flipper dans l'histoire ? Oui, je sais, c'est un mammifère (en particulier il met bas de la même manière), mais avouez qu'il se déplace plutôt comme un poisson. Donc je propose qu'on déclare une classe `Dauphin` qui hérite à la fois de `Poisson` et de `Mammifere`. Attention, notez bien que tous les langages orientés objet n'autorisent pas cette manipulation. Voici la déclaration de la classe.

```
class Dauphin : public Poisson, public Mammifere {
 ...

 public:
  void     deplacer(void) { Poisson::deplacer(); }
  Animal * engendrer(void) { return Mammifere::engendrer(); }
};
```

Problèmes

Cette déclaration soulève tout de même quelques problèmes. Tout d'abord, le dauphin hérite de la méthode `deplacer` à la fois de `Poisson` et de `Mammifere`. Il faut donc choisir laquelle sera effectivement appelée quand on voudra déplacer un dauphin. La solution est dans le corps de la méthode `deplacer`: celle-ci est redéfinie, et à l'aide de l'opérateur `::`, on appelle la méthode `deplacer` de la super-classe qu'on souhaite (ici c'est `Poisson`).

Le second problème se situe dans la construction même des instances. Revenons aux classes `Poisson` et `Mammifere`, et penchons-nous sur leur constructeur.

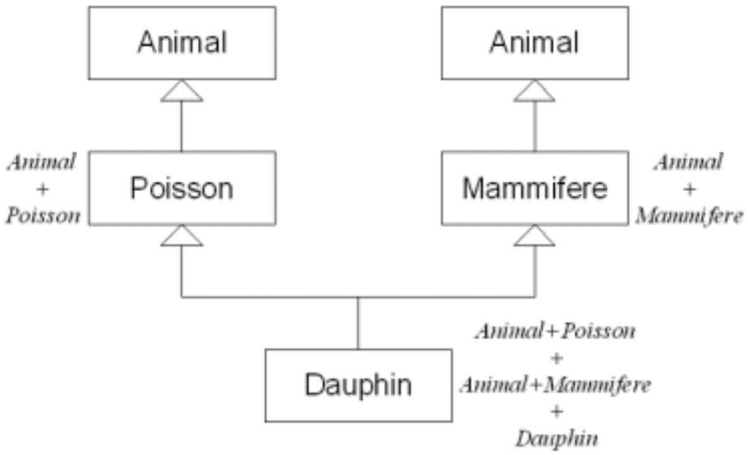
```
Poisson::Poisson(char * n,int x,int y,int p)
: Animal(n,x,y), _profondeur(p) {}
```

```
Mammifere::Mammifere(char * n,int x,int y,int v)
: Animal(n,x,y), _vitesse(v) {}
```

Nous avons vu aux chapitres précédents qu'avant l'appel au constructeur d'une classe, les attributs étaient construits. En fait, dans le cas d'un héritage, avant même la construction des attributs, il y a la construction de la partie de l'objet issue des super-classes. Ainsi, pour les classes `Poisson` et `Mammifere`, il faut construire la partie `Animal` avant la partie propre à chaque sous-classe. Dans le cas maintenant de la classe `Dauphin`, la construction est la suivante.

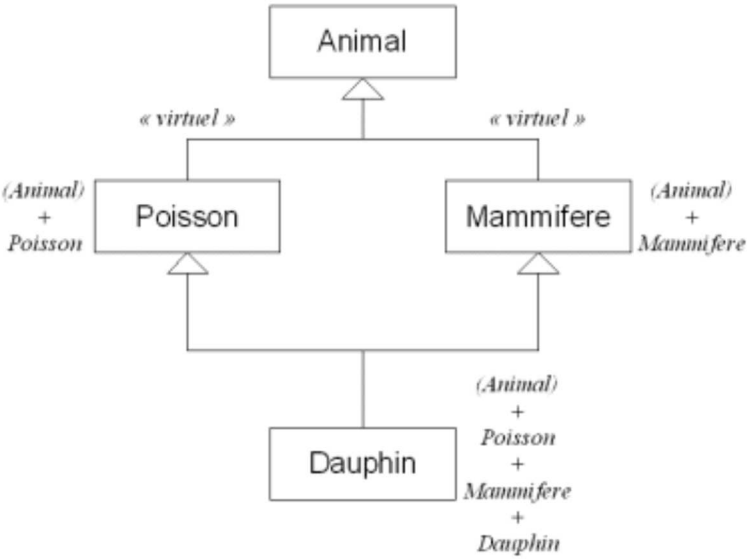
```
Dauphin::Dauphin(char * n,int x,int y,int p,int v)
: Poisson(n,x,y,p), Mammifere(n,x,y,v) {}
```

Le problème de l'héritage multiple devient alors flagrant: une instance `Dauphin` se retrouve avec deux fois la partie `Animal` (l'une issue de `Poisson` et l'autre de `Mammifere`), ce qui n'est pas du tout ce qu'on souhaite ici.



Héritage virtuel

Idéalement, il faudrait qu'un dauphin soit constitué d'une seule partie `Animal`, d'une sous-partie `Poisson` et d'une sous-partie `Mammifere`. Pour résoudre ce problème, il existe l'héritage *virtuel*. L'idée est qu'on hérite d'une classe, mais si par héritage multiple, on se retrouve avec plusieurs fois la même classe, alors on ne la considère qu'une seule fois.



Dans notre exemple, il faut donc effectuer les modifications suivantes.

```
class Poisson : public virtual Animal { ... };
```

```
class Mammifere : public virtual Animal { ... };

class Dauphin : public Poisson, public Mammifere {
 ...

 Dauphin(char * n,int x,int y,int p,int v)
 : Animal(n,x,y), Poisson(n,x,y,p), Mammifere(n,x,y,v) {}

 ...
};
```

Au niveau de la construction, il faut alors détailler le chaînage de la classe de base (i.e. la super-classe la plus haute dans la hiérarchie) jusqu'aux sous-classes immédiates. Même si les arguments `n`, `x`, `y` sont répétés pour le constructeur de `Animal`, de `Poisson` et de `Mammifere`, ils ne sont réellement pris en compte que dans la classe de base `Animal`. Ensuite, ils ne servent qu'à repérer les arguments significatifs pour chaque sous-classe, i.e. `p` et `v` dans notre exemple.

CLASSE ABSTRAITE ET INTERFACE

Nous présentons ici brièvement la notion de classe abstraite, qui peut être associée à la notion d'interface, qui est une alternative intéressante pour l'héritage multiple. Ce concept est notamment proposé dans le langage Java.

Méthode abstraite

Une méthode virtuelle peut ne pas avoir de corps. En effet, si l'on reprend l'exemple de la classe `Animal`, on peut souhaiter ne pas fournir de comportement par défaut aux méthodes virtuelles. Une solution peut être de définir les méthodes de la manière suivante.

```
virtual void     deplacer(void) {}
virtual Animal * engendrer(void) {}
```

On peut donc créer une instance de la classe `Animal` qui n'a pas vraiment de sens, des méthodes sans action pouvant être exécutées. Il peut être intéressant de définir alors des méthodes virtuelles qui n'ont pas de corps, ainsi elles ne pourront pas être exécutées directement. Ces méthodes sont dites *abstraites*, à l'opposé des autres qui sont dites *concrètes*. L'intérêt d'une méthode abstraite est double: elle empêche tout d'abord la classe d'être instanciée, ensuite elle oblige les sous-classes à proposer une implémentation.

Classe abstraite

Une classe *abstraite* est tout simplement une classe qui possède au moins une méthode abstraite. L'exemple suivant rend la classe `Animal` abstraite, en déclarant les méthodes virtuelles `deplacer` et `engendrer` abstraites.

```
virtual void     deplacer(void) = 0;
virtual Animal * engendrer(void) = 0;
```

Les méthodes, comme les fonctions, sont des pointeurs en C++. Pour déclarer une méthode abstraite, il suffit de lui affecter le pointeur `NULL`. Lorsqu'une classe est abstraite, il est impossible de créer une instance de cette classe, seule la manipulation de pointeurs ou de références est possible. Prenons l'exemple suivant, toutes les lignes sont autorisées exceptée la première.

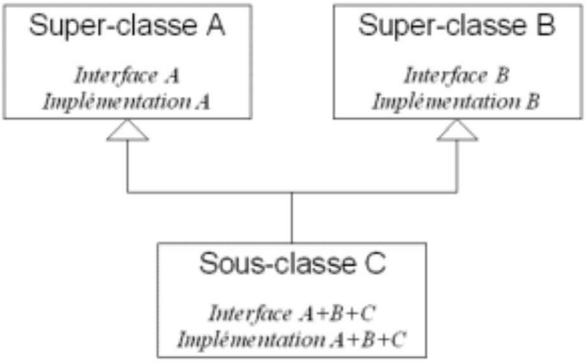
```
Animal a("un animal",0,0);
Poisson p("Maurice",10,20,3);
Animal * pa = &p;
```

```
Animal & ra = p;
```

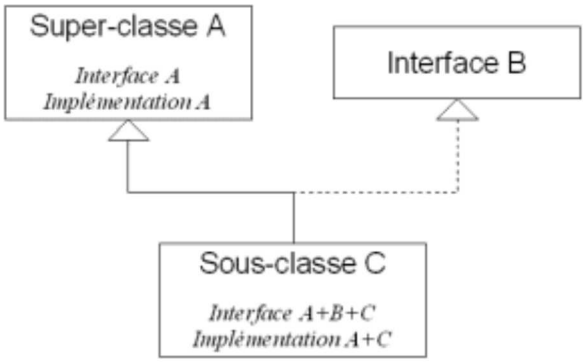
Interface et classe abstraite pure

Une classe abstraite qui ne possède que des méthodes abstraites (aucune méthode concrète, virtuelle ou non, et aucun attribut) est dite *abstraite pure*. Elle ne décrit en fait qu'une interface, i.e. un jeu de méthodes qu'il faudra implémenter dans les sous-classes si l'on souhaite les instancier. La notion de classe abstraite pure est très proche de la notion d'*interface* en Java. Ce langage interdit en effet l'héritage multiple de classes concrètes ou abstraites grâce à ce concept.

Une classe est formée de deux parties: une *interface* (qui décrit la partie visible: prototypes des méthodes publiques) et une *implémentation* (qui décrit la partie cachée: attributs et contenu des méthodes). Lors de la spécialisation d'une classe, son interface et de son implémentation sont héritées toutes les deux, comme le montre la figure suivante.



Dans le cas d'un héritage multiple, les problèmes surviennent au niveau de l'héritage des implémentations: méthodes avec le même nom mais un corps différent, ou héritage double d'une classe qui implique une duplication de l'implémentation. Pour éviter ces difficultés, Java empêche l'héritage multiple d'implémentations en imposant l'héritage simple pour les classes et en autorisant l'héritage multiple pour les interfaces.



MEMBRE DE CLASSE

Dans tous nos exemples, les attributs que nous avons déclarés sont propres aux instances, c'est-à-dire que chaque objet construit possède son jeu d'attributs qu'il est le seul à pouvoir modifier (à condition qu'ils soient encapsulés). Il est possible de déclarer des attributs propres non pas aux objets mais à la classe elle-même. Cela signifie qu'il n'existe qu'une seule instance de ces attributs dans tout le programme et qu'ils sont partagés entre toutes les instances de la classe. Le mot-clé `static` est utilisé pour déclarer de tels attributs. Ceux-ci sont appelés attributs *de classe* en opposition aux attributs communs dits *d'instance*. Dans notre exemple, nous pouvons imaginer un compteur du nombre d'instances de la classe `Animal` dans un programme. Pour cela, nous effectuons les modifications suivantes.

```
class Animal {
 ...

 protected:
  static int _nb_instance;

 public:
  static int getNbInstance(void) { return _nb_instance; }

  Animal(char * n,int x,int y)
  : _nom(n), _x(x), _y(y) { ++_nb_instance; }

  virtual ~Animal(void) { --_nb_instance; }

 ...
};
```

Pour manipuler ces attributs de classe, il est naturellement possible de créer des méthodes *de classe*, en utilisant de manière similaire le mot-clé `static`. Dans l'exemple, le constructeur et le destructeur sont utilisés pour mettre à jour le compteur. Ainsi, à tout moment, on connaît le nombre exact d'instances de la classe `Animal` dans un programme. Pour accéder à un membre de classe, on utilise l'opérateur `::` de la manière suivante.

```
std::cout << "Nombre animaux: " << Animal::getNbInstance();
```

En résumé, une méthode de classe est équivalente à une fonction, car une instance de la classe n'est pas nécessaire pour appeler l'une de ses méthodes statiques, même si cela reste possible comme le montre le code suivant.

```
Poisson p("Maurice",10,20,3);
...
std::cout << "Nombre animaux: " << p.getNbInstance();
```

La seule différence avec une fonction est qu'une méthode statique appartient à sa classe et peut donc accéder à tous ses membres statiques, cachés ou non. En ce qui concerne les attributs de classe, ils sont aussi très proches des variables globales, et à leur image, ils doivent être initialisés dans un fichier source `.cpp` et non pas dans un *header* `.hpp` (ce qui entraînerait des complications au niveau des inclusions de fichiers). Ainsi, dans un fichier `.hpp`, on aura le code suivant.

```
class Animal {
 ...
 protected: static int _nb_instance;
 ...
};
```

Et dans le fichier `.cpp` associé, on retrouvera le code suivant, qui initialise l'attribut de classe.

```
int Animal::_nb_instance = 0;
```

MECANISME *RTTI*

Le mécanisme RTTI (*Run-Time Type Information*) est très utile pour déterminer la classe réelle d'un objet en cours d'exécution, lorsque celui-ci est pointé ou référencé. Il s'agit du même type de contrôle que celui effectué par l'opérateur de conversion `dynamic_cast`. Le mot-clé `typeid` est utilisé pour retourner une structure de type `type_info` (ne pas oublier d'inclure le fichier `<typeinfo>`). Considérons le code suivant.

```
#include <typeinfo>

...
```

```
Poisson p("Maurice",10,20,3);
Mammifere m("Rantanplan",5,9,17);
Animal * pa = &p;
Animal * pb = &m;
...
std::cout << typeid(*pa).name();
```

La structure `type_info` contient des informations sur le type fourni à `typeid`. Notamment, comme le montre l'exemple, il est possible d'afficher le nom du type en question. Mais le plus intéressant est que les opérateurs `==` et `!=` ont été surchargés pour le type `type_info`. Ainsi, il est possible de vérifier si les pointeurs `pa` et `pb` pointent effectivement sur le même type d'objet.

```
if (typeid(*pa)==typeid(*pb))
 std::cout << "Ils sont de même type.";
else std::cout << "Ils ne sont pas de même type.";
```

L'opérateur `typeid` peut également s'appliquer sur un type, ce qui permet de tester directement qu'un pointeur référence bien, par exemple, un poisson.

```
if (typeid(*pa)==typeid(Poisson))
 std::cout << "C'est un poisson.";
else std::cout << "Ce n'est pas un poisson.";
```

Il y a tout de même un petit piège avec l'opérateur `typeid`. Pour obtenir le type réel d'un objet, il faut fournir sa référence et non pas son pointeur, comme le montre le tableau ci-dessous. En effet, le pointeur est considéré comme un type à part et il n'y a donc pas de lien d'héritage entre deux pointeurs, i.e. un pointeur `Poisson *` n'a aucun lien (pour `typeid`) avec `Animal *`, alors qu'une référence `Animal &` peut en fait être une référence sur un objet de type `Poisson`. Considérons le code suivant.

```
Animal * pp = new Poisson("Maurice",10,20,3);
Animal & rp = *pp;
```

Le tableau qui suit montre les égalités possibles entre les types, selon qu'il s'agisse de pointeurs ou de références.

	typeid(Animal)	**typeid(Poisson)**	**typeid(Animal *)**	**typeid(Poisson *)**
typeid(pp)	!=	!=	==	!=
typeid(rp)	!=	==	!=	!=
typeid(*pp)	!=	==	!=	!=
typeid(&rp)	!=	!=	==	!=

EXCEPTIONS

Gérer les erreurs dans un programme est toujours une chose très délicate. La technique habituelle consiste à traiter localement le problème, comme dans l'exemple suivant.

```
void deplacerPoisson(Animal * a) {
 Poisson * p = dynamic_cast<Poisson *>(a);

 if (p==0) {
  std::cerr << "Erreur: ce n'est pas un poisson." << std::endl;
  exit(1);
 }

 ... // Le poisson se déplace.
}
```

Le premier défaut d'une telle approche est de forcer un couplage entre la fonction et la manière d'afficher du programme. En effet, la fonction écrit ici l'erreur sur le flux standard, mais on pourrait imaginer que la fonction est utilisée dans une application graphique où le message d'erreur apparaît dans une boîte de dialogue. Une solution pourrait consister à utiliser une fonction qui se charge d'afficher l'erreur.

```
void afficherErreur(char * s) {
 std::cerr << "Erreur: " << s << std::endl;
 exit(1);
}

void deplacerPoisson(Animal * a) {
 Poisson * p = dynamic_cast<Poisson *>(a);

 if (p==0) afficherErreur("ce n'est pas un poisson.");
 ... // Le poisson se déplace.
}
```

L'autre défaut de l'approche est qu'il est difficile dans ce cas de reprendre le programme après le traitement de l'erreur. On a choisit ici de sortir du programme, mais on pourrait imaginer vouloir reprendre le cours du programme à un endroit bien précis. De ces principaux défauts sont nées les exceptions qui permettent une plus grande souplesse et une certaine factorisation dans la gestion des erreurs. Reprenons notre exemple.

```
void deplacerPoisson(Animal * a) {
 Poisson * p = dynamic_cast<Poisson *>(a);

 if (p==0) throw Chaine("ce n'est pas un poisson.");
 ... // Le poisson se déplace.
}
```

Lorsqu'une erreur est détectée, on choisit de la traiter localement ou de la transmettre à la méthode (ou à la fonction) appelante. C'est le rôle du mot-clé `throw` qui *jette* un objet, appelé une *exception*, à la méthode appelante. Dans l'exemple, `throw` transmet un objet `Chaine` créé à la volée en appelant explicitement son constructeur. La méthode courante est terminée en détruisant ses variables locales et ses arguments. Notez que toutes les allocations dynamiques de la méthode doivent être traitées manuellement. Une fois la méthode terminée, la main est rendue à la méthode appelante qui suspend alors son exécution et tente de traiter l'erreur.

```
int main(void) {
 Animal * a = new Mammifere("Rantanplan",5,9,17);

 try {
  deplacerPoisson(a);
  ...
 }

 catch(const Chaine & c) {
  std::cerr << "Erreur: " << c << std::endl;
  return 1;
 }

 catch(...) {
  std::cerr << "Erreur inconnue." << std::endl;
  return 2;
 }

 return 0;
}
```

Dans la méthode appelante, une partie de code est surveillée par un bloc `try`. Lorsqu'une exception est levée dans cette zone, l'exécution de la méthode est suspendue et reprend dans l'une des méthodes

`catch` qui suit le bloc `try`. Les méthodes sont testées dans leur ordre de déclaration. Dès que le type de l'argument d'une méthode `catch` correspond au type de l'exception levée, alors la méthode est exécutée. Dans notre exemple, si une exception de type `Chaine` est levée, alors c'est la première méthode `catch` qui est exécutée. En revanche, pour tous les autres types d'exception, c'est la seconde méthode (qui prend en paramètre n'importe quel type d'objet grâce au mot-clé `...`) qui est lancée. Notez que pour éviter toute recopie de l'exception, les méthodes `catch` reçoivent des références.

Dans l'hypothèse où une méthode `m` susceptible de recevoir une exception ne la gère pas (soit parce qu'aucun bloc `try` n'a été établi, soit parce qu'aucune méthode `catch` ne correspond au type de l'exception) alors l'exception est automatiquement transmise à la méthode qui a appelé `m`. Ce processus se répète jusqu'à la fonction `main` où l'erreur doit impérativement être traitée. Si ce n'est pas le cas, l'exception est lancée dans le vide et une erreur du genre *aborted* apparaît.

Dans notre exemple, nous avons simplement jeté un objet de type `Chaine`. Mais grâce à la STL, le C++ fournit une hiérarchie de classes d'exception. La classe de base (i.e. la classe en haut de la hiérarchie) est `exception`. Elle se décline ensuite en sous-classes pour divers types d'erreur: fichiers, débordement... (cf. cette page pour les détails). Voici un exemple simple qui permet de créer sa propre classe d'exception (très important pour éviter de réécrire le message d'erreur à chaque appel, mais également pour permettre des traitements différents au niveau des `catch`, en fonction de la classe de l'exception).

```
#include <stdexcept>

class ErreurConversionPoisson : public std::exception {
 public:
  const char * what(void) const throw()
  { return "ce n'est pas un poisson."; }
};

void deplacerPoisson(Animal * a) {
 Poisson * p = dynamic_cast<Poisson *>(a);

 if (p==0) throw ErreurConversionPoisson();
 ... // Le poisson se déplace.
}

int main(void) {
 Animal * a = new Mammifere("Rantanplan",5,9,17);

 try {
  deplacerPoisson(a);
  ...
 }

 catch(const std::exception & e) {
  std::cerr << "Erreur: " << e.what() << std::endl;
  return 1;
 }

 catch(...) {
  std::cerr << "Erreur inconnue." << std::endl;
  return 2;
 }

 return 0;
}
```

La classe `exception` implémente une méthode virtuelle `what` qui retourne le message associé à l'erreur. Pour créer sa propre classe d'exception, il suffit de redéfinir cette méthode avec son propre message (ne pas oublier d'inclure le fichier `<stdexcept>`). Notez qu'il est tout à fait possible de rajouter des attributs: par exemple, pour une erreur d'entrée/sortie, le nom du fichier qui pose

problème peut être mémorisé.

```
class ErreurFichier : public std::exception {
 protected:
  Chaine _fichier;

 public:
  ErreurFichier(char * s) : _fichier(s) {}

  const char * what(void) const throw() {
   Chaine c = Chaine("impossible d'ouvrir le fichier ")+_fichier;

   return (const char *)c;
  }
};

...
if (...) throw ErreurFichier("dummy.txt");
...
```

4. PATRONS DE COMPOSANT ET CLASSES GENERIQUES

Les patrons de composant sont l'une des particularités du langage C++. Ils sont souvent appelés *templates* (tiré du mot-clé utilisé dans le langage). Ce concept permet de paramétrer, dans un composant (une fonction ou une classe), le type de certaines données manipulées. Contrairement aux arguments d'une fonction ou d'une méthode, les inconnues ne sont pas des valeurs mais des types. Par la suite, nous ferons la différence entre le terme *paramètre*, employé pour désigner un paramètre d'un patron et donc un type, et le terme *argument*, employé pour désigner un paramètre d'une méthode et donc une variable.

Ce concept ne fait pas partie intégrante du paradigme objet, car il est tout à fait possible de définir le patron d'une fonction, notion qui n'est pas objet. Les patrons de composant ont d'ailleurs donné naissance à un nouveau type de programmation dite *générique*. Néanmoins, il est très courant de combiner la programmation générique avec l'approche orientée objet, en définissant des patrons de classe qu'on nomme *classes génériques*. Dans ce contexte, il est plus judicieux de considérer les patrons comme une extension de la programmation orientée objet.

Après avoir présenté les notions de base sur les patrons de fonction et de classe en C++, nous nous penchons sur des aspects plus particuliers comme la définition de paramètres par défaut, ainsi que l'instanciation partielle d'un patron, qui permet de spécialiser ce dernier pour certains paramètres.

CLASSE GENERIQUE ET META-CLASSE

Un patron de classe (respectivement de fonction) est un modèle de classe (respectivement de fonction), de la même manière qu'une classe est un modèle d'objet. La différence entre une classe générique et une classe simple se trouve dans l'instanciation du modèle qui fournit une classe et non pas un objet. On est d'ailleurs tenté de dire qu'une classe générique est une *méta-classe* (i.e. une classe qui est un modèle de classe), mais en C++ notamment, une classe générique n'est pas traitée comme une classe, il s'agit simplement d'un modèle. Le terme méta-classe est alors plutôt réservé aux langages comme Java qui considèrent les classes comme des objets, ce qui signifie qu'il existe des classes dont l'instance unique est une classe, et ce sont elles les méta-classes.

PATRON DE FONCTION

De la macrocommande au *template*

Pour illustrer ce qu'est un patron de fonction, nous avons choisi l'exemple de la fonction `max` qui est sensée retourner le maximum de deux valeurs. Avec une approche classique, on est obligé de définir cette fonction pour chaque type de donnée.

```
inline int    max(int a,int b) { return (a>b ? a : b); }
inline double max(double a,double b) { return (a>b ? a : b); }
```

Le principal défaut réside dans la réécriture manuelle du code pour chaque type de donnée. Une autre solution en C consiste à utiliser une macrocommande.

```
#define max(a,b) ((a)>(b) ? (a) : (b))
```

On évite ainsi de réécrire manuellement le code pour chaque type de donnée, mais on introduit d'autres problèmes. Tout d'abord, contrairement à la version précédente, aucune vérification de type

n'est effectuée. On peut alors comparer des éléments de nature différente. Ensuite, il peut se produire des effets de bord comme dans l'exemple suivant.

```
int x = ...;
int y = ...;
...
int z = max(x++,y++);
```

Grâce aux patrons, on peut éviter tous ces défauts. L'exemple suivant définit un modèle de la fonction `max` dont le paramètre `T` est le type des objets qui sont comparés.

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

Instanciation d'un *template*

Il faut bien comprendre que le code précédent n'est pas une fonction, mais un modèle. Ainsi, tel quel, il n'y a pas de compilation du code. Cela ne se produira que lorsqu'on *instanciera* le modèle, c'est-à-dire lorsqu'on précisera le véritable type des paramètres. Ainsi, du modèle `max`, on peut définir la fonction `max<int>` qui est l'instanciation du modèle pour `T = int`. Imaginons deux entiers (de type `int`) `i` et `j`, le code suivant instancie la fonction `max<int>`.

```
std::cout << max<int>(i,j) << std::endl;
```

Il y a tout de même deux remarques concernant l'instanciation d'un patron. Tout d'abord, comme il s'agit d'un modèle, un patron doit toujours se trouver dans un *header*, sinon il ne pourra pas être instancié chaque fois qu'on en a besoin. Ensuite, l'instanciation du patron correspond simplement à remplacer le paramètre `T` par un type donné, dans notre exemple `int`. Une fois ce code généré, il est compilé comme s'il avait été écrit directement par un programmeur. Il n'y a donc aucune différence (excepté l'automatisme) avec un copier/coller à la main du code. Cela signifie que pour chaque jeu de paramètres, une fonction est créée, même si le modèle n'est écrit qu'une seule fois. Néanmoins, si l'on fait appel au modèle plusieurs fois avec le même jeu de paramètres, la fonction associée n'est présente qu'une seule fois dans le programme (bien qu'elle puisse être compilée plusieurs fois, à cause de la compilation séparée).

En résumé, un patron (de classe ou de fonction) doit toujours être défini dans un *header*, ce qui peut conduire à une augmentation significative du temps de compilation. En outre, le patron est instancié pour chaque jeu (différent) de paramètres avec lequel il est appelé, ce qui peut également induire une augmentation significative de la taille du code généré.

Polymorphisme statique

Il n'est pas toujours nécessaire de préciser les paramètres d'un patron pour l'instancier. En effet, dans l'exemple précédent, les arguments `i` et `j` sont tous les deux de type `int`, le compilateur peut alors, en analysant le patron `max`, déduire que `T` est de type `int`. Le code suivant est donc valide et équivalent au précédent.

```
std::cout << max(i,j) << std::endl;
```

Ce mécanisme est appelé *polymorphisme statique*, puisqu'on écrit simplement le nom du patron et le compilateur décide seul de la fonction à instancier. Mais ce procédé implicite peut causer quelques problèmes, comme le montre l'exemple suivant.

```
int i,j;
long k,l;
Chaine a,b,c;
```

```
...
std::cout << max(i,j) << std::endl;
std::cout << max(k,l) << std::endl;
std::cout << max<long>(i,l) << std::endl;
std::cout << max(long(i),l) << std::endl;
std::cout << max(a,b) << std::endl;
```

Aux deux premières lignes, les fonctions `max<int>` et `max<long>` sont appelées implicitement. En revanche, si on ne précise pas le paramètre du patron à la troisième ligne, e.g. `max(i,l)`, quelle instanciation le compilateur va-t'il choisir, `max<int>` ou `max<long>` ? Comme la règle de conversion `int` vers `long` est implicite, le compilateur choisira certainement la seconde instanciation, mais pour éviter toute confusion, il est plus sage de préciser l'instanciation à ce moment-là. Il est également possible d'expliciter la conversion `int` vers `long` (cf. la quatrième ligne), le polymorphisme statique peut alors opérer correctement.

Notion de concept

Dans le cas de `max<Chaine>` (cf. dernière ligne de l'exemple précédent), la compilation de la fonction requiert un opérateur de comparaison `>` pour la classe `Chaine`, ce que le compilateur ne peut pas vérifier à l'avance. Cela va donc se traduire par une erreur de compilation de la fonction `max<Chaine>` sur la ligne qui tente d'appeler l'opérateur `>`. Ce message est malheureusement assez déroutant si la personne qui utilise le patron n'est pas celle qui l'a écrit, car l'erreur va pointer le code de la fonction.

Pour faciliter l'usage des patrons, il est très important, au moment de la définition d'un *template*, de spécifier quelle interface un paramètre doit respecter pour que le patron puisse être instancié. Dans notre exemple, il faudrait idéalement pouvoir indiquer que `T` doit posséder l'opérateur `>`. L'interface nécessaire à un paramètre pour l'instanciation de son *template* est plus souvent appelée *concept*. En C++, cette notion est implicite et il incombe au programmeur de bien documenter ses patrons pour informer au mieux l'utilisateur. Dans d'autres langages, comme notamment *Generic C#* et *Java Generics*, les extensions aux classes génériques de C# et Java, il est prévu des mots-clé pour spécifier directement dans le code les concepts que doivent respecter les paramètres des patrons.

Opérateurs logiques

Les opérateurs logiques `<`, `>`, `<=`, `>=`, `==` et `!=` sont souvent surchargés pour permettre la comparaison de deux objets d'une classe donnée. Mais c'est un peu fastidieux d'écrire les 6 opérateurs à chaque fois. En y réfléchissant, avec seulement `==` et `<`, on peut construire tous les autres.

```
template <class T>
inline bool operator > (const T & a,const T & b)
{ return (b<a); }

template <class T>
inline bool operator <=(const T & a,const T & b)
{ return (!(b<a)); }

template <class T>
inline bool operator >=(const T & a,const T & b)
{ return (!(a<b)); }

template <class T>
inline bool operator !=(const T & a,const T & b)
{ return (!(a==b)); }
```

Pour la classe `Chaine` par exemple, il suffit d'implémenter `==` et `<`, et automatiquement on dispose des 6 opérateurs logiques. Notez qu'une majorité de compilateurs, notamment la famille des GCC,

fournit maintenant les quatre opérateurs >, <=, >= et != automatiquement.

PATRON DE CLASSE, CLASSE GENERIQUE

De manière identique aux fonctions, il est possible de définir des patrons de classe, qu'on nomme communément *classes génériques*. Nous avons choisi d'illustrer ce concept avec l'exemple d'un vecteur d'éléments où le paramètre du *template* est justement le type des éléments. Voici l'interface de la classe générique (notez les similitudes avec la classe `Chaine`, qui est très proche de la classe `Vecteur<char>`).

```
template <typename T> class Vecteur {
 protected:
  T * _t; // Tableau d'éléments.
  int _n; // Nombre d'éléments.

 public:
  Vecteur(int = 10);
  Vecteur(const Vecteur &);

  Vecteur & operator=(const Vecteur &);
  const T & operator[](int) const;
  T &       operator[](int);
};
```

Supposons maintenant qu'on souhaite définir l'une des méthodes à l'extérieur de la définition du patron, par exemple l'opérateur =.

```
template <typename T>
Vecteur<T> & Vecteur<T>::operator=(const Vecteur & v) {
 int i = 0;

 if (this!=&v) {
  if (_t!=0) delete [] _t;
  _n=v._n;
  _t=new T[_n];
  while (i<_n) _t[i]=v[i++];
 }

 return *this;
}
```

Il faut donc préciser à nouveau le paramétrage. En outre, vous remarquerez qu'avant les `::` qui indiquent l'entrée dans la classe `Vecteur<T>`, l'utilisation du patron `Vecteur` doit être précisée avec le ou les paramètres pour l'instancier, c'est-à-dire `Vecteur<T>`. Une fois à l'intérieur du patron (i.e. une fois passés les `::`), l'instanciation du patron en `Vecteur<T>` est implicite. Notez également que le nom des constructeurs et du destructeur n'est jamais suivi de `<T>` (qu'ils soient définis à l'intérieur ou à l'extérieur du patron), il s'agit toujours de `Vecteur(...)` et `~Vecteur(void)`, et non de `Vecteur<T>(...)` et `~Vecteur<T>(...)`.

Tout dans le *header*

Nous l'avons déjà dit, la définition des patrons doit se trouver intégralement dans les *headers*, ce qui peut soulever quelques difficultés. Voici l'exemple plus complet du patron `Vecteur` auquel on ajoute un patron `Iterateur`. Un itérateur est un pointeur au sens objet qui référence un élément d'un conteneur (cf. les *designs patterns* du GoF). Voici tout d'abord le fichier `vecteur.hpp`.

```
#ifndef _VECTEUR_H_
#define _VECTEUR_H_
```

```
 #include "iterateur.hpp"

 template <typename T> class Iterateur;

 template <typename T> class Vecteur {
  protected:
   T * _t; // Tableau d'éléments.
   int _n; // Nombre d'éléments.

  public:
   Vecteur(int = 10);
   Vecteur(const Vecteur &);

   Vecteur & operator=(const Vecteur &);
   const T & operator[](int) const;
   T &       operator[](int);

   Iterateur<T> begin() const;
   Iterateur<T> end() const;
 };

#endif
```

Comme tout *header* qui se respecte, il possède un *gardien*. Son rôle est primordial ici puisque l'inclusion entre les fichiers `iterateur.hpp` et `vecteur.hpp` est cyclique (chacun inclut l'autre). Les gardiens permettent de casser cette boucle après l'inclusion de chaque fichier. Ainsi, si dans un programme on inclut `vecteur.hpp`, alors le préprocesseur inclura d'abord `iterateur.hpp` (le gardien de `vecteur.hpp` a été activé à sa première inclusion), puis il continuera `vecteur.hpp`. A l'opposé, si on inclut `iterateur.hpp` dans un programme, alors le préprocesseur inclura d'abord `vecteur.hpp` avant de poursuivre `iterateur.hpp`.

Dans la seconde situation, cela signifie que le patron `Vecteur` sera défini avant le patron `Iterateur`, ce qui pose un problème puisque la définition de `Vecteur` fait appel au patron `Iterateur`. Pour éviter tout problème, une pré-déclaration de `Iterateur` doit être effectuée (cf. la ligne juste après le `include`), cette déclaration est dite *avancée* (ou *forward declaration* en anglais). Elle permet de manipuler `Iterateur` avant sa véritable déclaration (cette notion est similaire à celle de prototype d'une fonction).

La déclaration avancée peut également être utilisée pour une classe simple, mais dans ce cas, seule la manipulation d'un pointeur ou d'une référence de cette classe est autorisée et il est impossible d'appeler une méthode à partir d'une déclaration avancée (le compilateur ne peut pas vérifier et donc encore moins compiler). Cela implique de déporter la définition des méthodes qui manipulent la classe en déclaration avancée dans le fichier source `.cpp` associé et d'inclure alors tous les *headers* nécessaires pour compléter la déclaration avancée.

Mais pour les patrons, ce problème ne se pose pas, puisqu'il ne s'agit que de modèles, l'instanciation et la compilation se faisant plus tard. A l'image de `vecteur.hpp`, voici le fichier `iterateur.hpp`.

```
#ifndef _ITERATEUR_H_
#define _ITERATEUR_H_

 #include "vecteur.hpp"

 template <typename T> class Iterateur;
 template <typename T> class Vecteur;

 template <typename T>
 bool operator==(const Iterateur<T> &,const Iterateur<T> &);

 template <typename T> class Iterateur {
  protected:
```

```
    T * _p; // Pointeur sur un élément du vecteur.

    Iterateur(T * p) : _p(p) {}

   public:
    Iterateur(void) : _p(0) {}

    T &         operator*(void) const { return *_p; }
    Iterateur & operator++(void);
    Iterateur   operator++(int);

   friend class Vecteur<T>;

   friend bool operator==<T>(const Iterateur<T> &,
                             const Iterateur<T> &);
  };

 #endif
```

Grâce aux méthodes `begin` et `end`, un vecteur peut créer des itérateurs pointant sur le début ou la fin de son tableau. Pour éviter une mauvaise initialisation des itérateurs, la construction d'un itérateur à partir d'un pointeur est protégée. La classe `Iterateur<T>` déclare alors la classe `Vecteur<T>` amie (attention, je n'ai pas écrit "le patron `Iterateur` déclare alors le patron `Vecteur` ami"). Ainsi, seul un vecteur peut créer des itérateurs initialisés à partir d'un pointeur.

Le patron `Iterateur` défini ici est inspiré des itérateurs de la STL: les opérateurs `++` déplacent l'itérateur et l'opérateur `*` renvoie l'élément pointé. Une dernière remarque, il est utile de pouvoir comparer deux itérateurs, l'opérateur `==` doit donc être surchargé. Il s'agit là aussi d'un patron. Celui-ci a besoin de comparer les attributs de deux itérateurs, c'est pourquoi il doit être ami de la classe `Iterateur`.

Remarquez que l'amitié est donnée ici à l'instance `operator==<T>` uniquement et non pas au patron. Tous les compilateurs ne comprennent pas la syntaxe utilisée ici pour déclarer l'amitié avec l'instance d'un patron. Ils utilisent alors une syntaxe plus simple qui indique simplement que la fonction amie est en fait une instance d'un patron, mais sans préciser de quelle instance il s'agit. Ils supposent que la signature de la fonction suffira à déterminer de quelle instance il s'agit. Voici comment serait alors définie l'amitié de la classe `Iterateur<T>` pour l'opérateur `==`.

```
friend bool operator==<>(const Iterateur<T> &,
                         const Iterateur<T> &);
```

La première écriture semble plus logique, dans la mesure où certaines instanciations ne peuvent pas être déduites totalement à partir de la signature d'une méthode (e.g. lorsque le type de retour de la fonction est un paramètre du patron). Néanmoins, par expérience, il resort que la seconde syntaxe est à l'heure actuelle celle qui est la plus tolérée.

Pour conclure, voici un petit exemple qui remplit un vecteur d'entiers avec des nombres pairs en utilisant les patrons définis précédemment.

```
Vecteur<int> v(10);
Iterateur<int> courant = v.begin();
Iterateur<int> fin = v.end();
int i = 1;

while (courant!=fin) {
 i*=2;
 *(courant++)=i;
}
```

ATTRIBUT STATIQUE

A l'image des variables globales, les attributs statiques doivent être initialisés dans un fichier source `.cpp` pour éviter tout conflit lors des inclusions. Mais, concernant les classes génériques, toute leur définition doit se trouver dans un *header*, même l'initialisation des attributs statiques. Prenons l'exemple suivant.

```
template <typename VEHICULE> class Usine {
 protected:
  static int _nb_instances;
  ...

 public:
  Usine(void) { ++_nb_instances; }
  ~Usine(void) { --_nb_instances; }

 ...
};

template <typename VEHICULE> int Usine<T>::_nb_instances = 0;
```

L'attribut statique sert ici à compter le nombre d'instances de chaque classe `Usine<...>` (remarque: il y a bien un compteur par instanciation du patron `Usine`). Pour l'initialiser, c'est la même approche que pour une classe, seulement on est dans le *header* et il faut préciser le paramétrage du patron.

PARAMETRE PAR DEFAUT

Il est possible de préciser, comme pour les arguments d'une fonction, un type par défaut pour le paramètre d'un patron. Prenons l'exemple d'un arbre binaire de recherche dans lequel on stocke des éléments en leur associant à une clé, et supposons une classe générique avec comme paramètres le type des éléments stockés et le type des clés.

```
template <typename ELEMENT,typename CLE> class Arbre;
```

Mais nous remarquons que la plupart du temps, les utilisateurs utilisent le type `int` pour les clés. Il est alors possible de spécifier ce type comme défaut pour le paramètre `CLE`. Voici la syntaxe.

```
template <typename ELEMENT,typename CLE = int> class Arbre;
```

Les deux lignes suivantes sont alors équivalentes. Il faut simplement remarquer, à l'image des arguments d'une méthode, que les paramètres ne peuvent être omis qu'en partant de la fin.

```
Arbre<Chaine,int> a;
Arbre<Chaine> b;
```

Il est également possible de fournir, non pas un type, mais une constante à un patron. Supposons la classe générique `Vecteur`, pour laquelle nous souhaitons regrouper dans une même classe tous les vecteurs de même taille. Il est alors possible de définir la taille d'un vecteur comme étant un paramètre du patron. Par conséquent, la taille d'un vecteur devient statique.

```
template <typename T,int N = 10> class Vecteur {
 protected:
  T t[N];

 ...
};
```

Les vecteurs déclarés ci-dessous appartiennent à des classes différentes.

```
Vecteur<int,10> v1;
Vecteur<int,8> v2;
```

Considérons maintenant un vecteur creux, c'est-à-dire qu'au lieu de stocker dans un tableau tous les éléments d'un vecteur, on considère qu'une certaine valeur est le fond (i.e. la valeur par défaut) et on ne stocke que les valeurs différentes du fond, en mémorisant la valeur même et sa position dans le vecteur. Imaginons maintenant qu'on souhaite écrire un patron dont la valeur de fond est un paramètre. Voici la déclaration de la classe générique.

```
template <typename T,T FOND> class VecteurCreux;
```

`FOND` est de type `T` et représente la valeur de fond. Attention, seules les constantes d'un certain type sont autorisées. Il s'agit normalement des types primitifs, i.e. les entiers, les flottants et les pointeurs. Cependant, il semblerait que l'usage ait été limité seulement aux types entiers, utiliser des flottants étant devenu obsolète (cf. GCC récent). En ce qui concerne les pointeurs, tous ne conviennent pas, il faut que le compilateur puisse s'assurer que le pointeur est global et constant, c'est-à-dire qu'il sera valide tout au long du programme (car l'instanciation du patron est globale). Tout ceci rend l'utilisation de paramètres constants dans un patron assez rare.

INSTANCIATION PARTIELLE

Spécialisation d'un *template*

Un patron de composant permet de définir une classe ou une fonction générique, c'est-à-dire qui reste la même (à un certain niveau d'abstraction naturellement) quels que soient ses paramètres. Cependant, il peut arriver que, pour un type précis, le modèle général du patron ne convienne pas. Il est alors possible de définir une classe ou une fonction différente pour ce type donné. Prenons l'exemple suivant.

```
template <typename T1,typename T2> class Paire {
 protected:
  T1 _val1;
  T2 _val2;

 public:
  paire(void);

  void afficher(std::ostream & o)
  { o << _val1.toString() << "," << _val2.toString(); }
};
```

Les concepts de `T1` et `T2` doivent implémenter une méthode `toString` qui transforme un objet `T1` ou `T2` en chaîne de caractères. Pour les types primitifs, cette méthode n'est pas disponible, et il est impossible de la rajouter. Il faut donc réécrire la classe `Paire` tout spécialement pour `int` par exemple, de la manière suivante.

```
template <> class Paire<int,int> {
 protected:
  int _val1;
  int _val2;

 public:
  paire(void);

  void afficher(std::ostream & o)
  { o << _val1 << "," << _val2; }
};
```

On utilise ici le mécanisme de l'*instanciation partielle*, sauf que dans cet exemple, tous les paramètres sont instanciés (il s'agit donc d'une véritable instanciation). Le code précédent décrit donc une classe simple (`Paire<int,int>`) et non plus une classe générique (`Paire<T1,T2>`). Avec cette spécialisation du *template*, notre problème n'est pas résolu, puisqu'il répond seulement

au cas où `T1` et `T2` sont de type `int`. Mais que se passe-t-il quand `T1` est entier et pas `T2` (ou inversement) ? Il faut donc là aussi spécialiser le patron.

```
template <typename T2> class Paire<int,T2> {
 protected:
  int _val1;
  T2 _val2;

 public:
  paire(void);

  void afficher(std::ostream & o)
  { o << _val1 << "," << _val2.toString(); }
};
```

Cette fois-ci, il s'agit d'une instanciation partielle de `Paire<T1,T2>` en posant `T1 = int`. Contrairement à l'instanciation `Paire<int,int>`, le code ci-dessus est encore un patron de classe, puisqu'il reste un paramètre (`T2`). Enfin, pour résoudre totalement notre problème, il faudrait aussi écrire l'instanciation partielle `Paire<T1,int>`.

```
template <typename T1> class Paire<T1,int> {
 protected:
  T1 _val1;
  int _val2;

 public:
  paire(void);

  void afficher(std::ostream & o)
  { o << _val1.toString() << "," << _val2; }
};
```

Notez que les trois instanciations sont nécessaires pour considérer tous les cas. Dans l'exemple suivant, chaque cas utilise l'une des instanciations sans qu'il n'y ait la moindre d'ambiguïté. On aurait pu en effet penser que les instanciations `Paire<T1,int>` et `Paire<int,T2>` entrent en conflit avec `Paire<int,int>` lorsque leur dernier paramètre est instancié par `int`, ce qui n'est pas le cas.

```
Paire<int,int>    p1;
Paire<int,Chaine> p2;
Paire<Chaine,int> p3;
```

Template récursif

L'instanciation partielle peut également être utilisée pour définir une récursivité statique, comme le montre l'exemple suivant.

```
template <int N> class Factorielle {
 public: enum { valeur = N*Factorielle<N-1>::valeur };
};

template <> class Factorielle<1> {
 public: enum { valeur = 1 };
};
```

Comme pour toute récursivité, il faut s'assurer de son arrêt. C'est le rôle ici de l'instanciation partielle `Factorielle<1>` qui stoppe les appels récursifs. Si l'on instancie `Factorielle<5>` par exemple, alors `Factorielle<4>`, `Factorielle<3>`, `Factorielle<2>` et `Factorielle<1>` sont également instanciées.

Comme tout se passe à la compilation, l'intérêt d'une telle récursivité est d'éviter le calcul à

l'exécution. Mais on s'aperçoit qu'une classe `Factorielle` sera créée pour chaque nombre qu'on lui fournira en paramètre. L'autre défaut de cette approche est que l'appel doit être déterminé à la compilation. Ainsi, la classe `Factorielle<5>` est valide, alors que `Factorielle<n>` où `n` est une variable n'a pas de sens en C++.

Nous avons choisi d'écrire la récursivité de la fonction factorielle par une classe générique, mais il est également possible de la définir comme un patron de fonction. Néanmoins, tous les défauts énumérés précédemment persistent.

```
template <int N> int factorielle(void)
{ return N*factorielle<N-1>(); }

template <> int factorielle<1>(void) { return 1; }
```

Dans le cas d'une récursivité dynamique, la profondeur, i.e. le nombre d'appels empilés, est limité par la taille de la mémoire centrale, et en particulier par celle de la pile où le programme s'exécute. Pour une récursivité statique, la limitation est effectuée par le compilateur qui n'autorise par défaut qu'une profondeur très faible (e.g. 16 appels). Très rapidement, il peut donc être nécessaire d'augmenter ce seuil. Avec un compilateur de la famille des GCC, l'option `-ftemplate-depth-x` permet de monter la limite à *x* appels. Notez que ce seuil ne limite pas simplement la récursivité statique, mais également l'utilisation d'un *template* pour en définir un autre. Ce qui signifie qu'un usage intensif des *templates*, même sans récursivité, peut conduire le programmeur à augmenter cette limite.

PATRON DE METHODE

Pour conclure ce chapitre, voici un dernier petit détail sur les patrons: il est possible de définir un patron de méthode. Cela signifie qu'une classe peut avoir un nombre indéterminé de méthodes au moment de son écriture, puisqu'à chaque nouvelle instanciation du patron, une méthode sera ajoutée à la classe. Néanmoins, comme toujours avec les *templates* en C++, tout se passe à la compilation, cela signifie qu'au moment de l'exécution, l'interface de la classe est figée. Pour illustrer, voici l'exemple d'une classe qui encapsule simplement un entier.

```
class Entier {
 protected:
  long _val;

 public:
  template <typename T> Entier(const T & v) : _val(long(v)) {}

 ...
};
```

Le constructeur ici est un patron de méthode. L'idée est de pouvoir fournir un constructeur d'objets `Entier` pour toute classe ou type primitif qui autorise la conversion vers un `long`. Telle quelle, la classe est compilée sans constructeur. Ce n'est qu'au moment où l'on tente de construire des objets `Entier` que des constructeurs sont instanciés. Par exemple, le code `Entier('c')` implique l'instanciation, et donc la compilation, du constructeur `Entier(const char &)`.

Il reste un détail de syntaxe à préciser. Supposons que le type de l'attribut `_val` soit un paramètre de la classe `Entier`.

```
template <typename E> class Entier {
 protected:
  E _val;

 public:
  template <typename T> Entier(const T &);
```

```
  ...
};
```

On peut se demander alors comment écrire le constructeur à l'extérieur du patron. Voici la syntaxe, mais attention, tous les compilateurs ne la supporte pas (notamment Visual C++ 7 qui impose une définition à l'intérieur du patron).

```
template <typename E> template <typename T>
Entier<E>::Entier(const T & v) : _val(T(v)) {}
```

Néanmoins, cette syntaxe semble être le standard qui devrait apparaître prochainement dans tous les compilateurs (cf. la famille des GCC).

5. LA BIBLIOTHEQUE *STL*

La bibliothèque STL (*Standard Template Library*) fait actuellement partie intégrante du langage C++. Il s'agit d'un ensemble de structures de données (des conteneurs) et d'algorithmes suffisamment performants pour répondre aux besoins usuels des programmeurs. Afin de rendre ces composants génériques, ils sont proposés pour la plupart sous forme de patrons de classe ou de fonction.

Les conteneurs sont des classes génériques et les algorithmes des patrons de fonction. Afin de rendre les conteneurs interchangeables et rendre ainsi l'utilisation d'un algorithme possible sur tous, des objets intermédiaires, nommés *itérateurs*, sont manipulés. Les algorithmes sont également conçus pour être extensibles, grâce au concept de *foncteur*. La STL fournit aussi des *adapteurs*, qui sont des classes permettant l'adaptation d'un conteneur pour lui procurer de nouvelles fonctionnalités.

L'objectif de ce chapitre n'est pas de présenter tous les détails de la bibliothèque, mais simplement de fournir les notions nécessaires à l'utilisateur pour l'exploiter au mieux. En ce qui concerne des documentations complètes, voici quelques liens.

- **SGI's STL**
  Documentation et code source de la STL fournie par SGI. Entre autres, l'index est très utile pour accéder rapidement aux fonctionnalités d'un conteneur ou d'un algorithme.

- **STL Quick Reference**
  Un document concis et précis sur les principales fonctionnalités de la STL. Il s'agit d'un aide-mémoire qui répertorie les prototypes des méthodes des conteneurs et des principaux algorithmes.

- **Les conteneurs de la STL**
  Un diagramme UML qui vient compléter le document précédent, il décrit les relations et les principales fonctionnalités des conteneurs de la STL.

PACKAGE

Espace de nommage, *namespace*

La STL est fournie sous la forme d'un *package*, les composants sont rassemblés sous un même espace de nommage (ou *namespace*). Cela signifie que pour utiliser un composant, il faut précéder son nom de l'espace de nommage, en l'occurrence `std`. Par exemple, pour utiliser l'algorithme `find` du *package* `std`, il faut écrire `std::find`. Pour éviter d'écrire à chaque fois le nom du *package*, il est possible de l'intégrer dans l'espace de nommage courant par la commande suivante.

```
using namespace std;
```

Par défaut, l'espace de nommage courant n'a pas de nom, et écrire `find` ou `::find` est donc équivalent. Mais il est possible de définir ses propres espaces de nommage.

```
namespace mon_espace {
 using namespace std;

 class MaClasse { ... find(...) ... };
}
```

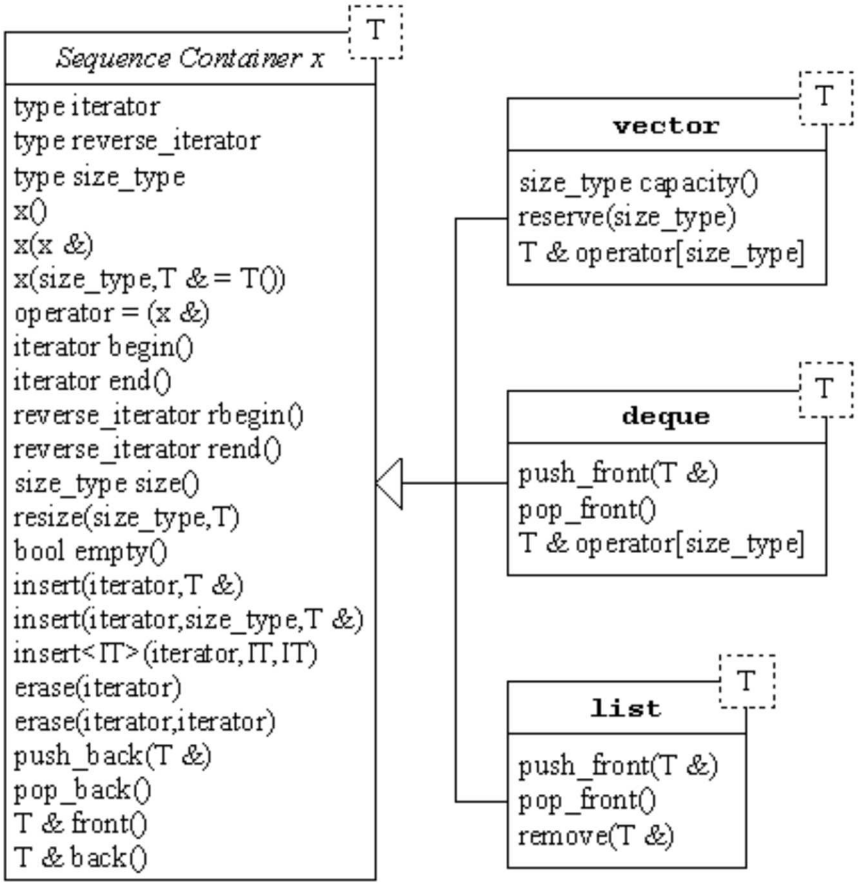
Dans l'espace de nommage `mon_espace`, la classe `MaClasse` est définie, et le *namespace* `std` est intégré à `mon_espace`. Cela signifie que de l'extérieur de l'espace de nommage, `std::find` et `mon_espace::find` sont équivalents. En revanche, dans l'espace de nommage `mon_espace`, `find` peut être appelée directement.

Inclusions

Les *headers* fournis par la STL n'ont pas d'extension. Par exemple, pour utiliser la fonction `std::find`, il faut inclure le fichier `<algorithm>`. L'espace de nommage `std` contient également toutes les fonctions et les types qui font partie du C standard. Par exemple, la fonction `strcpy` est devenue `std::strcpy`. En ce qui concerne les *headers* du C standard, leur nom est légèrement changé, l'extension est retirée et la lettre *c* est rajoutée en début de nom. Par exemple, en C, `strcpy` est déclarée dans `<string.h>`, alors qu'en C++, on inclura plutôt le fichier `<cstring>`.

CONTENEURS DE SEQUENCE

La bibliothèque STL propose différents types de conteneur. La première catégorie regroupe les *conteneurs de séquence*. Il s'agit de listes d'éléments dont l'ordre est entièrement contrôlé par l'utilisateur. Le schéma suivant montre les méthodes communes et les méthodes spécifiques aux trois types de conteneur de séquence: `vector`, `deque` et `list`. Une petite remarque concernant les notations: les classes en italique ne sont pas de véritables classes dans la STL, il ne s'agit que de classes abstraites utilisées pour montrer l'organisation des conteneurs. Dans la bibliothèque, les conteneurs sont simplement des classes génériques sans aucun lien explicite.



Outre les méthodes classiques telles que la recopie (par l'intermédiaire du constructeur de copie et de l'opérateur d'affectation), ces conteneurs proposent l'ajout (`push_back`) et la suppression (`pop_back`) d'un élément en fin de liste, et l'accès au premier élément (`front`) et au dernier élément (`back`). La méthode `size` permet de connaître le nombre d'éléments dans un conteneur, et

`resize` permet de redimensionner l'ensemble, soit en retirant des éléments (les derniers), soit en rajoutant des éléments (à la fin) à partir d'un modèle. La définition d'un conteneur est toujours présente dans le *header* du même nom, i.e. pour utiliser `vector`, il faut inclure le fichier `<vector>`.

vector

Le but du vecteur est de remplacer le tableau classique. Sa structure interne est un tableau qui est redimensionnable, automatiquement ou manuellement. A tout moment, il est possible d'ajouter un élément, il n'y a pas de limitation (hormis l'espace mémoire). Ainsi, si le tableau interne est plein au moment de l'ajout d'un nouvel élément, alors il est réalloué avec une taille plus importante et tous ses éléments sont recopiés dans le nouvel espace. La règle couramment employée consiste à doubler la taille chaque fois que le tableau est plein.

```
vector<int> v;

for (int i=0; i<10; ++i) v.push_back(i);
```

L'efficacité par rapport à un tableau est naturellement moindre, mais elle reste tout à fait acceptable (perte d'environ 10 %). En outre, il possible de contrôler à l'avance la taille du tableau interne (grâce à la méthode `reserve`), ce qui permet d'éviter la réallocation du tableau. Les performances sont alors équivalentes à celles d'un tableau.

```
vector<int> v;

v.reserve(10);
for (int i=0; i<10; ++i) v.push_back(i);
```

Il est aussi possible d'initialiser un vecteur rempli d'un certain nombre d'éléments. Ces derniers sont créés à partir du constructeur par défaut de leur classe.

```
vector<int> v(10);

for (int i=0; i<10; ++i) v[i]=i;
```

Il est possible d'accéder directement à un élément d'un vecteur à partir de son index, en utilisant l'opérateur `[]`. La numérotation des éléments débute à zéro, comme pour un tableau classique.

deque

L'intérêt de la *deque* (*double-ended queue*) par rapport au vecteur est de pouvoir ajouter (`push_front`) ou retirer (`pop_back`) un élément en début de liste. La structure interne de ce conteneur est un hybride entre une pile et une file d'attente. Les performances de la *deque* sont très bonnes pour l'ajout et la suppression d'éléments, que ce soit en tête ou en fin de liste. Elles surpassent même le vecteur au niveau de l'insertion, puisque les défauts liés à la réallocation sont évités.

list

La liste propose également un ajout et une suppression en début de liste. La structure interne du conteneur est une liste doublement chaînée. Son intérêt par rapport à la *deque* réside dans l'ajout et la suppression en milieu de liste, rendus possibles par l'intermédiaire des itérateurs. Grâce à la structure de liste chaînée, ces opérations sont très efficaces. Il existe également des fonctions spécifiques à la classe générique `list` telles que `slice` qui permet de scinder rapidement la liste.

ITERATEURS

Un *itérateur* est un pointeur, au sens objet, sur un élément d'un conteneur. Il s'agit d'un *design pattern* introduit par le GoF. En d'autres termes, c'est un objet fourni par un conteneur et qui pointe directement sur sa structure interne. Cela évite de dévoiler la structure interne à l'utilisateur, tout en fournissant un accès efficace aux éléments. Un itérateur sur un conteneur ne peut être créé que par le conteneur même. L'exemple suivant montre comment afficher le contenu d'un vecteur à l'aide d'itérateurs.

```
vector<int> v;
...
vector<int>::const_iterator courant = v.begin();
vector<int>::const_iterator fin = v.end();

while (courant!=fin) std::cout << *(courant++) << ' ';
```

Pour obtenir un itérateur, il faut utiliser les méthodes `begin` et `end` du conteneur qui retournent respectivement un itérateur sur le début de la liste et un itérateur sur l'élément fictif juste après la fin de la liste. Un itérateur dispose des opérateurs `++` pour avancer au prochain élément (attention, ne pas utiliser `--` pour reculer, ça ne marche pas toujours), et de l'opérateur `*` pour accéder à l'élément pointé. Les opérateurs `==` et `!=` sont également disponibles pour vérifier si deux itérateurs pointent au même endroit.

Le type des itérateurs est également fourni par le conteneur. Il dispose de types internes (e.g. `iterator`, `const_iterator`, `reverse_iterator`...) qui représentent différentes catégories d'itérateurs. Dans l'exemple, nous avons utilisé `const_iterator` qui représente un itérateur sans autorisation de modification de l'élément pointé. Dans l'exemple suivant, nous utilisons le type `iterator`, où la modification de l'élément pointé est autorisée.

```
vector<int> v;
...
vector<int>::iterator courant = v.end();
vector<int>::iterator fin = v.begin();

while (courant!=fin) *(courant++)*=2;
```

Dans cet exemple, tous les éléments du conteneur sont multipliés par 2. Un autre type intéressant d'itérateur, `reverse_iterator`, permet de parcourir la liste en sens inverse: `rbegin` retourne un itérateur sur la fin de la liste et `rend` retourne un itérateur sur l'élément fictif juste avant le début de la liste.

```
vector<int> v;
...
vector<int>::reverse_iterator courant = v.rbegin();
vector<int>::reverse_iterator fin = v.rend();

while (courant!=fin) std::cout << *(courant++) << ' ';
```

Mais quel est le véritable intérêt d'un itérateur ? On peut penser que disposer de méthodes de parcours directement dans le conteneur serait suffisant. Mais on peut instancier autant d'itérateurs qu'on souhaite pour un conteneur donné, et initier ainsi plusieurs parcours sur la structure (sans en connaître les détails), ce qui ne serait pas forcément évident si les méthodes d'accès étaient directement fournies par le conteneur. Il est également important de rendre les conteneurs interchangeables. Considérons la fonction suivante qui trie par ordre croissant tous les éléments du vecteur fourni en argument.

```
template <typename T> void trier(vector<T> & v) {
 int i = 0;
 int j;
```

```
 while (i<v.size()) {
  j=i+1;

  while (j<v.size()) {
   if (v[j]<v[i]) std::swap(v[i],v[j]);
   ++j;
  }

  ++i;
 }
}
```

Ce serait dommage de devoir réécrire cette fonction pour un autre type de conteneur. Donc, au lieu d'utiliser les indices pour accéder aux éléments (ce qui est propre au vecteur et à la *deque*), nous utilisons des itérateurs.

```
template <typename T> void trier(vector<T> & v) {
 typename vector<T>::iterator fin = v.end();
 typename vector<T>::iterator i = v.begin();
 typename vector<T>::iterator j;

 while (i!=fin) {
  j=i;
  ++j;

  while (j!=fin) {
   if (*j<*i) std::swap(*i,*j);
   ++j;
  }

  ++i;
 }
}
```

Vous remarquerez que pour utiliser le type interne `iterator` de `vector<T>`, il est précédé du mot-clé `typename`. La raison est simple, lorsqu'on dispose d'un paramètre `T` dans un patron, rien ne garantit qu'il possède un membre donné. En ce qui concerne les attributs et les méthodes, le compilateur fait confiance à l'utilisateur à la première vérification du *template*. Il effectuera le véritable test au moment de l'instanciation du patron. En ce qui concerne les types internes, c'est un peu différent, le compilateur prévient le programmeur par un message indiquant qu'il ne connaît pas le type. Ainsi, pour le code `T::iterator`, le compilateur demande confirmation par l'ajout du mot-clé `typename`. Dans le cas de `vector<T>::iterator`, le compilateur considère `vector<T>` aussi inconnu que `T`, même si le patron `vector` est connu (on ne sait jamais, il est possible à tout moment de définir une instanciation partielle de `vector`).

Dans la fonction précédente, il n'existe plus de code propre au vecteur, exceptée l'initialisation des itérateurs de début et de fin qui pourraient être fournis comme argument.

```
template <typename I>
void trier(const I & debut,const I & fin) {
 I i = debut;
 I j;

 while (i!=fin) {
  j=i;
  ++j;

  while (j!=fin) {
   if (*j<*i) std::swap(*i,*j);
   ++j;
  }

  ++i;
```

```
 }
}
```

La fonction est alors utilisable pour tout type de conteneur pour lequel il est possible de fournir des itérateurs. Ces derniers ont donc un rôle d'intermédiaires entre un algorithme et le conteneur auquel ils sont associés. Quelque soit leur conteneur, ils implémentent tous la même interface et sont donc totalement interchangeables.

Les itérateurs peuvent également être utilisés pour insérer un élément en milieu de liste. Les conteneurs de séquence disposent de la méthode `insert` qui permet d'insérer un élément à la position précédant celle pointée par l'itérateur fourni. Ainsi, `v.insert(v.begin(),10)` insère l'élément en début de liste et `v.insert(v.end(),10)` l'insère en fin de liste.

ALGORITHMES

Les algorithmes sont élaborés sur le même modèle que notre dernier exemple. C'est-à-dire qu'au lieu de passer le conteneur en argument, il faut fournir des itérateurs pour délimiter la liste d'éléments. Les algorithmes sont des patrons de fonction dans la STL et sont définis dans le *header* `<algorithm>`. Par exemple, il existe le patron `sort` qui trie les éléments d'un conteneur.

```
std::sort(v.begin(),v.end());
```

Nous n'allons pas détailler ici tous les algorithmes, nous vous invitons à consulter la documentation pour plus d'informations. Nous signalons juste la fonction `find`, qui recherche dans une liste, identifiée par un iterateur de début et de fin, un élément donné, et retourne alors comme résultat un itérateur sur cet élément.

```
vector<int>::const_iterator i = std::find(v.begin(),v.end(),28);
```

FONCTEURS

Il est possible d'étendre certains algorithmes de la STL. La méthode employée repose sur le concept de *foncteur* qui est une adaptation d'un *design pattern* du GoF appelé *visiteur*. Pour illustrer ce concept, revenons à notre exemple de fonction de tri. Nous souhaitons maintenant pouvoir paramétrer l'ordre du tri. En analysant la fonction, on s'aperçoit que l'ordre est simplement déterminé par le test qui vérifie qu'un élément doit se situer avant un autre. Pour rendre l'algorithme plus générique, il suffit de déporter ce test dans une fonction. Celle-ci sera alors passée en paramètre (i.e. en tant que pointeur de fonction) à la fonction de tri, et pourra alors être remplacée pour changer l'ordre du tri.

```
template <typename T,typename I>
void trier(const I & debut,const I & fin,
           bool (*estAvant)(const T &,const T &)) {
 I i = debut;
 I j;

 while (i!=fin) {
  j=i;
  ++j;

  while (j!=fin) {
   if (estAvant(*j,*i)) std::swap(*i,*j);
   ++j;
  }

  ++i;
 }
```

```
}
```

Pour définir par exemple un tri croissant, il suffit d'écrire une fonction qui vérifie qu'un élément est inférieur à un autre.

```
template <typename T>
bool estAvantCroissant(const T & a,const T & b) { return a<b; }

...

trier(v.begin(),v.end(),estAvantCroissant);
```

Ce genre de fonction peut être considérée comme un objet, c'est ce qu'on appelle un *visiteur*. Il s'agit d'un objet qui possède des méthodes indispensables pour compléter un algorithme, ce qui permet alors d'en modifier son comportement. Ainsi, la fonction `estAvantCroissant` peut être remplacée par la classe `OrdreCroissant`.

```
template <typename I,typename O>
void trier(const I & debut,const I & fin,const O & ordre) {
 I i = debut;
 I j;

 while (i!=fin) {
  j=i;
  ++j;

  while (j!=fin) {
   if (ordre.estAvant(*j,*i)) std::swap(*i,*j);
   ++j;
  }

  ++i;
 }
}

template <typename T> class OrdreCroissant {
 public:
  bool estAvant(const T & a,const T & b) const { return a<b; }
};

...

trier(v.begin(),v.end(),OrdreCroissant<int>());
```

Enfin, notamment pour éviter à l'utilisateur de retenir le nom des méthodes des visiteurs à chaque nouvel algorithme, la STL propose le *foncteur* qui est un visiteur possédant l'opérateur `()`. Ainsi, lorsque la méthode est appelée, le visiteur ressemble syntaxiquement à une fonction.

```
template <typename I,typename O>
void trier(const I & debut,const I & fin,const O & estAvant) {
 I i = debut;
 I j;

 while (i!=fin) {
  j=;
  ++j;

  while (j!=fin) {
   if (estAvant(*j,*i)) std::swap(*i,*j);
   ++j;
  }

  ++i;
 }
```

```
}

template <typename T> class OrdreCroissant {
 public:
  bool operator()(const T & a,const T & b) const { return a<b; }
};
```

La STL propose une fonction de tri, `sort`, qui possède le même prototype que notre dernière fonction `trier`. Pour trier un vecteur `v` dans l'ordre croissant, il suffit d'exécuter le code suivant.

```
std::sort(v.begin(),v.end(),OrdreCroissant<int>());
```

Si aucun foncteur n'est passé en paramètre, le foncteur `less` (équivalent à `OrdreCroissant`) est utilisé. En général, tous les algorithmes et les conteneurs de la STL qui nécessitent un foncteur pour une relation d'ordre utilisent par défaut `less`. Ainsi, le simple code suivant est possible.

```
std::sort(v.begin(),v.end())
```

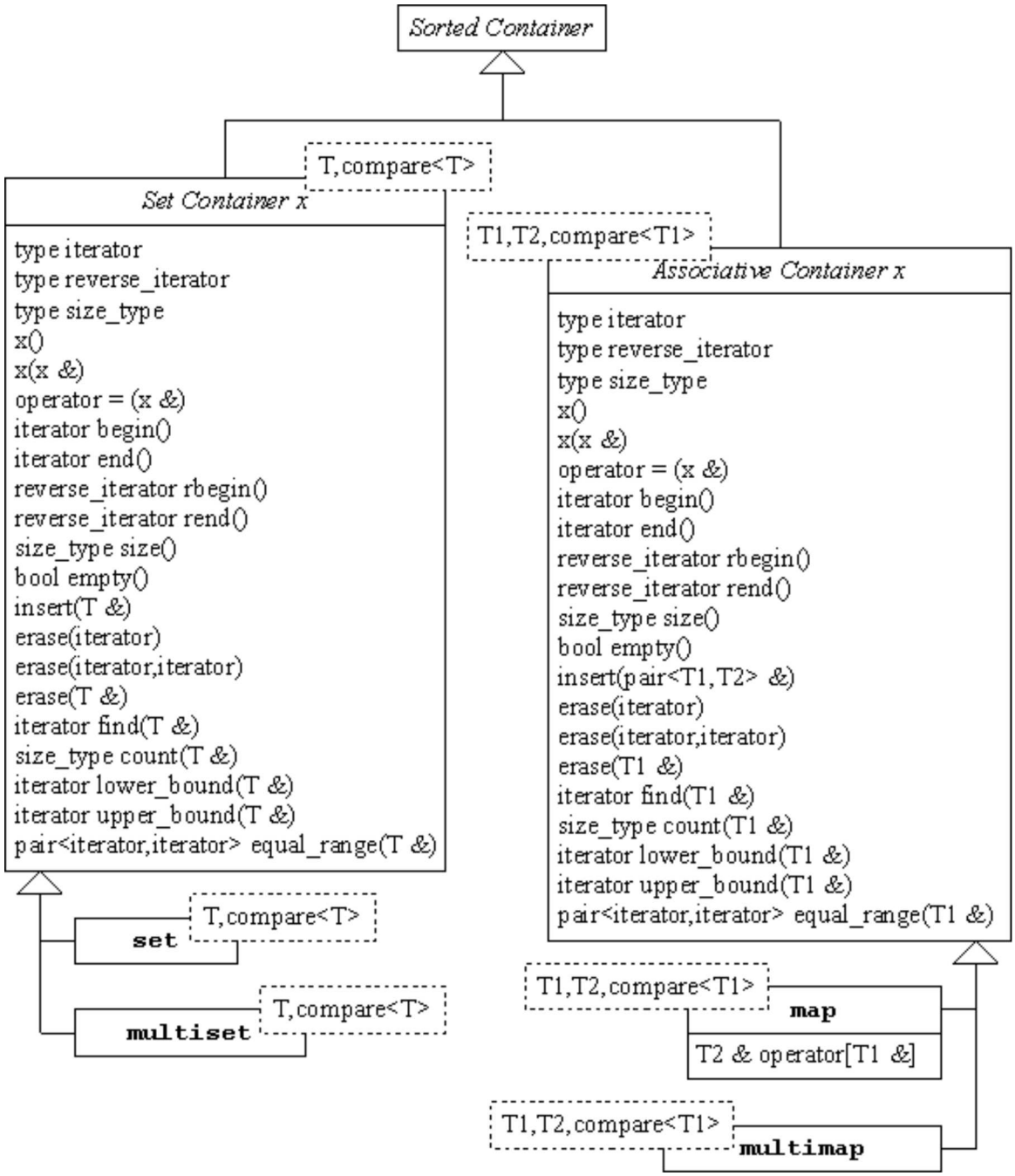
L'intérêt majeur des visiteurs réside dans leur capacité à posséder leurs propres données, ce qui n'est pas concevable avec des pointeurs de fonction. Considérons par exemple l'algorithme `generate` fourni par la STL. Il parcourt un conteneur (par l'intermédiaire d'itérateurs) et remplace chaque élément par une valeur fournie par un foncteur. Supposons qu'on souhaite remplir le conteneur de nombres pairs.

```
class GenerateurNombrePair {
 protected:
  int _x;

 public:
  GenerateurNombrePair(int x) : _x(x/2*2) {}

  int operator()(void) {
   int y = _x;

   _x*=2;
   return y;
  }
};

...

vector<int> v(10);

std::generate(v.begin(),v.end(),GenerateurNombrePair(8));
```

L'exemple précédent remplit le vecteur `v` d'une séquence de 10 nombres pairs, à partir de la valeur 8. L'avantage du visiteur par rapport à un pointeur de fonction est ici d'avoir une mémoire. En effet, à chaque appel, le générateur retourne la valeur suivante dans la séquence.

CONTENEURS TRIES

La seconde catégorie de conteneurs proposée dans la STL regroupe les structures de données triées. Le schéma suivant montre les méthodes communes et les méthodes spécifiques aux deux types de conteneur trié: les ensembles (i.e. `set` et `multiset`) et les conteneurs associatifs (i.e. `map` et `multimap`).



Notez que pour tous ces conteneurs, il est nécessaire de fournir un foncteur qui permet de trier les éléments. Par défaut, le foncteur `less` est utilisé. Les itérateurs sont bien sûr disponibles pour ces conteneurs.

Ensembles, *set* et *multiset*

La classe `set` représente un ensemble trié où les éléments sont uniques. Pour vérifier que deux éléments sont différents, il suffit d'utiliser la relation d'ordre fournie par un foncteur (`compare<T>` sur le schéma): `a!=b` si et seulement si `a<b` ou `b<a`. La classe `multiset` autorise en revanche les doublons.

Pour ajouter un élément, on utilise la méthode `insert` qui garantit, dans le cas d'un `set`, l'unicité de la valeur. Pour supprimer un élément, on utilise la méthode `erase`, en fournissant soit directement la valeur à retirer, soit un itérateur pointant sur l'élément à supprimer. Une méthode `find` est également disponible pour rechercher un élément, elle retourne un itérateur sur l'élément. Comme l'ensemble est trié, la recherche s'effectue relativement rapidement. La structure interne du conteneur est un arbre binaire de recherche.

Conteneurs associatifs, *map* et *multimap*

Les classes `map` et `multimap` sont des conteneurs dits *associatifs*, c'est-à-dire qu'ils associent une clé à chaque élément. Les données sont donc stockées par paire *(clé;élément)* dans le conteneur, et sont triées par rapport à la clé. Pour le conteneur `map`, il est possible d'accéder directement à un élément à partir de sa clé, grâce à l'opérateur `[]`. Mais attention, si la clé n'est pas présente dans le conteneur, une paire avec cette clé est automatiquement créée et ajoutée au conteneur, l'élément associé est construit par défaut.

Le conteneur `map` (respectivement le conteneur `multimap`) est en fait un conteneur `set` (respectivement un conteneur `multiset`) où les éléments sont des paires *(clé,élément)*. La STL fournit une structure pour représenter une paire dont voici la définition.

```
template <typename T1,typename T2> struct pair {
 T1 first;
 T2 second;

 pair(const T1 & a,const T2 & b) : first(a), second(b) {}
};
```

Ainsi, pour insérer un élément dans un conteneur `map`, il faut d'abord créer une paire pour l'associer à sa clé, et ensuite fournir cette paire à la méthode `insert`.

```
std::map<int,char *> m;

char * chiffres[] = { "zero", "un", "deux", "trois", "quatre",
                      "cinq", "six", "sept", "huit", "neuf" };

for (int i=0; i<10; ++i) {
 m.insert(std::pair<int,char *>(i,chiffres[i]));
 ++i;
}

std::cout << "5 s'ecrit: " << m[5] << std::endl;
```

Pour insérer une paire, il faut le plus souvent la créer à la volée en appelant son constructeur. Mais il faut tout de même préciser tous les paramètres du *template* `Pair` pour réussir l'instanciation. Afin d'éviter de préciser chaque fois ces paramètres, on utilise le patron de fonction `make_pair` qui crée une paire à partir de deux éléments. Voici tout d'abord la définition de ce patron.

```
template <typename T1,typename T2>
inline std::pair<T1,T2> make_pair(const T1 & a,const T2 & b)
{ return std::pair<T1,T2>(a,b); }
```

L'intérêt ici, c'est d'utiliser le polymorphisme statique fourni pour les patrons de fonction, comme le montre l'exemple suivant. Plus besoin de préciser les paramètres au *template* !

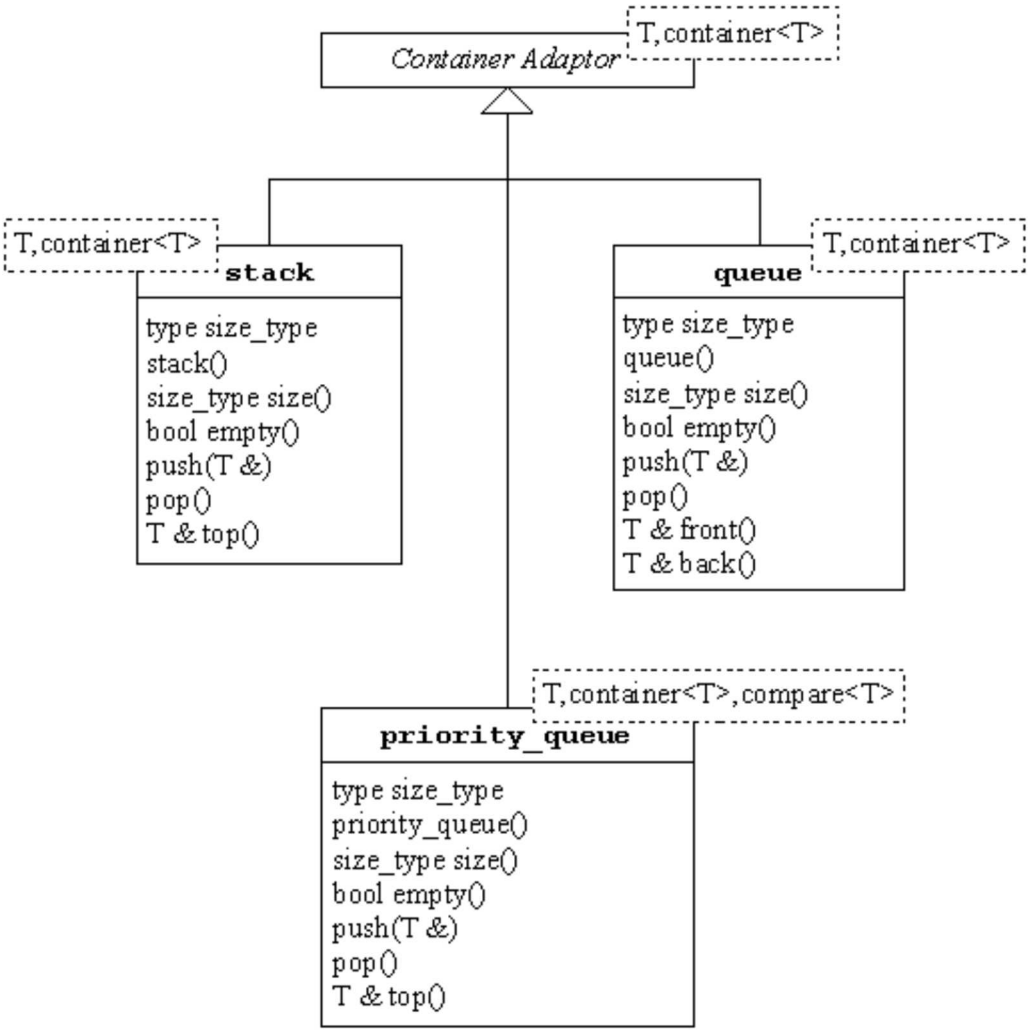
```
for (int i=0; i<10; ++i) {
 m.insert(make_pair(i,chiffres[i]));
 ++i;
}
```

ADAPTEURS

Lorsqu'une classe ne convient pas totalement à une utilisation donnée, il est possible de l'adapter pour modifier ses fonctionnalités. Dans ce but, un *design pattern* nommé *adapteur* est proposé par le GoF. La bibliothèque STL a emprunté le nom *adapteur* pour désigner des classes permettant d'adapter le comportement des conteneurs de séquence. Trois classes sont proposées:

- `stack`, pour adapter le conteneur au comportement d'une pile,
- `queue`, pour adapter le conteneur au comportement d'une file d'attente,
- `priority_queue`, pour adapter le conteneur au comportement d'une file d'attente à priorité.

Il faut bien comprendre que ces classes ne sont que des surcouches (on dit aussi *wrappers*) aux classes `vector`, `deque` et `list`. Pour créer un adapteur, il faut donc préciser le type de conteneur qu'on souhaite adapter (comme le montre le schéma suivant).



Dans le cas particulier de la file d'attente à priorité, il faut également fournir un foncteur qui permette de comparer les éléments, le plus grand étant le plus prioritaire. Ce foncteur repose sur le même modèle que celui de la fonction `sort` par exemple, et par défaut, le foncteur `less` est encore une fois utilisé. Voici un exemple d'utilisation de cette file à priorité.

```
std::priority_queue<int,vector<int>,less<int> > q;

for (int i=0; i<10; ++i) q.push(std::rand());

for (int i=0; i<10; ++i) {
 std::cout << q.top() << std::endl;
 q.pop();
}
```

Des entiers sont ajoutés aléatoirement dans la file à priorité. Ils sont ensuite affichés du plus grand au plus petit.

COMPLEXITE

Nous n'avons absolument pas présenté ici la complexité des algorithmes et des opérations liées aux conteneurs. Ces informations nous semblent peu pertinentes ici, dans la mesure où nous ne proposons qu'un aperçu rapide de la bibliothèque. Mais nous vous invitons à consulter la documentation avant d'utiliser un conteneur ou un algorithme. La connaissance de la complexité d'une opération est fondamentale dans le choix du composant le mieux adapté à une situation donnée. La documentation fournie par SGI (cf. l'introduction de ce chapitre) fournit tous ces renseignements.