Introduction au C++ et à la programmation objet

Version 1.6
(Novembre 2004)

Table des matières

Introduction

*Ce cours fait partie du cursus du **D.E.S.S. de bioinformatique** de l'Université Paul Sabatier*

Avertissement

Vous pouvez télécharger l'ensemble des fichiers constituant ce cours. Vous pourrez alors l'installer sur votre machine personnelle et le lire avec tout navigateur. Il vous suffira d'ouvrir le fichier `index.html`
Vous pouvez aussi installer ce cours sur votre serveur web, mais dans ce cas merci de commencer par lire le fichier LISEZ-MOI, et surtout de *ne rien modifier* sans mon accord.
Si vous trouvez des inexactitudes, ou si vous avez des idées d'améliorations, merci de me contacter.

Symboles et conventions utilisés dans ce cours

 Au boulot !!! Cliquez sur l'équerre et vous aurez quelques sujets d'exercices. Parmi eux, certains sont corrigés. Dans ce cas, promis-juré, j'ai réussi à compiler et à exécuter le programme, au moins avec gcc.

 Cliquez là-dessus pour accéder à une page destinée aux terriens normaux, qui vous donnera des explications plus simples à comprendre, en particulier grâce à des analogies avec le monde réel.
Il y a des exemples de code dans le cours. Je ne garantis pas qu'ils fonctionnent, tout simplement parce qu'ils sont pour la plupart incomplets, ou sont du "pseudo-code" plutôt que du code réel. Regardez les exercices si vous voulez du code qui fonctionne pour de vrai.
Peut-être voudrez-vous mettre en pratique les notions expliquées, sans obligatoirement regarder le corrigé des exercices. Dans ce cas, il vous faut savoir écrire un programme C++ minimum, par exemple celui qui écrit `hello world` sur l'écran.

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

main() {
    cout << "Hello world" << endl;
    cerr << "Hello buggy world" << endl;
};
```

Il s'agit d'instructions d'entrées-sorties. La première écrit sur la sortie standard, la seconde sur l'erreur standard. Les deux premières lignes vous permettent d'utiliser la bibliothèque standard. Prenez-les pour l'instant comme des formules magiques, ce n'est qu'à la fin du cours que vous comprendrez réellement de quoi il retourne

 Le C++ est un langage très simple à apprendre (mais si...) mais il y a tout-de-même quelques difficultés. Celles-ci sont indiquées comme ce paragraphe-ci.

 Certaines règles, dites "règles d'or", sont plutôt des règles de bon usage que des règles imposées par le compilateur. Elles sont indiquées comme ceci.

 Et chez les autres, comment ça se dit ? Derrière cette icône sont regroupées quelques comparaisons avec d'autres langages objets: certaines notions se retrouvent (mais se disent différemment), alors que d'autres notions sont absentes de certains langages.

Impasses...

J'ai volontairement fait l'impasse sur plusieurs aspects du langage:

Fichiers sources
: Le découpage en fichiers `.H` et `.C` n'est pas abordé.

Fonctions `inline`
: Les fonctions peuvent être déclarées `inline` pour améliroer la rapidité d'exécution; ce point n'est pas abordé en détails.

Héritage privé ou protégé
: L'héritage privé ou protégé n'est pas abordé.

Héritage multiple
: L'héritage multiple est à peine abordé.

R.T.T.I.
: Run Time Type Identification. Permet de déterminer quel est le type d'une variable. Dans de rares cas cela est indispensable, mais la plupart du tempsdes mécanismes basés sur l'héritage seront préférables.

stdlib
: La bibliothèque standard est présentée... un peu trop rapidement, et certainement de manière incomplète.

Le C++ est-il un langage trop compliqué ?

Les blagues suivantes sont tirées de www.rigoler.com

Combien de développeurs ça prend pour changer une ampoule?
: L'ampoule fonctionne correctement sur mon PC.

Combien de bêta testeurs ça prend pour changer une ampoule?
: Nous venons tout juste de constater que la pièce est dans l'obscurité; Pour l'instant, le problème n'a pas été résolu.

Combien de consultants ça prend pour changer une ampoule?
: On ne sait pas. Il n'y a pas encore eu d'étude de faisabilité

Combien de programmeurs ça prend pour changer une ampoule?
: Aucun, c'est un problème de hardware

mais surtout celle-ci...

Combien faut-il de programmeurs C++ pour changer une ampoule ?
: 6. Un pour la changer et 5 autres six mois plus tard pour comprendre comment il a fait.

Une autre blague (mais est-ce vraiment une blague...), rapportée par Stroustrup sur sa page d'accueil:

> "Did you really give an interview to IEEE... in which you confessed that C++ was deliberately created as an awful language for writing unmaintainable code to increase programmers' salaries?"

et la réponse de Stroustrup:

> "Of course not."

Faut-il plutôt travailler en java ?

A la question "java est-il le langage que vous auriez aimé créer si vous n'aviez pas dû assurer la compatibilité avec le langage C", Stroustrup répond [Stroustrup-faq]:

> "Much of the relative simplicity of Java is - like for most new languages - partly an illusion and partly a function of its incompleteness. As time passes, Java will grow significantly in size and complexity. It will double or triple in size and grow implementation-dependent extensions or libraries. That is the way every commercially successful language has developed. Just look at any language you consider successful on a large scale. I know of no exceptions, and there are good reasons for this phenomenon".

Est-il possible d'écrire de "gros" codes sans erreurs ?

Les paragraphes qui précèdent montrent bien l'atmosphère souvent polémique qui accompagne la naissance de nouveaux outils. Peut-être le C++ est-il complexe dans sa syntaxe, peut-être la notion de programmation objet n'est-elle pas simple à maîtriser, mais cette complexité est aussi le prix à payer pour utiliser des outils suffisamment puissants pour générer des applications elles aussi puissantes.
On considère ([Stroustrup-lang]) qu'il est toujours possible d'écrire un programme "qui marche" , même mal structuré, à condition que ce programme soit assez petit (moins de 1000 lignes de code).
Dès que le programme grossit (10000 lignes), il devient impossible de le maîtriser correctement s'il n'est pas structuré de manière rigoureuse. A chaque correction de bogue, de nouvelles bogues apparaîtront...
D'après Stroustrup, il est réaliste qu'un programmeur maîtrise un programme C++ de 25000 lignes... d'autre part, de part sa conception, le C++ facilite le travail en équipe, en permettant à plusieurs programmeurs de collaborer à la réalisation d'un gros projet.
On peut donc considérer que l'important, dans le C++, n'est pas tant la syntaxe en elle-même, que le paradigme objet sous-tendu par le langage. Autrement dit, l'important est de bien comprendre comment les données sont "vues" par le langage... et donc par le programmeur.

Un peu d'histoire...

Le C++ est apparu durant les années 80; il s'agit d'une création de Bjarne Stroustrup, un informaticien qui travaillait chez ATT sur de gros projets d'informatique distribuée. Quelques dates:

- 1965-1970 Le langage BCPL est largement utilisé pour la programmation système, en particulier chez ATT
- 1969-1972 Naissance du langage B, une évolution de BCPL, puis du langage C, lui-même une évolution de B
- 1978 The C Programming Language par Richie et Kernighan, ce livre a tenu lieu de manuel de référence du langage pendant plusieurs années.
- 1979 Bjorne Stroustrup commence à travailler sur un nouveau C ("C with Classes")
- 1983 La première version du C++ est utilisée en interne chez ATT
- 1983 Mise en place de la commission de normalisation du C par l'ANSI
- 1985 Première implémentation commerciale du C++
- 1985 Normalisation du C ("C Ansi")
- 1989 Premiers travaux de normalisation du C++
- Nov 1997 - Approbation du nouveau standard (Ansi C++) par le comité de normalisation

Pourquoi tous ces langages ?

Thompson, Khernigan et Richie travaillaient sur un projet de système d'exploitation (qui s'appellera un peu plus tard Unix...), en utilisant un mini-ordinateur Digital (en l'occurrence le Pdp 7): une machine, même pour l'époque, très limitée en puissance CPU, mais surtout en mémoire (8 Kilo-mots de 18 bits...). Venant d'autres projets dans lesquels ils avaient utilisé des langages de haut niveau, ils se refusaient à utiliser l'assembleur, trop lourd et difficile à coder. D'où leur intérêt pour des langages légers, permettant de faire à peu près la même chose que l'assembleur mais avec un code plus lisible et plus rapide à écrire.

Qu'est-ce qui les différencie ?

La syntaxe évolue bien entendu d'un langage à l'autre, mais comme ceux-ci sont fortement apparentés entre eux, les différences de syntaxe sont au fond mineures. Par contre, la manière dont les données sont traitées va beaucoup évoluer au cours du temps; BCPL et B sont des languages non typés: cela signifie que les données seront toutes rassemblées dans des tableaux de "mots", encore appelés "cellules". La taille de chaque cellule est tout simplement la taille de cellule élémentaire utilisée sur la machine (18 bits). Normal, au fond, pour un langage destiné à remplacer l'assembleur, tout en gardant ses principales caractéristiques. Cela a cependant deux conséquences importantes:

- La notion de pointeur ne présente aucune difficulté à implémenter
- La correspondance forte entre pointeurs et tableau est parfaitement naturelle.

En 1970, un nouvel ordinateur (DEC pdp 11) est acquis par l'équipe: dès lors, la taille des mots n'est plus 18 bits, mais 16 bits, et l'adressage se fait par octets. Le langage B est rapidement recodé pour fonctionner sur la nouvelle machine, mais de gros problèmes se posent alors:

- Les opérateurs mettant en jeu des pointeurs, conçus pour des machines sur 18 bits, deviennent très inefficaces avec la nouvelle machine.
- Le pdp 11 étant équipé de coprocesseur flottant, comment en tirer parti avec un langage ne connaissant que des données sur un mot, alors que 16 bits est insuffisant pour représenter un nombre réel ??? (18 bits auraient été suffisants...)

La seule manière de s'en sortir est alors de doter le langage de variables de différents *types*. Le nouveau langage, appelé C, inclut donc la notion de types: `int, char, float`, puis le type `struct` fait son apparition un peu plus tard.

Langage proche de la machine ou abstraction de données ?

On voit donc bien l'évolution, qui s'est poursuivie jusqu'au C++: on part d'une situation où le langage n'est au fond qu'un "super-assembleur", pour gagner petit à petit en abstraction, (type `struct` en particulier), ce qui permettra au langage d'être utilisé pour un usage plus général. Toutefois, on reste toujours "proche de la machine", de sorte que le C peut dans la plupart des cas remplacer totalement l'assembleur, même pour développer du code proche du hardware (pilotes de périphériques, par exemple).
Cette évolution a été poursuivie par Stroustrup lors de la conception du C++: le type `class`, qui est en quelque sorte l'aboutissement du type `struct`, permet d'implémenter un très haut niveau d'abstraction des données; en même temps, le C++ étant un surensemble du C, il reste un langage de choix pour l'informatique système.

La programmation objet

Poser un problème

Avant de se précipiter sur son clavier pour écrire un programme, il convient de réfléchir afin de poser correctement le problème... Or, la manière même dont le problème sera posé influe sur l'écriture du programme. D'où la notion de "paradigme de programmation". Or, s'il est possible d'implémenter tous les paradigmes en utilisant n'importe quel langage, cela sera plus ou moins facile selon le langage utilisé. Ainsi, il est possible de programmer en objet en utilisant le C... mais le C++, conçu dans cet objectif, **supporte** la programmation objet, par sa syntaxe d'une part, par les contrôles apportés tant au moment de la compilation que lors de l'exécution d'autre part.

La programmation objets expliquée aux programmeurs

Si vous êtes programmeur, mais habitué aux langages de programmation "procéduraux" (`pascal`, `fortran`, `C`, `perl`, etc.), ce chapitre est pour vous: il essaie d'expliquer comment on peut passer de la programmation procédurale à la programmation objet, via la programmation structurée.

 Mais si vous êtes débutant en programmation, vous êtes encore des "gens normaux", dans ce cas vous pouvez passer directement au chapitre suivant.

L'approche procédurale

Elle met l'accent sur l'action représentée par le programme: on doit "faire quelque chose", mais cette chose sera exécutée par étapes successives. Chaque étape elle-même peut être découpée. On arrive ainsi à des découpages de plus en plus fins, jusqu'à obtenir des fonctions élémentaires. Certaines de ces fonctions peuvent figurer dans des bibliothèques, cela permettra de les réutiliser plus tard pour d'autres projets.

Qu'est-ce qu'une fonction ?

Une fonction est un sous-programme caractérisé par:

- Son nom
- Des données en entrée (les paramètres)
- Une valeur de retour
- Une action sur le système, représentée par un algorithme

Variables locales ou globales

Les variables peuvent être locales, mais aussi globales: si `X` est une variable globale, une fonction `f1` peut modifier la valeur de `X`, mais une autre fonction `f2` peut également modifier cette valeur. Un programme bien structuré aura le moins possible de variables globales... mais celles-ci ne pourront pas être totalement évitées.

L'approche modulaire

Supposons que dans notre programme, deux fonctions `f1`, `f2` et `f3` accèdent toutes deux aux variables globales `A` et `B`, mais *ce sont les seules.* Dans ces conditions, peut-on vraiment dire que `A` et `B` sont des variables globales ?
Il est tentant de regrouper ces fonctions et ces variables: c'est la notion de module. Nous pouvons regrouper les trois fonctions `f1`, `f2` et `f3` d'une part, les deux variables `A` et `B` d'autre part, dans un module qui est constitué de deux parties:

- Une *interface* qui spécifie exactement les fonctions qui seront visibles depuis l'extérieur.
- Une *implémentation* qui contient des variables d'une part, du code d'autre part.

Interface publique...

Le module comporte une interface, c'est-à-dire un ensemble de fonctions (et éventuellement de variables), qui seules seront vues par l'utilisateur du module. Un soin particulier doit être apporté à l'écriture de l'interface, puisque la modification de celui-ci pourra avoir des conséquences sur le code utilisateur du module. La seule modification de type d'une variable de fonction, par exemple, peut entraîner une impossibilité de compilation de l'application.

...Implémentation cachée

Les algorithmes constituant le corps des fonctions seront cachés, en ce sens qu'ils ne seront pas visibles par les utilisateurs du module. En conséquence, il est possible de les modifier, par exemple pour améliorer leur performance ou pour corriger une erreur, sans que cela ait d'impact sur le reste du programme... à condition toutefois que l'interface reste inchangée (ou tout au moins qu'il y ait compatibilité entre l'ancienne et la nouvelle interface). Il est même possible de modifier le découpage en fonctions du module, en ajoutant ou en supprimant des fonctions: tant qu'on ne touche pas aux fonctions déclarées dans l'interface, pas de problème par rapport à l'extérieur.

Encapsulation des données

Les variables `A` et `B` étant cachées, on peut modifier leur type tout en limitant l'impact sur l'ensemble du programme. D'autre part, grâce à l'encapsulation, et en supposant que la fonction `f4` n'est pas intégrée à ce module, on est sûr d'éviter le bogue suivant (pas toujours très simple à détecter):

```
void f4 (){
   int A1;
   A=0; /* ERREUR, on voulait écrire A1=0 */
}
```

Si `A` avait été une variable globale, la ligne `A=0`, qui est une erreur du point-de-vue du programmeur, n'aurait pas été signalée par le compilateur, puisque `A` est accessible. Si `A` est "cachée" (encapsulée dans un module), le compilateur détectera une erreur: il sera alors aisé de la corriger.

Prototypage

Le mécanisme d'encapsulation des données, lorsqu'il est supporté par le langage de programmation, permet de travailler aisément en équipe sur un même projet: chaque programmeur écrit un module différent: il faut que tout le monde soit d'accord sur l'interface de chaque module, mais le codage lui-même peut se faire par chacun de manière indépendante.
Il est également possible de travailler par prototypes: lors de la phase de prototypage, l'interface est écrit mais le code est incomplet. Cela permet toutefois d'utiliser l'interface du module dans d'autres parties de l'application, et ainsi de tester la cohérence du modèle.

L'approche objets

Il est possible de dépasser l'approche modulaire. Supposons que l'on veuille, dans un programme, définir une structure de pile de caractères. On pourra écrire un module, avec les deux fonctions interfaces suivantes:

```
char pop();
void push (char);
```

On peut dire qu'un tel module est un "archéo-objet". Mais nous aimerions répondre aux trois questions suivantes:

1. Comment faire si nous avons besoin de plusieurs piles dans notre programme ?
2. Comment classer nos objets par "familles d'objets" ?
3. Comment faire si nous avons besoin de piles d'entiers ?

Des objets intégrés au système de typage

Donner un numéro d'identification aux piles:

On ajoute un paramètre aux fonctions `pop` et `push` définies ci-dessus, en l'occurence un numéro d'identification. Dès lors, on peut gérer autant de piles que l'on veut. Les fonctions interfaces deviennent:

```
char pop(int Id);
void push(int Id, char c);
```

La solution du C++

Le C++ apporte une solution beaucoup plus puissante: il permet au programmeur de définir un *"un type de données qui se comporte* [presque] *de la même manière qu'un type prédéfini"*.

Le module devient alors tout simplement une déclaration de type, on peut déclarer des variables de ce type; Les fonctions sont "attachées" à ces variables, de sorte qu'on écrira dans le code des lignes du style:

```
class stack {
   char pop();
   void push(char c);
}

stack A;
stack B;
stack C;

A.pop();
C.push(B.pop());
```

Opérations sur les objets

La phrase "type de données qui se comporte presque de la même manière qu'un type prédéfini" entraîne de nombreuses conséquences. Par exemple, on pourra déclarer un tableau d'objets de type pile de la manière suivante:

```
stack[10] Stacks;
```

Une variable pourra être initialisée:

```
stack S1=10;
```

On pourra recopier une variable dans une autre grâce à l'opérateur d'affectation:

```
stack A;
stack B;
...
B=A;
```

On pourra faire un transtypage (cast) d'un type dans un autre:

```
stack A;
int B;
...
B = (int) A;
```

On pourra même additionner deux piles avec l'opérateur +, les comparer avec >, etc. Le problème est bien sûr: quelle signification donner à ces opérations ? Si l'initialisation ne pose pas trop de problème, si l'affectation semble également évidente, que signifie additionner ou comparer deux piles ? Ces opérateurs étant définis par la personne qui définit l'objet, c'est également à elle de définir la signification précise des opérateurs du langage pour cet objet. C'est ce qu'on appelle la *surcharge des opérateurs*. Il n'est pas obligatoire de surcharger les opérateurs: si, dans l'exemple précédent, l'opérateur + n'est pas surchargé, l'opération `A + B` renverra tout simplement une erreur à la compilation.

Classer les objets

Nous avons maintenant à notre disposition autant de types de variables que nous voulons. Nous allons avoir rapidement besoin de définir une classification.

Un exemple tiré de la vie quotidienne

En effet, si nous prenons une comparaison avec la vie quotidienne, nous pouvons dire que nous avons à notre disposition: un couteau de cuisine, une Twingo, un tourne-vis, un couteau à beurre, une 205, un couteau à pain... et un raton laveur. On sent bien que nous aimerions écrire que nous avons des outils et des voitures; en l'occurrence un tourne-vis et plusieurs sortes de couteaux constituent les outils, alors que la 205 et la Twingo sont des voitures.

Un objet de type "shape"

Suposons que nous voulions définir une série d'objets permettant de dessiner des formes à l'écran ("circle", "triangle", "square"). Nous pouvons procéder de plusieurs manières:

La méthode Jacques Prévert

C'est celle correspondant aux couteaux de cuisine ci-dessus: il suffit de définir trois objets différents, un pour chaque forme désirée. On aura alors des déclarations du style:

```
class circle;
class triangle;
class square;
```

Si mettre tout sur le même plan est stupide dans la vie quotidienne, ce n'est pas plus malin dans un programme informatique... mais en outre, cette méthode conduira à réécrire sans arrêt la même chose... justement ce que le C++ voudrait éviter.

La méthode étiquette

Déjà un peu mieux... elle consiste à considérer qu'un cercle, un triangle, un carré sont des formes: nous créons donc une classe appelée `shape`, définie de la manière suivante:

```
enum kind {circle, triangle, square};
class shape {
   point center;
   color col;
   kind k;
public:
   point where() {return center};
   void draw();
};
```

Du point-de-vue de la conception, cela revient à dire qu'un cercle est une forme ayant l'étiquette "circle"... pas mal, mais pas fameux, car cela revient aussi à dire qu'il n'y a pas plus de différence entre un cercle rouge et un cercle noir qu'entre un cercle et un carré... même sans être un as en géométrie, on sent bien que cela ne correspond pas à la réalité...
Du point-de-vue de l'écriture du code, la fonction `draw` va tester le champ `kind`, et suivant les cas dessinera un cercle, un triangle, un carré... cela présente quelques sérieux inconvénients:

- Plus le nombre de formes sera grand, plus la fonction draw sera longue.
- Si je rajoute une nouvelle forme (ellipse, par exemple), je vais devoir modifier la fonction draw... avec tous les risques d'ajout d'erreurs.

Au fond, pour reprendre l'exemple concret précédent, tout se passe comme si les ingénieurs de Renault, lorsqu'ils ont conçu la Safrane, avaient réouvert le dossier de la Twingo et avaient modifié des dessins de celle-ci, en ajoutant des erreurs. Résultat: le jour de la sortie de la Safrane, les Twingo qui sortent de l'usine ont trois roues...
En fait, ce qui manque ici, c'est de distinguer entre propriétés génériques, communes à tous les objets de type shape, et propriétés spécifiques à certaines formes.

La méthode par l'héritage

L'héritage est précisément l'outil qui va nous permettre d'implémenter cette distinction: cela passera par la définition de 4 classes. Une classe "abstraite" comportant toutes les propriétés génériques, et trois classes comportant les propriétés spécifiques de chaque forme particulière. Voici le code de la classe de base:

```
class shape {
   point center;
   color col;
public:
   point where() {
      return center;
   };
   virtual void draw()=0;
}
```

La classe de base

La déclaration de fonction `draw` signifie qu'une forme doit pouvoir être dessinée, mais on ne sait pas encore, à ce stade, comment elle sera dessinée. Une conséquence de la présence de cette fonction est qu'il est impossible d'écrire dans un code quelque chose comme:

```
shape forme1;
```

Le compilateur refusera cela, parce que `shape` est une classe abstraite, c'est-à-dire une classe qui contient au moins une fonction virtuelle. Il ne sait pas quoi faire de cette fonction. En fait, cela est assez compréhensible: si je vous dis, "dessine-moi une forme"... et si vous êtes un ordinateur (donc complètement dépourvu d'imagination) vous ne saurez pas quoi

dessiner comme forme. De même, si je dis "cette variable est de type forme", c'est à peu près comme si je disais "cette variable est un machin". Pas très précis... Par contre, rien ne m'empêche de passer à une fonction une variable de type `shape`:

```
void arriere_plan (shape s);
```

Au fond, même si je suis Dieu, je ne peux pas dire "Bien, aujourd'hui je vais créer un machin", mais n'importe qui a le droit de demander son machin à un ami...

Les classes dérivées

Un cercle, un triangle, un carré sont des formes ayant chacune leur particularité. Nous écrirons cela de la manière suivante:

```
class circle: public shape {
   int radius;
   public:
   void draw();
};

class triangle: public shape {
   ...;
   public:
      void draw();            // dessine-moi un triangle
      coord getheight(1);     // renvoie la hauteur principale  du triangle
};

class square: public shape {
   int cote;
public:
   void draw();
};
```

Ainsi, nos trois classes dérivées "héritent" des caractéristiques génériques de leur classe de base, mais ajoutent des caractéristiques particulières propres à chacune. Du point-de-vue de la modélisation, nous avons bien trois objets, qui sont un cercle, un triangle, un carré... mais ces trois objets sont *aussi* une forme. On a donc réussi à introduire un fort degré d'abstraction de données... tout en gardant, à l'intérieur des fonctions, du code C, donc "proche de la machine".
Du point-de-vue de l'implémentation, cela nous conduit à écrire trois versions différentes de la fonction `draw`. Il est bien préférable d'écrire plusieurs petites fonctions qu'une grande... surtout, lorsque je devrai rajouter une nouvelle forme, je n'ai pas à revenir sur les formes déjà définies, il y a donc moins de risques d'erreurs.

Définir des piles de n'importe quoi

Revenons à notre histoire de pile de caractères: justement, c'est d'une pile d'entiers dont j'ai besoin. Et dans un autre programme, j'aurai besoin d'une pile d'autre chose... Ce cas peut-il se traiter, lui aussi, par la méthode d'héritage ? Certainement pas: d'un point-de-vue conceptuel, on ne peut pas dire qu'une pile d'entiers et une pile de réels soient des cas particuliers d'un objet plus général, comme a pu le faire avec les cercles, les triangles ou les carrés... par contre, on peut dire qu'une pile est toujours une pile, et que si on peut empiler des caractères il n'y a aucune raison pour qu'on ne puisse pas empiler autre chose que des caractères, en réutilisant les mêmes algorithmes. De même que dans la plupart des langages, on peut déclarer des tableaux d'entiers, de réels, de structures... Le C++ permettra d'implémenter ce concept par des types paramétrés (encore appelés des *modèles*. On pourra par exemple définir des piles de formes en déclarant:

```
stack<shape> pile-de-formes;
```

Héritage correct: faire du neuf avec du vieux

Une conséquence de ce qui précède est que nous allons être capables de "faire du neuf avec du vieux": puisque je sais passer à une fonction une variable de type `shape`, je peux dès la première version du programme écrire par exemple une fonction qui me tapisse mon écran avec toujours la même forme. Lors de l'appel de la fonction, je devrai bien sûr préciser si je veux dessiner des cercles, des carrés ou des triangles... mais surtout, si dans dix ans ils me prend la fantaisie de vouloir dessiner des ballons de rugby (des ellipses), il me suffira de créer un objet de type `ellipse`, et de recompiler *le vieux programme* qui, lui, restera inchangé. Le mécanisme d'héritage et de fonctions virtuelles me garantit que cela fonctionnera. D'où une énorme souplesse pour faire évoluer les programmes.


Cela fonctionnera *uniquement* à condition que les relations d'héritages soient "proprement" définies. En particulier, lorsqu'une classe hérite d'une autre, les fonctions virtuelles de la classe dérivée doivent *faire au moins autant de choses* que celles de la classe de base, et elles ne *doivent pas avoir des exigences supérieures* . En d'autres termes, la classe dérivée doit être *une extension* de sa classe de base, *pas une restriction*.

Des bibliothèques d'objets

Il est possible d'écrire des bibliothèques d'objets, qui utiliseront soit la généricité soit les relations d'héritages. Ces bibliothèques seront utilisables par les programmeurs, qui pourront éventuellement continuer à créer de nouveaux objets qui hériteront des objets de la bibliothèque. Nous verrons ici rapidement la `stdlib`

La programmation objets expliquée aux gens normaux

Ce chapitre s'adresse aux gens "normaux", c'est-à-dire aux personnes n'ayant pas d'expérience de programmation, quelque soit le langage utilisé. Les programmeurs chevronnés, eux, feraient mieux de lire le chapitre précédent

Une cafétéria informatisée

Imaginons que nous devons modéliser une cafétéria automatique, comme on en trouve souvent sur les aires de repos des autoroutes, constituée d'un certain nombre de cafetières en libre-service.
Pour corser la chose, il y a trois types différents de cafetières:

- Les machines à café utilisant du café en grain
- Les machines à café utilisant du café moulu
- Les machines à café utilisant du café soluble

Le café en programmation procédurale

En programmation procédurale, on s'intéressera essentiellement à ce que la machine doit *faire*; en l'occurrence, on réfléchira à la manière de préparer le café lorsqu'un utilisateur l'aura demandé. Les *objets* de l'application (en l'occurrence les spécifications des différentes machines à café) n'occuperont pas une place centrale dans notre réflexion.
Ainsi, dans notre exemple, on écrira trois algorithmes différents, correspondant aux trois manières de faire le café suivant le type de machine. Cela pourra par exemple se traduire par une fonction du type:

```
int faire_le_cafe (int cafid, int caftyp, int cafforce);
```

où `cafid` est un "identificateur de cafetière" (il faut bien donner une adresse ou un nom différents à chaque cafetière pour pouvoir les différentier), `caftyp` est le type correspondant de la cafetière, alors que `cafforce` est un nombre (de 1 pour du jus de chaussette à 10 pour du café italien) donnant une idée de la force du café désiré. La fonction `faire_le_cafe` cache sous un interface commun plusieurs algorithmes différents (au moins un pour chaque type de cafetière); il peut être astucieux d'ailleurs d'écrire trois fonctions différents, d'où la structure suivante pour la fonction `faire_le_cafe`:

```
int faire_le_cafe(int cafid, int caftyp, int cafforce) {
    int rvl=0;
    switch (caftyp) {
       case CAFE_EN_GRAIN:
            rvl = faire_le_cafe_en_grain(cafid,cafforce);
            break;
       case CAFE_MOULU:
            rvl = faire_le_cafe_moulu(cafid,cafforce);
            break;
       case CAFE_SOLUBLE:
            rvl = faire_le_cafe_soluble(cafid,cafforce);
            break;
       default:
            rvl = 9;  /* Erreur, type inconnu */
    }
    return rvl;
}
```

Il faut connaître pour chaque machine quel est son type, cela peut faire l'objet d'un tableau `caf_types`:

```
int caf_types[15];
```

Pour connaître le type de la machine numéro 10, il suffit de lire la valeur de `caf_types[10]`.
Par ailleurs, on a besoin de connaître l'état des différentes machines à café; par exemple, un client a-t-il demandé un café ? On peut donc imaginer une fonction, appelée `lire_etat`, qui ira lire l'état de la machine à café. Elle reverra par exemple le code 0 pour dire "machine prête", et le code 1 pour dire "quelqu'un a demandé un café". On peut bien sûr imaginer encore d'autres codes pour dire par exemple que le réservoir d'eau est vide, etc.
Un programme déclenchant N machines afin qu'elles fassent toutes du café ressemblera en fin de compte à celui-ci:

```
for (int i=0; i < N; i++) {
    if (lire_etat(i)==1) {
       int rvl = faire_le_cafe(i,caf_types[i],cafforce);
       if (rvl == 0) {
          printf "le cafe est pret\n";
       } else {
          printf "Machine en panne\n";
       };
    }
}
```

Le café en programmation objet

Tout cela est bien beau, mais on voit d'emblée que cette manière de procéder peut poser quelques problèmes:

- Ecrite de cette manière, la programmation n'a qu'un très lointain rapport avec la manière dont nous pensons, de sorte qu'elle a un côté très artificiel, y compris lors de la conception du programme.

- Le programme risque de se révéler fort peu robuste: en effet, si l'on désire le modifier afin d'ajouter un nouveau type de machine à café, on devra modifier la fonction `faire_le_cafe`, afin d'ajouter une condition à l'instruction `switch`. Pour peu que le programme soit complexe, il y aura un grand nombre de fonctions écrites sur le modèle de `faire_le_cafe`. Il n'est pas évident de les retouver toutes lorsqu'on doit ajouter un nouveau modèle de cafetière. D'autant plus qu'elles seront probablement dispersées un peu partout dans le code. Lors de cette modification, on risque d'ajouter des erreurs dans les lignes de code *correspondant aux cafetières existant déjà*.

Modéliser une cafetière

Il est bien plus naturel de raisonner en termes d'objets, car c'est notre manière quotidienne de penser: il s'agit en effet de faire fonctionner un ensemble *d'objets*. Ceux-ci:

- Ont un grand nombre de points en commun (il s'agit toujours de machines à café)
- Sont cependant de plusieurs types
- Ont une certaine structure (en fait, ils contiennent d'autres objets), ainsi qu'un comportement vis-à-vis du monde extérieur:
    - On peut leur "donner l'ordre" de faire le café.
    - On peut leur "demander" dans quel état ils sont.

La programmation objets va permettre de *modéliser* cet ensemble d'objets et ainsi d'écrire un programme bien plus proche de la manière de penser humaine. D'autre part, la structure du programme est telle que la modification d'un objet ne devrait pas avoir d'impact sur les caractéristiques d'un autre objet, ou sur les caractéristiques générales du programme; le programme sera donc plus robuste, et plus facile à maintenir.

qu'est-ce qu'une cafetière ?

Nous dirons qu'une cafetière est un objet. Cet objet est un ensemble de composants:

- Réservoir d'eau
- Réservoir de café
- Réservoir de sucre
- Réservoir de cuillers
- Réservoir de gobelets
- Résistance chauffante

Lorsque nous écrirons le programme, il nous suffira de représenter ces objets par des *variables*, ainsi il sera possible d'écrire:

```
cafetiere A;
```

de la même manière qu'on écrit en C:

```
int B;
```

Une cafetière est une cafetière, mais il y en a plusieurs sortes

Par ailleurs, nous avons vu qu'il pouvait exister plusieurs types de cafetières: or, même si une machine avec réserve de café en grain est de conception différente d'une machine avec réserve de café en poudre, nous savons parfaitement qu'il s'agit dans les deux cas de cafetières, c'est-à-dire que ces deux objets ont un grand nombre de caractéristiques communes. Nous allons exprimer cette relation dans notre programme objet, en utilisant les relations d'*héritage*: nous aurons donc un nouveau type de variable (`cafetiere_grain`), différent du type `cafetiere`, mais qui *hérite* de ce type un grand nombre de caractéristiques. Simplement, il *ajoute* de nouvelles caractéristiques à ce type. Ces caractéristiques nous permettront de préciser à la machine le procédé exact pour faire le café, alors que les caractéristiques communes permettent de dire à quelles conditions un objet peut légitimement être appelé cafetière.

boutons marche-arrêt ou autres réglages

Lorsqu'on se trouve à l'extérieur de la cafetière, on n'a pas accès aux pièces composant la cafetière. De même, dans notre programme, un mécanisme permettra de *cacher* les objets situés à l'intérieur de l'objet `cafetiere`.
Par contre, on a un moyen de remplir le réservoir d'eau ou le réservoir de café, de savoir combien d'eau il reste dans le réservoir, combien de sucre dans la réserve de sucre, etc. De même, on a plusieurs boutons de réglage (café fort, moyen, faible), et on a un bouton pour déclencher la mise en route du café. Dans notre modèle, cela correspond à des `fonctions` que nous appellerons *méthodes*, ou encore *fonctions-membres*. Les fonctions-membres sont partie prenante de l'objet, de la même manière que le bouton de marche-arrêt de la (vraie) cafetière est une partie de la cafetière. Mais il s'agit de la partie "interface" avec le monde extérieur. Pour faire du café italien, nous pourrons alors écrire dans notre programme:

```
cafetiere_grain A;
A.force(10);
A.faire_le_cafe();
```

On voit que le style de programmation est considérablement différent de ce qui précède; alors que précédemment, le numéro de la cafetière, le type de cafetière, et peut-être encore d'autres informations sont passées en paramètre à une fonction `faire_le_cafe`, cette fois la fonction `faire_le_cafe` est directement intégrée à la variable `A`, ce qui est bien plus proche de la réalité: c'est bien la machine à café, qui intègre un mécanisme permettant de faire du café; de la même manière, l'algorithme expliquant à l'ordinateur comment le café devra être fait se trouve "intégré" à l'objet de type `cafetiere`.
Plus précisément, on va pouvoir dire au programme que toute cafetiere doit avoir *au moins* une fonction `faire_le_cafe`, mais il est simple d'imaginer que cette fonction sera très différente suivant le type de cafetière. La seule chose de sûre, c'est que cette fonction devra exister.
Puisque la fonction est intégrée à l'objet `cafetiere`, il peut aller chercher les informations dont il a besoin *directement à l'intérieur* de l'objet: c'est ainsi que le paramètre `cafforce` de tout-à-l'heure a été retiré; il n'est plus nécessaire, car la force du café a été fixée par la fonction `force()`, qui est elle aussi une méthode de notre objet. A l'inverse de la fonction `faire_le_cafe()`, par contre, on peut très bien imaginer que pour certains types de machines la fonction `force()` est inexistante (dans ce cas la force du café est prédéfinie et ne peut être ajustée par l'utilisateur: c'est moins confortable, mais ça fait tout-de-même du café). Donc la fonction `force()` sera une méthode de `cafetiere_grain`, *pas* de `cafetiere`.

Voyants

La fonction appelée précédemment `lire_etat` devient, elle aussi, une fonction-membre: elle joue alors le rôle d'un voyant.

Programme principal

Nous pouvons maintenant réécrire l'ébauche de notre programme:

```
cafetiere C[100];
... initialiser C[i] avec des objets de type cafetiere_grain,
cafetiere_poudre, cafetiere_soluble ...
for (int i=0; i < N; i++) {
    if (C[i].lire_etat()==1) {
       int rvl = C[i].faire_le_cafe();
       if (rvl == 0) {
          printf "le cafe est pret\n";
       } else {
          printf "Machine en panne\n";
       };
    }
```

```
}
```

Des programmes plus robustes

Puisque les objets de type `cafetiere` sont tout simplement des variables comme les autres, il est possible, par exemple, de les ranger dans un tableau. Chaque élément d'un tableau sera donc une `cafetiere`, cependant certains éléments peuvent être un objet de type `cafetiere_grain` alors que d'autres peuvent être un objet de type `cafetiere_soluble`. De sorte que lorsque la fonction `C[i].faire_le_cafe()` est appelée, rien ne dit qu'il se passe en réalité la même chose à chaque itération du tableau. C'était déjà le cas en programmation fonctionnelle, la différence ici est tout simplement que la structure `switch` précédente a disparu: elle est remplacée par un mécanisme d'appel de fonctions intégré au systè lui-même. Cela conduit à une *totale séparation* entre les différents types de cafetière, donc si je rajoute dans un ou deux siècles un nouveau type de cafetière, je ne risque plus d'ajouter des erreurs et de faire planter les autres types. Le code est à la fois plus clair, parce que plus proche de la pensée humaine, *et* plus robuste, en ce sens qu'une modification quelque part risque moins d'introduire des erreurs ailleurs. Ces deux caractéristiques conduisent à un code plus simple à maintenir.

Classes et objets

Une *classe* est une manière de décrire un type d'objets: on peut le voir comme un moule, qui servira à la fabrication des objets proprement dits. Une classe n'est pas un objet, au moins d'un point-de-vue conceptuel.
Un objet se caractérise par trois choses:

- Un état
- Un comportement
- Une identité

L'état d'un objet:

L'état d'un objet est la combinaison de ses propriétés: celles-ci elles-mêmes peuvent être des objets. Par exemple, l'état de la machine à café sera:

- En fonctionnement
- Prête à faire le café
- Réservoir ? café plein ou vide
- etc.

L'état d'un objet dépend de son histoire: si la machine à café est plusieurs fois en fonctionnement, son réservoir finira par se vider.

Le comportement d'un objet:

Le comportement d'un objet est ce qu'il est capable de réaliser, la manière dont il réagira à des messages extérieurs, ... le comportement corresopnd donc au code qui sera inséré à l'intérieur de l'objet afin de la faire agir. Un changement d'état peut générer un comportement particulier (le réservoir à café se vide: l'objet téléphone à la maintenance), et réciproquement une action entraînera un changement d'état.

L'identité d'un objet:

Si l'on veut que plusieurs objets cohabitent dans le programe, il faudra bien les différentier d'une manière ou d'une autre: l'identité est là pour cela. Le programmeur n'a généralement pas à s'en préoccuper, elle est directement gérée par le système. L'identité de l'objet en C++ est simplement l'adresse de l'objet dans la mémoire.

Des objets "métiers"

Les objets métiers sont des objets qui doivent être compris par toute personne du domaine concerné: par exemple, l'objet cafetière devrait être aisément compréhensible par tous ceux qui s'occupent réellement de la machine à café. Par contre, si on met les machines à café dans un tableau, qui lui-même sera un objet, ce tableau n'est utile que pour le déroulement du programme lui-même: les gens "du métier" n'ont aucune raison de s'y intéresser.

Relations entre objets

Les objets vont communiquer entre eux en s'envoyant des *messages*: dans la réalité, ces messages sont tout simplement des appels de fonctions, comme on l'a vu. Les objets sont en relation entre eux (un objet en relation avec aucun autre objet ne servirait à rien). On distingue plusieurs types de relations:

Association
: C'est la plus courante, aussi la moins forte: c'est la relation qui lie un enseignant et ses étudiants, par exemple.

Agrégation
: C'est la relation qui lie un objet et ses composants: la cafetière et son réservoir d'eau, de café, de gobelets, par exemple. Elle est plus forte que la relation d'association, toutefois une cafetière peut exister sans son réservoir à café.

Composition
: La relation la plus forte (utilisée rarement). Un immeuble de 10 étages ne peut exister sans tous ses étages, et réciproquement: on exprimera donc dans le code le fait que si l'objet immeuble est détruit, tous les objets le composant seront détruits également.

Une sorte de
: On pourra exprimer le fait qu'une machine à café particulière est "une sorte de" machine à café.

Un tas de
: Des objets particuliers (les conteneurs) nous permettront de mettre nos objets dans des structures de données: on pourra donc avoir "un tableau de cafetières", "une pile de cafetière", "une queue de cafetières", etc.

Les relations d'association, d'agrégation et de composition s'expriment en insérant des variables membres dans une définition de classe (). Dans le cas de la relation de composition, il convient de s'assurer que les objets sont construits ou détruits ensemble. La relation "est une sorte de" s'exprime grâce à l'héritage ( ). Les autres relations s'expriment par les modèles ()

Emmanuel Courcelle <emmanuel.courcelle@toulouse.inra.fr>
sommaire

Les types de base

Déclarations de variables

Les types de base du `C++` sont les mêmes que les types du `C`, avec les extensions suivantes:

- Le type `bool`
- La notion de référence
- Le type `class` (fondamental, car c'est lui qui permet de définir les objets)

La portée d'un nom

Un nom [de type, de variable, de fonction, ...] n'est utilisable qu'entre telle et telle ligne du code. Cet espace est appelé la portée du nom. La portée du nom `A` est définie de la manière suivante:

- Elle commence à la ligne de déclaration de `A`
- Elle se termine à la fin du bloc dans lequel la variable `A` a été définie. La fin de bloc est marquée par une accolade fermante `}`.


Une variable peut être déclarée à n'importe quel endroit du code, alors que le C impose une déclaration de variable en début de bloc uniquement. Exemple:

```
...
   {
   ...
   int A=5;    // DEBUT DE LA PORTEE DE A
   ...
   };          // FIN DE LA PORTEE DE A
...
```

Nom global

Un nom global est un nom défini à l'extérieur de toute fonction, de toute classe, de tout espace de noms. Un nom global sera donc accessible dans tout le programme.


Ne pas confondre variable globale et variable locale à la fonction main:

```
...
int A;              // variable globale

int main() {
    int B;          // variable locale a main
    int C=f1();
};

int f1() {
    ...
    int C=A;        // pas de pb, A est accessible
    C += B;         // ERREUR B n'est pas connu ici
    return C;
};
```

Masquage et résolution de portée

Comme les trains à un passage à niveau, un nom peut en cacher un autre ! Pour masquer un nom, il suffit de déclarer le même nom dans un bloc interne au premier. Le masquage sera terminé dès la fin du bloc interne. Bien entendu, rien n'empêche de masquer un nom de variable par un nom de fonction, par exemple... Il est toutefois recommandé d'éviter de jouer avec cela, c'est une source de bogues pas évidents à détecter.
Un nom masqué *ne peut être utilisé* , sauf s'il s'agit d'une variable globale: dans ce cas, l'opérateur de résolution de portée `::` permet d'utiliser le nom global, comme on le voit dans l'exemple ci-dessous:

```
int A=1;
main () {
   int B=1;
   int A=2;
   cout << A << "\n";          // renvoie 2
```

```
   cout << ::A << "\n";        // renvoie 1
   {                           // ouverture d'un nouveau bloc
      int B=2;
      cout << B << "\n";       // renvoie 2
   }
```

Variables...

La portée d'une variable est bien entendu la portée du nom de cette variable. Concrètement, la mémoire est allouée dès le début de la portée, et la mémoire sera rendue au système dès la fin de la portée.

✔ Une exception à la règle de la portée: dans l'exemple ci-dessous, la portée de la variable `i` est le *bloc* situé en-dessous de l'instruction `for`. C'est parfaitement logique, car cela permet de définir des variables muettes dans les boucles `for`, mais ce comportement est différent de ce qu'on connaît en `C`.

```
...
for (int i=0; i<10;i++)    // DEBUT DE PORTEE DE i
   {...
    ...
   };                      //  FIN DE PORTEE DE i
```

La structure d'une déclaration de variables

Une déclaration de variables comprend:

1. Un descripteur optionnel (`const`, `extern`, `virtual`, ...)
2. Un type de base *obligatoire* (`int` etc., ou type défini par le programmeur)
3. Un déclarateur *obligatoire*. Celui-ci est constitué de:
   - un nom choisi par le programmeur
   - un ou plusieurs opérateurs de déclaration.
4. Un initialiseur optionnel.

Exemple:

```
int* A [];
```

La déclaration de variables ci-dessus est constituée de la manière suivante:

1. Pas de descripteur
2. Type de base: `int`
3. Déclarateur constitué de:
   - Le nom `A`
   - Les deux opérateurs de déclaration **`*`** et **`[]`**. Les opérateurs *postfixés* (**`[]`**) ayant une priorité supérieure aux opérateur *préfixés* (**`*`**), on a déclaré un tableau de pointeurs, non pas un pointeur vers un tableau.
4. Pas d'initialiseur.

Quelques déclarations légales:

```
char* ctbl[] = {"bleu","blanc","rouge"};    // 3 parties sont spécifiées
const int A = 2;                            // 4 parties
int B;                                      // 2 parties seulement
```

quelques descripteurs sur lesquels nous reviendrons à plusieurs reprises:

`const`
: Permet de définir une variable constante. Par exemple, un objet constant. Nous verrons dans la suite que cela peut apporter quelques complications.

`mutable`
: Utilisé dans les déclaration de classes, en lien avec `const`

`static`
: Utilisé dans les déclarations de classes ou de fonctions

`virtual`
Utilisé dans les déclaration de classes

Les types prédéfinis

Le type `bool`

Un booléen peut prendre les valeurs `true` ou `false`. Il est possible de convertir un booléen en entier et vice-versa; dans ce cas, `true` se convertit en `1`, `false` se convertit en `0`. Dans l'autre sens, `0` est converti en `false` et tout entier non nul est converti en `true`.

Les types `char` ou numériques

Les types usuels du C sont utilisables en C++:

- **`char`**, **`unsigned char`**, **`signed char`**
- **`int`**, **`short int`**, **`long int`**
- **`unsigned int`**, **`unsigned short int`**, **`unsigned long int`**
- **`float`**, **`double`**, **`long double`**

Le type void

Il s'agit d'un "pseudo-type", qui veut dire "rien". On peut l'employer comme type de pointeur;

```
void*
```

signifie "pointeur vers n'importe quoi". `void` peut aussi être employé comme type de retour de fonction:

```
void  f()
```

signifie "fonction qui ne renvoie aucune valeur" (procédure en pascal).

✔ La déclaration suivante:

```
fonction();
```

devrait en toute logique correspondre à une fonction qui ne renvoie aucune valeur. Hélas... le C considère que le type `int` est renvoyé par défaut. Le C++ a bien sûr suivi le même usage, pour des raisons de compatibilité. Donc pour ne rien renvoyer il faut spécifier:

```
void fonction_1();
```

Les types `struct` et `union`

Disons pour simplifier que ces types sont utilisés en C++ de la même manière qu'en C: même si l'on peut déclarer des fonctions-membres dans les structures, il est préférable de s'en tenir au type `class` décrit ci-dessous pour définir des objets, et utiliser `struct` et `union` pour le code de plus bas niveau.

Pointeurs (*), références (&), descripteur const

✔**Note typographique**. On le verra dans la suite, il est aisé de confondre:
- les deux significations du caractère * dans: `int * x` et `y = *x`
- les deux significations du caractère & dans: `int &` et `x = & y`

Pour faciliter les choses, on écrira:
- **`int* x`** ou **`int& y`** pour les déclarations
- **`*x`** ou **`&y`** pour les opérateurs.

Notons que la norme du C++ permet d'insérer un espace entre le caractère et le nom de la variable ou du type, mais cette présentation est plus claire pour le lecteur, et correspond bien à la réalité du compilateur: en effet, dans une déclaration `int * x` ou `int & y` , il s'agit bel et bien d'utiliser les *types* `int*` ou `int&`.

Quelques analogies avec le monde dit "réel"

Soit un objet de type... `homme`. Comment cet objet peut-il se manipuler, et quelle analogie peut-on faire avec la vie "réelle" ?

Initialisation, opérateur = : Un peu de Science Fiction

Contrairement à ce qu'on pourrait penser, ces opérations, qui paraissent les plus simples (par analogie avec les maths), sont en fait les plus lourdes pour l'ordinateur...

Initialisation = Clônage

```
homme jacques;
homme paul = jacques;
```

`paul` est obtenu par "clônage" à partir de `jacques`. Les deux objets sont parfaitement identiques lorsque le code ci-dessus est exécuté, mais ensuite ils vivent chacun leur vie, et leur destin peut être différent dans la suite du programme.

✔L'initialisation comme l'affectation ne sont pas des opérations simples a priori: elles se traduisent au minimum par une copie bit à bit, qui peut être longue si les objets sont gros, et éventuellement par des opérations plus complexes comme l'allocation de ressources. On essaira donc de les éviter dans la mesure du possible, tout au moins lorsqu'on a affaire à des objets élaborés.

Opérateur= : Je me prends pour un autre.

```
homme pierre;
...
pierre = paul;
```

Ici, `pierre` a une vie avant sa rencontre avec `paul`. L'opérateur= va "jeter" toutes les donné qui concernent pierre et les remplacer par celles de `paul`. Ensuite, chaque objet vit sa vie, comme précédemment...

✔Ces opérations permettent d'obtenir deux objets *égaux*, on l'a vu, mais pas *identiques*.

Références = Surnoms

```
homme pierre;
homme& pierrot = pierre;
homme& tonton = pierre;
```

La situation ci-contre est bien plus courante: tout simplement, `pierre` porte plusieurs surnoms. Les uns l'appelleront `pierrot`, les autres `tonton`. Dans tous les cas,il s'agit de la même personne (du même objet). Tout ce qui arrivera à `pierre` arrivera aussi à `pierrot`, puisqu'il s'agit du même individu. De même qu'une personne peut avoir autant de surnoms qu'on le souhaite, de même un objet peut avoir un nombre illimité de références. Mais il n'y a jamais qu'un seul objet.

✔ Cette fois, on a obtenu deux objets *identiques* (donc aussi égaux).

Pointeurs = Attention, on vous montre du doigt

```
homme pierre;
homme* ce_mec = pierre;
homme* le_type_la_bas = pierre;
homme* encore_un_bebe= new(homme);
```

`pierre` est montré du doigt une fois, deux fois, ... autant de fois que vous le désirez: donc `homme` désigne un objet a priori compliqué, mais `homme*` désigne tout simplement le doigt qui pointe sur un homme. (en C++, comme en C,on a autant de types de doigts différents que d'objets pointés. Cela permet d'éviter de nombreuses erreurs de programme).

✔ Bien entendu, on peut avoir autant de pointeurs que l'on veut. Mais chaque pointeur est un nouvel objet. Les pointeurs sont délicats à manier, simplement parce qu'il est possible de "casser" le lien entre pointeur et objet pointé. Cela peut amener deux situations ennuyeuses:

- Le pointeur est détruit, mais pas l'objet pointé. Si rien ne pointe plus sur lui, on peut perdre sa trace dans la mémoire de l'ordinateur... celui-ci a alors la migraine (fuites de mémoire).
- L'objet pointé peut disparaître, alors que le pointeur continue de pointer sur lui: Risque important de plantages aléatoires.

Pierre va chez le coiffeur...

Soit la fonction `coupe` qui a deux paramètres: le coiffeur et le client, le coiffeur coupant les cheveux au client.

Passage des paramètres par valeur

```
void coupe(homme coiffeur, homme client);
...
homme pierre;
homme jacques;
coupe(jacques, pierre);
```

`pierre` ainsi que `jacques` sont ici passés par valeur. Autrement dit, arrivés au salon de coiffure, la machine clône `pierre` d'une part, `jacques` d'autre part, et c'est le clône du coiffeur qui va couper les cheveux au clône de pierre. Après quoi, les deux clônes sont détruits, et `pierre` repart avec les cheveux longs. L'histoire est stupide, certes, mais ce

genre d'erreurs arrive fréquemment (en C++, en tous cas).

Passage du client par référence

```
void coupe(homme coiffeur, homme& client);
...
homme pierre;
homme jacques;
coupe(jacques, pierre);
```

`pierre` est passé par référence à la fonction `coupe`: `client` est tout simplement un surnom qu'on lui donne dans ce contexte. `jacques` est toujours passé par valeur, de sorte que dans cette histoire, c'est le clône de `jacques` qui coupera les cheveux à son `client`, qui se trouve être `pierre`. Pas de problème, le clône de `jacques` est par définition aussi bon coiffeur que `jacques` lui-même. Mais le clônage n'est-il pas une opération un peu compliquée, simplement pour une histoire de coupe de cheveux ? Un avantage à signaler: si `pierre` est mécontent du travail du coiffeur, il pourra toujours casser la figure au clône de `jacques`, `jacques` lui-même ne sera pas touché... en termes plus techniques, si la variable locale `coiffeur` est modifiée par le programme, cela n'aura pas d'impact sur `jacques` (pas d'effets de bords).

Passage du coiffeur par const référence

```
void coupe(const homme& coiffeur, homme& client);
...
homme pierre;
homme jacques;
coupe(jacques, pierre);
```

dans le contexte de la fonction `coupe`, `pierre` s'appelle maintenant `client`, alors que `jacques` s'appelle `coiffeur`. Plus besoin d'opérations compliquées comme le clônage, alors qu'un surnom fait si bien l'affaire. De plus, le descripteur `const` protège le coiffeur contre les clients mécontents: même si `pierre` est mécontent de sa coupe, il ne pourra pas casser la figure à son coiffeur (car l'état de celui-ci ne peut changer, à cause de `const`). D'un point-de-vue technique, la variable locale `coiffeur` ne peut être modifiée, il ne peut donc là non plus y avoir d'effets de bords. Ainsi, la sémantique (signification) de cet appel et celle de l'appel précédent sont les mêmes, simplement le code est ici plus optimisé.

Un accouchement difficile

Voici l'histoire d'un accouchement à haut risque, suite à l'exécution de la fonction `coït`...

Retour d'un paramètre par valeur

```
humain coit(homme& h, femme& f) {
   ...
   humain enfant = h + f;
   ...
   return enfant;
```

```
};

homme pierre;
femme marie;
humain loulou = coit(pierre,marie);
```

Une drôle de manière de faire un enfant: l'enfant nait dans le contexte de la fonction `coit`, mais à la fin de la fonction, on en fait un clône, on sort le clône et on massacre l'enfant. Merci de ne pas prévenir le comité d'éthique... C'est long, compliqué et immoral, mais ça marche.

Retour d'un paramètre par référence

```
humain& coit(homme& h, femme& f) {
   ...
   humain enfant = h + f;
   ...
   return enfant;
};

homme pierre;
femme marie;
humain& loulou = coit(pierre,marie);
```

Voilà qui n'est encore pire: l'enfant, après sa naissance, est retourné sous le nom `loulou`... mais tout-de-suite après il est détruit, puisqu'il s'agit d'une variable local à la fonction `coit`, qui n'existe donc que le temps que la fonction est exécutée.

✔ Attention, cela ne veut pas dire qu'on ne doit pas renvoyer de références en sortie d'une fonction. On ne doit pas renvoyer de référence sur un objet interne à la fonction, car cet objet cesse d'exister lorsque la fonction a fini son exécution. Le pire, c'est que... ça peut marcher: rien ne dit que le système aura détruit tout-de-suite l'objet. Mais gare au plantage si vous changez de conditions (de compilateur, par exemple).

Retour d'un paramètre par pointeur

```
humain* coit(homme& h, femme& f) {
   ...
   humain* enfant = new humain(h,f);
   ...
   return enfant;
};

homme pierre;
femme marie;
humain* nouveau_ne = coit(pierre,marie);
```

Cette fois, ça va mieux: l'enfant est créé par `new`, mais il est *quelque part ailleurs*, et *il n'a pas été baptisé (pas de nom)*. On ne sait que l'appeler `*enfant`. Seul le pointeur est interne à la fonction. On renvoie (par une recopie) le pointeur à l'extérieur, et on détruit le pointeur d'origine (mais cela n'a aucune importance, l'enfant est préservé).

Retour au monde virtuel...

Le type référence

Soit le programme suivant:

```
int A=3;
int& a=A;
A++;
cout << "valeur de A = " << A << "valeur de a = " << a << "\n";
```

Le programme renvoie 4 pour A comme pour a. Que s'est-il passé ? La ligne `int &a=A` qui signifie "référence", revient à déclarer un *synonyme* à A (même adresse mémoire, mais nom différent). L'adresse en mémoire sera donc *la même* pour A et pour a.

✔ La déclaration suivante *dans un programme ou une fonction* n'a pas de sens:

```
int & a;              // ERREUR DE COMPILATION !!!
```

En effet, un synonyme est un synonyme, encore faut-il préciser de quoi on est synonyme. Par contre, cette déclaration *en tant que membre d'une classe* a un sens: on précisera de quoi on est synonyme lors de l'initialisation de la classe.). De même, une telle déclaration dans une liste de paramètres d'une fonction a une signification

✔ On *ne peut pas* changer de "cible": une fois qu'on a dit que `a` est synonyme de `A`, `a` reste synonyme de `A` durant toute sa portée. Dans le code suivant:

```
nt A=3;
int& a=A;

int B=5;
a=B;
```

L'expression: **a=B** changera la *valeur* de `a`, donc aussi la valeur de `A`.

Le type pointeur

De même que ci-dessus, Le programme suivant imprimera deux fois le chiffre 4:

```
int A=3;
int* a;

a = &A;
A++;
cout << "valeur de A = " << A << "valeur pointee par a = " << *a << "\n";
```

a est un *pointeur*; A l'inverse des références, il est possible (*quoique dangereux*) de ne pas l'initialiser; d'autre part, `a` peut pointer sur n'importe quelle variable, ainsi que le montre le code suivant:

```
int A=3;
int B=6;
int* a;

a= &A;
cout << "valeur de A = " << A << "valeur pointee par a = " << *a << "\n";
a= &B;
cout << "valeur de B = " << B << "valeur pointee par a = " << *a << "\n";
```

✔Dans l'expression `a= &B` le signe `&` est un *opérateur*. Il ne s'agit pas d'une déclaration de type comme dans le paragraphe précédent: le même symbole a donc deux significations différentes.

Une référence, pour quoi faire ?

Les principales utilisations des références sont les suivantes:

- Le passage des paramètres aux fonctions
- La déclaration de membres de classes
- La valeur de retour renvoyée par les fonctions

Les deux premières utilisations sont utiles pour:

- Economiser de la place en mémoire, si les objets à passer prennent beaucoup de mémoire.
- Utiliser le polymorphisme

Passage des paramètres par référence

Le programme ci-dessous renvoie 5:

```
void f(int X) {
   X=0;
};

main() {
   int A=5;
   f(A);
   cout << A << "\n";
};
```

En effet, lorsque la variable `X` est passée à la fonction `f`, sa *valeur* est recopiée dans la variable locale `X`. C'est la copie locale de `X` qui est mise à zéro, elle sera détruite dès le retour de la fonction. Par contre, le programme ci-dessous renvoie 0:

```
void f(int& X) {
   X=0;
};

main() {
   int A=5;
   f(A);
   cout << A << "\n";
};
```

En effet, la déclaration `int& X` dans le prototype de la fonction `f` indique un passage des paramètres par référence. `X` est donc un *synonyme* de la variable passée, et non plus une recopie. En conséquence, la ligne `X=0` dans `f` remet à 0 la variable `A`. Passer un paramètre par référence revient donc à passer un paramètre à la fonction, tout en laissant à celle-ci la possibilité de modifier la valeur de la variable ainsi passée, donc d'en faire aussi une *valeur de retour*

Pourquoi renvoyer des références ?

Pour deux raisons principales:

- Renvoyer une référence est très rapide, car une référence n'est qu'une adresse (quelques octets). Par contre, renvoyer un objet peut être coûteux en ressources, car cela met en jeu le constructeur de copie, et l'objet peut être très volumineux
- Renvoyer une référence permet aussi de renvoyer une "`lvalue`", c'est-à-dire quelque chose qui peut se mettre à gauche d'un signe `=`

Regardons en effet le programme suivant:

```
int A,B;
int& renvAouB(bool s) {
  return (s==true ?) A : B;
};

main() {
  A = 10;
  B = 20;
  cout << A << B <<"\n";  // ecrit 10 20
  renvAouB(true) = 5;
  cout << A << B <<"\n";  // ecrit 5 20
};
```

La fonction renv renvoie une *référence* vers la variable `A`. Il est donc légal d'écrire `renv(true)=5` même s'il peut paraître surprenant de mettre à gauche du signe égal un appel de fonction.
Ce mécanisme est utilisé par les objets définis par la bibliothèque standard, en particulier **map**, `vector` etc. Il est également courant, dans beaucoup de fonctions-membres ou d'opérateurs surchargés, de renvoyer une référence, par exemple une référence à l'objet courant `*this`

 La fonction suivante a de fortes chances de planter à l'exécution: en effet, elle renvoie une référence vers une variable *locale*, et lorsque l'instruction `return` est exécutée, cette variable est détruite... le résultat est théoriquement non prédictible, mais moi je vous prédis pas mal d'embêtements.

```
int A;
int& renv() {
  int A=99;
  return A;      // boum !!! plantage probable.
};
```

Autres langages objets

Langage	Pointeur	Référence
C++	**OUI**	**OUI**
perl	**OUI**[1]	NON
java	NON	NON
python[2]	NON	NON

1. Une *référence* perl est en fait équivalent à un *pointeur* C ou C++. Par contre, étant donné que perl intègre un "ramasse-miette" (gestion de la mémoire), l'utilisation des références en perl est assez différente de celle des pointeurs en C/C++.
2. La situation en python est un peu particulière: en effet, dans ce langage, toutes les variables sont des pointeurs vers des zônes de mémoire. Il faut avoir cela présent à l'esprit lorsqu'on réalise des affectations.

Le descripteur const

Utilisation avec des références:

Pourquoi passer les paramètres par référence ? Pour deux raisons:

- Le paramètre est utilisé *en entrée*, mais aussi *en sortie*... cf. ci-dessus.
- Le paramètre est bien un paramètre d'entrée pur, mais sa recopie prendrait un temps processeur non négligeable: cela peut être le cas si l'on passe une variable de type `struct` ou `class` avec un grand nombre de champs.

Dans le second cas, il y a danger: en effet, si l'un ou l'autre champ de l'objet passé en paramètre est modifié, on se retrouve avec un "effet de bord" non désiré, erreur pas simple à détecter... dans ce cas, le C++ offre un moyen bien pratique d'éviter cela: le descripteur `const`, placé devant la déclaration du paramètre, assure que celui-ci *ne pourra pas* être modifié par la fonction. Ainsi, le programme suivant ne pourra pas être compilé:

```
void f( const int& X) {
   X=0;          // Erreur, car X est constant
};
```

Passage des paramètres par pointeur

Les possibilités offertes par le passage de paramètres par référence rendent obsolète l'équivalent en C: le passage des paramètres par pointeurs. Voici deux programmes équivalents, à vous de décider lequel est le plus lisible:

En C:

```
void f(int* X) {
   *X=0;
};

main() {
   int A=5;
   f(&A);
};
```

En C++:

```
void f(int& X) {
   X=0;
};

main() {
   int A=5;
   f(A);
};
```

Le programme C++ est bien plus lisible, ne serait-ce que parce que c'est lui qui minimise l'utilisation des signes barbares tels que & ou *.

 Il n'est utile de passer les paramètres par pointeur *que si le cas pointeur=NULL* doit être envisagé:

```
void f( int* X) {
   if (X == NULL) {
      ...faire quelque chose
   } else {
      ...faire autre chose
   };
};
```

Il n'est pas possible de gérer ce cas avec des références, puisqu'une référence n'est par définition "synonyme" de quelque chose.

✔ Une fonction prenant des paramètres par `const &` ne *doit jamais renvoyer ce paramètre*... il y a risque important de crash. Exemple:

```
const int& f(const int& x) {
  return x;
};

main() {
  int A = 10;
  int B = f(A);
  int C = f(4);
};
```

La ligne `int B = f(A)` ne pose pas de problème, par contre que se passe-t-il avec la ligne `int C = f(4)` ? Le compilateur crée une variable temporaire de type entier, l'initialise à 4, appelle la fonction `f` qui renverra une référence à cette variable temporaire... et *supprime juste après la variable temporaire*. Résultat, on se retrouve avec une référence qui pointe sur... rien du tout, risque important de plantages. Voir ci-dessous (chapitre gestion de la mémoire) d'autres exemples de gags du même genre

Utilisation de const avec des pointeurs

Le descripteur `const` peut s'employer également avec des pointeurs, de sorte que les différentes déclarations ci-dessous sont légales, et empêchent d'écrire certaines instructions... donc empêchent de faire certaines erreurs:

```
const int* a      = new(int);
*a = 10;                        // Erreur car *a est constant
int* const b      = new(int);
b = new(int);                   // Erreur car b est constant
const int* const c = new(int);
*c = 10;                        // Erreur car *c est constant
c  = new(int);                  // Erreur car c est constant
```

✔ L'expression **const int* a** *ne garantit pas* que ***a** ne changera jamais de valeur. Il garantit *uniquement* qu'il sera impossible de taper quelque chose dans le style `*a=10`. Mais le code suivant montre qu'il est parfaitement possible que `*a` change de valeur. Il suffit pour cela qu'un autre pointeur, non constant, soit défini avec la même adresse:

```
int A=10;
const int* a = &A;
cout << "*a = " << *a << "\n";
A=100;
cout << "*a = " << *a << "\n";
```

Le type class

Le type class va nous permettre de créer différents objets. C'est donc grâce à ce type qu'il est possible de faire de la programmation objets en C++.

✔ Attention, une déclaration de classe est une *déclaration de type*. Or, un objet est *une variable*. Une classe va permettre de créer (on dit aussi instancier) des objets d'un certain type. En d'autres termes, une classe est un *moule*, elle sert à créer

des objets, mais elle n'est pas un objet elle-même.

Sections privées, protégées, publiques

Voici la déclaration d'une classe qui implémente des nombres complexes:

```
class complexe {
public:
   void init(float x, float y);
   void copie(const complexe& y);
private:
   float r;
   float i;
}
```

Il s'agit d'une déclaration très proche du type `struct` du C. Cependant, par rapport à la `struct` du C, plusieurs différences fondamentales:

- Certains membres sont des fonctions
- On peut déclarer des membres qui sont eux-mêmes des types (on parle alors de types locaux).
- Les membres sont déclarés dans différentes sections avec des notions de protection par rapport à l'extérieur

On retrouve ainsi la notion de protection (encapsulation) des variables et des fonctions propre à la programmation structurée, mais intégrée au système de typage, puisqu'il s'agit de déclarer un nouveau type de données. Ce qui correspond à l'implémentation se trouve dans la section `private`, alors que ce qui correspond à l'interface se trouve dans la section `public`. En d'autres termes, l'intérieur de l'objet (son squelette) se trouve dans la section `private`, alors que l'interface avec le monde extérieur (les boutons, voyants, en un mot son comportement) se trouve dans la section `public`.

Section private

Tout ce qui est déclaré dans cette section sera utilisable *uniquement* (ou presque, il y a aussi les amis) à partir d'une variable de même classe; ainsi, dans l'exemple ci-dessus, le code:

```
complexe X;
...
X.r=0;
X.i=0;
```

produira une erreur à la compilation, car `r` et `i` étant des membres privés, ils ne sont pas accessibles à partir "de l'extérieur". Par contre, si `X` et `Y` sont deux complexes, le code écrit *dans les fonctions-membres de la classe complexe* peut atteindre les variables privées *de toutes les variables de type complexe*, ainsi qu'on le voit dans l'exemple ci-dessous (fonctions `init` et `copie`):

```
class complexe {
public:
   void init(float x, float y) {r=x;   i=y;};
   void copie(const complexe& y)  {r=y.r; i=y.i;};
private:
   float r;
   float i;
}
```

Notations

La fonction `init` accède aux membres privés de la variable elle-même. Dans ce cas, il suffit de les appeler par leur nom de membre (il ne peut y avoir d'ambiguïté) et l'expression `r=x` signifie "affecter la partie réelle de ce complexe à la valeur passée par paramètre".

La fonction `copie` accède aux membres privés de la variable, mais `aussi` aux membres privés du complexe y. Dans ce cas, il faut spécifier le nom de variable en plus du nom de champ, d'où l'expression `y.r`

Section public

Tout ce qui est déclaré dans cette section sera utilisable depuis l'extérieur de l'objet. Ainsi, dans l'exemple précédent les fonctions `init` et `copie` peuvent être appelées depuis le programme principal:

```
complexe X;
...
X.init(0,0);
```

Section protected

Cette section sera décrite plus tard, lorsque nous aborderons l'héritage

Fonctions membres

Les membres d'une `class` peuvent être soit des types, soit des variables, soit des fonctions. Dans ce dernier cas, on parle de fonctions membres, ou encore de méthodes.

Définition des fonctions-membres

Dans l'exemple précédent, nous avons déclaré et défini les deux fonctions-membres à l'intérieur de l'objet lui-même (voir plus loin la différence entre déclaration et définition). Cela offre deux avantages:

- Code plus compact et plus lisible pour de petites fonctions
- Rapidité d'exécution plus importante, car la fonction est ainsi déclarée implicitement *inline*, ce qui veut dire que le compilateur mettra directement les instructions dans le code, en évitant ainsi la perte de temps dûe aux appels de fonctions.

Toutefois, cela est difficilement concevable pour des fonctions plus longues. Dans ce cas, on ne met dans la déclaration de classe que la déclaration de la fonction, sa définition viendra plus tard... oui, mais alors il faudra bien spécifier l'appartenance de cette fonction à une classe donnée. Cela se fait avec l'opérateur de portée `::` (Voir ci-dessous les exemples).

Fonctions et classes amies

Il est possible de donner l'accès aux membres privés et protégés de la classe à certaines fonctions définies par ailleurs dans le programme, ou à *toutes* les fonctions membres d'une autre classe: il suffit de déclarer ces fonctions ou ces classes dans la section `public` (il s'agit d'une fonctionnalité de l'interface) en ajoutant devant la définition de fonction le mot-clé `friend`. Nous reparlerons des fonctions amies lors de la discussion sur la surcharge des opérateurs

✔ Une fonction-membre d'une classe a accès aux données privées *de tous les objets de sa classe*. Cela revient à dire que l'unité de protection n'est pas l'objet, mais *la classe*. Et la notion de fonction amie, et surtout de classe amie permet encore d'élargir cette notion de protection au "groupe de classes". On peut se poser la question suivante: n'y a-t-il pas contradiction entre l'encapsulation des données d'une part et cette notion d'amies d'autre part ? Bien évidemment si: à manier avec précaution... toutefois, dans certains cas, il est utile de déclarer des classes amies: certaines "abstractions" ne sont pas nécessairement implémentées par une seule classe, mais par deux ou plusieurs classes. Dans ce cas, les différentes classes participant à cette abstraction devront avoir accès aux mêmes données privées... sans quoi nous devrons enrichir l'interface de manière exagérée, au risque justement de *casser* le processus d'encapsulation.

Accès aux données

 Dans chaque section, on peut trouver des types, des variables, ou des fonctions. Cependant, même si le langage ne l'impose pas, il est préférable de s'en tenir aux usages suivants:

- Les types (énumérations notamment) peuvent être définis aussi bien dans la section `public` que dans la section `private`.
- Les variables ne seront définies *que* dans la section `private`: en effet, les variables jouent en quelque sorte le rôle de squelette de l'objet, elles définissent sa *structure interne*
- Les fonctions peuvent être définies aussi bien dans la section `private` que dans la section `public`.
    - Dans la section `private`, on trouvera les fonctions qui participent au *fonctionnement interne* de l'objet.
    - Dans la section `public`, on trouvera les *fonctions d'interface*. En particulier, on trouvera des fonctions permettant de modifier les variables privées (*mutator*), ou encore des fonctions permettant de lire la valeur de ces variables (*accessor*). Le fait de passer par des fonctions pour ces opérations, plutôt que de déclarer simplement la variable dans la section `public`, offre une très grande souplesse, car les fonctions membres peuvent parfaitement faire autre chose, en interne, que de simplement écrire ou lire une variable.

Cela permettra donc de contrôler très précisément l'accès aux données. La contrepartie étant, bien sûr, une plus grande lourdeur, puisqu'il y a plus de fonctions à écrire. Notre objet `complexe` pourrait devenir:

```
class complexe {
public:
   void init(float x, float y) {r=x;   i=y; _calc_module();};
   copie(const complexe& y)  {r=y.r; i=y.i; m=y.m;};
   float get_r() { return r;};
   float get_i() { return i;};
   void set_r(float x) { r=x; _calc_module();};
   void set_i(float x) { i=x; _calc_module();};
   float get_m() {return m;};
private:
   float r;
   float i;
   float m;
   float _calc_module();
}

float complexe::_calc_module() {
    m = sqrt(r*r + i*i);
}
```

Nous venons d'introduire un nouveau champ: `m`, qui représente le module. La fonction `_calc_module` est une fonction privée, appelée automatiquement dès que la partie réelle ou la partie imaginaire du complexe est modifiée. Ainsi, les fonctions `set_r` et `set_i` modifient les champs `r` et `i` de notre objet, mais elles font *aussi* autre chose: elles lancent le calcul du module. Il ne serait pas possible d'implémenter ce type de fonctionnement en utilisant pour `r` et `i` des champs publics. Le prix à payer est toutefois l'existence des fonctions `get_r`, `get_i` et `get_m`, qui sont triviales. Etant déclarées `inline` dans le corps de l'objet, elles ne causeront cependant pas de perte de performance.
Par ailleurs, il est évident que le champ `m` ne *doit pas* être public: en effet, si tel était le cas, le code suivant:

```
complexe X;
X.init(5,5);
X.m=2;
```

serait autorisé par le compilateur, avec un résultat désastreux (aucune cohérence dans les champs de l'objet). On peut bien sûr se demander s'il est utile de programmer un objet complexe de cette manière. Après tout, il serait aussi simple de lancer le calcul du module directement dans la fonction `get_m`... bien sûr, mais cette manière de faire présente certains

avantages:

- L'objet `complexe` ainsi défini est cohérent, puisqu'on est assuré que le module, maintenu par l'objet lui-même, sera toujours correct. Et la variable `m` peut être utilisée par d'autres fonctions membres, puisque l'on est sûr qu'elle est en permanence à jour.
- Supposons un programme qui initialise de temps en temps des complexes, mais qui passe son temps à utiliser le module des complexes dans d'autres calculs: cet objet se révèlera très performant, puisque le calcul du module ne sera effectué que lors de l'initialisation. Cet argument, peut être très fort lorsqu'il s'agit de calculs coûteux en ressources.

Mais peut-être qu'au cours du développement, nous allons justement nous apercevoir que le programme passe son temps à initialiser des complexes, et n'utilise le calcul du module qu'une fois de temps en temps. Dans ce cas, l'argument ci-dessus se renverse, et cette implémentation conduit à un objet peu performant. Qu'à cela ne tienne, nous allons réécrire l'objet *complexe*:

```
class complexe {
public:
   void init(float x, float y) {r=x;   i=y;};
   copie(const complexe& y)  {r=y.r; i=y.i;};
   float get_r() { return r;};
   float get_i() { return i;};
   void set_r(float x) { r=x;};
   void set_i(float x) { i=x;};
   float get_m() {return sqrt(r*r+i*i);};
private:
   float r;
   float i;
}
```

Le nouveau `complexe` est plus simple que le précédent, il calcule le module uniquement lorsque l'on en a besoin: il n'est donc plus nécessaire de maintenir le champ `m`.

Par contre, il a un autre défaut: à chaque appel de `get_m()`, le module est recalculé, ce qui peuimportantt s'avérer coûteux si les appels à cette fonction sont nombreux. La version suivante de `complexe` résoudra ce problème. Le module est calculé *uniquement* en cas de besoin, c'est-à-dire non pas lors de *chaque* appel à `get_m()`, *uniquement* lors du premier appel à `get_m()` suivant une modification du module. Voici le code, qui se complique un peu:

```
class complexe {
public:
  void init(float x, float y) :r(x),i(y),m(0),m_flg(false) {};
  void copie(const complexe& y )  {r=y.r; i=y.i; m=y.m;};
  float get_r() { return r;};
  float get_i() { return i;};
  void set_r(float x) { r=x; m_flg=false;};
  void set_i(float x) { i=x; m_flg=false;};
  float get_m() const;
private:
  float r;
  float i;
  bool m_flg;
  float m;
  void _calc_module() {m=sqrt(r*r+i*i);};
};

float complexe::get_m() {
  if (!m_flg) {
    _calc_module();
    m_flg=true;
  };
```

```
  return m;
};
```

Ce qui est remarquable, c'est que dans ces trois versions, *seule l'implémentation a changé*. Autrement dit, tout le code qui *utilise* cet objet restera identique. C'est très important, car ce code est peut-être gros, peut-être écrit par d'autres personnes, etc. D'où l'importance de *bien spécifier l'interface*, et de ne mettre dans l'interface *que des fonctions*: une fonction triviale un jour peut se révéler compliquée le lendemain, si son interface est la même le passage de l'une à l'autre sera indolore. Passer d'une opération d'affectation de membre à un appel de fonction (ou réciproquement) est une autre histoire... Cet argument de maintenabilité du code vaut largement que l'on écrive des fonctions triviales comme `get_r` ou `get_i`...

✔ Il ne faut pas abuser des fonctions `get_xxx` et `set_xxx`: en effet, attention à ne donner ainsi l'accès *qu'à certains membres* privés. Sans cela, donner un accès, même réduit, à tous les membres privés, risque de vous conduire à nier la notion d'encapsulation des données, et de rendre problématique l'évolution de l'objet.

Constructeurs

Nous avons dit précédemment que les types définis par l'utilisateur devaient se comporter "presque" comme les types de base du langage. Cela est loin d'être vrai pour ce qui est de notre objet `complexe`: par exemple, pour déclarer une variable réelle, nous pouvons écrire `float X=2;` Comment faire pour déclarer un objet complexe, tout en l'initialisant à la valeur (2,0), par exemple ? Actuellement, nous devons écrire:

```
complexe X;
X.init(2,0);
```

Ce n'est pas génial... d'une part le code est assez différent de ce qu'il est pour initialiser des réels ou des entiers, mais surtout que se passe-t-il si nous oublions d'appeler la fonction init ? Cet oubli est possible, justement parce que l'initialisation du `complexe` se fait de manière différente des autres types.
C'est pour résoudre ce problème que le C++ propose une fonction membre spéciale, appelée constructeur. Le constructeur possède deux spécificités:

- Le nom est imposé (même nom que le nom de la classe).
- Il ne renvoie aucune valeur.

```
class complexe {
public:
  complexe(float x, float y) :r(x),i(y),m(0),m_flg(false) {};
  void copie(const complexe& y )  {r=y.r; i=y.i; m=y.m;};
  float get_r() { return r;};
  float get_i() { return i;};
  void set_r(float x) { r=x; m_flg=false;};
  void set_i(float x) { i=x; m_flg=false;};
  float get_m() const;
private:
  float r;
  float i;
  bool m_flg;
  float m;
  void _calc_module() {m=sqrt(r*r+i*i);};
};
```

Rien n'a changé, à part la fonction `init`, remplacée par le constructeur (`complexe`). Mais cela change tout: en effet, on peut maintenant écrire dans le programme utilisateur de la classe:

```
float A = 5;
...
complexe X(2,0);
```

On voit qu'on a une déclaration "presque" équivalente à ce qu'on a avec un type prédéfini. La différence provient *uniquement* de ce que nous avons besoin de deux paramètres pour initialiser un complexe, et non pas un seul comme pour un entier ou un réel. Mais nous verrons au paragraphe suivant qu'il y a moyen de faire encore mieux.

Constructeur prédéfini

En fait, il n'est pas indispensable de définir un constructeur: si l'on supprime le constructeur de la définition de classe précédente, le programme compilera toujours. Simplement, il ne sera pas possible d'initialiser explicitement l'objet. En d'autres termes, l'expression `complexe X;` sera valide, mais l'expression `complexe X(0,0)` sera refusée par le compilateur. Le compilateur appellera simplement le constructeur par défaut de l'objet... Attention toutefois, celui-ci n'initialisera pas les membres de l'objet.

✔ Le (ou les, cf. plus loin) constructeurs définis pas l'utilisateur ne s'ajoutent pas au constructeur par défaut, ils le *remplacent*. Autrement dit, nous avons le choix entre:

- Une classe `complexe` sans constructeur. Dans ce cas, l'expression `complexe C;` sera acceptée, mais l'expression `complexe C(0,0);` sera refusée.
- Une classe `complexe` avec constructeur, telle que définie ci-dessus. Dans ce cas, l'expression `complexe C;` sera refusée, mais l'expression `complexe C(0,0)` sera acceptée.

Bien sûr, il y a moyen de dépasser ces limitations, nous verrons comment un peu plus tard.

Initialisation des membres

Le constructeur est le lieu idéal pour faire deux choses:

- Initialiser les données membres
- Allouer des ressources (mémoire en particulier, mais aussi éventuellement ouverture d'une connexion réseau, création d'un fichier, etc).

En fait, ces deux actions sont différentes. Il existe une syntaxe particulière, permettant de mettre en valeur ces différences: l'initialisation des membres peut se faire *avant* le bloc de définition de la fonction constructeur, mais *après* le nom de la fonction, comme on le voit dans le code suivant:

```
class complexe {
private:
   float r;
   float i;
   ...
public:
   complexe(float x, float y) : r(x), i(y) {_calc_module();};
   ...
}
```

Cette manière de procéder est intéressante, car elle sépare proprement les deux fonctions du constructeur: initalisation des membres d'une part, exécution de code (allocation de mémoire ou autre ressource) d'autre part. S'il n'y a rien d'autre à faire que des initialistations, le corps de la fonction peut être vide: dans ce cas, on doit écrire des accolades vides `{}` à la suite de la liste d'initialisation.

✔ Lorsqu'un membre est déclaré en tant que référence, la *seule manière* de l'initialiser est de passer par la liste d'initialisation:

```
class objet {
private:
   complexe& X;
   ...
public:
   object (const complexe& C) : X(C) {};
};
```

Destructeur

Nous avons vu qu'une fonction, le constructeur, est appelée lors de la création de la variable. De même, une autre fonction, le *destructeur*, est appelée lors de sa destruction. Le destructeur possède les spécificités suivantes:

- Le nom est imposé (caractère ~ suivi du nom de la classe)
- Il ne renvoie rien.
- Il ne prend pas de paramètre

Un des rôles du constructeur est de demander au système certaines ressources: fichier, mémoire, etc. Il faut bien un jour rendre la ressource au système, c'est précisément le rôle du destructeur: fermeture de fichier, libération de mémoire,...

Le type complexe en mode debug

A titre d'exemple pour l'utilisation du constructeur et du destructeur, nous allons adjoindre un système de débogage à notre objet `complexe`: le constructeur écrira un message sur l'erreur standard, tandis que le destructeur écrira un autre message. Ainsi, même dans le code le plus compliqué, nous aurons toujours une trace de la création ou de la destruction de la variable. Cela pourrait s'écrire de la manière suivante:

```
class complexe {
public:
   complexe(float x, float y);
   ~complexe();
   ...
private:
   ...
}

complexe::complexe(float x, float y) {
   r=x;
   i=y;
   _calc_module();
   cerr << "Creation d'un objet de type complexe\n";
}

~complexe::complexe() {
   cerr << "Destruction d'un objet de type complexe\n";
}

main() {
   ...
   if (...) {
      complexe A(0,0);   // Appel du constructeur
      ...
   };                                    // Appel du destructeur
```

A l'exécution, ce programme enverra un message sur l'erreur standard dès que l'instruction `complex A(0,0);` sera exécutée (c'est-à-dire à l'entrée du `if`), et à nouveau lors de la destruction de la même variable, c'est-à-dire lors du passage sur l'accolade fermante (`}`) (fin de la portée de la variable).

Le descripteur `static`

Le code ci-dessus envoie un message lors de chaque appel du constructeur et du destructeur. Cela peut être une aide précieuse lors de la mise au point du programme, mais il serait souhaitable de pouvoir inhiber ce fonctionnement: lorsque le programme sera mis en exploitation, le mode debug n'aura plus aucune raison d'être. Même en période de déboguage,

nous voulons avoir la possibilité de passer ponctuellement en mode débug, ou de le désactiver.
Voici un premier essai:

```
class complexe {
public:
   complexe(float x, float y): r(x),i(y),debflg(false) {};
   ~complexe();
   void set_debug() { debflg=true;};
   void clr_debug() { debflg=false;};
   ...
private:
   ...
   bool debflg=false;
}

complexe::complexe(float x, float y) {
   ...
   if (debflg) {cerr << "Creation d'un objet de type complexe\n";};
}

~complexe::complexe() {
   if (debflg) {cerr << "Destruction d'un objet de type complexe\n";};
}
```

Ce code nous pose deux problèmes:

1. Le constructeur n'enverra jamais de message: en effet, `debflg` est false par défaut, et l'objet aura déjà été créé, donc le constructeur aura déjà été appelé lorsque nous serons en mesure d'appeler la fonction `set_debug`.
2. Il serait fastidieux... et pour tout dire sujet à bien des erreurs, d'appeler `set_debug` ou `clr_debug` pour chaque objet, de manière individuelle. Nous avons besoin au contraire d'un membre et d'une fonction-membre qui puisse contrôler le mode debug simultanément pour *tous* les objets `complexe`.

Une donnée membre statique

La déclaration suivante résout une partie de notre problème:

```
class complexe {
public:
   complexe(float x, float y): r(x),i(y) {};
private:
   ...
   static bool debflg;
}
```

Le descripteur `static` signifie que `debflg` est un membre *commun à tous les objets de type* `complexe`: alors qu'un membre "ordinaire" est spécifique à chaque *objet*, un membre statique sera spécifique à chaque *classe d'objet*. Du point-de-vue de l'allocation mémoire, on peut considérer qu'il s'agit d'une référence à une zône de mémoire allouée ailleurs. Du coup:

- la déclaration `static bool debflg` ne provoquera pas de nouvelle allocation mémoire
- il est interdit d'initialiser ce membre comme on le ferait avec un membre ordinaire.

Cette seconde restriction est compréhensible; en effet, un initialisateur posé au même endroit que la déclaration aurait pour conséquence la réinitialisation du membre statique à chaque création de variable de type complexe. Ce qui rendrait ledit membre complètement inutile. Il faudra donc avoir quelque part dans le code une définition de variable:

```
bool complexe::debflg=false;
```

Fonctions membres statiques

Le code ci-dessus présente encore un gros inconvénient: il est impossible de jouer avec `debflg` avant d'avoir créé au moins une variable de type `complexe`. La solution est d'utiliser, en plus du membre statique `debflg`, deux *fonctions-membres* statiques; `set_debug` et `clr_debug`. De même que les membres statiques sont *liés à une classe d'objets*, les fonctions-membres statiques sont *liées à une classe d'objet*, pas à un objet.
Le code devient alors:

```
class complexe {
public:
   ...
   static void set_debug() { debflg=1;};
   static void clr_debug() { debflg=0;};
private:
   ...
   static bool debflg;
   ...
};

bool complexe::debflg=false;

main () {
   complexe::set_debug();        // passe en mode debug
   ...
   complexe::clr_debug();        // sort du mode debug
```

✔ Les membres statiques ou les fonctions-membres statiques sont des choses *très différentes* des membres ou fonctions-membres ordinaires:

- Ils obéissent aux règles de portée des membres
- Les fonctions-membres statiques ont le droit d'accéder aux membres privés de l'objet, par contre ils doivent pour cela utiliser le pointeur `*this`

✔ Les fonctions membres statiques ressemblent beaucoup aux fonctions amies :

- De même qu'une fonction amie de la classe, elles ont accès à l'ensemble des membres et des fonctions membres (private, protected, public) de celle-ci.
- Par contre, une fonction amie peut être:
  - Soit une fonction "ordinaire"
  - Soit une fonction membre d'une autre classe

  Une fonction membre statique d'une classe est... uniquement une fonction membre de cette classe: c'est donc uniquement la portée qui différencie fonction membre statique et fonction amie.

Le descripteur const

Le descripteur `const` est un des plus utilisés parce que très utile, mais il est aussi un des plus délicats à utiliser. Toute variable peut être déclarée comme `const`, ce qui veut dire que cette variable est en fait... une constante.

✔ Puisqu'il sera impossible, une fois la constante déclarée, de modifier sa valeur, il est indispensable de l'initialiser. Donc l'expression `const int A;` produira un message d'erreur, alors que `const int A=5;` sera accepté par le compilateur.

const, pourquoi faire ?

Il est utile, par exemple lorsqu'on passe un objet par référence à une fonction, d'exprimer le fait que cet objet est constant, c'est-à-dire que toute opération visant à *modifier explicitement* l'objet doit être interdite.

membres constants

Un objet peut avoir une donnée membre constante. Soit une classe appelée `tableau`. Dans son constructeur, cette classe alloue de la mémoire pour un tableau d'entiers. La mémoire est rendue au système dans le destructeur. La taille du tableau est constante durant toute la durée de vie de l'objet (une fois que le tableau existe, il n'est pas prévu qu'on puisse lui changer sa taille). Par contre, la taille du tableau peut être choisie lors de la construction de l'objet.
Afin de faire ressortir dans le code cette spécificité, et afin d'être sûr qu'un bogue ne modifie pas la taille du tableau inopinément, on utilise une donnée membre constante. Le code ci-dessous est incorrect:

```
class tableau {
  const int bfrsz;
  char* buffer;
public:
  tableau(int);
  ~tableau() {delete buffer;};
};

tableau::tableau(int sz) {
  bfrsz=sz;           // PAS POSSIBLE CAR bfrsz EST CONSTANT
  buffer = new char[sz];
};

void main() {
  tableau B(1024);
};
```

Cela marcherait si `bfrsz` n'était pas déclaré `const`. En effet, l'expression `bfrsz=sz;` ne peut pas avoir de sens, puisque bfrsz est une constante. Par contre, la technique des initialisateurs de membres, décrite ci-dessus, fonctionne très bien dans ce cas:

```
class tableau {
  const int bfrsz;
  char* buffer;
public:
  tableau(int);
  ~tableau() {delete buffer;};
};

tableau::tableau(int sz) : bfrsz(sz) {
  buffer = new char[sz];
};

void main() {
  tableau B(1024);
};
```

Objets constants

Il est bien sûr possible d'utiliser le descripteur `const` avec des objets, pas seulement avec des variables de types prédéfinis.

Les fonctions membres constantes

Par exemple, si nous retournons à notre objet complexe, on pourrait définir le complexe constant i par: `const complexe i(0,i);`

Oui, mais nous avons un problème: le code suivant ne compilera jamais.

```
class complexe {
public:
   complexe(float, float);
   float get_r() { return r;};
   float get_i() { return i;};
   ...
private:
   float r;
   float i;
   ...
};
main() {
   const complexe i(0,1);
   float X = i.get_i();
}
```

En effet, *personne* ne peut garantir au compilateur que la fonction `get_i()` ne va pas elle-même modifier l'objet i. Il est clair que *certaines* fonctions-membres doivent être utilisables sur des objets constants (`get_r, get_i` par exemple), parce qu'elles ne vont *pas modifier* cet objet (ce sont des *accessor*), alors que d'autres fonctions ne peuvent pas être utilisées dans ce contexte (`set_r, set_i`), car elles vont modifier l'objet (ce sont des *mutator*). Il suffit d'ajouter le mot-clé `const` *après* la définition de la fonction pour définir un accessor. Dans ce cas, toute tentative de modification de l'objet (qui serait une incohérence dans le code) sera détectée par le compilateur. Notre objet complexe s'écrit donc:

```
class complexe {
private:
   float r;
   float i;
   ...
public:
   complexe(float, float);
   float get_r() const { return r;};
   float get_i() const { return i;};
   ...
};
main() {
   const complexe i(0,1);
   float X = i.get_i();
}
```

Le descripteur mutable

Essayons d'utiliser le descripteur `const` avec le complexe troisième version écrit plus haut. Il y a un problème avec la fonction `get_m()`. En effet, pour pouvoir utiliser cette fonction avec un objet constant, il faut lui attribuer le descripteur `const`... Or, le compilateur refusera, car `get_r()` ne fait pas que de renvoyer la valeur du module, il lui arrive également de le *calculer*. Donc, les membres `m` et `flg_m` seront modifiés. Que se passe-t-il ? Cela veut-il dire que cette implémentation est incompatible avec le fait de déclarer des complexes constants ? Ce serait une sévère limitation: c'est l'implémentation la plus efficace !

Pour s'en sortir, il faut tout d'abord remarquer que `get_m` ne va pas *réellement* modifier l'objet. Cette fonction modifie deux membres privés, mais *uniquement* pour des raisons d'implémentation. En fait, vis-à-vis de l'extérieur, rien n'a changé: on parle de *constante logique*, par opposition aux *constantes physiques*. Les champs qui ont le droit de varier tout en laissant l'objet constant du point-de-vue logique sont affublés du descripteur `mutate`. Dans notre cas, il s'agit des champs `m` et `m_flg`. L'objet devient alors:

```
class complexe {
public:
```

```
  ...
  float get_r() const { return r;};
  float get_i() const { return i;};
  float get_m() const;
private:
  ...
  mutable bool m_flg;
  mutable float m;
  void _calc_module() const {m=sqrt(r*r+i*i);};
};

float complexe::get_m() const {
  if (!m_flg) {
    _calc_module();
    m_flg=true;
  };
  return m;
};
```

Le pointeur *this

Supposons que l'on veuille modifier à la fois la valeur de la partie réelle et la valeur de la partie imaginaire de notre nombre complexe. Nous pouvons écrire le code suivant:

```
complexe C(0,0);
C.set_r(2);
C.set_i(3);
```

Or, les fonctions `set_r` et `set_i` agissent sur le complexe C. Il est utile de se débrouiller pour qu'elles renvoient le complexe qu'elles viennent de modifier, plutôt que rien du tout. Cela permet par exemple d'écrir le code suivant:

```
C.set_r(2).set_i(3);
```

Cette expression ne peut avoir un sens que si la valeur renvoyée par `set_r(2)` est une *référence* vers le même objet que `C`: dans ce cas `C.set_r(2)` exécute la fonction `set_r(2)`, renvoie `C`, de sorte que `C.set_r(2).set_r(3)` est équivalent à `C.set_r(3)`

Le C++ offre un outil pour arriver à ce résultat: il s'agit du pointeur `*this`. Ce pointeur est une variable privée prédéfinie qui pointe *toujours* sur l'objet dans lequel on se trouve. Pour arriver au résultat ci-dessus, il suffira donc de renvoyer `*this` comme valeur de retour. D'où la définition suivante des fonctions `set_xxx`:

```
class complexe {
public:
   complexe& set_r(float x) { r=x; _calc_module(); return *this;};
   complexe& set_i(float x) { r=i; _calc_module(); return *this;};
private:
   ...
```

Le pointeur `*this` est très utilisé pour les opérateurs, et prendra tout son sens avec eux. Voilà au passage une nouvelle utilisation de la référence en tant que valeur de retour d'une fonction.

Langage	class	private, public, protected	Fonctions membres	Constructeurs	Destructeurs	Fonctions amies	Fonctions membres statiques	Don mem stati
C++	**OUI**	**OUI**	**OUI**	**OUI**	**OUI**	**OUI**	**OUI**	**OUI**
perl	NON	NON	**OUI**	**OUI**	**OUI**	NON	NON	**OUI**
java	**OUI**	**OUI**	**OUI**	**OUI**	NON	NON	**OUI**	**OUI**
python	**OUI**	NON	**OUI**	**OUI**	**OUI**	NON	**OUI**	**OUI**

1. Le tableau associatif `%self` dans lequel on met généralement les données des objets en perl, joue toutefois le même rôle que `this`.

Surcharger fonctions et opérateurs

Modifier une fonction sans remettre en cause l'existant

Il est fréquent, dans un processus de développement, de se trouver confrontés au problème suivant: Lors du codage de la version initiale du programme, nous avons utilisé une fonction `f(X)`, où `X` est un entier. Or, justement dans la seconde version du programme, nous sommes capables de travailler non plus seulement avec des entiers, mais aussi avec des complexes. Il nous faudra donc une fonction `f(X)`, où `X` est un nombre complexe. Autre situation fréquente: l'algorithme de `f` s'est un peu compliqué, et maintenant nous avons besoin de passer à `f` un second paramètre... Comment allons-nous faire ? Deux solutions si nous travaillons en C:

- Changer le prototype de la fonction... mais cela veut probablement dire réécrire en partie le code qui *utilise* cette fonction `f`. Qui dit réécriture du code dit risque d'ajout d'erreurs.
- Créer une nouvelle fonction `f_compl`, qui prendra un nombre complexe comme paramètre... faisable, mais illogique, sachant que `f` et `f_compl` font exactement la même chose: si deux fonctions font la même chose, on a envie de leur donner le même nom. Programmes plus simples à lire, donc à comprendre, donc à maîtriser.

Déclaration et définition de fonctions

Une *fonction* comprend une ou deux parties distinctes:

- *La déclaration* (optionnelle): Le nom de la fonction, les paramètres dont elle a besoin, et ce qu'elle renverra.
- *La définition*: la déclaration (redite au besoin), suivie de ce que fait la fonction (le code).

Déclaration

Une *déclaration de fonction* comprend trois parties:

- Le type de retour (éventuellement `void` si elle ne renvoie rien)
- Le nom de la fonction
- La liste des arguments avec leur type (voir toutefois ci-dessous)

Il est possible, mais pas indispensable de spécifier le nom des arguments: ceux-ci sont considérés par le compilateur comme des variables muettes, et sont ignorés. Leur type, par contre, est une information importante et ne doit pas être omis... sauf exception signalée ci-dessous. Voici un exemple de déclaration:

```
int f1 (int,int,int);
```

Définition

Une *définition de fonction* comprend deux parties:

- La *déclaration* de la fonction, *avec cette fois le nom des paramètres*
- Le *corps de la fonction*, sous la forme d'un bloc

✔ La déclaration de fonction est optionnelle, seule est indispensable sa définition.

✔ Si une fonction est déclarée avant d'être définie, *déclaration et définition doivent être identiques* (mêmes nom, types de paramètres, valeur de retour): l'ensemble de ces trois caractéristiques constituant la *signature* de la fonction.

Surcharge de fonctions

Il est possible de déclarer et définir plusieurs fonctions ayant même nom, *à condition que les listes de leurs arguments diffèrent*: cela résoud en partie le problème que nous avons évoqué plus haut, comme on le voit ci-dessous:

```
float fonction (float x) {
   float y = 3 * x + 2;
   return y;
}

complexe fonction (const complexe & x) {
   complexe y(0,0);
   y.set_r (3*x.get_r() + 2);
   y.set_i (3*y.get_i());
   return y;
}
```

✔ Il n'est *pas possible* de surcharger une fonction par une autre fonction qui aurait même nom et même liste d'arguments *mais une valeur de retour différente*.

✔ *Rien* ne garantit que les deux versions de la fonction `f` ci-dessus font la même chose: c'est au programmeur de s'en assurer, afin que le code reste compréhensible.

✔ Ce mécanisme est extrêmement puissant, en ce sens qu'il va nous permettre de donner un même nom à plusieurs fonctions, travaillant sur des paramètres de types différents. Mais comme souvent, ce qui donne de la simplicité à l'homme est source de complication pour la machine... il n'est pas toujours évident pour le compilateur de décider quelle version de la fonction sera utilisée. Il peut même y avoir parfois ambiguïté. D'où l'existence de règles de surcharge, qui ne seront pas explicitées ici.

Surcharge et constructeurs: le constructeur de copie

Un constructeur est une fonction "presque" comme une autre... donc, il n'y a pas de raison pour qu'on ne puisse pas la surcharger. La surcharge du constructeur permet de fournir plusieurs possibilités d'initialisation, à partir de plusieurs types d'objets.
Un de ces constructeurs est particulièrement important: il s'agit du constructeur de copie, qui va nous permettre d'initialiser un objet à partir d'un autre objet *de la même classe*.

```
class complexe {
public:
   complexe(float x,float y) : r(x), i(y) {};
   complexe(const complexe& c) : r(c.r),i(c.i) {
       cout << "ici constructeur de copie de complexe" << endl};
private:
   float r;
   float i;
   ...
}

main() {
   const complexe j(0,1);
   complexe A=j;
}
```

✔ Attention au prototype du constructeur de copie. En particulier, le passage *par référence* est fondamental: si l'on essaie de passer l'objet par valeur, on demande au compilateur de faire une copie de l'objet afin de la passer au constructeur de copie.

 De même que le langage offre un constructeur par défaut, de même il offre un constructeur de copie par défaut. Celui-ci fait tout simplement une copie membre à membre. Lorsque le constructeur par défaut est suffisant, *utilisez celui-ci*. Mais lorsque le constructeur doit aussi faire autre chose (comme dans l'exemple ci-dessus), vous devez fournir un constructeur de copie.

Valeurs par défaut des arguments

Dans une définition de fonction, il est possible de spécifier des valeurs par défaut à chaque argument. Il s'agit là encore d'un moyen très important pour modifier une fonction sans tout remettre en cause; Soit par exemple le code suivant:

```
float mult (float x) {
    return 2 * x;
};

main() {
   ...
   int y = f (4.5);
};
```

Supposons qu'on désire modifier la fonction `mult` afin qu'elle soit capable de multiplier son argument par n'importe quel nombre entier, et pas seulement 2. L'ancienne version correspodrait toujours à une multiplication par deux. Nous donnons donc 2 comme valeur par défaut au second paramètre, ce qui s'écrit: `float mult (float x, int m=2);`. A partir de là, seront acceptés:

- Les appels "à l'ancienne mode" type `mult (x)`
- Les appels "à la nouvelle mode" type `mult (x,3)`

Voilà ce que cela donne dans notre exemple:

```
float mult (float x, int m=2) {
    return  m * x;
};

main() {
   ...
   int y = f(4.5);         // meme resultat que ci-dessus
   int z = f(4.5,3);       // cela etait impossible avec la version precedente
};
```

 Les arguments ayant des valeurs par défaut *se trouvent obligatoirement* en fin de liste: sinon, le compilateur n'aurait aucun moyen de savoir de quels arguments vous parlez (il n'y a pas, en C++, de possibilité de fournir des arguments nommés, comme en perl ou en fortran 90).

Valeurs par défaut et constructeurs

Nous avons eu précédemment quelques ennuis avec le constructeur de la classe `complexe`, tel qu'il était défini alors. La solution à nos problèmes est toute simple: il suffit d'utiliser des valeurs par défaut pour les paramètres passés au constructeur. Voici le code:

```
class complexe {
private:
  ...
public:
  complexe(float x=0, float y=0) {r=x;   i=y; _calc_module();};
  ...
```

```
};

main() {
  complexe C;        // sous-entendu initialiser a 0
  complexe C1(2);    // sous-entendu initialiser a (2,0) [reel]
  complexe C2(2,2);
}
```

Une facilité d'écriture

Le code ci-dessus permet d'écrire:

```
main() {
   complexe A(5);
   complexe B=5;
   complexe C;
};
```

Les deux premières lignes ont exactement la même signification, simplement `C=5` est plus parlant.Tout le monde comprend que l'initialisation d'un complexe par un réel donne un complexe avec une partie imaginaire nulle. D'autre part, la troisième ligne conduit à l'initialisation à 0 d'un nombre complexe.

Le mot-clé explicit

La facilité d'un jour devient handicap le lendemain: en effet, revenons sur la classe tableau; Puisque le constructeur ne comporte qu'un seul paramètre, nous pouvons écrire le code suivant:

```
main() {
   tableau B = 1024;
}
```

Là, il n'est pas du tout évident, lorsqu'on lit le code ci-dessus, que cela signifie "allouer un buffer de *taille* 1024 octets"... Le concepteur de `tableau` devrait donc inhiber cette écriture, qui se révèle inadéquate. Cela se fait en utilisant le mot-clé `explicit` devant la définition du constructeur:

```
class tableau {
  ...
public:
  explicit tableau(int);
};
```

Maintenant, le code `tableau B = 1024;` provoquera une erreur de compilation, tandis que `tableau B(1024)` passera correctement. Cette dernière syntaxe est plus claire que la précédente, dans la mesure où l'on comprend qu'il y a appel de fonction avec passage de paramètres. On sait alors qu'il faut consulter la documentation afin de savoir précisément ce que fait cette fonction.

✔ Les constructeurs jouent également le rôle de convertisseurs de types: cela peut dans certains cas provoquer des problèmes à la compilation, car le compilateur "se trompe" dans la conversion. Dans ce cas, le mot-clé `explicit` est indispensable pour inhiber la conversion indésirable.

Surcharger les opérateurs

Lorsque nous écrivons le code suivant, en C:

```
int A=2;
int B=3;
int C;
double A1=2.1;
double B1=3.1,
double C1;
main() {
   C = A + B;
   C1= A1+B1;
}
```

Nous utilisons la surcharge des opérateurs "sans le savoir", tel M.Jourdain faisant de la prose. En effet, du point-de-vue des instructions en langage machine, l'opérateur + ne produira pas le même code dans la première et dans la seconde ligne. Dans le premier cas, on fait une addition en arithmétique entière, dans le second cas on fait l'addition en arithmétique flottante.
Le C++ permettra de donner une signification à l'opérateur + (ainsi qu'à tous les opérateurs du langage) spécifique pour chaque classe définie.

Opérateurs et fonctions

L'expression: `C = A + B` peut être vue comme une manière différente d'écrire un appel de fonction. En effet, on pourrait aussi écrire: `C = add(A,B)` Le résultat serait le même que l'expression ci-dessus, mais le code nettement moins lisible. Le C++ respecte tout simplement la convention suivante: lorsqu'il rencontre une instruction `C = A + B`, il exécute en réalité l'instruction `C = operator+(A,C)`. Donc exactement comme ci-dessus, mais simplement le nom réservé n'est pas `add` mais `operator+`
La fonction `operator+` doit accepter deux paramètres de type `complexe` en entrée, et elle doit renvoyer également un complexe, d'où le prototype suivant:

```
complexe operator+(const complexe&, const complexe&);
```

L'addition de trois complexes peut s'écrire `D = A + B + C` soit (l'opérateur + étant associatif à droite): `D = A + (B + C)`, ou encore `D = A + operator+(B,C)` soit `D = operator+(A,operator+(B,C))` Il va sans dire que la première écriture est bien plus compréhensible que la dernière, cependant il est bon de l'avoir présente à l'esprit, en particulier lorsqu'on définit le prototype de la fonction.
La forme utilisant un appel de fonction et la forme utilisant les opérateurs *sont équivalentes*. Simplement, la surcharge des opérateurs va permettre à l'utilisateur de nos objets d'écrire un programme plus élégant.

✔ Il ne s'agit pas de *créer* de nouveaux opérateurs, il s'agit bien de *surcharger* les opérateurs existants. Ni plus, ni moins. Les règles de priorité et d'associativité définies pour les opérateurs du langage s'appliquent également aux opérateurs surchargés.

Opérateurs: le bestiaire

Les tables ci-dessous indiquent:

- La liste (*non exhaustive*) des opérateurs, leurs significations, et la possibilité ou non de surcharge
- Les priorités et associativités des principaux opérateurs

Principaux opérateurs du C++

Opérateurs	signification	Surcharge	Intérêt de la surcharge
::	Résolution de portée	NON	
.	Sélection de membre	NON	
+= -= *= /= %=	Opérateurs unaires arithmétiques.	OUI	Opérations arithmétiques unaires et performantes
+ - * / %	Opérateurs binaires arithmétiques.	OUI	Opérations arithmétiques binaires
++ --	Incrémentation, décrémentation	OUI	Itérateurs
=	Opérateur d'égalité.	OUI	Clônage entre deux objets.
>> <<	Décalage à gauche ou à droite	OUI	entrée sortie.
[]	Accès aux membres d'un tableau	OUI	Indiçage généralisé
()	Appel de fonction	OUI	Objets-fonctions
!	Opération logique	OUI	Permet de comparer un objet à true/false
== !=	Egalité, non égalité	OUI	Egalité, non égalité entre deux objets
> < >= <=	Inégalités	OUI	Inégalités
->* .*	Sélection de membre depuis un ou vers un pointeur.	OUI	Contrôle de l'accès aux membres
& *	Pointeur, référence.	OUI	Objets à comptage de référence, itérateurs

int long short float double etc.	Conversion de types	OUI	Conversion vers un type prédéfini depuis un objet

✔ Vous pouvez mettre *n'importe quoi* dans le code. Rien (sinon votre bon sens) ne vous empêche de mettre une multiplication dans un opérateur +. Autrement dit, c'est un jeu d'enfant de faire dire à un programme C++: `16 = 4 + 4`... Mais bien sûr ce n'est pas fait pour cela ! Au contraire, le seul intérêt de la surcharge des opérateurs est que les *utilisateurs* de vos objets pourront écrire des programmes plus clairs. Utilisez la dernière colonne du tableau ci-dessus afin de surcharger vos opérateurs à bon essient.

Priorité et associativité des opérateurs

Opérateurs (par priorité descendante)											Associativité	() [] -> . -->
!	~	++	--	+	-	*	&	(int)	sizeof		<--	
*	/	%									-->	
+	-										-->	
<<	>>										-->	
<	<=	>	>=								-->	
==	!=										-->	
&											-->	
^											-->	
|											-->	
&&											-->	
||											-->	
?:											<--	
=	+=	-=	*=	/=	%=	&=	^=	|=	<<=	>>=	<--	
,											-->	

Fonction membre ou fonction ordinaire ?

🔷 Faut-il spécifier un opérateur comme une fonction-membre ou comme une fonction ordinaire (éventuellement amie) ? La règle générale est la suivante:

- Opérateur unaire (ex: ++). Fonction membre
- Opérateur binaire (ex: +). Fonction ordinaire

En effet, un opérateur unaire modifie par nature l'objet sur lequel il opère (`a +=3` modifie `a`). Il est donc cohérent d'en faire une fonction-membre. Un opérateur binaire, par contre, opère sur *deux* objets. En faire une fonction-membre revient à "privilégier" de manière arbitraire l'un des deux objets. Au mieux c'est incohérent, au pire cela ne fonctionnera pas.

Les quatre opérations

Le code suivant montre une implémentation de l'opérateur += sur la classe `complexe`. += est implémenté en tant que fonction membre:

```
class complexe {
private:
```

```
  ...
public:
  ...
  complexe& operator+= (const complexe&);
};

complexe& complexe::operator+=(const complexe& c) {
  r += c.r;
  i += c.i;
  return *this;
};
```

Le code suivant montre l'implémentation de l'opérateur +, qui est simplement une fonction ordinaire, prenant deux complexes comme paramètres, et renvoyant un autre complexe:

```
complexe operator+(const complexe& a, const complexe& b) {
  complexe r=a;
  r += b;
  return r;
};
```

Nous avons défini *deux opérateurs* (+ et +=), mais seul l'un d'entre eux (+=) accède aux données privées. Cela signifie que si nous modifions l'implémentation de `complexe` (hypothèse réaliste, nous avons déjà vu trois implémentations différentes) nous n'aurons *qu'un seul* opérateur à modifier: moins de travail, surtout moins de risque d'erreur.

 Attention aux types de retour des opérateurs: en effet, `operator+=` renvoie un `complexe&`, tandis que `operator+` renvoie simplement un `complexe`. Pourquoi ? Il est toujours préférable de renvoyer une référence plutôt qu'un objet, pour des questions de performances: en effet, renvoyer un objet signifie effectuer une copie, opération éventuellement longue pour des objets volumineux, alors que renvoyer une référence signifie renvoyer simplement... une adresse. Opération très rapide, et indépendante de la taille de l'objet. C'est ainsi que `operator+=` renvoie une référence. Par contre, `operator+` renvoie un `complexe`. Ce serait en effet une erreur dans ce cas de renvoyer une référence, car celle-ci pointerait sur une *variable locale*. Cela a d'ailleurs une conséquence dans le code que nous écrirons lors de l'utilisation de ces opérateurs: ainsi il sera plus performant d'écrire `a += b` que d'écrire `a = a + b` , bien que les deux écritures soient autorisées et signifient la même chose. C'est vrai dès que `a` et `b` sont des objets.

Les opérateurs d'incrémentation ou décrémentation

Les opérateurs ++ et -- peuvent bien sûr être surchargés, cependant un problème se pose: en C comme en C++, les versions prédéfinies de ces opérateurs peuvent être:

- soit préfixées
- soit postfixées

L'opération est la même, simplement la valeur de retour sera différente:

- dans le cas `++i`, on incrémente, *puis* on évalue le résultat et on le renvoie
- dans le cas `i++`, on évalue la variable, on incrémente, mais on renvoie la variable *avant incrémentation*

Il est possible (et même recommandé) d'utiliser la même distinction avec des opérateurs surchargés. La convention adoptée par le langage est d'effectuer deux déclarations de fonctions différentes:

- `operator++()` pour la version *préfixée*
- `operator++(int)` pour la version *postfixée*

 Dans le cas de l'opérateur postfixé, on doit:

- faire une copie locale de l'objet
- renvoyer la copie de l'objet, donc impossible de renvoyer une référence.

d'où surcoût (qui peut ne pas être négligeable, suivant la taille de l'objet). Moralité: utilisez *toujours* la version préfixée, sauf nécessité absolue.

Les opérateurs ++ et -- servent à définir des itérateurs.

L'opérateur d'affectation

Affectation n'est pas initialisation

En dépit des apparences, les deux lignes de code ci-dessous ne sont pas équivalentes.

```
int A=4;
A=5;
```

En effet, la première ligne correspond à une déclaration de variable avec *initialisation*, alors que la seconde ligne correspond à une *affectation*. L'initialisation est une affectation *précédée d'une allocation de mémoire*. Dans le cas de l'initialisation, la fonction appelée est le *constructeur de copie*, dans le cas de l'affectation il s'agit de l'`operator=`. Afin d'éviter de réécrire du code, la manière habituelle de procéder est de définir une fonction privée de copie, fonction qui sera appelée par le constructeur *et* par l'opérateur d'affectation. Cela pourrait donner par exemple, pour notre objet `tableau`:

```
class tableau {
public:
  explicit tableau(int);
  tableau(const tableau&);
  tableau& operator=(const tableau&);
  ~tableau() {delete buffer;};

private:
  int bfrsz;
  char* buffer;
  void copie(const tableau&);

};

void tableau::copie(const tableau& b) {
  ... copier le buffer de b ...
};

tableau::tableau(int sz) {
  buffer = new char(sz);
};

tableau::tableau(const tableau b&) {
  buffer = new char(b.bfrsz);
  copie(b);
};

tableau& tableau::operator=(const tableau&) {
  delete buffer;
  bfrsz = b.bfrsz;
  buffer = new char(bfrsz);
  copie(b);
};

void main() {
  tableau B(1024);
  tableau A(1024);
  A=B;

};
```

 A la vue du code ci-dessus, on pourrait être tenté d'écrire pour l'opérateur d'affectation:

```
  tableau& tableau::operator=(const tableau& b) {
  delete *this;
  this = new tableau(b);
  return *this;
};
```

Je détruis l'objet puis je le reconstruis en utilisant le constructeur de copie... et ce faisant, je scie tout simplement la branche sur laquelle je suis assis !!! A éviter...

Auto références

Il est important de prévoir, dans les opérateurs d'affectation, le cas *a priori* stupide où une variable est affectée à elle-même: cela est un cas de figure tout-à-fait possible, par le jeu des pointeurs et des références. Or, `operator=` risque alors de provoquer un plantage: en effet, lors d'une autoréférence, l'expression `delete buffer` supprimera aussi bien le buffer de l'objet cible que le buffer de l'objet source... de sorte qu'on en vient à copier n'importe quoi dans la fonction `copie`. Le code correct pour `operator=` sera donc:

```
tableau& tableau::operator=(const tableau& b) {
  if (this != &b) {
     delete buffer;
     buffer = new char(b.bfrsz);
     copie(b);
  };
  return *this;
};
```

Le trio infernal

Le trio infernal est constitué par les trois fonctions suivantes:

- Constructeur de copie
- Opérateur d'affectation
- Destructeur

Si l'une de ces trois fonctions est inexistante, le compilateur en produira une version par défaut. Dans le cas du constructeur de copie ou de l'affectation, la version par défaut consiste en une simple copie membre à membre. De sorte que de nombreuses classes se contentent de la version fournie par défaut.

 *Si vous fournissez l'une de ces trois fonctions, fournissez les trois*. Sinon, gros risques de plantages, le compilateur se chargeant de fournir ses versions "à lui" de la ou des fonctions manquantes...

La mise en cage du trio infernal

Lorsque l'on écrit un objet, on peut parfaitement *empêcher* les utilisateurs de l'objet en question d'utiliser copie et constructeur de copie: pour cela, il suffit de les définir *dans la section private*. Ainsi, seul l'objet lui-même (c'est-à-dire *vous*, le concepteur) sera capable de les utiliser. Vous interdisez aux *utilisateurs* de l'objet toute copie de celui-ci. Dans ce cas, constructeur de copie et opérateur d'affectation peuvent d'ailleurs être des fonctions vides...

```
class unique {
private:
   unique& unique(const unique&) {};
   unique& operator=(const unique&) {};
   ...
```

Nous verrons lors du chapitre sur l'héritage une mise en cage un peu moins brutale de l'opérateur = .

Conversions et opérateurs

Conversions vers une classe

Nous avons défini un opérateur +, mais celui-ci ne nous permet que d'ajouter deux complexes entre eux. Et pourtant, le code suivant est valide:

```
complexe A(1,1);
float B=1;
complexe C;
C = A + B;
C = B + A;
```

En fait, dans un cas comme celui-ci, le compilateur cherche à effectuer des conversions de types. Puisque nous avons défini des valeurs par défaut pour les paramètres du constructeur, le compilateur sait générer un complexe à partir d'un entier. Il sait donc faire une conversion de types entier vers complexe. Toutes les conversions de types seront donc traitées à l'aide de constructeurs surchargés.

Conversions depuis une classe

Cependant, comment allons-nous effectuer une conversion de type *depuis* la classe `complexe` *vers* un type de base du langage ? La technique ci-dessus ne le permet pas, car le compilateur ne peut deviner *ce que signifierait* une telle conversion. Nous allons alors définir, puis utiliser, un *opérateur de conversion*. Dans le cas des nombres complexes, par exemple, nous pourrions considérer qu'une conversion d'un complexe vers un entier consiste à prendre la partie réelle du complexe. D'où la définition suivante:

```
class complexe {
     public:
       ...
       operator float() {return r;};
     private:
       ...
     };
```

A partir de maintenant, on peut faire une conversion de type comme on a l'habitude en C:

```
...

complexe J(1,0);
float I = (float)J;
```

✔ Lorsqu'on définit un `operator <type>`, il ne *faut pas* spécifier de type de retour. C'est un peu bizarre, mais assez logique, compte-tenu du fait que le type est déjà spécifié dans le nom de l'opérateur lui-même.

Autres opérateurs

Certains opérateurs seront évoqués un peu plus loin:

- << et >> sont utilisés pour les entrées-sorties
- *, ->, ->*, `new` et `delete` permettent de définir des fonctions avancées de gestion de mémoire
- `[]` permet de définir des opérateurs d'accès aux tableaux.
- `()` permet de simuler des accès à des tableaux multidimensionnels
- `()` permet également de définir des objets fonctions: un objet fonction est un objet dont *la seule raison d'être* est

d'encapsuler un appel de fonction. cf. les exercices sur ce chapitre pour plus de détails, et le chapitre sur la bibliothèque standard, qui fait largement appel à cette notion d'objets fonctions

---

Langage	Surcharge d'une fonction	Valeurs par défaut des arguments	Surcharge des opérateurs
C++	**OUI**	**OUI**	**OUI**
perl	**Possible (1)**	**OUI**	**Possible (2)**
java	**OUI**	NON	NON
python	NON	**OUI**	**OUI**

1. En perl, le typage des données n'existe pas vraiment... du coup, le passage d'arguments est assez "souple" pour qu'il soit possible de surcharger une fonction, même si le langage ne le prévoit pas explicitement
2. La surcharge des opérateurs est possible, en perl, par l'intermédiaire du module OVERLOAD. D'autre part, le fonction `tie` peut être considérée comme une sorte de surcharge des opérateurs d'accès aux tableaux.

Héritage

Jusqu'à présent, nous avons vu comment définir des classes, qui permettront de créer des objets dans notre programme. Or, pour l'instant, nous ne pouvons exprimer que certaines relations entre nos objets: les relations d'association, d'agrégation et de composition, qui s'expriment par le fait qu'une donnée membre d'un objet peut être elle aussi un objet. L'héritage est l'outil qui va nous permettre d'exprimer entre nos classes une relation de type "est une sorte de". Cet outil est très puissant, car grâce à lui nous pourrons déclarer des classes d'objets très généraux, puis progressivement "spécialiser" ces classes d'objets. Cette spécialisation est aussi une "extension" de la classe de base: si l'on reste dans les généralités, on n'a pas grand-chose à dire. Plus les choses se précisent, plus on doit détailler. Donc à chaque étape du processus, on rajoutera du code. Le fait de partir d'une classe de base générale permet deux choses fondamentales intervenant à deux niveaux différents dans le processus de développement:

- *niveau conception:* Exprimer directement dans le code des concepts ayant un haut niveau d'abstraction.
- *niveau codage:* Coder quelques fonctions ou données qui seront "réutilisées" dans les classes dérivant de la classe de base

 Il est important de considérer *en priorité* les arguments de conception avant de définir une relation d'héritage. La réutilisation du code lorsqu'on en sera au codage doit être la conséquence d'une bonne conception, pas l'inverse.

L'héritage simple

Reprenons l'exemple des machines à café abordé lors de l'introduction à la programmation objet pour les gens normaux. Imaginons un objet permettant de modéliser une machine à café, modèle de base, mais avec du café soluble:

```
class modele_soluble {
public:
  int faire_le_cafe();
  bool plus_de_sucre();
  int lire_etat();

  private:
  reservoir_eau r;
  reservoir_sucre s;
  reservoir_cafe f;
  reservoir_cuillers c;
  reservoir_gobelets g;

  verser_un_verre_deau();
  verser_une_dose_de_cafe();
  donner_une_cuiller();
  donner_un_gobelet();
  verser_une_dose_de_sucre();
}
```

Or, dans la gamme de machine, il exite un modèle à peine plus sophistiqué, qui permet *aussi* de faire du chocolat. A part cette fonctionnalité supplémentaire, la machine est identique. Pourquoi se fatiguer à tout réécrire ? Le C++ nous offre la possibilité d'exprimer justement cela: notre nouvel objet hérite du premier; cela s'écrit ainsi:

```
class modele_choco: public modele_soluble {
public:
  int faire_le_chocolat();
  int lire_etat();
```

```
private:
  reservoir_chocolat o;
  verser_une_dose_de_chocolat();
}
```

La relation d'héritage peut aussi se représenter graphiquement:

```
  modele_soluble
       ^
       |
       |
       |
  modele_choco
```

ou encore de manière plus compacte:

```
modele_soluble
     modele_choco
```

C'est cette dernière représentation que nous utiliserons dans l'avenir.
On dit que la classe `modele_choco` est *dérivée* de `modele_soluble`, `modele_soluble` étant (comme son nom l'indique) une *classe de base* par rapport à `modele_choco`. `modele_choco` est en quelque sorte un "sous-type" de `modele_soluble`, et les variables de type `modele_choco` sont *aussi* des variables de type `modele_soluble`. Les `modele_choco` sont donc des *sortes de* modele_soluble, ils constituent *aussi* une *extension* de `modele_soluble`
On trouve dans la section publique de `modele_choco` deux fonctions:

- `faire_le_chocolat`: cette fonction est nouvelle pour ce modèle.
- `lire_etat`: cette fonction existait déjà dans le `modele_soluble`, mais c'est une nouvelle version que nous proposons dans `modele_choco`: en effet, il s'agit ici de lire l'état des différents réservoirs afin de savoir s'ils ne sont pas vides: il faut lire l'état du réservoir de chocolat, ce qui justifie qu'on réécrive une nouvelle version de cette fonction. Peut-être s'agit-il d'une réécriture complète, peut-être la nouvelle fonction `lire_etat` va-t-elle appeler la fonction `lire_etat` de la classe de base, avant d'exécuter le code spécifique au `modele_choco`.

Accès aux données: la section protected

Tel qu'il est défini `modele_choco` ne fonctionnera pas: en effet, les données `private` de `modele_soluble` *ne sont pas accessibles* par les classes dérivées. De ce fait, la fonction `faire_le_chocolat` n'aura pas accès au réservoir d'eau... gênant.

✔ On peut être surpris, à première vue, que les classes dérivées n'aient pas accès à la section `private` de leurs classes de bases... et pourtant, c'est la moindre des choses:

- Cela serait trop simple de "casser" le mécanisme d'encapsulation simplement en faisant une classe qui dérive d'une autre... les sections `private` n'auraient dès lors plus aucune raison d'être.
- Très souvent, c'est *l'utilisateur d'une bibliothèque d'objets* qui va dériver de nouveaux objets d'un objet de la bibliothèque. Or, il serait très imprudent pour celui-ci d'utiliser des données de la section `private`: en effet, celles-ci peuvent changer du jour au lendemain; seul l'interface est "garanti" par le concepteur de la bibliothèque.

L'idéal pour une classe dérivée est donc de n'utiliser que les fonctions-membres publiques de la classe de base, au même titre que n'importe quelle autre fonction. Cependant, certaines fonctionnalités ou certaines données n'ont pas à être utilisées par tout le monde, mais seulement par les classes dérivées: ainsi dans notre cafetière, tous les modèles de cafetière doivent avoir accès au réservoir d'eau, mais les *utilisateurs* de la machine à café n'ont pas de raison, eux, d'y avoir accès. D'où la nécessité de créer une nouvelle section: `protected`; tout ce qui sera déclaré dans cette section sera utilisable par les classes dérivées, *et uniquement par elles*. Ainsi, dans notre exemple, on mettra dans `protected` des fonctions d'accès aux éléments constitutifs de la machine à café (`get_reservoir_eau()`). On considère qu'une bonne conception doit respecter les règles suivantes:

- Dans la section `private`, se trouvent les données encapsulées par l'objet, ainsi que quelques fonctions membres correspondant au conctionnement interne de l'objet.
- Dans la section `protected`, ne se trouvent **que** des fonctions-membres.
- Dans la fonction `public`, ne se trouvent **que** des fonctions-membres

D'où une nouvelle définition de notre `modele_soluble`:

```
class modele_soluble {
public:
  int faire_le_cafe();
  bool plus_de_sucre();
  int lire_etat();

protected:
  verser_un_verre_deau();
  verser_une_dose_de_cafe();
  donner_une_cuiller();
  donner_un_gobelet();
  verser_une_dose_de_sucre();

private:
  reservoir_eau r;
  reservoir_sucre s;
  reservoir_cafe f;
  reservoir_cuillers c;
  reservoir_gobelets g;

}

class modele_choco: public modele_soluble {
public:
  int faire_le_chocolat();
  int lire_etat();

protected:
  verser_une_dose_de_chocolat();

private:
  reservoir_chocolat o;
}
```

On voit que les fonctions permettant d'agir sur le réservoir d'eau, de sucre, etc. sont passées de la section `private` à la section `protected`. Il en est de même pour la fonction `verser_une_dose_de_chocolat()` du `modele_choco`: cela permettra éventuellement de reprendre le processus de dérivation et de concevoir le `modele_capuccino` qui propose également du capuccino:

```
class modele_capuccino: public modele_choco {
public:
  int faire_le_capuccino();
}
```

Le capuccino étant un mélange de café et de chocolat, la fonction `faire_le_capuccino` a en effet besoin des *deux* fonctions `verser_une_dose_de_chocolat()` et `verser_une_dose_de_cafe()`. Le faire que ces dernières soient déclarées dans des sections `protected` des classes de base de `modele_capuccino` autorise cet accès, tout en l'interdisant aux programmes utilisateurs de l'objet, qui n'ont besoin *que* des fonctions interfaces (`faire_le_capuccino()` en l'occurrence).
En conclusion, nous retiendrons qu'il est *toujours* possible de dériver une classe à partir d'une autre. La classe dérivée aura

toujours accès aux fonctions publiques de sa classe de base. Mais il est souhaitable de *prévoir* la dérivation, en définissant une section `protected` "réservée" aux classes dérivées.

Constructeurs...

En reprenant les classes définies ci-dessus, que se passe-t-il lorsqu'on écrit:

```
modele_capuccino machine_1;
```

Il se passe les choses suivantes:

1. Allocation de mémoire.
   Autant d'octets que nécessaire compte-tenu des champs de `modele_capuccino` *et* de ses classes de base.
2. Appel du *constructeur par défaut* de `modele_soluble`
3. Appel du *constructeur par défaut* de `modele_choco`

Mais lorsqu'on écrit:

```
modele_capuccino machine_2 = machine_1;
```

il se passe:

1. Allocation de mémoire
2. Appel du *constructeur par défaut* de `modele_soluble`
3. Appel du *constructeur par défaut* de `modele_choco`
4. Appel du *constructeur de copie* de `modele_capuccino`

ce qui est probablement un bogue: en effet, il aurait été préférable d'appeler le constructeur de copie de `modele_soluble` et de `modele_choco`. *Le compilateur n'appellera de lui-même que le constructeur par défaut*, aussi faut-il lui dire *explicitement* quel constructeur appeler. Cela peut se faire dans les listes d'initialisation de `modele_capuccino` et de `modele_choco`:

```
class modele_capuccino: public modele_choco {
public:
   modele_capuccino(const modele_capuccino& mac) : modele_choco(mac) {...};
...
}
class modele_choco: public modele_soluble {
public:
   modele_choco(const modele_choco& mac) : modele_soluble(mac) {...};
...
}
```

Les expressions `modele_choco(mac)` et `modele_soluble(mac)` sont correctes, puisque `mac` étant un modele_capuccino, `mac` est *aussi* un `modele_soluble` *et* un `modele_choco`.

✔ Si on utilise les contructeurs de copie par défaut (ce est le cas, sauf lorsque l'on ne peut faire autrement), tout se passe bien.

...et destructeurs

Lorsque `Machine_a_cafe` est détruit, il se passe la séquence suivante:

1. Le destructeur de `modele_capuccino` est appelé
2. Le destructeur de `modele_choco` est appelé
3. Le destructeur de `modele_soluble` est appelé
4. La mémoire est rendue au système

✔ Si on utilise avec des pointeurs ou des références, on peut avoir quelques surprises, qui nécessitent l'utilisation de destructeurs virtuels.

Fonctions-membres

Observons le code ci-dessous, dans lequel on gère trois machines à café, de trois types différents, par l'intermédiaire d'un tableau de pointeurs:

```
modele_soluble* machines[3];
machines[0] = new(modele_soluble);
machines[1] = new(modele_choco);
machines[2] = new(modele_capuccino);
for (int i=0; i<3; ++i) {
    machines[i]->lire_etat();
};
```

Le problème avec le code ci-dessus est le suivant: puisque nous avons déclaré notre tableau `machines` comme un tableau de `modele_soluble*`, ce sera la fonction `lire_etat()` de `modele_soluble` qui sera appelée à chaque fois. Ce fonctionnement n'est pas cohérent, et conduira à une mauvaise lecture de l'état des machines, au moins dans le cas des machines 1 et 2: l'état du réservoir de chocolat ne sera jamais rapporté. Pourtant, cela devrait marcher, car une machine du type `modele_choco` est *aussi* une machine du type `modele_soluble`, par définition de l'héritage.

Fonctions virtuelles

Il est possible de remédier à ce problème, comme on le voit dans le programme ci-dessous. Nous avons modifié la déclaration des trois classes de machines à café. Seules figurent ici les déclarations de `lire_etat()`, le reste étant inchangé.

```
class modele_soluble {
public:
  ...
  virtual int lire_etat();

private:
  ...
}

class modele_choco: public modele_soluble {
private:
  ...

public:
  ...
  virtual int lire_etat();
}

class modele_capuccino: public modele_choco {
public:
  ...
  int lire_etat();
}

modele_soluble* machines[3];
machines[0] = new(modele_soluble);
machines[1] = new(modele_choco);
machines[2] = new(modele_capuccino);
```

```
for (int i=0; i<3; ++i) {
    machines[i]->lire_etat();
};
```

Cette fois la ligne `lire_etat()` correspondra non pas seulement à un appel de `modele_soluble::lire_etat()`, mais pour les machines 1 et 2 à un appel de `modele_choco::lire_etat()` . En effet, même si `machines` est déclaré comme un tableau de `modele_soluble*`, on a mis dans le tableau des pointeurs vers des objets *de types `modele_choco` ou `modele_capuccino`*. Le mot-clé `virtual` placé devant la fonction `lire_etat()` indique au compilateur qu'il doit laisser tomber le *typage statique* au profit de *l'édition de liens dynamique* lors de l'appel de cette fonction..
Ce mécanisme fondamental va nous permettre de réaliser la gymnastique suivante:

- Déclarer des pointeurs, ou des paramètres de fonctions en tant que classe de base
- Passer à la fonction un objet d'une classe dérivée: l'ordinateur retrouve ses petits, il connait le type de l'objet réel lors de l'exécution.

C'est *au moment de l'exécution* que la machine connaît le type précis de l'objet (lors de la compilation, on sait pas a priori qui va appeler la fonction). D'où le terme *édition de liens dynamique*. Remarquons qu'une fois de plus, on retrouve une manière de penser parfaitement naturelle: si je vous passe une casserole (classe de base) en vous demandant de la laver (fonction qui opère sur le type générique casserole), je vous passe en réalité une casserole bien particulière, et pas toujours la même (hier c'était le vieux chaudron à confiture qui me vient de ma grand-mère, aujourd'hui c'est une casserole en aluminium, dans les deux cas il s'agit d'un type de casserole particulier). Dans les deux cas vous allez la laver... mais suivant le type de casserole, vous vous y prendrez différemment.

✔ Lorsque vous redéfinissez une fonction virtuelle, **la nouvelle définition doit avoir la même valeur de retour** que la fonction originale. Sinon, le compilateur refusera votre code.

🔷 Si vous surchargez une fonction virtuelle, la fonction surchargée ne sera pas virtuelle automatiquement: il vous faut le spécifier explicitement.

Fonctions virtuelles pures

Nous avions envisagé précédemment la possiblité d'avoir des cafetières avec du café en grain et d'autres avec du café moulu. Hériter depuis `modele_soluble` n'aurait pas de sens pour définir ces nouveaux types de machines à café: en effet, on ne désire pas disposer d'une machine qui puisse faire à la fois du café soluble et du "vrai" café, on désire *remplacer* le café soluble par du vrai café. D'autre part, on désire conserver le principal, à savoir garder quelque part la caractéristique machine à café. D'où une nouvelle classe, appelée `cafetiere`:

```
class cafetiere {
public:
  bool plus_de_sucre();
  virtual int faire_le_cafe()=0;
  virtual int lire_etat()=0;

protected:
  verser_un_verre_deau();
  donner_une_cuiller();
  donner_un_gobelet();
  verser_une_dose_de_sucre();

private:
  reservoir_eau r;
  reservoir_sucre s;
  reservoir_cuillers c;
  reservoir_gobelets g;

}
```

```
class modele_soluble: public cafetiere {
public:
  virtual int faire_le_cafe();
  virtual int lire_etat();

protected:
  verser_une_dose_de_cafe();

private:
  reservoir_cafe_soluble f;
}

class modele_poudre: public cafetiere {
private:
  reservoir_cafe_en_poudre f;

protected:
  preparer_cafe_soluble();

public:
  virtual int faire_le_cafe();
  virtual int lire_etat();
}

class modele_grain: public modele_poudre {
public:
  virtual int faire_le_cafe();
  virtual int lire_etat();

protected:
  moudre_le_cafe();

private:
  reservoir_cafe_en_grain g;
}
```

La classe `modele_soluble` est elle-même dérivée d'une classe que j'ai appelée **cafetiere**; cela dit bien ce que cela veut dire: on dit au compilateur ce que devra savoir faire cette machine pour être considérée comme une machine à café. Le minimum est qu'elle sache... faire du café. D'où la fonction virtuelle `faire_le_cafe`. Mais comme nous aurons plusieurs manières de faire le café, nous nous gardons bien à présent de lui dire *comment* faire le café; autrement dit, nous ne faisons que *déclarer* la fonction `faire_le_cafe`, nous ne la définissons pas. L'expression **`=0`** indique au compilateur qu'il devra chercher la définition de la fonction dans une classe dérivée.
Deux classes dérivent de `cafetiere`: `modele_soluble`, que nous avons déjà rencontré, et `modele_poudre`. Ces deux classes définiront les fonctions `faire_le_cafe` et `lire_etat`, qui restent des fonctions virtuelles, afin que nous puissions sans encombre continuer le processus de dérivation. Et en effet, la dernière classe `modele_grain` hérite du modèle de café en poudre: les mécanismes sont bien les mêmes, simplement il faut moudre le café avant de le préparer.
Le diagramme des classes devient alors le suivant:

```
cafetiere
    modele_soluble
         modele_choco
              modele_capuccino
```

```
    modele_poudre
         modele_grain
```

Grâce au jeu des fonctions virtuelles, le code ci-dessous permettra *toujours* de préparer le café, quelque soit le modèle de cafetière:

```
cafetiere machines* [10];
... initialiser le tableau machines* ...
for (int i=0; i<10; i++) {
    cafetiere &c=*machines[i];
    if (c.lire_etat()) {       // la machine est-elle prête ?
        c.faire_le_cafe();
    } else
        cout << "La machine n'est pas prête, revenez plus tard\n";
};
```

La présence des fonctions virtuelles pures dans la classe `cafetiere` est indispensable: en effet, sans elles, le code `c.lire_etat()` et `c.faire_le_cafe()` *ne serait pas compilé*: le compilateur a besoin de savoir que ces fonctions existent de dans la classe `cafetiere`. Sinon, comment pourrait-il savoir que je n'ai pas tout simpement tapé une erreur ? Toutefois, à chaque appel de la fonction `cafe`, la bonne version de `faire_le_cafe` ou de `lire_etat` sera appelée, de sorte que le café sera toujours parfaitement préparé: c'est magique.

Classes de bases abstraites

Le code suivant ne sera jamais compilé:

```
  cafetiere C1;
  cafetiere tabC1[10];
```

En effet, la présence des fonctions virtuelles pures `faire_le_cafe` et `lire_etat` empêche le compilateur de générer un objet de type cafetiere, et a fortiori un tableau d'objets.En d'autres termes, *on ne peut pas* dire au compilateur "donne-moi de la mémoire et initialise cette cafetière". Il veut savoir *précisément* de quoi il s'agit. La classe `cafetiere` est une *classe de base abstraite*, (c'est-à-dire une classe qui a *au moins une* fonction virtuelle pure). En tant que telle, on peut la passer en paramètres (par référence ou par pointeur) à une fonction, mais on ne peut pas déclarer d'objet de cette classe.
Ainsi, le code suivant ne pose aucun problème:

```
  cafetiere* C1;
  cafetiere* tabC1[10];
  tabC1[0] = new(modele_soluble);
  tabC1[1] = new(modele_grain);
  ...
```

En effet, il est toujours possible de déclarer un pointeur sur une classe abstraite. Le type *réel* de la classe est précisé lorsque le pointeur sera initialisé, le plus souvent en utilisant l'opérateur `new`.

Constructeurs virtuels...

ça n'existe pas: le principe d'une fonction virtuelle est que la fonction *réellement* appelée à l'exécution est la fonction correspondant au type de l'objet existant. Encore faut-il que l'objet soit existant, ce qui n'est pas le cas lors de l'appel d'un constructeur.

...et destructeurs virtuels

là, ça existe: et c'est même fort utile. En effet, le code suivant risque d'entraîner des résultats catastrophiques:

```
for (int i=0; i<10; i++) {
    delete machines[i];
```

Le problème ici est que *seul* le destructeur de la classe de base a été appelé. Si le destructeur de la classe `modele_poudre` devait faire quelque chose de particulier (rendre la mémoire allouée pour le `reservoir_cafe_en_poudre`, par exemple), c'est raté. Le destructeur n'a pas été appelé, et il y a un réservoir à café qui traîne au fond de la mémoire... Il est donc toujours préférable de déclarer pour la classe `cafetiere` un destructeur virtuel... quitte à ce que celui-ci ne fasse rien:

```
class cafetiere {
  virtual ~cafetiere() {};
};
```

 On pourrait bien sûr considérer que tous les destructeurs doivent être virtuels. Malheureusement, déclarer une fonction virtuelle prend des ressources... c'est d'ailleurs pour cela que toutes les fonctions ne sont pas virtuelles. Généralement, on peut considérer que la déclaration de la *première* fonction virtuelle prend pas mal de ressources, les fonctions suivantes nettement moins. D'où la recommandation suivante: *toute classe abstraite, ainsi que les classes qui en dérivent, doivent avoir un destructeur virtuel.* Une classe non abstraite ayant peu ou pas de classes dérivées le constructeur virtuel a moins de chances d'être indispensable

Appel de fonctions virtuelles depuis le constructeur ou le destructeur

L'exemple ci-dessous peut poser un problème:

```
class modele_poudre: public cafetiere {
public:
  modele_poudre() { ... lire_etat();};
}

class modele_grain: public modele_poudre {
public:
  modele_grain() { ...lire_etat();};
};

main() {
   modele_grain G;
}
```

En effet, lors de la construction de la classe `modele_grain`, c'est le *constructeur de `cafetiere`, puis celui de `modele_grain`* qui sont appelés d'abord. Or, ce dernier appelle la fonction virtuelle `lire_etat()`. Est-ce donc la version de la classe dérivée `modele_grain` qui sera appelée ? Surtout pas: lors de l'exécution du constructeur de `modele_poudre`, la partie de l'objet spécifique à `modele_grain` n'est *pas encore* initialisée... on risquerait donc d'avoir des plantages ou autres erreurs inexpliquées. D'où la règle suivante: c'est *la fonction de la classe de base* (ici la classe `modele_grain`) qui sera appelée par le constructeur. Il en est de même pour le destructeur.

 Afin de clarifier le code (la règle ci-dessus est tout de même assez subtile), on recommande de *qualifier complètement* les fonctions virtuelles dans les constructeurs ou les destructeurs en utilisant l'opérateur de portée `::`. On écrirait donc :

```
class modele_grain: public modele_poudre {
public:
  modele_grain() {modele_grain::lire_etat();};
};
```

L'opérateur d'affectation dans une classe de base

Si l'opérateur d'affectation (`operator=()`) d'une classe dérivée doit être défini (c'est-à-dire si l'opérateur d'affectation fourni par le compilateur n'est pas suffisant), celui-ci *doit* appeler l'opérateur d'affectation de sa classe de base: sinon, *tous les champs ne seront pas copiés !!!*. Par contre, s'il n'est pas nécessaire de définir un opérateur d'affectation particulier pour une classe dérivée, le compilateur se chargera correctement d'appeler l'opérateur d'affectation de la classe de base.

```
class modele_grain: public modele_poudre {
public:
    modele_grain& operator=(const modele_grain& c) {
       modele_poudre::operator=(c);  // copie de la partie commune
       g = c.g;                      // copie du reservoir de café en grains
    };
}
```

Protéger l'opérateur d'affectation d'une classe de base abstraite

L'opérateur d'affectation d'une classe de base abstraite doit normalement être déclaré *dans une section* `protected`, quitte à déclarer un `operator=` *uniquement dans ce but*. En effet, par défaut l'`operator=` est public. Mais pour une classe abstraite, cela risque de conduire à des catastrophes:

```
void f(cafetiere& c1, const cafetiere& c2) {
   c1 = c2;
};

main() {
  modele_capuccino P;
  modele_grain G;
  f (P,G);
}
```

Cela aboutit à écrire une égalité entre une machine à café en grains et une machine à capuccino (café soluble+chocolat). Il est clair que cela ne peut pas fonctionner... et pourtant le compilateur a bêtement accepté. D'où le code suivant:

```
class cafetiere {
protected:
    void operator=(const cafetiere& c);
}
```

L'opérateur = étant déclaré dans la section protected, il peut être utilisé par les classes dérivées, *mais il est inaccessible pour le reste du monde*. De sorte que la fonction suivante:

```
void f(cafetiere& c1, const cafetiere& c2) {
   c1 = c2;
};
```

ne sera pas compilée

✔ Lorsque `opérator=` est `protected`, il n'est pas nécessaire qu'il renvoie quelque chose, puisqu'il ne sera utilisé *que* indirectement par les classes dérivée. D'où le `void` comme type de retour de la fonction.

Autres langages objets...

Langage	Héritage	mot-clé virtual	Classe abstraite	Héritage multiple
C++	**OUI**	**OUI**	**OUI**	**OUI**
perl	**OUI**(1)	NON(3)	**Possible** (6)	**OUI**
java	**OUI**(2)	NON(3)	**OUI** (4)	NON(5)
python	**OUI**	NON(3)	**Possible** (6)	NON(5)

1. Utilisation du tableau `@ISA` (Est une sorte de)
2. Mot-clé `extends` (étendre les fonctionnalités de)
3. En java, toutes les fonctions sont des fonctions virtuelles
4. On définit une classe abstraite et on spécifie les fonctions qui doivent être redéfinies, par le mot-clé `abstract`
5. Mais les mots-clés `interface` et `implementation` correspondent aux mêmes fonctionnalités que l'héritage multiple.
6. Pour définir une classe abstraite en perl ou en python, il suffit de s'arranger pour que certaines fonctions (les "fonctions membres virtuelles pures") génèrent une exception.

Les modèles

Nous sommes maintenant capables d'exprimer plusieurs types de relations entre objets:

- Associations, agrégation, composition:
- Est une sorte de

Nous allons voir dans ce chapitre comment exprimer la relation: *Est une liste de*, ou *Est un tas de*, etc.; nous avons défini au début du cours une classe appelée `complexe`, qui comprend essentiellement deux membres privés, `r` et `i`. Or, ces deux nombres étaient définis comme des `float`, ce qui peut paraître un peu restrictif... Ne faudrait-il pas plutôt définir les parties réelle et imaginaire des complexes comme des double, pour avoir une meilleure précision lors des calculs numériques ? Nous ne voulons pas avoir à choisir, en tous cas *pas au moment d'écrire l'objet* `complexe`: les algorithmes utilisés seront les mêmes, quelque soit le type des champs utilisés pour la partie réelle et pour la partie imaginaire. D'où la notion de *modèles*.
Les modèles sont très utilisés pour créer des objets de type conteneurs: ce sont des objets qui vont en "contenir" d'autres. Nous en reparlerons dans le chapitre sur la bibliothèque standard

Classes paramétrées

Définition d'une classe paramétrée

Voici la nouvelle définition de la classe `complexe`, en utilisant des modèles:

```
template<class NUM=float> class complexe {
public:
  complexe(NUM x=0, NUM y=0);
  complexe(const complexe<NUM> & );
  ~complexe();
  operator NUM();
  complexe<NUM> & operator=(const complexe<NUM> &);
  complexe<NUM> & operator+= (const complexe<NUM> &);
  NUM get_r() const { return r;};
  NUM get_i() const { return i;};
  void set_r(NUM x) { r=x; m_flg=false;};
  void set_i(NUM x) { i=x; m_flg=false;};
  NUM get_m() const;
  static void set_debug() { debflg=true;};
  static void clr_debug() { debflg=false;};

private:
  NUM r;
  NUM i;
  mutable bool m_flg;
  mutable NUM m;
  static bool debflg;
  void _calc_module() const {m=sqrt(r*r+i*i);};
};
```

Le nom de classe est précédé par le mot-clé `template<class NUM>` (que l'on traduit par *modèle*,ou *patron*), ce qui signifie qu'un certain *type* sera utilisé lors de l'instantiation de cette classe. D'autre part, dès que le nom complexe est utilisé pour désigner la classe qui est en train d'être définie, on *doit* citer le paramètre, d'où la notation un peu lourde

`complexe<NUM>` partout dans la définition des paramètres.

✔ Dans l'expression `class NUM`, le mot `class` signifie `type`... On peut utiliser n'importe quel type, que ce soit un type défini en tant que classe ou que ce soit un type prédéfini. Une autre syntaxe peut d'ailleurs être utilisée, elle consiste à remplacer `class` par `typename`

✔ Lorsqu'on *déclare* la classe elle-même, les paramètres ne doivent pas être spécifiés à nouveau derrière le mot-clé `class`. De la même manière, le nom des constructeurs ou du destructeur restent inchangés.

✔ Si vous définissez une fonction *en-dehors de la portée de la déclaration de classe*, vous devrez répéter `template<NUM>`. Sinon, le compilateur ne saura comment interpréter `complexe`**`NUM`**. Ainsi, dans l'exemple ci-dessus, le destructeur devra être défini de la manière suivante:

```
template <class NUM> complexe<NUM>::~complexe()
```

Ce qui signifie:

- Indiquer que `NUM` correspond à un template en cours de définition
- Le nom de la classe est `complexe<NUM>`
- Le nom de la fonction est `~complexe` (destructeur, en l'occurrence)

Paramètres par défaut

Dans le code ci-dessus, on remarque la structure `class NUM=float`, qui revient à donner à `NUM` une valeur par défaut (`int` en l'occurrence, afin de retrouver le même fonctionnement que ci-dessus).

Instantiation

Le programme principal ressemblera à ce qui suit:

```
typedef complexe<>  complexe_float;
typedef complexe<double> complexe_double;

main() {
  complexe_float F;
  complexe_double D;
}
```

Le `typedef` ne fait *rien d'autre* que de déclarer un *synonyme*. Autrement dit, il ne crée pas un nouveau type. Son rôle n'est pas indispensable, mais simplement *très souhaitable* pour la lisibilité du code: en effet, la syntaxe `complexe<float>` se révèle difficilement lisible.
Dans le cas des flottants, il suffit de déclarer `complexe<>`, le symbole `<>` étant là pour rappeler qu'il s'agit bien d'un modèle.

Paramètres utilisables

On peut mettre plusieurs paramètres dans les modèles. Ces paramètres peuvent être de deux sortes:

- Le mot-clé `class` suivi d'un nom de type existant (prédéfini ou non).
- Un type prédéfini (généralement `int`) suivi d'un nombre, pour exprimer par exemple une dimension.

✔ On peut aussi mettre en paramètre un paramètre de modèle déjà défini au préalable. Par exemple:

```
template<class T, T VALEUR> class machin {
   ...
}
```

commence par définir `T` comme un type (`int` ou `float`, par exemple), puis VALEUR est un paramètre de même type que T. On pourra alors instantier le modèle par:

```
machin<int,6> m1;
machin<float,3.14> m2;
```

Exemple: La classe tableau

Ainsi, la classe `tableau` définie ci-dessus peut être réécrite en utilisant un modèle:

```
template<size_t BFRSZ> class tableau {
  char buffer[BFRSZ];
  void copie(const tableau<BFRSZ> &);
public:
  tableau(){};
  tableau(const tableau &);
  tableau<BFRSZ> & operator=(const tableau<BFRSZ> &);
  ~tableau() {delete buffer;};
};

template<size_t BFRSZ> tableau<BFRSZ>::tableau(const tableau & b ) {
  copie(b);
  cout << "copie effectuee par const de copie\n";
};

template<size_t BFRSZ> void tableau<BFRSZ>::copie(const
tableau<BFRSZ> & b) {
  memcpy(buffer,b.buffer,BFRSZ);
};
```

Cette implémentation est intéressante, parce qu'on n'a plus besoin d'utiliser l'opérateur `new` dans le constructeur. Par contre, on ne peut plus choisir la taille du buffer lors de l'exécution du code, mais lors de la compilation.

Modèles de fonctions

Les fonctions peuvent, elles aussi, être déclarées par l'intermédiaire d'un modèle. Ainsi, une fonction `sort` qui implémente un algorithme de tri sur un vecteur (objet jouant le rôle de tableau) peut-elle être déclarée de la manière suivante:

```
template<class T> void sort(vector<T> &)
```

Dans ce cas, on trie un vecteur d'objets de types T. Ce que sera le type T n'est pas connu à ce stade: c'est lors de l'utilisation de la fonction que cela sera précisé, ainsi qu'on le voit ci-dessous:

```
vector<int> V1;
vector<string> V2;
...
sort(V1);   // trie le vecteur d'entiers V1
sort(V2);   // trie le vecteur de chaînes de caractères V2
```

Spécialisation

Il n'est pas toujours possible de s'en tenir à l'écriture du code général, tel que le modèle l'implémente: ne serait-ce que pour des raisons de performance, il est parfois nécessaire de réécrire le code pour certains types particuliers. Par exemple, un tableau d'objets peut parfaitement être implémenté sous forme de modèle (et *est* implémenté dans la bibliothèque standard,

d'ailleurs), mais on conçoit qu'un tableau de booléens devrait être radicalement différent, tout simplement parce que des booléens ne tiennent pas sur un ou plusieurs octets, mais bien sur un `bit`.

Quelques conseils...

Les modèles permettent de définir des objets de manière extrêmement générale, en ce sens ils constituent un outil très puissant. Mais, chaque médaille ayant son revers, ils sont d'une utilisation assez délicate. Il semble d'ailleurs que, encore actuellement, tous les compilateurs n'implémentent pas les modèles de manière complètement identique, ce qui n'est pas pour faciliter les choses...
Il est important de connaître la syntaxe des modèles, car elle est employée en permanence dans la bibliothèque standard: cela est tout-à-fait compréhensible, dans la mesure où une bonne partie de la bibliothèque standard est constituée de "conteneurs" et d'algorithmes associées, c'est-à-dire d'objets encapsulant des structures de données: seuls les modèles permettent de les décrire de manière générale
Quelques conseils, pour ne pas se noyer dans les modèles:

- *toujours* utiliser des `typedef`, afin de n'entrer qu'une seule fois le modèle lui-même avec toute sa complexité.
- Lors de *l'écriture* de nouveaux modèles, il convient de démarrer d'abord avec des types ordinaires (`int`, par exemple) ou des constantes littérales. Mettez au point votre code, *alors seulement* remplacez vos types et vos constantes par des paramètres, afin de gagner en généralité. Cette approche progressive permettra d'isoler les problèmes liés à l'utilisation des modèles, et les problèmes plus classiques liés à tout programme en cours d'écriture.

Langage	Modèles
C++	**OUI**
perl	NON
java	NON
python	NON

Exceptions

Que faire en cas d'erreur ?

Que le programme doit-il faire lorsqu'il découvre une condition d'erreur lors de son exécution: par exemple une division par 0, ou encore l'ouverture d'un fichier inexistant (ou qui ne peut pas être lu pour un problème de permissions), etc. Le problème est généralement le suivant: L'erreur se produit dans une bibliothèque; un objet doit ouvrir un fichier dont on lui a passé le nom en paramètres. L'objet voit bien qu'il y a une erreur (il n'a pas trouvé le fichier), mais comment doit-il réagir ? En fait, il ne *le sait pas*... parce que ce n'est pas à lui de réagir. C'est au programme utilisateur de l'objet: suivant les cas, la réaction de celui-ci sera soit l'arrêt du programme avec impression d'un message, soit tentative de reprise après avoir demandé à l'utilisateur un nouveau nom de fichier, soit génération automatique d'un nouveau nom de fichier, ... L'objet doit donc se borner à prévenir le programme appelant qu'il y a eu une erreur. Il y a trois moyens:

- Retourner un code d'erreur comme valeur de retour de la fonction.
- Mettre un code d'erreur dans une valeur globale (`errno` en C)
- Générer une exception.

Les deux premiers moyens présentent certains inconvénients: tout d'abord, si la fonction, dans son fonctionnement normal, doit déjà renvoyer une valeur, comment renvoyer le code d'erreur ? Par une valeur illégale peut-être, mais ce n'est pas toujours possible... ainsi une fonction qui doit renvoyer un entier ne peut renvoyer une valeur illégale. D'autre part, une bonne partie du code utilisateur risque d'être consacrée au traitement des erreurs... à condition que le programmeur ait assez de courage ou de conscience professionnelle. On estime que dans certains cas le code peut *doubler* simplement à cause du traitement d'erreurs. De manière plus fondamentale, on aura une totale imbrication du code de traitement d'erreurs et du code de l'application, d'où une relativement mauvaise lisibilité du code. Le C++ offre un système d'exceptions qui améliore considérablement la situation.

Une analogie avec la vie courante

Afin de bien comprendre le système des exceptions, imaginons l'administration des postes d'un pays quelconque: l'organisation est la suivante:

1. Service National des Postes
2. Service Régional des Postes
3. Bureau de Postes de quartier
4. Facteur

Il faut comprendre cette analogie de la manière suivante:

1. Le Service National des Postes correspond au programme principal
2. Le Service Régional correspond à une fonction appelée par le programme principal
3. Le Bureau de postes de quartier correspond à une fonction appelée par la fonction précédente
4. Le facteur est une méthode appartenant à un objet utilisé par la fonction précédente

Imaginons donc le facteur, en train de distribuer le courrier. La plupart du temps, tout se passe correctement et la mission du facteur est menée à bien. Mais quelques problèmes peuvent survenir; par exemple, une lettre adressée à M. Dupond est notée 105 rue des Mimosas, alors que les dupond habitent au 15 rue des Mimosas: le facteur connaît le quartier, il mettra l'enveloppe dans la boîte aux lettres des Dupond, même si l'adresse est mauvaise. Il s'agit d'un cas d'erreur qui a pu être corrigé par le facteur - par l'objet. Rien ni personne ne sera au courant qu'il y a eu un problème avec cette enveloppe. Autre problème possible: une lettre est adressée à M. Durand, or il n'y a pas de M. Durand dans le quartier. Cette fois, le facteur mettra la lettre dans une boîte appelée "Adresses inconnues", et il continuera sa tournée.

Voilà que le facteur tombe sur une lettre qui comporte la bonne adresse, mais il s'agit d'une rue située dans un autre quartier: le facteur mettra la lettre dans une nouvelle boîte, appelée "autres quartiers".
A la fin de sa tournée, le facteur regarde l'état de ses deux boîtes: si elles sont vides, il rentre chez lui tout simplement. La méthode "facteur" a fait son travail sans histoire. Si au moins l'une des deux est pleine, le facteur, avant de rentrer chez lui, va déposer à un endroit réservé à cet usage, au bureau de postes, le ou les cartons contenant les lettres en cause: il "lance une exception"; celle-ci sera traitée soit au niveau du bureau de poste du quartier, soit au niveau supérieur; *mais en aucun cas le facteur ne prend de décision à propos de cette lettre*: ce n'est tout simplement pas son travail.
Le bureau de poste de quartier, voyant qu'il y a une "exception", va alors la traiter: si l'adresse située sur la lettre "autres secteurs" correspond à un secteur géré par ce bureau de poste, il suffira de la donner à un autre facteur pour que le problème soit résolu. L'erreur a été corrigée au niveau Bureau de Poste, et personne à un plus haut niveau n'en saura rien. Sinon, le bureau de poste la renvoie à l'échelon supérieur (régional) qui se chargera du problème, à moins qu'il ne le renvoie à nouveau à un échelon supérieur...
C'est un système analogue qui est employé par le C++ pour traiter les exceptions:

- Si une fonction peut traiter l'exception, elle la traite.
- Si une fonction ne peut pas la traiter, elle renvoie un objet; celui-ci sera intercepté au niveau d'appel immédatement supérieur: à ce niveau de décider s'il sait traiter l'exception ou s'il la renvoie lui aussi au niveau immédiatement supérieur

Il est possible de renvoyer ainsi n'importe quel objet, et de mettre donc dans cet objet n'importe quelle information: un code d'erreur, par exemple, avec une chaîne de caractères explicative, mais aussi des données (d'autres objets, par exemple) permettant aux niveau supérieurs de traiter effectivement l'exception.

Le système d'exceptions

Supposons qu'on ajoute à notre type complexe un constructeur ayant pour objet l'initialisation du complexe à partir de données se trouvant dans un fichier. Vu qu'il s'agit d'un complexe un peu spécial, nous allons créer une nouvelle classe, `fcomplexe`, qui dérive de la classe `complexe.`. Le code du constructeur correspondant se trouve ci-dessous:

```
fcomplexe::fcomplexe(const char* fn) {
  ifstream input(fn);
  float x,y;
  input >> x;
  input >> y;
  input.close();
  set_r(x);
  set_i(y);
};
```

Il n'y a ici *aucun traitement d'erreur*. Si le fichier est inexistant, le résultat sera n'importe quoi. Si le fichier existe mais contient une seule ligne, la partie réelle sera correctement initialisée, mais pas la partie imaginaire. Voici comment nous pouvons ajouter un traitement d'erreur:

Les hiérarchies d'objets exceptions

Nous allons tout d'abord définir quelques objets, sous forme de classe ou de structure, généralement très simples, mais liés entre eux par une relation d'héritage. En particulier, tous dériveront de la hiérarchie d'exceptions standards, qui est partiellement décrite ci-dessous:

Les exceptions standards

Les exceptions ci-dessous sont définies dans la bibliothèque standard, elles sont disponibles dès que l'on a inclut le fichier

d'en-tête <stdexcept>.

```
exception
    logic_error
        domain_error
        length_error
        out_of_range
        invalid_argument
    runtime_error
        overflow_error
        underflow_error
        range_error
```

Les exceptions de type logic_error ne devraient normalement pas se produire, sinon peut-être en période de déboguage. Les exceptions de type runtime_error, par contre, ne peuvent pas toujours être évitées.
Toute exception dérivant de l'une des exceptions précédentes pourra fournir une fonction virtuelle appelée what(), qui renvoie une chaine de caractères décrivant l'exception. Par ailleurs, un constructeur recevant un argument de type chaine permettra de fixer la valeur de la chaine renvoyée par what. Bien sûr, toute autre extension est également possible.

Exceptions complexes

```
class CE : public runtime_error {
public:
   const char* what() const {return "Complexe exception";};
};

class CE_F: public CE {
public:
   CE_F(char* filename) : _name(filename) {};
   const char* what() const {
      string pb = "Complexe exception = File pb, filename = "+_name;
      return pb;
   };
protected:
   _name;
};

class CE_O: public CE_F {
 public:
  CE_O(char* filename, int e) : CE_F(filename), _err(e) {};
  int geterror() const { return _err;};
 private:
  int _err;
};

class CE_R: public CE_F {};
```

La classe CE définit les "exceptions complexes" de manière générale, c'est-à-dire que toutes les exceptions générées par la classe complexe seront de ce type. Ces exceptions peuvent être de plusieurs sortes: un problème de fichier, avec notre classe fcomplexe, mais aussi un problème numérique (division par zéro). Les problèmes de fichiers donneront lieu à une exception de type CE_F. A nouveau, plusieurs problèmes peuvent se présenter: fichier impossible à ouvrir parce que inexistant (CE_O), ou fichier ayant un mauvais format (CE_F). La classe CE_O est un peu plus compliquée que les autres, car elle contiendra le code d'erreur.

La déclaration de fonction

Bien que ce ne soit pas obligatoire, il est fortement conseillé de montrer lors de la déclaration qu'une fonction donnée est susceptible de générer une exception. Par exemple, dans le cas de l'objet `fcomplexe`, nous aurons la déclaration suivante:

```
class fcomplexe: public complexe {
  char* fn;
public:
  fcomplexe(const char* ) throw(CE);
  char* get_fn() const throw() {return fn;};
}
```

`throw(CE)` dans la déclaration du constructeur signifie qu'en cas de malheur, celui-ci est susceptible de générer une exception de type `CE`. La fonction `get_fn`, elle, ne peut générer aucune exception. Cela est indiqué par `throw()`.

✔ Il n'est pas obligatoire de mettre la spécification `throw` dans les déclarations de fonction. Cependant, cela est préférable: en effet, les utilisateurs de vos classes (vous ou un autre) sauront alors, simplement en lisant le fichier d'entête de la classe, quelles exceptions la fonction est susceptible de générer. Si `throw` n'est pas spécifié, ils seront obligés de lire le code source.

✔ Le compilateur vérifiera la cohérence entre déclaration et définition de fonction. Ainsi, la définition du constructeur de `fcomplexe`, devra ressembler à ce qui suit:

```
fcomplexe::fcomplexe(const char* fn) throw(CE) {
...
}
```

Par contre, le compilateur *ne vérifiera pas* si vous êtes menteur ou pas: en d'autres termes, s'il n'y a aucune instruction `throw` dans le corps de la fonction, aucune erreur ne sera générée et personne ne vous mettra en garde.

✔ Si vous avez spécifié une ou plusieurs exceptions dans la déclaration de fonction, et que c'est en fait *une autre* exception qui est générée, dans ce cas une fonction particulière, appelée `unexpected`, est appelée. Généralement, la conséquence de cette fonction est l'arrêt du programme.

La génération d'exception

L'exception est générée, dans le corps de la fonction, par la fonction `throw`. Le paramètre de `throw` *est un objet d'un type défini par l'utilisateur*. Voici le code de notre constructeur:

```
fcomplexe::fcomplexe(const char* fn) throw(CE) {
  ifstream input(fn);
  if (! input) {
    throw(CE_O(fn,input.rdstate())); // L3
  };
  float x,y;
  input >> x;
  if (! input) {      // L1
    CE_R e;
    throw(e);
  };
  input >> y;
  if (! input) {      // L2
    throw(CE_R());
  };
  input.close();
  set_r(x);
  set_i(y);
```

```
};
```

La ligne L1 montre une déclaration de variable de type CE_R, puis l'envoi de cette variable par la fonction throw. La ligne L2 montre exactement la même chose, mais en utilisant une variable temporaire, directement générée par l'appel du constructeur. C'est la forme standard de la génération d'exceptions. La ligne L3 montre la même chose avec une exception de type CE_0, cependant cette fois le code d'erreurs est passé en paramètres au constructeur.

✔ Pour passer un objet temporaire à la fonction throw, il faut appeler son constructeur. En conséquence, throw (CE_R) ne fonctionnera pas. Il faut bien spécifier l'appel du constructeur, à savoir throw(CE_R()).

La capture des exceptions

Notre constructeur va donc générer des exceptions le cas échéant, encore faut-il que la fonction appelante les "capture". Voici un exemple de fonction appelante, qui va demander à l'utilisateur un nom de fichier, jusqu'à ce que ce fichier puisse être ouvert:

```
fcomplexe* create_compl() {
  fcomplexe* fc;
  for (;;) {
    String fn;
    cout << "entrer un nom de fichier ";
    cin >> fn;
    try {
      fc = new fcomplexe(fn);
      break;
    }
    catch (CE_0 e) {
      cerr << "Erreur d'ouverture de fichier - etat " << e.err << "\n";
      cerr << "Entrez un autre nom\n";
      continue;
    }
  };
  return fc;
};

void main() {
  complexe::set_debug();
  fcomplexe* D = create_compl();
}
```

Tout se passe dans une boucle infinie (for(;;)), qui permettra la reprise en cas d'erreur d'ouverture de fichier. L'appel du constructeur (via l'opérateur new) est à l'intérieur d'un bloc try. Ce bloc est suivi (*attention*, le bloc try se termine par un }, *pas* par un };) d'un ou plusieurs blocs catch. catch prend un paramètre (de même qu'une fonction), le type de ce paramètre est le type d'exception que l'on cherche à capturer (ici CE_0). L'exception est traitée dans le bloc catch.

✔ En fait, plusieurs programmes de capture d'exceptions auraient pu être écrits, suivant la finesse avec laquelle on veut traiter les exceptions:

- On pourrait se contenter de capturer les exceptions de type CE
- Dans un traitement plus grossier, on peut capturer les exceptions de type run_time
- Dans un traitement encore plus grossier, on peut se contenter de capturer les exceptions de type exception.

🔷 **Règle d'or:** Si la syntaxe ne le rend pas obligatoire, il est important de bien définir les exceptions de manière hiérarchique, et de préférence comme des classes dérivées de la classe exception. Cela permet en effet le traitement hiérarchisé des exceptions, ainsi qu'on vient de le voir.

Exceptions non capturées

Et que se passe-t-il si une exception `CE_R` est générée ? Elle ne sera pas traitée: à la place, elle sera transmise à nouveau à la fonction appelante. Dans l'exemple ci-dessus, puisque `main` ne prévoit aucune capture d'exception, l'exception se terminera par un arrêt du programme. Mais il est parfaitement possible d'envisager un traitement des exceptions restant à capturer par la fonction `main`; le code de celle-ci devient alors:

```
void main() {
  complexe::set_debug();
  try {
    fcomplexe* D = create_compl();
    cout << *D << "\n";
  }
  catch (CE_F) {
    cout << "Erreur de format de fichier\n";
  }
  catch (...) {
    cout << "Autre exception\n";
  };
}
```

Le premier bloc `catch` correspond aux erreurs de format de fichier, qui peuvent effectivement être renvoyées par `fcomplexe`. Le second correspond à toutes les autres exceptions... en l'occurence pas grand-chose pour un programme aussi simple.

Renvoyer les exceptions

Un bloc `catch` peut parfaitement se terminer par le renvoi d'une exception. Cela permet par exemple de définir dans une fonction quelques traitements spécialisés et de renvoyer le traitement des exceptions restantes au code appelant. A cet effet, on peut appeler la fonction **`throw()`** qui renvoie l'exception en cours de traitement.

Exceptions et ...

...constructeurs

Le système des exceptions est *le système de traitement d'erreurs* à employer pour des constructeurs d'objet, *à l'exclusion de tout autre*: en effet, on pourrait par exemple imaginer une variable `err` qui indiquerait que l'objet est construit, certes, mais dans un état "bizarre", donc pas vraiment utilisable. C'est ce qu'on appelle les "objets zombies"... cela peut conduire à des comportements inattendus (variables internes non initialisées, par exemple), à moins que l'objet soit suffisamment bien écrit pour que toutes les fonctions-membres testent la valeur de `err` afin de s'assurer que l'objet n'est pas un zombie... mais dans ce cas, que de code inutile ! A l'inverse, si le constructeur est interrompu par une exception, *l'objet ne sera pas construit*... Or, un vrai mort vaut mieux qu'un faux zombie, qui ira prétendre le contraire ?
Si une exception a lieu dans le constructeur d'une classe dérivée, comme c'est le cas avec le type `fcomplexe`, le processus de construction est interrompu... mais le constructeur de la classe de base a déjà été appelé. Avant que la main soit rendue au bloc `catch` de la fonction appelante, le destructeur de la classe de base sera alors autmatiquement appelé, de sorte qu'on ne risque pas d'avoir de "fuite de mémoire".

...destructeurs

Le système des exceptions est le système de traitement d'erreurs *à ne pas employer* avec les destructeurs: en effet, un destructeur peut être appelé lors du déroulement normal du programme; mais il peut aussi être appelé lors de la génération *d'une autre exception*. Dans ce cas, le programme sera immédiatement arrêté.
Evidemment, rien n'empêche un destructeur d'appeler des fonctions qui, elles, sont suceptibles de générer une exception. Mais dans ce cas, ces appels de fonction doivent être encadrés par des blocs `try...catch`, et *aucune exception ne doit s'échapper* du destructeur. Cela signifie que les destructeurs, s'ils ont une erreur à faire remonter, devront trouver un autre système. Par exemple écrire sur une fenêtre ou dans un fichier de log.
 Cette dissymétrie peut paraître surprenante à première vue... mais en fait, en informatique comme dans la vie, il est bien plus simple de détruire que de construire, et surtout alors qu'on peut avoir du mal à construire une maison, rien ne devrait pouvoir vous empêcher de la détruire... De même, le constructeur peut rencontrer un grand nombre de problèmes (ressources impossibles à trouver, par exemple), mais normalement le destructeur ne *devrait pas* générer d'erreur... ou alors, c'est grave, car cela signifie que le système refuse de récupérer une ressource.

Langage	Exceptions
C++	**OUI**
perl	NON
java	**OUI**
python	**OUI**

Gestion de la mémoire

Ce paragraphe traite des pointeurs, des problèmes liés à l'allocation dynamique de mémoire, et des moyens qui existent de résoudre ces problèmes... plutôt d'éviter leur apparition, car en ce domaine le préventif est bien plus aisé que le curatif...

Qu'est-ce que l'allocation dynamique de mémoire ?

Nous avons vu précédemment que la durée de vie d'une variable s'étendait durant toute la portée de son nom. La mémoire est allouée, et le constructeur de l'objet est appelé en début de portée; le destructeur est appelé et la mémoire est rendue au système à la fin de portée. Les objets utilisés ainsi utilisent une partie de la mémoire vive appelée `la pile`. La structure de pile est en effet parfaitement adaptée à la gestion des règles de portée. Or, il peut être intéressant de stocker des données à des endroits de la mémoire qui ne seront pas sujets soumis aux règles de portée; cela peut se faire grâce à:

- d'une part l'utilisation de pointeurs
- d'autre part à des opérateurs permettant de gérer explicitement l'allocation mémoire et sa libération.

L'allocation-libération de mémoire étant à la charge du programmeur, elle peut se faire dans n'importe quel ordre. La structure de pile n'est alors plus adaptée, et de fait la mémoire est allouée dans une autre zône de la mémoire, appelée `le tas` (`heap`).

 Le programmeur doit *effectivement* gérer la libération de mémoire... s'il ne le fait pas, ou s'il le fait mal, les pires conséquences (à savoir un plantage du programme) peuvent arriver.

Pointeurs empilés, objets entassés

La seule zône de mémoire que le programme *peut adresser directement* est la pile. Les pointeurs se trouveront donc quelque part dans la pile, au même titre que n'importe quelle variable. Les *objets pointés*, par contre, se trouveront dans le tas. Il doit y avoir *en permanence* un lien entre ces deux zônes de mémoire. Garder ce lien intact est la première préoccupation d'une bonne gestion de la mémoire.

Objets perdus et fuites de mémoire

Lorsqu'un objet est alloué dynamiquement, *au moins un pointeur* doit pointer sur lui: sinon, le lien évoqué ci-dessus est brisé, et l'objet est inutilisable. On peut dire qu'il est perdu, mais surtout *la mémoire correspondante* est perdue. Avant de briser le lien, il aurait fallu rendre la mémoire au système. Suivant les cas de figure, cela peut être grave ou pas. Par exemple, si l'allocation de mémoire a lieu dans une boucle, à chaque itération de la boucle on perd un peu de mémoire... d'où l'expressoin fuite de mémoire. Si le nombre d'itérations est important, il y a un moment où le système refusera de donner de la mémoire supplémentaire au programme, et celui-ci sera interrompu brutalement.

Les pointeurs qui pendouillent

Il est parfaitement possible de faire pointer plusieurs pointeurs vers le même objet. Mais dans ce cas si l'objet est détruit (car le programmeur a consciencieusement rendu la mémoire au système) les autres pointeurs pointeront sur une zône de mémoire qui ne contient plus de données valides... soit elle contient n'importe quoi ("du jargon" (`garbage`)), soit elle contient de nouvelles données, mais qui ne sont peut-être pas structurées de la même manière que les précédentes. Le pointeur va donc "pendouiller", et si on cherche à l'utiliser, il peut se passer n'importe quoi, mais le pire est à craindre.

Propriétaires et référents

Compte tenu de ce qui précède, on voit donc qu'on peut définir deux sortes de pointeurs:

- Un pointeur *propriétaire* de l'objet. Il a la responsabilité de la destruction de cet objet. L'objet est le *référent*.
- Les autres pointeurs. Ils peuvent accéder à l'objet tant que celui-ci existe, mais ils ne sont pas censés s'occuper de sa destruction.

Bien sûr, la propriété d'un objet peut passer d'un pointeur à l'autre. D'autre part, il faut bien avoir présent à l'esprit que ces notions, importantes lors de la phase de conception, *ne sont pas présentes dans le langage lui-même*: le C++ ne comprend en effet aucune gestion de la mémoire, celle-ci restant à la charge du programmeur. Cependant, des objets (dont l'un d'entre eux se trouve déclaré dans la bibliothèque standard) vont pouvoir nous aider.

Opérateurs et fonctions

Les opérateurs `new` et `delete`

L'opérateur `new` est utilisé pour allouer de la mémoire pour un objet, `delete` est utilisé pour redonner la mémoire au système. Le (ou les) paramètres passés à `new` seront passés au constructeur de l'objet:

```
main() {
   const complexe J(0,1);
   complexe* C = new complexe(5,5);
   *C = J;
   delete C;
};
```

Les opérateurs `new[]` et `delete[]`

Ils servent à allouer de la mémoire pour un *tableau* d'objets. Ils ne peuvent etre utilisés qu'à la condition qu'existe pour notre objet un constructeur par défaut, à qui on puisse ne pas passer de paramètres. C'est le cas pour notre objet `complexe`, nous pouvons donc écrire:

```
main() {
   const complexe J(0,1);
   complexe* C = new complexe[100];
   for (int i=0; i<100; ++i) {
       C[i] = J;
   }
   delete C[];
};
```

Un tableau de 100 complexes est dynamiquement alloué. Les complexes sont tous initialisés à 0 (constructeur par défaut), puis affectés à la valeur `J`. Enfin, le tableau est détruit et la mémoire est rendue au système.

✔ On ne peut allouer un tableau d'objets de cette manière *que* si les objets en question possèdent un constructeur par défaut. Il n'est pas possible de passer des paramètres aux constructeurs des objets créés.

Fonctions malloc, free, realloc

Nous avons en `C` des fonctions d'allocation dynamique de mémoire: `malloc` et `free` pour allouer de la mémoire et la rendre au système, `realloc` pour refaire une allocation mémoire lorsque le bloc précédemment alloué est trop juste.
Tout cela est réutilisable, à condition de prendre quelques précautions:

- `new` alloue la mémoire, *puis* appelle le constructeur. `malloc n'appelle pas le constructeur`. Attention, donc à l'initialisation correcte de l'objet.
- `delete` appelle le destructeur *puis* rend la mémoire au système. `free` n'appelle pas le destructeur. Attention donc au code qui ne sera pas exécuté
- `realloc` *ne doit pas être utilisé* dans le cas d'objets de type class: `realloc` va provoquer une copie de la mémoire bit à bit, ce qui risque de provoquer des catastrophes dans certains cas.
- *Soyez cohérent*: si vous avez utilisé `new` pour créer un objet, utilisez `delete` pour le détruire. Si vous avez utilisé `alloc` pour créer un objet, utilisez `free` pour le détruire.

Conclusion: pour du code C++, il n'y a aucune raison de ne pas utiliser les opérateurs du C++, `new` et `delete`. Mais il faut savoir que le code C écrit avec `malloc` et `free` (même `realloc` dans le cas de zônes d'entiers ou de caractères, par exemple) reste utilisable.

Allocation mémoire et constructeurs-destructeurs

Le constructeur d'un objet est l'endroit rêvé pour appeler `new`. De même, le destructeur du même objet est l'endroit rêvé pour appeler `delete`.

Constructeurs de copie

Attention au constructeur de copie; à chaque copie, il faudra prendre une décision; il peut en effet se présenter plusieurs cas de figure:

1. L'objet source est propriétaire d'un objet référent, l'objet copié pointe vers le référent sans être propriétaire.
2. L'objet source est propriétaire d'un référent, à la suite de la copie l'objet copié devient le nouveau propriétaire.
3. Le référent de l'objet source est copié dans une autre zône mémoire, et l'objet copié est propriétaire de la copie du référent.

✔ Des trois solutions ci-dessus, la première est *très dangereuse*: en effet, elle risque fort d'aboutir à des objets "irresponsables" vis-à-vis de l'allocation mémoire. Cette solution est toutefois acceptable lorsque les objets référents comptent eux-mêmes les références. La seconde solution peut être implémentée par un `auto_ptr`.

Objets utilisés pour la gestion de la mémoire

L'objet `auto_ptr`

`auto_ptr` fait partie de la bibliothèque standard du C++ . Il sert à définir un pointeur "intelligent"... en tous cas fort sympathique, ayant les caractéristiques suivantes:

- Un `auto_ptr` est *toujours* propriétaire de l'objet sur lequel il pointe
- Lorsque l'`auto_ptr` est détruit, l'objet sur lequel il pointe est détruit également. On est donc assuré de ne pas avoir de pointeur pendouillant.
- Si l'on copie un `auto_ptr`, l'`auto_ptr` original perd l'objet pointé, de sorte que s'il est détruit, le référent ne sera pas détruit. Par contre, si le *nouvel* `auto_ptr` est détruit, le référent sera détruit puisque la copie est le nouveau propriétaire.

Le code suivant, qui utilise des objets de type `complexe`, illustre l'utilisation d'`auto_ptr`:

```
typedef auto_ptr<complexe> complexe_ptr;

void main() {
  complexe::set_debug();
  complexe* c3;                              // pointeur ordinaire
  {
    complexe_ptr c2(new complexe);           // allocation d'un auto_ptr.
```

```
    c3=c2.get();                                // c3 pointe sur le meme complexe
    complexe_ptr c1(new complexe(5,5));         // allocation d'un auto_ptr
    cout << "c1 = " << c1.get() << " c2 = " << c2.get() << " c3 = " << c3 <<
"\n";
    c2=c1;                                      // c2 devient proprietaire du complexe
                                                // son ancien referent est detruit
                                                // c3 pendouille
    c3=c2.get();                                // c3 raccroche
    cout << "c1 = " << c1.get() << " c2 = " << c2.get() << " c3 = " << c3 <<
"\n";
    cout << *c3 << "\n";
  };                                            // destruction de c2 et du referent
  cout << *c3 << "\n";                          // ATTENTION c3 pendouille
}                                               // ca a deja plante (probablement)
```

Un objet de type `auto_ptr` peut donc avantageusement être alloué dans le constructeur à la place d'un objet ordinaire. La destruction du référent sera automatique au moment de la destruction de l'objet, sans même qu'il soit nécessaire de le spécifier dans le destructeur. Dans le code ci-dessus, tous nos ennuis viennent en effet uniquement de `c3`, déclaré comme `complexe*`.


En fait, l'important n'est pas là: après tout, vous êtes bien assez malin pour ne pas oublier d'écrire les quelques lignes de code du destructeur correspondant aux instructions `delete`. L'important, c'est qu'il est fort possible que le constructeur, *après avoir alloué un ou plusieurs pointeurs*, génère une exception (fonctionnement normal pour un constructeur). La construction de l'objet est alors interrompue, et *le destructeur n'est pas appelé*. Résultat: une fuite de mémoire. L'`auto_ptr` est donc la solution élégante, car lorsque l'auto_ptr sera détruit, le référent sera lui aussi détruit.

Objets gestionnaires de ressources

Pour éviter les ennuis évoqués ci-dessus, arrangez-vous pour qu'un objet *ne gère qu'une seule ressource*. Eventuellement, si vous devez gérer trois ressources, rien ne vous empêche d'utiliser trois objets différents, quitte à les insérer dans un autre objet.


Les trois principes fondamentaux pour gérer les ressources sont:

Propriété
: A chaque ressource allouée, correspond un et un seul objet gestionnaire, qui sera propriétaire de cette ressource.

Responsabilité
: L'objet gestionnaire est responsable de la ressource, et il est le seul responsable

Simplicité
: L'objet gestionnaire ne fait rien d'autre.

`auto_ptr` est un bon exemple d'objet dont l'unique raison d'être est la gestion d'une ressource (en l'occurrence la mémoire).

Produire du code robuste, malgré les exceptions

On l'a vu, les exceptions peuvent avoir pour conséquence l'apparition de fuites de mémoire ou de pointeurs pendouillants. Voici quelques astuces permettant d'éviter ces désagrément.

auto_ptr

Utiliser `auto_ptr` le plus souvent possible. On ne le dira jamais assez...

p=NULL

Après avoir exécuté `delete p`, *on se retrouve avec un pointeur pendouillant*. Donc, remettez les choses en ordre *dès la ligne suivante* (à moins, bien sûr, qu'on ne sorte de la portée). Par exemple avec `p=NULL;` En effet, s'il arrive qu'un second appel `delete` soit lancé (par exemple à partir du destructeur), il ne se passera rien si `p` vaut `NULL`, alors que le résultat sera catastrophique si `p` pendouille.

✔ Le code suivant est *dangereux*:

```
delete p ;
p = new toto();
```

🔹Si `toto` envoie une exception, `p` continue à pendouiller. Il vaut mieux faire:

```
delete p ;
p = NULL;
p = new toto();
```

Les objets à comptage de référence

Il est possible d'implémenter des objets qui *comptent* le nombre de pointeurs mis sur eux. De cette manière, il est aisé, lorsque ce nombre arrive à 0, de faire en sorte que ces objets s'auto-détruisent. Cela passe, bien sûr, par la surcharge des opérateurs `=, *, ->`. Signalons au passage que perl utilise un comptage de références sur *toutes* ses variables.

La bibliothèque standard

La bibliothèque standard est en principe livrée avec le compilateur, à condition que celui-ci soit assez récent pour respecter la norme ANSI de 1997 du C++: il est donc important d'utiliser toujours *la version la plus récente possible* de son compilateur. La bibliothèque comprend une multitude d'objets et de fonctions fort utiles, dont nous présentons brièvement ici *les plus importants*.

Documentation

L'accès à la documentation de la bibliothèque standard est la première difficulté... vous trouverez ci-dessous deux liens intéressants, sinon les pages `man` (sous Unix) de votre système sont une autre source d'informations... une autre solution est d'aller lire directement les fichiers d'include, généralement sous `/usr/include/CC` ou autre chemin du même style. La page de références donne aussi quelques pointeurs utiles

✔ En lisant les fichiers d'include, vous aurez accès à l'interface, mais *aussi* à une partie de l'implémentation... N'*utilisez pas* ce que vous aurez appris de l'implémentation. Utilisez *uniquement* les spécifications de l'interface... En effet, seul l'interface est "garanti" ne pas changer d'un compilateur à l'autre, ou d'une mise à jour à l'autre de votre compilateur.

Compléments sur le langage

Certaines particularités du langage n'ont pas encore été abordées: elles peuvent en effet être ignorées, dans une première approche du C++. Cependant, il est nécessaire de les connaître, ne serait-ce que pour comprendre la documentation ou le code de la bibliothèque standard.

Les espaces de noms

Un problème se pose dès lors que l'on travaille avec des bibliothèques venant de plusieurs sources différentes: les noms de classes, de fonctions, de types, etc. employés par les uns et par les autres peuvent parfaitement entrer en conflit; Une manière classique de résoudre le problème est d'inciter les développeurs à utiliser des noms ayant peu de chances d'entrer en conflit, par exemple en mettant devant chaque nom un préfixe particulier. Ainsi, si la bibliothèque `GenialeBibliotheque` utilise le type `string`, elle l'appellera `GenialeBibliotheque_string` . Si la bibliothèque `SublimeBibliotheque` définit elle aussi un type `string`, elle l'appellera `SublimeBiblitoheque_string`. Cependant, le C++ offre un mécanisme bien plus astucieux;

La déclaration `namespace`

Chacune des deux bibliothèques encapsule toutes ses déclarations dans un bloc appelé `namespace`, comme on le voit ci-dessous:

```
namespace GenialeBibliotheque {
   class string {
      ...
   }
}
```

pour la première bibliothèque et:

```
namespace SublimeBibliotheque {
```

```
   class string {
      ...
   }
}
```

pour la seconde.

Les aliases d'espaces de noms

On conseille de donner des noms longs pour les espaces de noms: plus le long sera long plus le risque de conflit sera faible. Par contre, plus le long sera long plus la souffrance sera aigûe pour l'utilisateur. Mais, heureusement, nous avons la possibilité de définir des aliases d'espaces de noms comme suit:

```
namespace GB = GenialeBibliotheque;
namespace SB = SublimeBibliotheque;
```

L'utilisation des espaces de noms

A partir de là, comment le code utilisateur va-t-il prévenir le compilateur qu'il veut utiliser l'un ou l'autre des espaces de noms ? Trois solutions:

- L'opérateur de portée
- Les déclarations `using`
- La directive `using`

L'opérateur de portée permet de spécifier la totalité du nom, comme indiqué ci-dessous:

```
GB::string S;
```

La notation est lourde, conduisant à un code peu lisible.
Les *déclarations* `using` sont intéressantes, elles permettent de déclarer, objet par objet, le nom de l'espace de nom correspondant. L'opérateur de portée figure dans la déclaration, mais ensuite il est sous-entendu. Par exemple:

```
using GB::string;
string S;
```

On a autant de contrôle qu'avec la méthode précédente, mais sans la lourdeur de la notation.
La *directive* `using` est plus radicale: elle dit au compilateur que tous les objets de l'espace de noms considéré doivent être accessibles:

```
using namespace GB;
string S;
```

Sympathique, car la notation est allégée mais attention: si un nouveau symbole apparaît lors de la prochaine version de la bibliothèque SublimeBibliotheque, par exemple la classe `class1`, un nouveau conflit peut être généré dans le code utilisateur... il est toujours très désagréable d'avoir de nouvelles erreurs de compilation parce qu'on a changé de version de bibliothèque. La solution la meilleure semble donc bien être celle des déclarations `using`.

L'espace de noms de la bibliothèque standard

Tous les symboles définis par la bibliothèque standard se trouvent dans l'espace de noms `std`. D'autre part, les en-têtes de la bibliothèque standard ne comprennent pas le traditionnel `.h`. D'où les quelques lignes suivantes que l'on trouve en tête des programmes:

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

✔Certains compilateurs nécessitent l'utilisation d'une option de compilation pour utiliser la bibliothèque standard. A vérifier dans votre documentation.

L'instruction `typedef`

L'instruction `typedef` (issue du `C`) ne définit pas, contrairement à ce que son nom pourrait laisser croire, un nouveau type. Elle ne fait que donner un autre nom (synonyme) *à un type existant*. Il est recommandé de s'en servir, car l'utilisation systématique des modèles dans la bibliothèque standard rend le code difficilement compréhensible si on ne l'utilise pas.

Les types locaux

Nous avons vu qu'il est possible de définir des types à l'intérieur d'une classe: on parle alors de *types locaux*. La bibliothèque standard fait largement appel à la notion de types locaux, par exemple pour définir des énumérations:

```
class semaine {
public:
   enum jour {lundi,mardi,mercredi,jeudi,
              vendredi,samedi,dimanche};
   jour j;
...
}

...
if (j == semaine::lundi) {
   ...
}
```

Remarquez la notation `semaine::lundi`, utilisant l'opérateur de portée `::`
Les itérateurs décrits ci-dessous, apparaissent eux aussi comme des types locaux dans les prototypes de la bibliothèque standard. D'où la syntaxe que nous verrons ci-dessous:

```
typedef vector <int>::iterator i_int;
```

L'instruction `typename`

Dans certains cas, lors de la définition de modèles, il n'est pas simple pour le compilateur de déterminer si une déclaration est une déclaration de type ou de variable: en fait, cette détermination n'est possible que lors de l'instantiation du modèle. Or, suivant qu'il s'agit d'un type ou d'une variable, l'utilisation ultérieure de la chose sera bien différente. On doit donc pouvoir dire au compilateur que tel objet est soit une variable, soit un type. La règle est la suivante:

- Soit rien n'est spécifié, dans ce cas il s'agit d'une variable
- Si on veut forcer une définition de type, on emploie `typename`

Les conteneurs

Les conteneurs sont des objets qui *en contiennent* d'autres: ce terme général inclue un grand nombre de structures de données: les tableaux, les listes, les queues, ... mais aussi les tableaux associatifs. Comme on pouvait s'y attendre, les conteneurs sont implémentés grâce aux *modèles*: on devra donc spécifier "ce que" le conteneur contient lors de la déclaration de la variable. Par rapport à un tableau classique "à la C", un conteneur offre un certain nombre d'avantages:

- Allocation dynamique et automatique de mémoire
- Possibilité d'insérer ou supprimer les éléments, pour certains types de conteneurs

- Utilisation d'algorithmes prédéfinis sur ces conteneurs

✔ *Ne pas utiliser* de tableaux "à la C" en C++: il est bien plus sûr et aussi performant (à condition de prendre quelques précautions) d'utiliser à la place un conteneur de type `vector`
La surcharge des opérateurs permettra d'écrire du code lisible (opérateur `[]` pour l'accès aux données dans un vecteur ou opérateur `++` pour les itérateurs, par exemple). Suivant les opérations que l'on doit réaliser, on devra utiliser tel ou tel type de conteneur, afin d'obtenir la meilleure performance. D'ailleurs, certaines opérations ne seront pas implémentées sur *tous* les conteneurs, non pas parce que ce ne serait pas possible, mais parce que ce serait inefficace.

Conteneurs d'objets ou conteneurs de pointeurs ?

Un conteneur doit-il contenir des *objets* ou des *pointeurs vers des objets* ? Le problème avec les conteneurs est qu'on est amené, lorsqu'on remplit le conteneur, à *copier* les objets. Si le conteneur contient beaucoup d'objets, si les objets eux-mêmes sont gros, cela peut conduire à une utilisation excessive de la mémoire. Dans ce cas, il peut être intéressant d'utiliser un conteneur de pointeurs.

✔ On a déjà évoqué les problèmes posés par les pointeurs: les conteneurs de pointeurs n'y échappent pas, bien entendu. En particulier, attention, si l'objet est détruit le pointeur correspondant doit être retiré du conteneur, faute de quoi il pendouillera.
Une possibilité offerte par les conteneurs de pointeurs: mettre un même objet "dans" plusieurs conteneurs. Attention toutefois à ce qui précède: lors de la destruction de l'objet, il faudra supprimer *plusieurs pointeurs* dans *plusieurs conteneurs* différents. Une solution peut être alors d'utiliser des objets qui comptent leurs références

✔ L'objet `auto_ptr` *n'est pas* une bonne solution dans ce cas: lors de la copie d'un `auto_ptr`, l'objet source est en effet détruit.

Conteneurs hétérogènes ou homogènes ?

Reprenons l'exemple des `shape` : un conteneur hétérogène pourrait contenir des `shape*`; cela permettrait de mettre dans le conteneur n'importe quelle forme; le problème est au moment de récupérer l'objet: qu'est-ce que j'ai bien pu mettre là dedans ? Les conteneurs de classes de bases sont en fin de compte les seuls conteneurs hétérogènes vraiment utiles: grâce Ã l'utilisation des relations d'héritage et des fonctions virtuelles, vous n'aurez pas à vous poser cette question.

🔹 La règle d'or:

- Utilisez les *conteneurs de valeurs*, sauf si vous ne pouvez faire autrement.
- Dans ce cas, utilisez les *conteneurs de pointeurs.*
- Utilisez les *conteneurs homogènes* autant que possible.
- En cas de besoin, les conteneurs hétérogènes sont utilisables également. Ils seront implémentés sous la forme de conteneurs de pointeurs vers une classe de base.
- En aucun cas, vous ne pourrez utiliser de *conteneurs de références.*

Conteneurs séquentiels et conteneurs associatifs (ordonnés).

Les conteneurs de la bibliothèque standard se regroupent en deux familles principales:

- Conteneurs séquentiels
- Conteneurs ordonnés, ou encore conteneurs associatifs

Les conteneurs séquentiels permettent au programmeur de contrôler l'ordre dans lequel les éléments sont insérés.
Les conteneurs ordonnés déterminent eux-mêmes l'ordre dans lequel les éléments sont rangés: ils seront mis dans un ordre permettant de les rechercher très rapidement. Un accès par clé permet d'ailleurs cette recherche: il s'agit donc de conteneurs associatifs, qui fonctionnement de la même manière qu'un tableau associatif en perl, ou un dictionnaire en python.

✔ Certains conteneurs, en particulier `map` et `multimap`, utilisent la `struct` modèle `pair`, qui comprend deux champs: `first` et `second`, ainsi qu'on peut le voir ci-dessous:

```
typedef pair<int,int> p_i_i;
```

```
p_i_i p;
p.first=67;
p.second=54;
```

Les conteneurs de la bibliothèque standard

Les conteneurs disponibles dans la bibliothèque standard sont présentés rapidement ci-dessous:

Conteneurs séquentiels généralistes:

`vector`
: Tableau à une dimension

`list`
: Liste doublement chaînée

`deque`
: Ressemble à un vecteur, sauf que l'insertion et la suppression en tête de liste sont plus performantes.

`queue`
: File d'attente (premier entré, premier sorti). Une queue est un adaptateur de conteneur, qui repose (par dé faut) sur un conteneur de type deque.

`stack`
: Pile (dernier entré, premier sorti). C'est un adaptateur de conteneur.

Conteneurs ordonnés généralistes:

`map`, `multimap`
: Tableau associatif, dans le cas de multimap plusieurs éléments peuvent avoir la même clé

`set`, `multiset`
: Ensemble (set), dans le cas de multiset on peut avoir plusieurs fois le même élément.

`priority_queue`
: File d'attente. On accède uniquement à l'objet situé en haut de la queue. De plus, le conteneur garantit que l'objet situé en haut est le "plus grand". Il s'agit d'un adaptateur de conteneur, qui repose sur un vecteur.

Conteneurs spécialisés:

`string, wstring`
: Chaînes de caractères

`bitset`
: Tableau de booléens.

✔ Les type `bitset`, `multimap`, `multiset`, `priority_queue` ne seront pas abordés plus avant dans ce cours.

Types définis sur les conteneurs

Les conteneurs définissent des *types* en tant que membres publics dont les plus importants sont écrits ci-dessous. On supposera dans ce qui suit qu'on travaille avec l'un de ces deux objets:

- `seq<objet>` (un conteneur séquentiel)
- `ord<cle,valeur>` ou `ord<cle>` (un conteneur ordonné paramétré par un type de clé et éventuellement de valeur).

✔ Même si cela peut paraitre lourd à première vue, il est important d'utiliser ces types, pour générer du code portable:

peut-être que sur votre système le type `size_type` est équivalent à `unsigned int`. Mais sur un autre système, `size_type` risque d'être en fait équivalent à `unsigned long`. D'où de gros soucis de portabilité.

`value_type`
: Type d'élément: `seq<objet>::value_type` n'est autre que `objet`

`reference`
: **`value_type&`** (ici **`objet &`**)

`const_reference`
: **`const value_type &`** (ici **`const objet &`**)

`size_type`
: Numéros d'indices, nombre d'éléments, etc.

`iterator`
: **`value_type*`**, (ici **`objet *`**) permet de balayer le conteneur

`const_iterator`
: Méme chose, mais garantit que les objets récupérés ne seront pas modifiés.

`reverse_iterator`
: Pour balayer le conteneur à l'envers

`const_reverse_iterator`
: no comment

`difference_type`
: Différence entre deux itérateurs

Cas des conteneurs associatifs

Les conteneurs associatifs définissent également les types suivants:

`Key` ou **`key_type`**
: Clé d'accès aux éléments. (Ici, `cle`)

`T` ou **`data_type`**
: Type d'éléments stockés. (Ici, `valeur`)

`pair<key,T>`
: Les objets stockés dans un `map` (donc `value_type`). On accède à la clé par le champ `first`, à la valeur par le champ `second`. A noter que les paires sont rangées par ordre croissant du champ `key`. Il est possible de définir ce qu'est cet ordre (cf. la documentation de l'objet `map`).

Quelques fonctions membres ou opérateurs définis sur les conteneurs

Ce paragraphe expose quelques fonctions-membres (les plus utiles) définies sur les conteneurs les plus souvent utilisés: autant dire qu'il ne prétend en aucune façon à l'exhaustivité. Ne sont en effet considérés ici que les conteneurs suivants: `vector, list, deque, queue, stack, map, set`.

- **Itérateurs**

  `iterator begin()`, **`const_iterator begin()`** (tous)
  : Retourne un itérateur qui pointe sur le premier élément. Permet donc de démarrer une boucle pour faire défiler tous les éléments du conteneur.

  `iterator end()`,**`const_iterator begin()`** (tous)
  : Retourne un itérateur qui pointe après le dernier élément. Permet donc de donner une condition de fin à la boucle. Cet objet est aussi utilisé par de nombreuses fonctions (algorithmes), par exemple `find` pour signifier que l'objet recherché n'a pas été trouvé.

```
typedef conteneur<int>::const_iterator citr;
for (citr i=V1.begin(); i!=V1.end(); ++i) {
cout << *i << endl;
}
```

- **Accès aux éléments**

reference () const, **const_reference () const** (stack)
Renvoie l'élément situé en haut de la pile. Temps d'exécution constant. La pile ne doit pas être vide.

reference front() const, **const_reference front() const** (vector,list,deque)
Renvoie le premier élément

reference back() const,**const_reference back() const** (vector,list,deque)
Renvoie le dernier élément

```
vector<int> V1;
...
int f = V1.front();
int b = V1.back();

cout << "Premier entier " << f  << endl;
cout << "Dernier entier " << b  << endl;
```

reference operator[](size_type n), **const_reference operator[](size_type n)** (vector,string,deque)
Permet d'accéder à un élément indicé. Attention, il n'y a pas de contrôle de débordement. Le premier élément a l'indice 0.

```
string s = "hello";
cout << s[0]; // renvoie h
```

data_type & operator[](const key_type& k) (map)
Permet d'accéder à un élément à partir de sa clé. Si l'élément n'existe pas, un nouvel enregistrement est créé et inséré dans le conteneur. En conséquence, il n'existe pas de version `const` de cet opérateur.

```
map<string,string> couleurs;
...
couleurs["foreground_color"]="rouge";
couleurs["background_color"]="blanc";
```

iterator find(const key_type & k), **const_iterator find(const key_type & k)** (map,set)
Renvoie un itérateur pointant sur l'élément de clé k. S'il n'y a pas d'élément de clé k, renvoie `end()`

```
map<string,string> couleurs;
...
map<string,string>::iterator i = couleurs.find["foreground_color"];
if (i==couleurs.end()) {
   cout << "Pas trouvé" << endl;
} else {
   cout  << "clé=" << i->first << "valeur=" << i->second << endl;
```

- **Opérations d'insertion et de suppression d'éléments.**
  `void push(const value_type & x)`(stack)
  Insère `x` en haut de la pile. Temps d'exécution constant.
  `void pop(const value_type & x)`(stack)
  Supprime l'élément situé en haut de la pile. Temps d'exécution constant. La pile ne doit pas être vide.
  `void push_front(const value_type & x)`, **`void push_back(const value_type & x)`** (string, deque,list,vector)
  Insère `x` respectivement au début ou à la fin. Attention toutefois, il est peu performant d'insérer une valeur au début de la séquence dans le cas d'un `string` ou d'un `vector`.
  `void pop_front()`, **`void pop_back()`** (deque,list,vector)
  Supprime `x` respectivement au début ou à la fin. Attention toutefois, il est peu performant de supprimer une valeur au début de la séquence dans le cas d'un d'un `vector`. Ces fonctions ont un comportement indéfini si le conteneur est vide: utiliser la fonction `empty()` auparavant.
  `iterator insert(iterator p,const value_type & x)` (vector,list,deque)
  Insère `x` avant `p` et retourne un itérateur pointant sur `x`.
  `void insert(iterator p, In first, In last)` (vector,list,deque)
  Insère les éléments de l'intervalle semi-fermé `[first,last[` immédiatement avant `p`. Le code suivant insère tout le vecteur `V1`, moins les trois premiers éléments, au début de la liste `L1`. Cette insertion est performante, puisque le conteneur de destination est une liste. Par ailleurs, on remarquera que la nature des conteneurs source et destination n'a aucune importance.

```
vector<int> V1;
list<int> L1;
...
vector<int>::iterator i=V1.begin()+3;
L1.insert(L1.begin(),i, V1.end());
```

  `iterator erase(iterator p)` (vector,deque,list)
  `void erase(iterator p)` (map,set)
  Supprime l'élément sur lequel pointe `p`. La performance dépend du type de conteneur. Pour un conteneur séquentiel, renvoie un itérateur pointant sur le successeur immédiat de l'élément détruit, éventuellement `end()`.
  `iterator erase(iterator first, iterator last)` (vector,deque,list)
  `void erase(iterator first, iterator last)` (map,set)
  Supprime tous les éléments de l'intervalle semi-ouvert `[first,last[`. La performance dépend du type de conteneur. Pour un conteneur séquentiel, renvoie un itérateur pointant sur le successeur immédiat du dernier élément détruit, éventuellement `end()`.
  `void remove(const value_type & valeur)` (list)
  Supprime de la liste tous les éléments égaux à `valeur`
  `void unique()` (list)
  Supprime de la liste tous les éléments conécutifs égaux de la liste, sauf un.
  `void clear()` (tous)
  Efface tous les éléments.
- **Opérations affectant l'ordre des éléments *(uniquement list)***
  `void reverse()`
  Retourne l'ordre des éléments dans la liste.
  `void sort()`
  Trie les éléments de la liste en utilisant `operator<`
- **Opérations diverses**

size_type size() const (tous)
renvoie le nombre d'éléments dans le conteneur.
bool empty() const (tous)
renvoie `true` si le conteneur est vide.

---

string: les chaînes de caractères

Ce conteneur permet de définir un type chaîne de caractères et de le manipuler simplement et efficacement. Voici quelques-unes des fonctions-membres associés:

Construction de chaînes
On peut utiliser un constructeur par défaut, un constructeur permettant d'initialiser le `string` à partir d'un `char *`, ainsi que beaucoup d'autres constructeurs.

Concaténation de chaînes
Surcharge de l'opérateur **+**

```
string A="hello";
string B="world";
string C = A + "   " + B;   // C contient hello world
...
```

Tester l'égalité entre deux chaînes.
Comme d'habitude, les opéteurs **==**, **!=**, **<**, **>** etc. sont là pour cela.

Connaître la longueur d'une chaîne.
Par la fonction-membre **length()**.

Trouver un caractère ou une sous-chaîne dans une chaîne
La fonction-membre **find** renvoie un nombre de type **string::size_type**. Si le caractère n'existe pas, elle renvoie le membre constant **string::npos**. Comme pour les autres conteneurs, le premier élément a pour indice 0.

```
...
string::size_type p1 = C.find('o');       // renvoie 4
string::size_type p2 = C.find('o',p1+1);  // renvoie 9
string::size_type p3 = C.find('z');       // renvoie npos
if (p3 == string::npos) {
   cout << "Pas de lettre z " << endl;
...
```

Touver un caractère différent d'un caractère donné:
La fonction-membre **find_first_not_of** est faite pour cela:

```
...
string::size_type c1 = C.find(' ');                  // renvoie 5
string::size_type m2 = C.find_first_not_of(' ',c1);  // renvoie 8
...
```

Définir une sous-chaîne
Fonction-membre **substr**. Ses deux paramètres sont: la position à partir de laquelle démarrer la sous-chaîne, et la longueur de celle-ci.

```
...
string milieu = C.substr(p1,p2-p1+1); // renvoie o   wo
...
```

Effacer une sous-chaîne
: Fonction-membre **`erase`**. Ses deux paramètres sont: la position à partir de laquelle démarrer la sous-chaîne, et la longueur de celle-ci. Renvoie la chaîne après la suppression réalisée:

```
...
C.erase(p1,p2-p1+1);   // C vaut maintenant hellrld
string milieu = C.substr(p1,p2-p1+1); // renvoie o   wo
...
```

Insérer une ous-chaîne
: Fonction-membre **`insert`**. Ses deux paramètres sont: la position à partir de laquelle démarrer l'insertion, et la chaîne à insérer.

```
...
C.insert(1,milieu);    // C vaut maintenant ho   woellrld
...
```

Ajouter des caractères en fin de chaîne
: La fonction-membre **`push_back`** est utilisable dans ce but.

Travailler avec des *`char *`*
: De nombreuses fonctions C utilisent non pas l'objet **`string`**, mais le type **`char *`**. La fonction **`c_str()`** permet de passer d'une `string` vers un `char *`, alors que le constructeur de chaîne permet de passer d'un `char *` vers une `string`, ainsi qu'on l'a vu au début de ce chapitre.

```
...
char * c = C.c_str();
...
```

Les itérateurs

Un itérateur est un objet associé à un conteneur, qui va permettre de balayer l'ensemble des objets se trouvant dans le conteneur, et ceci sans avoir aucune idée de la structure de données sous-jacente; des boucles `for` ou `while` permettront de balayer le conteneur, *exactement* comme on balaierait un tableau classique en C:

```
typedef vector <int> v_int;
typedef vector <int>::iterator v_int_it;

v_int V1;
V1.push_back(1);
V1.push_back(2);
V1.push_back(3);
...
V1.push_back(10);
```

```
for (v_int_it i=V1.begin();i!=V1.end();++i) {
  cout << *i << endl;
};
  ...
```

✔ La ligne `cout << *i << endl` ne signifie pas que `i` est un véritable pointeur: `i` est un *objet* qui implémente une *abstraction* de la notion de pointeur: l'opérateur * est ici surchargé.

✔ `V1.begin()` et `V1.end()` sont tous deux des itérateurs, mais alors que `V1.begin()` pointe sur le *premier* élément du conteneur, `V1.end()` pointe *après* le dernier élément. De la sorte, pour balayer l'ensemble du conteneur, il faut:

- Initialiser l'itérateur sur `V1.begin()`
- Incrémenter (++) l'itérateur à chaque itération, de préférence à l'aide de la version préfixée
- Sortir de la boucle dès que l'itérateur est égal à `V1.end()`

Itérateurs valides et invalides

Un itérateur qui pointe sur un élément est dit *valide*. Dans ce cas, `*i`, renvoie un élément du conteneur. Un itérateur peut être valide à un certain moment de l'exécution du programme, puis être invalide un peu plus tard. Un itérateur peut être *invalide* pour les raisons suivantes:

- Il n'a pas été initialisé
- Le conteneur a été redimensionné (par des insertions ou des suppressions, par exemple).
- Le conteneur a été détruit
- L'itérateur pointe sur la fin de la séquence (`V1.end()` dans l'exemple ci-dessus).

Les conteneurs se différencient par la manière dont les itérateurs deviennent invalides: par exemple, le code suivant a toutes les chances d'invalider `i`:

```
...
i=V1.begin() + 4;      // pointe sur le 5 è élément.
V1.insert(i,100000,0); // insère plein de cases juste avant i
cout << *i <<endl;     // plantage car i est devenu invalide
```

En effet, lors de l'insertion de 100000 entiers, il y a fort à parier que le vecteur s'est trouvé une autre zône de mémoire, de sorte que l'intérateur s'est mis à pendouiller bêtement. Par contre, le code suivant ne présentera pas de problème:

```
typedef list <int> l_int;
typedef list <int>::iterator l_int_it;

l_int L1;
L1.push_back(1);
L1.push_back(2);
L1.push_back(3);
...
L1.push_back(10);

l_int_v1 i = L1.begin();
i++;i++;i++;i++;       // pointe sur le 5 è élément.
L1.insert(i,100000,0); // insère plein de cases juste avant i
cout << *i <<endl;     // Pas de plantage
```

En effet, l'objet de type `list` s'arrange pour que `i` reste toujours valide dans ce cas.

Les différentes catégories d'itérateurs

Les itérateurs peuvent être classés en plusieurs catégories; suivant les catégories auxquelles ils appartiennent, nous avons plus ou moins d'opérations à notre disposition;

Intérateurs In (Input)

On ne peut effectuer que 4 opérations:

1. *égalité* `j = i`
2. *Incrémentation* `++i` ou **i**++
3. *Déréférencement, en lecture seule* **`A = *i`**
4. *Test d'égalité* **i == j**

✔ Il est impossible de faire **`*i = A`** avec cet itérateur.

✔ Il est impossible de déréférencer plus d'une fois le même élément. Ainsi, le code **`A = *i; B = *i`** ne marche pas.

On doit penser à cet itérateur comme à un objet permettant de lire un fichier.

Intérateurs Out (Output)

On ne peut effectuer que 3 opérations:

1. *égalité* `j = i`
2. *Incrémentation* `++i` ou `i++`
3. *Déréférencement, en écriture seule* **`*i = A`**

✔ Il est impossible de faire **`A = *i`** avec cet itérateur.

✔ Il est impossible de déréférencer plus d'une fois le même élément. Ainsi, le code **`*i = A; *i = B`** ne marche pas.

On doit penser à cet itérateur comme à un objet permettant d'écrire un fichier.

Intérateurs For (Forward)

Permet de balayer une séquence du début à la fin, mais sans retour en arrière possible. Les opérations supportées par les itérateurs In *et* Out sont également disponibles avec cet itérateur. On peut également déréférencer l'itérateur autant de fois qu'on veut.

✔ Partout où un itérateur `In` ou `Out` est requis, vous pourrez fournir un itérateur `For`.

Intérateurs Bi (Bidirectionnels)

Tout ce qui est permis par l'itérateur `For`, avec en plus:

1. *Décrémentation* **--i** ou **i--**

✔ Partout où un itérateur `In` ou `In`, `Out` ou `For` est requis, vous pourrez fournir un itérateur `Bi`.

Intérateurs Ran (Random)

Tout ce qui est permis par l'itérateur `Bi`, avec en plus:

1. *Opérateur d'indexation* **i[3]**
2. *Ajout ou suppression d'entiers* **j=i+3; j=i-4; i +=2;**
3. *Opérateur -: la différence entre deux itérateurs random est un entier* **`if (j-i==4)...`**

✔ Partout où un itérateur `In` ou `In`, `Out`, `For` ou `Bi` est requis, vous pourrez fournir un itérateur `Bi`.

✔ Un itérateur random offre les mêmes possibilités qu'un pointeur conventionnel.

Les itérateurs `const`

La fonction suivante ne pourra pas être compilée:

```
typedef list<int> li;
typedef list<int>::iterator ili;

void  pl(const li & L) {
  for(ili i=L.begin();i!=L.end();i++) {
    cout << *i <<  endl;
  };
};
```

En effet, le type `ili` ne peut s'appliquer sur un objet constant, puisqu'il est susceptible de modifier cet objet. Il faut dans ce cas déclarer un `const_iterator`, d'où le code ci-dessous:

```
typedef list<int> li;
typedef list<int>::const_iterator cili;

void  pl(const li & L) {
  for(cili i=L.begin();i!=L.end();i++) {
    cout << *i <<  endl;
  };
};
```

Les itérateurs `reverse`

Ils permettent de balayer la séquence en sens inverse. Leur description sort toutefois du cadre de ce cours.

Spécifier un intervalle à l'aide de deux itérateurs.

Un grand nombre de fonctions membres, ou d'algorithmes, demandent deux itérateurs en entrée, afin de spécifier un intervalle: *Dans tous les cas*, cet intervalle est semi-ouvert:

- Fermé sur la borne inférieure
- Ouvert sur la borne supérieure

En effet, utiliser des intervalles de ce type présente plusieurs intérêts:

- La borne supérieure de l'intervalle étant en-dehors de celui-ci, on peut utiliser la fonction `end()`, présente dans tous les conteneurs, pour signifier qu'on va jusq'au bout de celui-ci: puisque `end()`. Même si le dernier élément change au cours du temps, (lorsqu'on ajoute un élément en bout de conteneur, par exemple) `end()` ne changera pas
- Dans le cas d'itérateurs "Random", la différence entre borne supérieure et borne inférieure renverra toujours le nombre d'éléments se trouvant entre les deux bornes: pas besoin de se creuser la tête pour résoudre des problèmes tordus d'intervalles...

✔ Lorsque vous spécifiez un intervalle par `[first,last[` `first` et `last` doivent être deux itérateurs valides *du même conteneur*. Le type de conteneur n'a par contre aucune importance.

Qui fait quoi ?

Les tableaux ci-dessous ne sont qu'un résumé des paragraphes ci-dessus. En particulier, on a insité ici sur les différences de performances entre les différents conteneurs, suivant les fonctions membres.

- o Fonction-membre présente

- **const** Fonction-membre présente, performance constante (quelque soit le nombre d'éléments dans le conteneur)
- **O(n)** Fonction-membre présente, durée proportionnelle au nombre d'éléments
- `O(log(n))` Fonction-membre présente, durée proportionnelle au logarithme du nombre d'éléments
- + surcoût à prévoir de temps en temps (lors de demandes automatiques d'allocation mémoire, par exemple)

Fonctions membres "non mutantes"

Conteneurs	Itérateurs	Fonctions membres				
		`(r)begin()`, `(r)end()`	`front()`, `back()`	`[]`	`find()`	`lower_bound()`, `upper_bound()`
`vector`	**Ran**	**o**	**o**	**const**		
list	**Bi**	**o**	**o**			
deque	**Ran**	**o**	**o**	**const**		
`queue`						
`priority_queue`						
stack						
`map`	**Bi**	**o**		**O(log(n))**	**O(log(n))**	
multimap	**Bi**	**o**			**O(log(n))**	**O(log(n))**
`set`	**Bi**	**o**			**O(log(n))**	
multiset	**Bi**	**o**			**O(log(n))**	**O(log(n))**

Fonctions membres "mutantes"

Conteneurs	Fonctions membres						
	`push_back`, `pop_back`	`push_front`, `pop_front`	`pop`	`push`		`insert()` `erase()`	`clear()`
`vector`	**O(n)+**					**O(n)+**	**o**
list	**const**	**const**				**const**	**o**
deque	**const**	**const**				**O(n)**	**o**
queue			**const**	**const+**			
`priority_queue`			**O(log(n))**	**O(log(n))**	**o**		
`stack`			**const**	**const+**	**o**		
`map`						**O(log(n))+**	**o**
`multimap`						**O(log(n))+**	**o**
`set`						**O(log(n))+**	**o**
multiset						**O(log(n))+**	**o**

Algorithmes

La bibliothèque standard fournit également des algorithmes (fonctions) qui permettent de réaliser toutes sortes d'opérations

sur les éléments d'un conteneur. Il sera souvent nécessaire de passer un objet fonction en paramètre; par exemple l'algorithme **find_if** qui recherche dans un conteneur les éléments qui répondent à une condition donnée: la condition sera passée sous forme d'un objet fonction à un paramètre, qui renverra soit `true`, soit `false` (un tel objet est appelé un *prédicat*).
Les principales fonctionnalités sont les suivantes:

- Appeler une fonction pour chaque objet d'un conteneur (`for_each`, équivalente à la fonction prédéfinie `map` du perl).
- Trouver un élément, une paire d'éléments
- Trouver la première occurrence, ou la dernière occurrence d'une valeur dans une séquence.
- Compter les occurrences d'une valeur dans une séquence
- copie, remplace, copie si telle condition est réalisée,...
- retire un élément, retire tous les éléments correspondant à une condition particulière,...
- Trie, effectue une recherche binaire (sur une séquence triée)
- recherche le minimum ou le maximum
- etc.

Un exemple d'utilisation d'algorithmes

L'extrait de code ci-dessous montre une manière amusante d'imprimer tous les éléments d'un conteneur (quelque soit le conteneur, d'ailleurs):

```
class sortie {
    public:
    sortie (const string & s): msg(s){;};
    void operator()(int i) {
        cout << msg << i << "\n";
    };
    private:
    string msg;
};

...

vector<int> V1;
...

sortie s("coucou ");
for_each(V1.begin(), V1.end(),s);
```

La classe `sortie` permet de définir un objet-fonction, c'est-à-dire un objet comportant `operator()`. Le constructeur de cet objet initialise un membre privé. Par la suite, on crée un objet de type `sortie`, et on appelle la fonction `for_each` en lui passant deux itérateurs (pour définir un intervalle) ainsi que l'objet-fonction précédemment créé. Pour plus de détails, aller voir la documentation de `for_each`.

Les entrées-sorties

Les entrées-sorties se font à travers plusieurs objets "flots" dont la hiérarchie se trouve partiellement résumée ci-dessous:

```
ios
   istream
       ifstream
```

```
       istringstream
       iostream -------
   ostream             |
       ofstream        iostream
       ostringstream   |    fstream
       iostream -------     stringstream
```

Les objets de type ostream ou istream

Pour ouvrir un fichier, il suffit d'instancier un objet de type `ostream` (ouverture en écriture), de type `istream` (ouverture en lecture), ou encore de type `iostream` (ouverture en lecture-écriture)
Pour fermer le fichier ouvert, le plus simple est de détruire l'objet, par exemple d'attendre la fin du bloc afin qu'il soit automatiquement détruit (il n'est alors pas nécessaire de se préoccuper de la fermeture du fichier), ou dans le cas d'un objet créé avec l'opérateur `new` d'appeler l'opérateur `delete`.:

```
{                             // debut de bloc
   ofstream F1("toto");       // ouvre un fichier en sortie et l'appelle toto
   ...
}                            // fin de bloc = fermeture du fichier
```

Lecture-écriture en binaire

Pour écrire ou lire des données en binaire, c'est-à-dire simplement écrire (lire) une image d'un bloc de mémoire, il suffit d'utiliser les fonctions-membres `write` ou `read`.

```
char bfr[50000];
int size;
...
{
   ofstream OUT("toto-bin",ios::binary);  // ouvre un fichier en écriture
   OUT.write(bfr,size);                   // recopie le buffer
}                                         // ferme le fichier
...
{
   ifstream IN("toto-bin",ios::binary);  // ouvre un fichier en écriture
   IN.read(bfr,size);                    // recopie le buffer
}                                        // ferme le fichier
```

✔ Attention aux débordements de buffers en utilisant ces fonctions !!! Rien ne garantit que size n'est pas supérieur à 50000 dans l'exemple ci-dessus.

Fonctions get, put, getline

La fonction `out.put(c)` où `c` est un `char`, permet d'envoyer ce caractère sur la sortie. Elle est strictement équivalente à `out << c`.
La fonction `in.get(c)` où `c` est un `char` est plus intéressante, dans la mesure où, contrairement à l'opérateur `>>`, elle ne fait *aucune* interprétation du caractère lu. Elle permet par exemple de lire les espaces ou les '\n', qui sont interprétés par `>>`.
La fonction-membre `getline` est bien pratique pour lire un fichier ligne par ligne. Son prototype est le suivant:

```
getline(char* buffer, int taille, char delimiteur='\n')
```

Le délimiteur par défaut est le caractère de fin de ligne, tandis que la possibilité de fixer une taille permet de s'assurer que le buffer ne déborde pas. Juste après un appel `getline`, la fonction-membre `gcount()` permet de savoir quelle est la

longueur de la chaine de caractères effectivement lue.

✔ Il existe aussi une fonction `getline` (qui *n'est pas* une fontion membre) permettant de lire une ligne dans un objet de type `string`. On pourrait penser que cette fonction est bien sympathique, puisqu'il n'y a plus à se préoccuper des débordements de tampons... mais attention, elle va environ *quatre fois plus lentement* que la fonction-membre `getline`... A utiliser avec parcimonie, donc.

Entrées-sorties formattées: << et >>

Les opérateurs << et >> sont surchargés de sorte qu'ils permettent respectivement l'entrée ou la sortie:

```
string A = "voici une chaine";
int B = 56;
objet O;                    // Objet dont le type a été défini plus haut
...

ofstream F1("toto");        // ouvre un fichier en sortie et l'appelle toto
F1 << A << " " << B << "\n"; // Ecrit A, puis B
F1 << O << "\n";            // Ecrit O
```

En particulier, tous les types de base du langage peuvent être utilisés avec ces opérateurs. Mais par ailleurs, lorsque vous écrivez la définition de la classe `objet`, vous pouvez surcharger cet opérateur, afin de vous permettre par la suite d'écrire: `F1 << O << "\n";` Nous verrons plus tard comment écrire un tel opérateur.

Le contrôle du format

Nous avons plusieurs outils à notre disposition pour contrôler le format d'écriture ou de lecture:

- Les manipulateurs
- Les fonctions membres
- Les bits de contrôle

Les manipulateurs

Les manipulateurs s'interposent entre les données imprimées pour agir sur le fonctionnement du flots. Particulièrement bien intégrés au systéme d'entrées-sorties, ils sont d'un usage très pratique.

Les fonctions membres

Un flots étant tout simplement un objet, plusieurs fonctions membres sont définies afin de fixer la valeur de tel ou tel paramètre. Il existe aussi des fonctions membres pour lire la valeur fixée actuellement. Cela permet, par exemple, de sauvegarder un paramètre avant de le modifier, pour ensuite lui redonner sa valeur initiale.

Les bits de contrôle

Plus classique: certains paramètres sont accessibles via le positionnement de bits de contrôle. Les fonctions `setf` et `unsetf` permettent respectivement d'activer et désactiver les bits de contrôle, tout en renvoyant leur ancienne valeur pour sauvegarde éventuelle.

Précision, largeur de champ, caractère de remplissage

Fonctions membres	Résultats

```
float x = 4.56782765;
int savprec=cout.precision();
cout.precision(3);
cout << "x = ";
cout.width(7);
cout << x << "cm\n";

cout << "x = ";
cout.width(7);
cout.fill('#');
cout << x << "cm\n";
```

```
x =4.57cm


x = ###4.57cm
```

Manipulateurs	Résultats

```
float x = 4.56782765;
cout << setprecision(3) << "x = "
     << setw(7) << x << "cm\n";
cout << setprecision(3) << setfill('#')
     << "x = " << setw(7) << x << "cm\n";
```

```
x =    4.57cm

x = ###4.57cm
```

La fonction `width` ainsi que le manipulateur `setw` ne précisent la largeur d'impression *que* pour la prochaine opération de sortie. Il n'en est pas de même pour la plupart des autres fonctions ou manipulateurs,dont l'effet est permanent.

Alignement à gauche ou à droite

champs de bits	Résultats

```
float x = -4.56782765;
cout.precision(3);
cout.fill('-');
cout.setf(ios::left,ios::adjustfield);
cout << "x = ";
cout.width(7);
cout << x << "cm\n";

cout.setf(ios::right,ios::adjustfield);
cout << "x = ";
cout.width(7);
cout << x << "cm\n";
```

```
x = -4.57  cm


x =   -4.57cm
```

Notation scientifique, fixe

champs de bits	Résultats

float x = 4567.82765;
cout.precision(3);
cout.setf(ios::fixed,ios::floatfield);
cout << "x = ";
cout.width(10);
cout << x << "cm\n";
 	
cout.setf(ios::scientific,ios::floatfield);
cout.precision(3);
cout << "x = ";
cout.width(10);
cout << x << "cm\n";
	x = 4567.828cm
	
	x = 4.568e+03cm

Afficher en hexadécimal ou en octal

champs de bits	Résultats

int i =987654;
cout.setf(ios::showbase);
cout << "x = ";
cout.width(10);
cout << i << "..\n";
 	
cout.setf(ios::oct,ios::basefield);
cout << "x = ";
cout.width(10);
cout << i << "..\n";
 	
cout.setf(ios::hex,ios::basefield);
cout.setf(ios::showbase);
cout << "x = ";
cout.width(10);
cout << i << "..\n";
 	
cout.setf(ios::uppercase);
cout << "x = ";
cout.width(10);
cout << i << "..\n";
	x = 987654..
	
	x = 03611006..
	
	x = 0xf1206..
	
	x = 0XF1206..

Manipulateurs	**Résultats**
cout.unsetf(ios::showbase); cout << "x = "<< dec << i << "..\n"; cout << "x = "<< oct << i << "..\n"; cout << "x = "<< hex << i << "..\n";	x = 987654.. x = 3611006.. x = f1206..

Jouer avec les tampons

champs de bits

```
int i =987654;
cout.setf(ios::unitbuf);
cout << "x = " << i;
```

Manipulateurs
`cout << "x = "<< i << flush;`

`unitbuf` assure que toute opération de sortie sera envoyée immédiatement (le tampon n'est pas utilisé); cela peut être important par exemple lorsqu'on écrit dans un pipe, pour assurer une bonne syncrhonisation. Le manipulateur `flush` vide le tampon immédatement.

Ecrire... ou lire l'état du flot

L'opérateur `()` est surchargé de sorte que l'on puisse écrire:

```
OUT << ...;
if (OUT) {
    ...    // Pas de pb
    ..
};
```

De même, l'opérateur `!` est surchargé de sorte que l'on puisse écrire:

```
OUT << ...;
if (!OUT) {
    ...    // BIG PB
    ..
};
```

Par ailleurs, plusieurs fonctions-membres permettent de tester plus précisément l'état du fichier. Parmi elles, la plus utile est `eof()` permettant de détecter la fin de fichier.

Surcharger l'opérateur <<

L'opérateur **<<** peut être surchargé lorsqu'on écrit une classe, ainsi qu'on le voit ci-dessous avec notre classe complexe; Le mieux est de définir une fonction amie, en utilisant un prototype analogue au prototype ci-dessous:

```
class complexe {
...
  friend ostream & operator<<(ostream& os, const complexe& c) {
    os << "Partie reelle = " << c.r << " Partie imaginaire = " << c.i;
    os << endl;
  }
}

...

complexe A;
cout << A << "\n";
```

La dernière opération de sortie écrit:

```
Partie reelle = 0 Partie imaginaire = 0
```

Surcharger l'opérateur >>

L'opérateur **>>** peut lui aussi être surchargé. Attention, il est bon de s'assurer que les données fournies par l'utilisateur correspondent bien à ce que l'on attend, et dans le cas contraire il faut agir sur l'état du flot, afin que le programme principal sache que quelque chose d'anormal est arrivé.
Le programme ci-dessous montre l'opérateur **>>** ainsi que le programme principal qui va avec; on voit qu'on utilise l'opérateur `!` sur le flot `cin` afin de détecter les erreurs, et de boucler le cas échéant. Tout cela ne peut fonctionner que parceque **>>** positionne correctement les bits d'état lorsqu'il rencontre un problème.

```
class complexe {
...
  friend istream& operator>>(istream& is, complexe& c) {
    char sep;
    float r,i;
    is.clear();                          // remise a 0 du status
    is >> r >> sep;
    if (sep == ',') {
      is >> i;
      c.r = r;
      c.i = i;
    } else {
      string dump;
      getline(is,dump);                  // pour jeter la fin de ligne
      is.clear(ios::badbit|is.rdstate()); // pb en lecture
    };
    return is;
  }
};

main() {
  complexe C1;
  do {
    cout <<"Entrer reel,imag:";
  } while(!(cin >> C1));
  cout<< "Voici le complexe = " << C1 << endl;
};
```

Les itérateurs de flots

Une autre manière d'utiliser les flots est de faire "comme si" il s'agissait d'un conteneur, et d'utiliser un itérateur, ainsi qu'on peut le voir ci-dessous pour l'entrée:

```
istream_iterator<int> input (cin);
istream_iterator<int> end_of_input;
while(input != end_of_input) {
    int i = *input;
    ...
    ++input;
}
```

✔ La première ligne associe un itérateur de flot à un flot existant (ici `cin`). La seconde ligne construit un itérateur de flot sans l'associer à quoique ce soit: il s'agit par convention d'un signal de fin de fichier.
Et pour la sortie:

```
ostream_iterator<int> output(cout);
while(...) {
```

```
    *output = ...;
    ++output;
}
```

✔ La ligne `*output = ...` utilisera en fait l'opérateur **<<** de l'objet situé à droite du signe **=**.

Remplir un conteneur à partir d'un fichier

Le code ci-dessous utilise la fonction `back_insert_iterator` afin de générer à partir d'un conteneur, un type particulier d'itérateurs: un itérateur d'insertion. Celui-ci sert à insérer les objets dans le conteneur par la fin. Associé aux itérateurs de flots, on remplit le conteneur avec une ligne de code... mais surtout, le fait qu'on lit depuis un fichier est totalement masqué: changez les itérateurs `i` et `i_end` par des itérateurs définissant un intervalle, par exemple, le code sera exactement le même. Elégant, n'est-il pas ?

```
vector<int> V1;

ifstream data("file.data");
istream_iterator<int> i(data);
istream_iterator<int> i_end;
back_insert_iterator<vector<int> > bii1 = back_inserter(V1);

copy(i,i_end, bii1);
```