COURS DE C++ . NET, JAVA, C++ DE BTS 2 IRIS

Année scolaire : 2005-2006

TABLE DES MATIERES

INTRODUCTION

Bonjour à tous, après de longues vacances bien méritées, je suis de retour pour vous expliquer ce que l'on fait en 2ème année de BTS IRIS.

Rassurer vous, nous allons continuer sur la lancée de l'année précédente. Le tout c'est de ne pas oublier ce que vous avez apprit l' année dernière.

Des le premier jours, nous avons commencé par une petite révision au niveau des classes, de leur déclarations, de leur fonctionnement et sur ce que l'on doit y trouver.

! A première vu, je peut vous le dire, pour tout le monde, la reprise ça a été plutôt dure. !

Fini, de se plaindre et rentrons dans le vif du sujet.

Ah oui, une dernière chose, pour ceux qui ont lu mon premier cour, je reprend la même légende de couleur pour marquer, ce qui à mon goût, le plus important.

Encore merci, de lire mon cour.

MENBRES STATIQUES – MENBRES DE CLASSES

Nous avons remarqué que lorsque nous créons des objets d'une même classe, chaque objet possède ses propres attributs.

L'ensemble de ces attributs, nous permettent de définir l'état de cette objet à un instant (T).

Qu'est ce qu'un attribut de classe ?

Il peut permettre aux différents objet issu de la classe de partager les informations de telle sorte que lorsqu'un des objets modifie cette information, que tout le monde soit informé de cette modification. Nous pouvons dire que cette information est mise en commun, représenté par un attribut de la classe et non de l'objet.

Définition d'un attribut statique ?

Dans le langage de C++, un attribut de classe corresponds à une variable globale, dont la portée est limité à la classe.
Celui –ci est donc déclarer dans la mémoire statique et donc par conséquence utilisé le préfixe « static ».
Sans oublier de l'initialiser avec des valeurs correcte.
Cette initialisation doit être lancer au moment du lancement du programme comme pour des variables globales.

ATTENTION : Cet attribut aura donc une durée de vie correspondant à la durée de vie du programme.

Ex :

```
public static unsigned int taille ;
```

Cette exemple de code permet de déclarer un attribut de classe TEST.h « statique » du type entier non - signe.

Pour initialiser cette attribut, nous utiliserons le ligne suivante :
Unsigned TEST ::taille=10 ;

Si nous avions dut modifier la variable taille, alors tous les objets auraient automatiquement été modifier.

Les méthodes ont la possibilité d'accéder aux attributs statiques comme pour tous les autres attribut. Soit, le constructeur peut tout à fait intervenir et modifier un attribut de classe en sachant toutefois que l'initialisation a déjà eu lieu au moment du lancement du programme et qu'il s'agit, dans ce cas là, que d'une évolution.

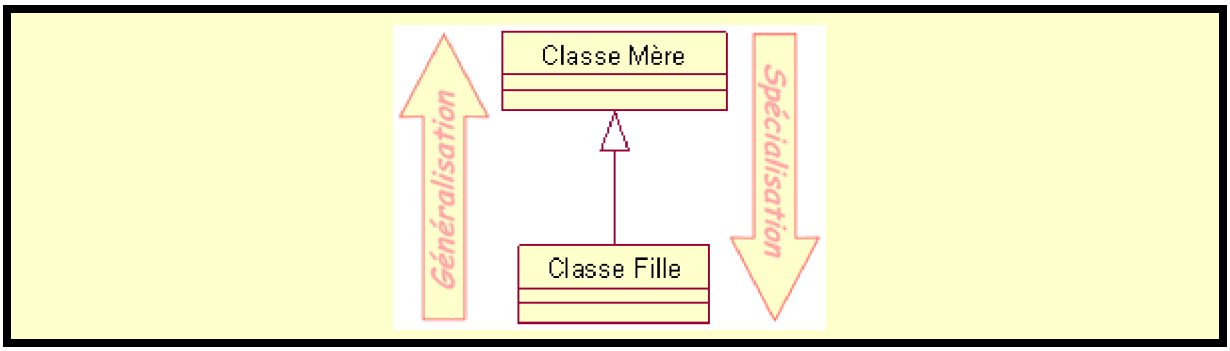
Méthode statique ?

Cela s'applique aussi aux méthodes de la classe, nous imaginons que leur rôle est totalement indépendant d'un quelconque objet. Une méthode peut aussi être déclarer comme « static ».

ATENTION : Une methode « static » n'a pas de pointeur « this » et qu'il est donc absolument interdit d'accéder à un attribut non statique.

HERITAGE – GENERALISATION ET SPECIALISATION

Le mécanisme d'héritage permet de mettre en relation un certain nombre de classes ayant des caractéristiques communes (attributs et comportements) en respectant une certaine filiation.



Généralisation :

La généralisation se représente par une flèche qui part de la classe « fille » vers la classe « mère ». Cela corresponds exactement à ce que nous avions avant sans l'héritage, sauf qu' avec cette technique, nous évitons toutes les duplications. Toutes les caractéristiques communes n'apparaissent plus dans chacune des classes « filles », alors qu'elles sont bien présentes implicitement.

En terme de mathématicien, cela s'appel faire une factorisations.

Les héritages peuvent être défini sur plusieurs niveau.

Classes abstraites :

Déclarer une classe « abstraite » revient à interdir la création d'objet issu de cette classe. En opposition, à une classe « concrète », dont le plus souvent déclarer par défaut.

Spécialisation :

Dans la conception des classes, nous pouvons avoir une démarche inverse de la généralisation, c'est-à-dire, cette fois-ci, de partir plutôt de la classe mère pour aboutir ensuite aux classes filles.

Le concept d'héritage constitue l'un des fondements de la programmation orientée objet.

Il vous autorise à définir une nouvelle classe, dite dérivée, à partir d'une classe existante dite de base. La classe dérivée héritera donc des potentialités de la classe de base.

Il ne sera pas utile de la recompiler, ni même de disposer du programme source correspondant (exception faite de sa déclaration).

En outre, l'héritage, n'est pas limité à un seul niveau : une classe dérivée peut devenir à son tour classe de base pour une autre classe.
Nous voyons apparaître la notion d'héritage comme outil de spécialisation croissante.

Le polymorphisme :

Le terme polymorphisme indique qu'une entité peut apparaître suivant plusieurs formes.

Normalement, le principe même de l'héritage, c'est que lorsque une méthode est décrite sur une classe parente, elle est automatiquement hérité par les classes enfants.

Cette technique s'appelle une redéfinition, c'est-à-dire que dans la classe dérivée, nous allons redéfinir une méthode qui porte le même nom avec une signature identique (polymorphisme) que la classe de base.

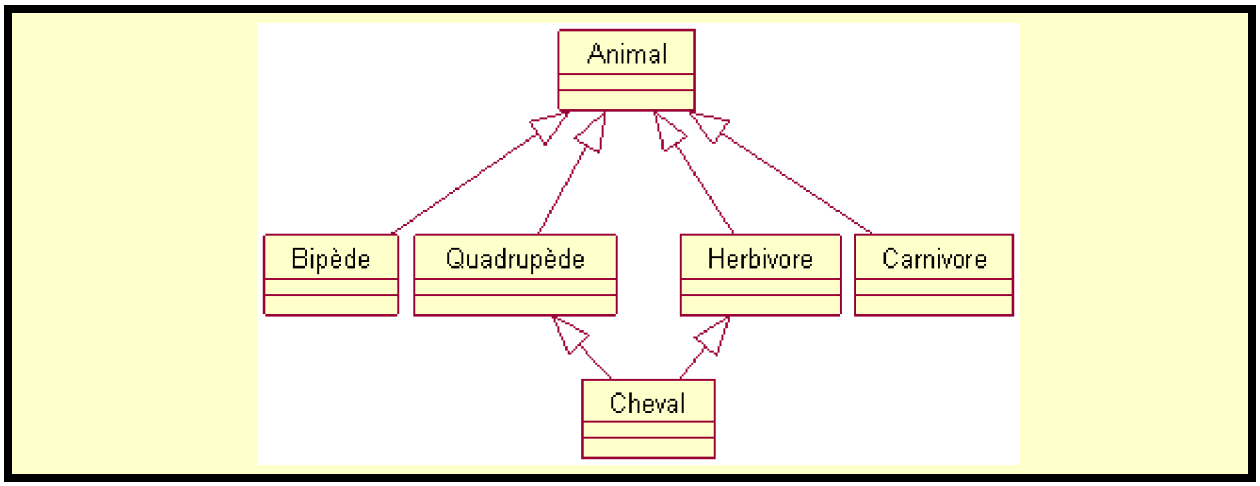
Pour information :

Une surdéfinition (ou surcharge) permet d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe, avec une signature différente, pour que le système puisse s'y retrouver.

Une redéfinition permet de fournir une nouvelle définition d'une méthode d'une classe ascendante et ainsi de substituer la description qui en été faite. Nous avons également le même nom que la méthode parente mais surtout avec une signature rigoureusement identique.

Héritage multiple et classe paramétrable :

Ainsi une classe peut être issue de plusieurs classes, c'est ce qui s'appelle l'héritage multiple.



Ainsi, le cheval est à la fois un quadrupède et un herbivore. Pour finir, il est bien évidemment possible de proposer l'héritage sur des classes paramétrables.

HERITAGE SIMPLE

Dans le chapitre précédent, nous avons évoqués les principes généraux de l'héritage sans tenir compte d'un langage quelconque. Ce chapitre sera donc consacré à l'étude de l'héritage simple codé avec le langage C++.

Déclaration de l'héritage simple (Classes sans constructeurs)

Pour indiquer une dérivation, il faut utiliser l'opérateur « : » (déjà utilisé dans les listes d'initialisation, ici il s'agit d'une liste de dérivation) suivit de nom de la classe de base. Pour un héritage classique, il est nécessaire de faire une dérivation publique.

Ex : class « fille » : public nom_ classe _ mère

Dans la déclaration dans le .h, nous mettons toujours la classe « mère » avant la « fille ».

Utilisation de tous les membres de la classe dérivée :

Lorsqu'une classe dérivée hérite d'une classe de base, elle s'approprie de tout le comportement de la classe de base. La classe dérivée récupère donc l'ensemble des attributs et des méthodes de la classe de base. C'est comme si nous avions une seule classe avec une fusion de tous les attributs et de toutes les méthodes.

Contrôle des accès :

Effectivement, l'enfant récupère tout ce que possède le parent. Mais attention, il existe quand même un petit problème d'accessibilité.

Les statuts des membres d'une classe

1. privé : Le membre (généralement l'attribut) n'est accessible qu'aux méthodes de la classe (publiques ou privées). De l'extérieur, il n'est pas possible d'atteindre ce membre. Même sa descendance ne peut pas y accéder directement. Seule l'amitié donne des droits d'accès privilégiés pour atteindre les attributs privés.
2. public : le membre est cette fois-ci accessible non seulement aux méthodes de la classe et aux fonctions amies, mais également à l'utilisateur de la classe, les enfants compris.

Il existe une règle de conception de hiérarchie qui stipule que la classe dérivée ne doit pas avoir besoin de connaître les détails de l'implémentation de la classe de base. Je pense qu'il est bon d'avoir en tête cette démarche. Elle correspond à un principe de sécurité maximale.

La classe de base ne possède pas toujours des méthodes de lecture pour atteindre les attributs. Il serait alors souhaitable d'être moins rigoureux et de permettre uniquement à la descendance (l'enfant ou l'enfant de l'enfant) l'accès à tous les attributs (ou éventuellement une partie) de la classe parente. Il existe un nouveau statut qui propose cette autorisation particulière. Il s'agit du statut protected .

Les statuts des membres d'une classe

3. protected : Les membres protégés se présentent comme des membres privés pour l'utilisateur de la classe de base, mais ils sont comparables à des membres publics pour la classe dérivée et pour toute la descendance.

Pour l'utilisateur des classes dérivées, les membres protégés continuent à être considérés comme des membres privés.

Accès des attributs dans l'héritage

Nous pensons souvent que pour avoir le maximum de souplesse, il suffit de déclarer protégés tous les attributs de la classe de base. Ainsi les enfants dans toute la hiérarchie peuvent y accéder sans aucune restriction. Toutefois cela peut être dangereux puisque un développeur par héritage peut modifier le comportement prévu par la classe de base.

Il ne faut pas que les attributs de la classe de base soient systématiquement protégés. Je pense que par réflexe, il faut d'abord les considérer comme privés, parce qu'il n'est pas toujours nécessaire que les enfants puissent accéder directement à tout ce que font les parents.

Dérivation publique, protégée et privée :

Dérivation public :

« class nom_deuxieme classe : public nom_premiere classe »

Les membres hérités d'une classe de base publique conservent leurs niveaux d'accès à l'intérieur de la classe dérivée. C'est le comportement normal prévu.

Dérivation privée :

« class nom_deuxieme classe : nom_premiere classe »

Les membres hérités publics et protégés d'une classe de base privée deviennent des membres privés de la classe dérivée.

Cette technique permet d'interdire, aux utilisateurs d'une classe dérivée, l'accès aux membres publics de sa classe de base. Cela sous-entend que seules les méthodes de la classe dérivée devront être utiliser, sinon il faudra redéfinir les méthodes de la classe de base.

Dérivation protected :

« class nom_deuxiemeclasse : protected nom_premiereclasse »

Les membres hérités publics et protégés d'une classe de base protégée deviennent des membres protégés de la classe dérivée. Nous retrouvons le même principe que pour une dérivation privée, la seule différence concerne les enfants éventuels de la classe dérivée, puisque dans ce cas là, les petits enfants peuvent atteindre des membres protégés.

RAPPEL :

Les membres *publics* de la classe de base sont accessibles « à tout le monde », c'est-à-dire à la fois aux méthodes, aux fonctions amies de la classe dérivée ainsi qu'aux utilisateurs de la classe dérivée.

Les membres *protégés* de la classe de base sont accessibles aux méthodes et aux fonctions amies de la classe dérivée, mais pas aux utilisateurs de cette classe dérivée.

Les membres *privés* de la classe de base sont inaccessibles à la fois aux méthodes ou aux fonctions amies de la classe dérivée et aux utilisateurs de la classe dérivée.

Constructions :

Pour une classe, à chaque fois que nous fabriquons un nouvel objet, le souci porte sur sa phase de création.

Lorsque nous parlons de l'état de l'objet, il s'agit en fait de la valeur de chacun des attributs. C'est le (ou les) constructeur qui gère ce genre de problème au moment de la phase de création.

La classe de base et la classe dérivée possèdent un constructeur par défaut :

La création d'un objet se déroule en quatre phases :

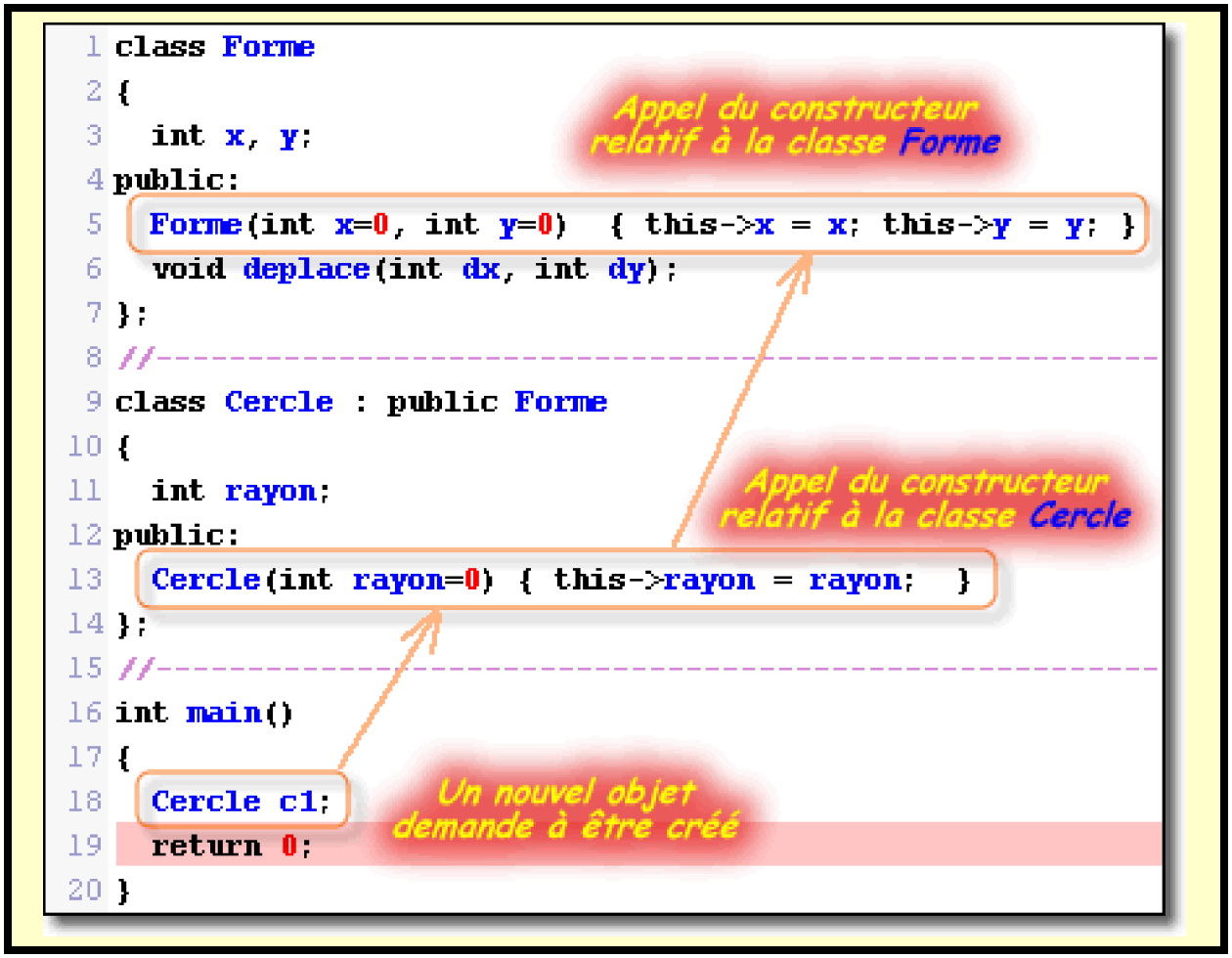
Allocation mémoire nécessaire pour contenir l'ensemble des attributs que comporte l'objet. Puisqu'il s'agit d'un héritage, l'objet alloue l'espace mémoire pour ses propres attributs ainsi que l'espace mémoire nécessaire aux attributs délivrés par l'héritage. Sinon l'objet ne serait pas complet.

Appel du constructeur de la classe dérivée. Ce constructeur est appelé mais pas encore exécuté. En effet, le constructeur de la classe dérivée ne s'occupe uniquement de ce qui fait la spécificité de la classe, c'est-à-dire, initialiser ses propres attributs. Avant d'effectuer cela, il faut être sûr que toute la structure générale soit elle-même bien initialisée. C'est la classe de base qui s'occupe justement de la structure de base et qui gère l'initialisation de ces propres attributs. Donc, avant l'exécution du constructeur de la classe dérivée, c'est le constructeur de la classe de base qui est appelé.

Appel et exécution du constructeur de la classe de base. A moins que la classe de base soit elle-même une classe dérivée d'une autre classe de base, les instructions qui constituent le corps du constructeur sont exécutées. Au minimum, ces instructions consistent à donner une valeur correcte aux attributs afin que l'objet par la suite n'ait pas de comportement aléatoire. (Si cette classe de base est également une classe dérivée, le système appelle d'abord le constructeur de sa classe de base avant l'exécution du constructeur).

Exécution du constructeur de la classe dérivée. Puisque la partie générale est bien initialisée, nous pouvons nous occuper de la partie spécifique à la classe dérivée. Les instructions du corps du constructeur sont donc exécutées. Il s'agit également d'initialiser les attributs relatifs à la classe dérivée.

VOIR ILLUSTRATION CI-DESSOUS :



La création d'un objet passe systématiquement par ces quatre phases. Ainsi, nous sommes sûr que l'objet est correctement initialisé. Chaque phase joue son rôle et s'occupe que d'une petite partie, ce qui rend la lecture plus facile. La maintenance s'en trouvera également d'autant plus simplifiée.

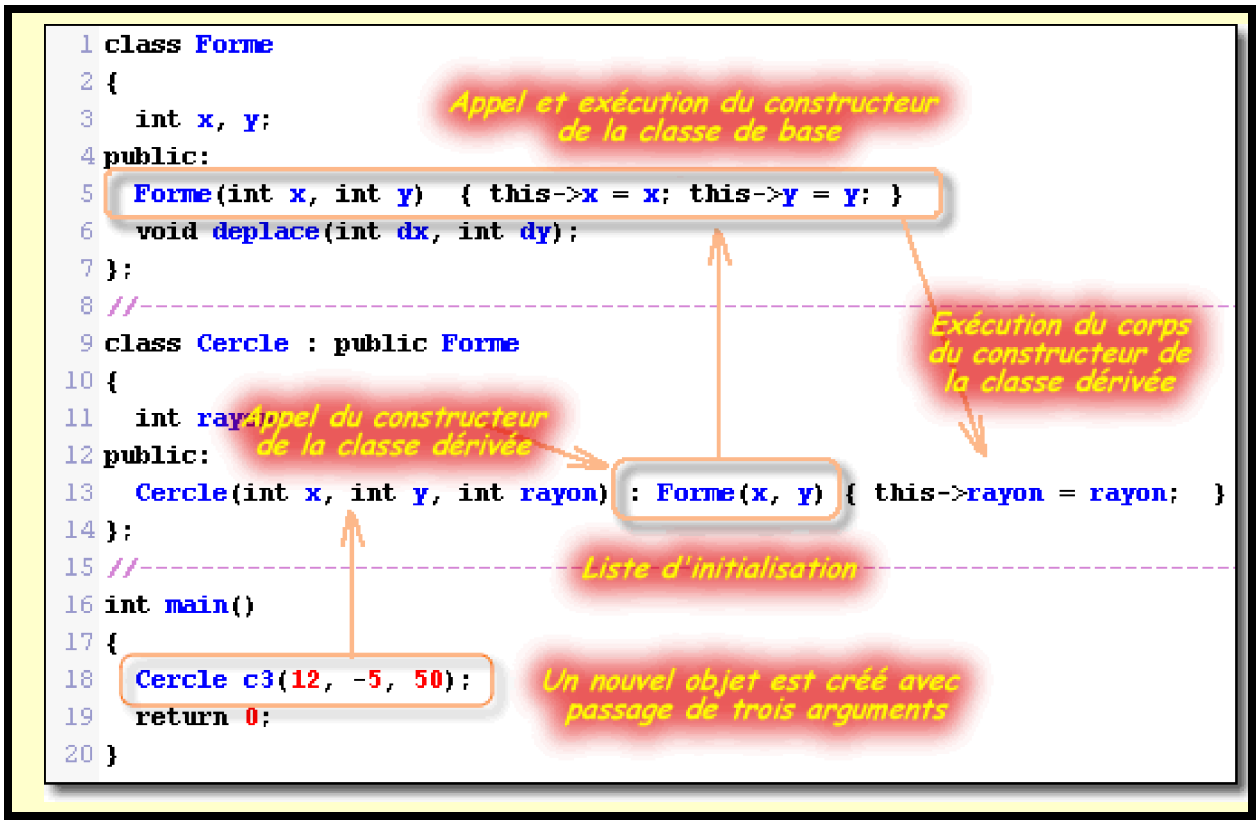
La classe de base dispose d'un constructeur par défaut et la classe dérivée d'un constructeur avec un paramètre :

Par rapport au scénario précédent, rien ne change vraiment, les quatre phases sont appelées dans le même ordre.

La classe de base dispose d'un constructeur avec paramètres :

Lorsque nous disposons d'un constructeur par défaut, l'appel se fait implicitement, c'est-à-dire automatiquement. Lorsque nous avons un constructeur avec arguments, cette fois-ci, il est nécessaire de faire un appel explicite afin d'envoyer les bons arguments au constructeur pour que l'initialisation des attributs correspondent à l'objet désiré.

Du coup, pour la classe dérivée, il est nécessaire de disposer au moins d'un constructeur qui fasse un appel explicite au constructeur de la classe de base en propageant les bons arguments nécessaires aux attributs généraux relatifs à la classe de base.




```
1  class Forme
2  {
3    int x, y;
4  public:
5    Forme(int x, int y)  { this->x = x; this->y = y; }
6    void deplace(int dx, int dy);
7  };
8  //-----------------------------------------------------------
9  class Cercle : public Forme
10 {
11   int rayon;
12 public:
13   Cercle(int x, int y, int rayon) : Forme(x, y) { this->rayon = rayon;  }
14 };
15 //-----------------------------------------------------------
16 int main()
17 {
18   Cercle c3(12, -5, 50);
19   return 0;
20 }
```

Quelque soit les situations, nous disposons toujours des quatre mêmes phases pour la création de l'objet et toujours dans le même ordre.

Comportements par défaut et héritage

Constructeur de copie – écriture implicite :

Par défaut, le constructeur de copie propose une copie membre à membre. Les attributs de l'objet à créer sont initialisés par rapport aux attributs de l'objet (qui sert de copie) passé en argument. Le même comportement par défaut reste vrai pour un objet de la classe dérivée.

Un constructeur de classe de base est toujours invoqué avant l'exécution du constructeur de la classe dérivée. Le constructeur de copie est également un constructeur. Il ne fait donc pas exception à cette règle, ce qui est logique.

Lorsque le constructeur de copie de la classe dérivée fait référence au constructeur de la classe de base, il passe par la liste d'initialisation.

Constructeur de copie – écriture explicite :

Vous savez que le comportement par défaut n'est pas toujours souhaitable, notamment lorsque nous disposons de variables dynamiques au sein même de la classe. L'héritage n'exclut pas cette problématique, il faut alors gérer la situation. En fait trois cas peuvent se présenter :

La classe de base possède au moins une variable dynamique, mais pas la classe dérivée : Dans ce cas, il faut uniquement redéfinir le constructeur de copie de la classe de base. Lorsque nous tenterons de créer un objet de la classe dérivée par copie, l'appel du constructeur de copie de la classe de base se fera implicitement sans aucun problème particulier.

La classe de base et la classe dérivée possèdent toutes les deux des variables dynamiques : Dans ce cas, il faut redéfinir, bien entendu, les deux constructeurs de copie. *Mais attention, lors de la définition du constructeur de copie de la classe dérivée, vous devez impérativement faire un appel explicite au constructeur de copie de la classe de base grâce à la liste d'initialisation. L'appel implicite au constructeur de copie de la classe de base ne marche pas dès que vous redéfinissez le constructeur de copie de la classe dérivée* .

La classe dérivée possède une variable dynamique, mais pas la classe de base : *Vous êtes donc obligé de redéfinir le constructeur de copie de la classe dérivée, ce qui implique que là aussi, vous devez faire un appel explicite au constructeur de copie par défaut à l'aide de la liste d'initialisation.*

Opérateur d'affectation :

Lorsque nous utilisons l'affectation par défaut, une copie membre à membre est réalisée.
L'ensemble des attributs est bien copié d'un objet vers l'autre, avec d'abord, la copie des attributs relatifs à la classe de base.

Destructeur :

Les différentes phases qui constituent la destruction de l'objet s'effectuent dans l'ordre inverse de la construction, c'est-à-dire :

1.Appel du destructeur de la classe dérivée.

2.Exécution du destructeur de la classe dérivée .

3.Appel et exécution du destructeur de la classe de base.

4.Libération de la mémoire utilisée par l'objet .

Ces quatre phases existent que le ou les destructeurs soient redéfinis ou pas. Les destructeurs sont à redéfinir dans le cas où les classes disposent de variables dynamiques.

Redéfinition des méthodes :

ATTENTION : Il ne faut pas mélanger la redéfinition et la surdéfinition.

1. Une surdéfinition (ou surcharge) permet d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe, avec une signature différente, pour que le système puisse s'y retrouver.

2. Une redéfinition permet de fournir une nouvelle définition d'une méthode d'une classe ascendante et ainsi de substituer la description qui en été faite. Nous avons également le même nom que la méthode parente mais surtout avec une signature rigoureusement identique. La redéfinition constitue la base du polymorphisme.

Même si c'est très rarement utilisé, vous pouvez faire un appel explicite à une méthode d'une classe ascendante.

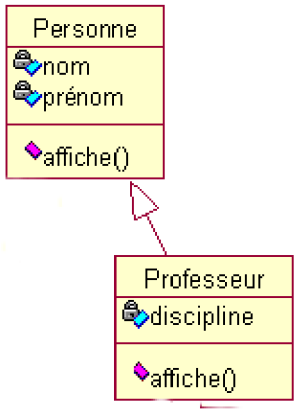
LE POLYMORPHISME

Le terme *polymorphisme* décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.

Le *polymorphisme*, en informatique, désigne *un concept de la théorie* des types, selon lequel un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence. Effectivement, nous avons découvert que les classes issues d'une même hiérarchie sont compatibles.

```
Classe mère &p = Classe fille( « Lagafe » , « Gaston » ) ;
```

L'objet p peut aussi bien faire référence à une « Classe mère » qu'à une « classe fille » ou à tout autre classe créée ultérieurement faisant partie de cette hiérarchie. C'est ce principe là qui offre une grande richesse à la programmation orientée objet.



Ici, nous avons en exemple, une « classe fille » ayant pour noms « Professeur » qui hérite d'une « classe mère » ayant pour noms « Personne ».

Maintenant, voyons quels sont les principes du polymorphisme.

Les interactions entre objets sont écrites selon les termes des spécifications définies, non pas dans les classes dérivées des objets, mais dans leurs classes de base. Cela permet d'écrire un code détaché des particularités de chaque classe, et d'obtenir des mécanismes suffisamment généraux pour être valides dans le futur, quand seront créées de nouvelles classes.

Le terme polymorphisme désigne en fait le polymorphisme du comportement, c'est-à-dire la possibilité de déclencher les méthodes différentes en réponse d'un même message. *Chaque classe dérivée hérite de la spécification des méthodes de ses classes de base*, mais a *aussi la possibilité de modifier localement le comportement de ces méthodes, afin de mieux prendre en compte les particularités de chacun.* C'est le principe même de la redéfinition des méthodes comme affiche().

Attention, cela parait facile, mais cependant, le polymorphisme engendre des restrictions en C++.

Par défaut, et contrairement au langage Java, les classes créées dans une hiérarchie dans le langage C++ n'intègrent pas le polymorphisme. Ici, cela entraîne une modification sur le comportement général de la (ou des) méthodes afin qu'effectivement le polymorphisme soit opérationnel dans notre hiérarchie.

Pour cela, certains critères doivent être respectés :

Pour qu'une méthode soit désignée comme polymorphe, *elle devra impérativement être virtuelle.*

Le polymorphisme est uniquement activé quand un *objet de classe dérivée est indirectement adressé via une référence ou un pointeur vers une classe de base.*

Dans la suite, nous allons découvrir pourquoi ces deux critères sont nécessaires.

Il existe *un lien de parenté entre les classes d'une même hiérarchie*, nous avons découvert qu'il existe, du coup, une certaine compatibilité. Elle consiste, ici en C++, en un système de conversions implicites, mises en œuvres automatiquement.

Ces conversions sont les suivantes :

D'un objet d'un type dérivé dans un objet d'un type de base (l'inverse n'est pas possible),

D'un pointeur (ou d'une référence) sur une classe dérivée en un pointeur (ou une référence) sur une classe de base.

Qu'est ce qu'une méthode virtuelle ?

Une méthode virtuelle est une méthode particulière invoquée au moyen d'un pointeur ou d'une référence sur une classe de base ; elle est liée dynamiquement au moment de l'exécution. L'instance invoquée est déterminée par le type de classe de l'objet adressé par le pointeur ou la référence. La résolution d'une méthode virtuelle est transparente à l'utilisateur.

Il suffit de placer le mot réservé « virtual » devant la méthode que nous désirons rendre virtuelle et le tour est joué. Par ailleurs, il n'est pas nécessaire de déclarer virtuelles les méthodes redéfinies dans les classes dérivées, elles le sont automatiquement.

Voici la syntaxe correspondante en C++ :

```
virtual void affiche() ;
```

Cette instruction indique au compilateur que les éventuels appels de la méthode affiche() doivent utiliser une ligature dynamique et non plus une ligature statique

Qu'est-ce qui se passe quand le compilateur rencontre cette instruction ?

Lorsque le compilateur rencontre cette instruction, il ne décidera pas de la méthode à appeler.

Il se contentera de mettre en place un dispositif permettant de n'effectuer le choix de la méthode qu'au moment de l'exécution de cette instruction, ce choix étant basé sur le type exact de l'objet ayant effectué l'appel.

Plusieurs exécutions de cette même instruction pouvant appeler des méthodes différentes.

Conclusion : Nous voyons que nous pouvons intégrer le polymorphisme vraiment très simplement. Il suffit de déclarer la ou les méthodes voulues de la classe de base comme virtuelles. La seule difficulté finalement, se situe au moment de la phase de conception durant l'élaboration des diagrammes UML. C'est effectivement à ce moment là qu'il faut décider si une hiérarchie de classes propose le polymorphisme ou pas. Dans l'affirmative, il est en effet souvent nécessaire de rajouter de nouvelles méthodes dans la classe ancêtre, alors que ce n'était pas spécialement prévu au départ.

En voyant cette simplicité, nous pourrions nous dire que nous n'avons pas besoin de nous poser autant de questions. Nous pouvons systématiquement spécifier toutes les méthodes comme virtuelles, puisque nous rajoutons un seul mot sur chacune des méthodes de la classe de base. L'étude suivante va nous montrer que ce n'est pas aussi simple.

Quel est en réalité le mécanisme qui tourne derrière tout cela ?

Nous venons de voir que le polymorphisme est très simple à implémenter dans le langage C++.

Nous pouvons nous passer de toutes autres connaissances subsidiaires. Or, cependant pour les programmeurs avertis, il est peut être intéressant d'avoir une compréhension plus fine du mécanisme interne, en prenant connaissance de l'implantation de la ligature dynamique.

D'une manière générale, *lorsqu'une classe comporte au moins une méthode virtuelle, le compilateur lui associe une table contenant les adresses des méthodes virtuelles correspondantes*.

D'autre part, *tout objet d'une classe comportant au moins une méthode virtuelle se voit attribuer par le compilateur, outre l'emplacement mémoire nécessaire à ses attributs, un emplacement supplémentaire de type pointeur, contenant l'adresse de la table associée à sa classe*.

Voyons maintenant, quelles sont les propriétés d'une méthode virtuelle !

Dans ce chapitre, nous allons faire un certain nombre de remarques afin que les méthodes virtuelles soient correctement implémentées.

Les méthodes virtuelles doivent impérativement existées pour qu'elles puissent être adressées à l'aide de la table correspondante. *Elles sont donc nécessairement non* inline (si vous la déclarez *inline*, le compilateur fabrique une véritable méthode).

Le mot *virtual* se place uniquement dans la déclaration de la classe. Lorsque vous définissez *la méthode à l'extérieur de la classe*, vous *ne* devez *plus re-spécifier le mot* virtual *devant la signature de la méthode.*

La *redéfinition* d'une méthode virtuelle dans une classe dérivée doit réaliser *une adéquation parfaite* (nom, signature, et type de retour) avec la méthode virtuelle déclarée dans la classe de base. Il n'est pas nécessaire de re-spécifier le mot *virtual*. *Si la re-déclaration dans la classe dérivée ne réalise pas une adéquation parfaite, la méthode n'est pas gérée comme une méthode virtuelle de la classe dérivée.* Dans ce cas là, *il s'agira tout simplement d'une surdéfinition*.

Il n'est pas obligatoire que toutes les classes dérivées redéfinissent impérativement toutes les méthodes virtuelles données par la classe de base. C'est notamment le cas lorsque la méthode héritée de la classe de base fait déjà tout ce qu'il faut. Par contre, rien n'empêche à une classe ultérieurement dérivée de ces classes dérivées de proposer, elle, la redéfinition de la méthode virtuelle, même si son proche parent ne l'a pas fait.

Lorsque nous avons une redéfinition des destructeurs dans une hiérarchie de classe qui comporte des méthodes virtuelles, il est généralement préférable que les destructeurs fassent également partie des tables des adresses des méthodes virtuelles.

En effet, lorsque nous réalisons un *delete* sur un pointeur d'une classe de base, le bon destructeur est alors pris en compte. Vous obtenez ce comportement en déclarant *virtuel* le destructeur de la classe de base.

```
39 int main( )
40 {
41   Personne *p;
42   p = new Professeur("NomProfesseur", "PrénomProfesseur", "Discipline");
43   p->affiche();
44   cout << endl;
45   delete p;
46   Eleve *e = new Eleve("NomElève", "PrénomElève");
47   e->ajoutNote(15); e->ajoutNote(8); e->ajoutNote(10);
48   p = e;
49   p->affiche();
50   delete p;
51   return 0;
52 }
```

Doit faire appel au destructeur de la classe Professeur

Doit faire appel au destructeur de la classe Elève

Lorsque qu'une méthode virtuelle est invoquée à l'intérieur d'un des constructeurs de la hiérarchie, c'est toujours la méthode virtuelle de la classe de base qui est sollicité.

En effet, puisque nous sommes en phase de création, les tables des adresses des méthodes virtuelles n'existent pas encore. Nous ne pouvons donc pas intégrer le polymorphisme sur un constructeur, il faut que l'objet soit d'abord créé. *Du coup, un* constructeur *ne peut jamais être* virtuel.

LES CLASSES ABSTRAITES

Comment est mise en œuvre une classe abstraite ?

Contrairement aux autres langages, comme le JAVA par exemple, il n'est pas possible d'indiquer directement qu'une classe est abstraite.

En réalité, pour que des méthodes soient réellement désignées comme abstraites ou bien indiquer qu'elles ne font vraiment rien, *elles doivent être explicitement initialisées à 0*. Donc à la suite de la signature de la méthode, vous devez placer « = 0 ; ». Ces méthodes sont appelées, dans le langage C++, des *méthodes virtuelles pures*.

Lorsque vous héritez d'une telle classe, vous devez impérativement, si vous désirez que votre classe devienne concrète, redéfinir toutes les méthodes virtuelles pures. Si cette condition n'est pas réalisée, la classe dérivée est elle-même une classe abstraite.

Finalement, *dans le langage C++*, une classe est abstraite lorsqu'elle dispose d'au moins une méthode abstraite, c'est-à-dire une méthode virtuelle pure.

Une fois, qu'une *classe* est définie *abstraite*, il n'est *plus possible de créer un objet relatif à cette classe*. Une erreur de compilation se produit si vous tentez l'expérience.

FONCTIONS ET CLASSES GENERIQUES <TEMPLATE>

Qu'est ce qu'une classe générique ?

La généricité permet d'avoir des fonctions et des classes paramétrables, c'est-à-dire que, au moment où nous en avons besoin, nous précisons le type à utiliser pour ladite fonction ou ladite classe. C'est le type qui est paramétrable. Ce concept nous permettra d'avoir des écritures plus concises et ainsi d'éviter de nombreuses surdéfinitions. La généricité est souvent appelée « template » (patron – évocation de la haute couture), ou également « modèle ».

Implémentation et utilisation d'une classe générique :

```
1  template <class Type>
2  Type min(Type x, Type y)
3  {
4    return x<=y ? x : y;
5  }
6  //------------------------
7  int main()
8  {
9    double y = min(2.0, -3.5);
10   int z = min(2, 18);
11   return 0;
12 }
```

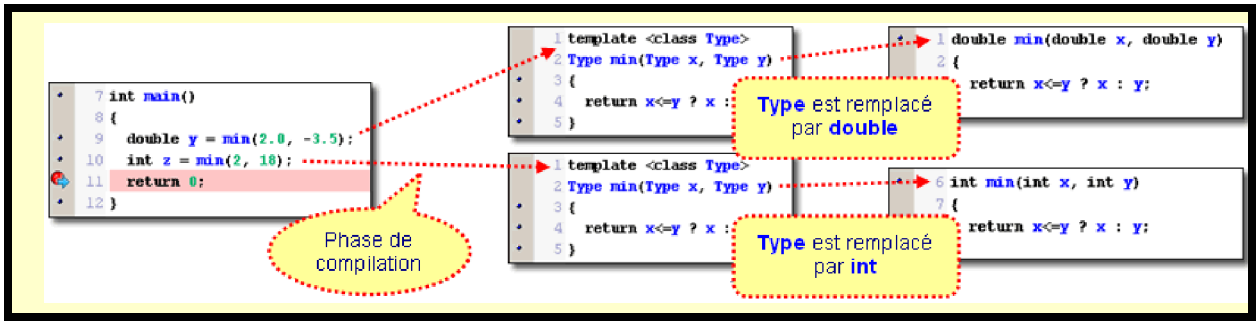
Le paramètre est utilisé dans le modèle et remplace les types prédéfinis « **double** » et « **int** » que nous avions au préalable.

« **template** » indique que nous mettons en place un nouveau modèle. Après ce mot réservé nous trouvons systématiquement les opérateurs de séparations « **< >** » à l'intérieur desquels nous spécifions les paramètres.

Si nous avons une liste de paramètres, nous devons utiliser l'opérateur virgule « **,** » pour les séparer. Il existe deux sortes de paramètres :

- ***Paramètre de type*** : c'est le cas le plus fréquent, et c'est justement l'intérêt des modèles. Lorsque nous utilisons un tel modèle, le compilateur détermine le type et fabrique la fonction en conséquence. Dans ce cas, c'est bien le type qui est paramétrable. Pour identifier un paramètre de type, vous devez utiliser le mot réservé « **class** » (class indique qu'il s'agit d'un type) suivi du nom du paramètre qui représentera le type. Ce nom est un identificateur classique, et c'est vous qui décidez de son appellation.
- ***Paramètre non type*** : qui représente une expression constante et qui est une déclaration de paramètre ordinaire du style : « **int taille** ». Nous exploiterons un exemple lorsque nous aborderons les classes génériques.

Qu'est ce qui se passe au niveau de la compilation ?



Le schéma ci-dessus, nous montre clairement, qu'est ce qui se passe lors de l'utilisation des « templates » après compilation du programme principal faite.

Grâce aux « templates », nous pouvons créées et adaptées à volonté des méthodes utilisable à tout moment. C'est le moyen, le plus efficace pour concevoir une boite à outils.

De cette façon, l'utilisateur de la méthode avec un type primaire désiré pourra utilisé votre méthode sans aucun problème.

La classe contenant des « templates » s'appelle en langage de programmeur « un modele ».

Le modèle nous sert en fait à composer, une fois pour toute, les lignes de codes nécessaires à l'élaboration de fonctions surdéfinies et c'est le compilateur qui finalement travaille pour nous. Cela nous évite d'écrire des lignes de codes identiques.

Fonctions génériques :

Pouvons-nous mettre le mot « inline » avec les fonctions génériques ?

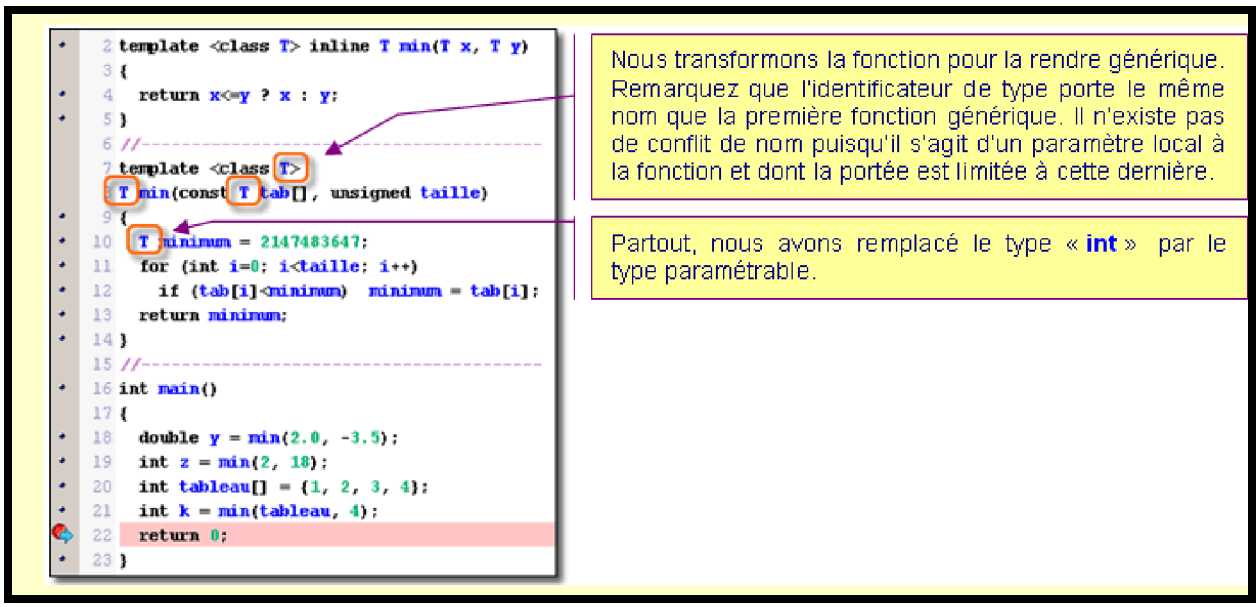
Nous pouvons modéliser n'importe quel type de fonction et le fait qu'elle puisse être « inline » ne change absolument rien.

Pouvons-nous surdéfinir une fonction générique ?

L'avantage de ces modèles c'est d'écrire très peu de ligne de codes.

Nous pouvons donc faire coexister des fonctions génériques avec des fonctions classiques. Le tout, c'est de proposer des signatures différentes pour que le compilateur soit à même de comprendre le souhait du programmeur et donc de résoudre la surdéfinition.

Pouvons-nous rendre une fonction standard en une fonction générique ?



Classes génériques - Conception :

Les classes aussi peuvent être génériques.

La syntaxe demeure identique à l'écriture des fonctions génériques. Partout où le type « int » ou bien tout autres types primaire sont utilisés, vous le remplacez par le paramètre « Type » puisque c'est lui qui est défini dans le modèle.

Mise à part l'écriture de la partie paramétrable, les classes génériques demandent très peu d'investissement supplémentaire. Il ne faut pas hésiter à utiliser cette technique.

Classes génériques – Utilisation :

Il existe toutefois une petite différence dans l'utilisation des classes génériques par rapport aux fonctions génériques.

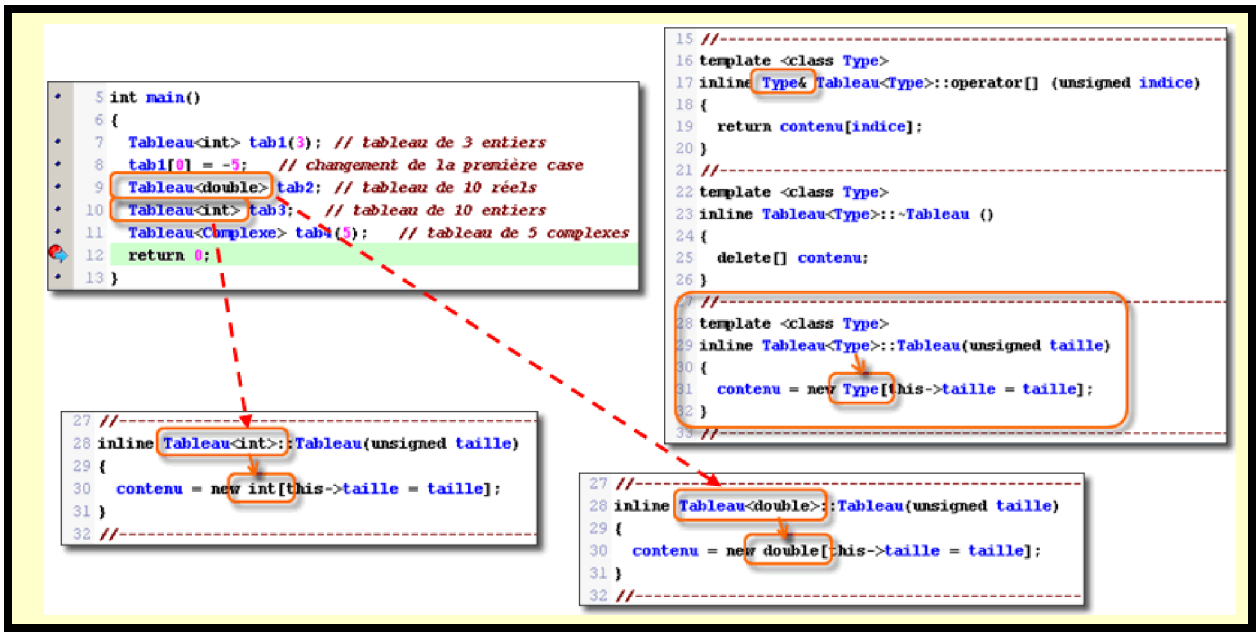
Le compilateur, au moment de l'appel d'une fonction générique, contrôle la signature proposée et détermine le type demandé pour effectivement fabriquer la fonction avec le type désiré. Sans spécification supplémentaire, le compilateur arrive à connaître le type demandé.

Dans le cas d'un objet, c'est différent. Lorsque nous déclarons cet objet, le type de certains attributs n'apparaît pas au moment de la déclaration puisque seul le nom de l'objet est visible.

En conséquence, il est nécessaire, durant la création d'indiquer le ou les types voulus. La syntaxe est simple d'utilisation. Après le nom de la classe, vous devez préciser le type voulu entre « < > ».

Le nom de votre type devient alors l'argument de votre classe générique. Du coup, la syntaxe de votre code est très lisible, en ce sens que nous voyons bien qu'il s'agit, par exemple, d'un tableau d'entiers ou d'un tableau de complexes.

Classes génériques – Définition des méthodes :



D'habitude, lorsque nous développons des classes, nous plaçons la déclaration de la classe dans un fichier en-tête alors que la définition de ses méthodes se trouve dans le fichier source correspondant qui porte le même nom, mais dont l'extension de fichier est « *.cpp ».

Dans le cas d'une classe générique, c'est différent. N'oubliez pas qu'il s'agit d'un prototype (d'un patron) qui servira à la fabrication (réelle) de plusieurs classes de types différents. Dans ce contexte, la déclaration de la classe ainsi que la définition des méthodes doivent se trouver entièrement dans le fichier en-tête.

En effet, tout ce qui est générique est utilisé uniquement par le « préprocesseur » du compilateur, et n'oubliez pas que cette phase particulière de la compilation propose de transformer un texte par un autre. Ce n'est que lorsque le texte a été mis en place que la compilation réelle s'effectue.

En revenant sur notre exemple, à l'utilisation, nous avons besoin d'avoir d'une part un tableau d'entiers et d'autre par un tableau de réels. Bien que le code interne soit identique, il existe tout de même des différences. Nous ne traitons pas des réels comme des entiers. Il faut donc qu'il y ait, par exemple, un constructeur pour un tableau d'entier et un constructeur pour un tableau de réel puisque la réservation mémoire dynamique est totalement différente. Les réels prennent plus de place en mémoire. Pour un tableau de complexe, c'est encore pire.

Finalement, les méthodes sont également paramétrées. Dans ce contexte, à la définition de chaque méthode, vous devez impérativement utiliser la syntaxe complète des « templates ».

Classes génériques – Paramètre non type :

Nous avons vu qu'il existait deux types de paramètres pour les fonctions et les classes génériques : les paramètres de type, et les paramètres non type.

```
1  #ifndef Tableau_H
2  #define Tableau_H
3  //---------------------------------------------------------------------
4  template <class Type, unsigned taille> class Tableau
5  {
6    Type contenu[taille];
7  public:
8    Type& operator[] (unsigned indice) { return contenu[indice]; }
9  };
10 //---------------------------------------------------------------------
11 #endif
```

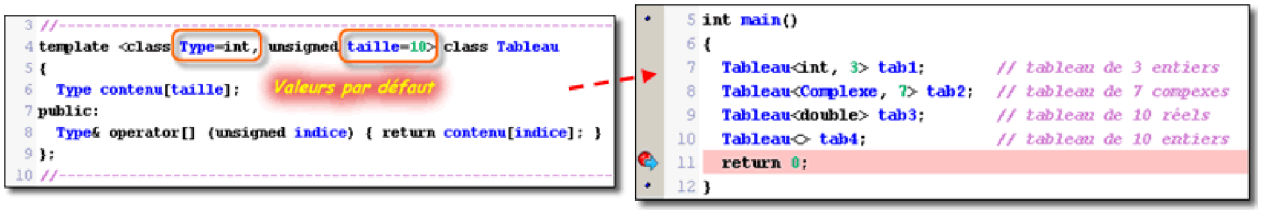
Paramètre non type qui sert de dimension au tableau, donc sans variable dynamique

Dans le cas des classes génériques, le paramètre « non type » peut s'avérer particulièrement utile, notamment pour implémenter un tableau de type quelconque en spécifiant, dès le départ, la dimension du tableau, et c'est ce dernier élément qui servira de paramètre non type. En effet, dans ce cas là, le paramètre attend une valeur entière non signée. Il ne s'agit en aucun cas de proposer un type mais bien une valeur.

```
3  int main()
4  {
5    Tableau<int, 3> tab1; // tableau de 3 entiers
6    tab1[0] = -5;    // changement de la première case
7    Tableau<int, 3> tab2 = tab1; // construction de copie
8    Tableau<int, 3> tab3;    // tableau de 3 entiers
9    tab3 = tab1;     // affectation
10   tab3[2] = 3;     // changement de la dernière case
11   return 0;
12 }
```

```
(Tableau<int, 3>) tab1 = { { -5, 256, 1 } }
(Tableau<int, 3>) tab2 = { { -5, 256, 1 } }
(Tableau<int, 3>) tab3 = { { -5, 256, 3 } }
```

En prenant un paramètre non type pour spécifier la dimension, le code de la classe devient extrêmement simple. En effet, comme la dimension est connue en moment de la création de la classe, il est possible de mettre comme attribut un tableau et non plus un pointeur vers une variable dynamique. Du coup, nous n'avons plus besoin de redéfinir tous les comportements par défaut. Il suffit de redéfinir l'opérateur de crochets « [ ] ».



STANDARD TEMPLATE LIBRARY

Introduction :

Le langage C++ est un langage intéressant puisqu'il offre avec une grande souplesse d'écriture.

Par rapport au langage C dont il hérite, il propose beaucoup plus de choses comme, bien entendu, toute la philosophie objet, mais également la possibilité de redéfinir les opérateurs ainsi que la possibilité de fabriquer des modèles.

Malheureusement, il conserve aussi quelques tares issues de son prédécesseur, notamment les tableaux et les chaînes de caractères.

N'oubliez pas que les tableaux ne sont pas de véritables tableaux, mais des pointeurs vers des mémoires consécutives. Même si elle traite de caractères, la chaîne n'est pas mieux lotie puisqu'elle est également considérée comme un tableau (et donc comme un pointeur).

Les ingénieurs de Hewlett Packard ont développé beaucoup d'autres classes comme les nombres complexes, les nombres binaires, des classes conteneurs comme les listes, les piles, les ensembles, etc. Tous ces éléments sont rassemblés dans une bibliothèque standard et sont construits sous forme de modèles. Cette librairie s'appelle STL (Standard Template Library).

Quels sont les contenus disponible dans la STL ?

Les conteneurs séquentiel :

Il est très fréquent d'avoir besoin de stocker dans une même entité mémoire un ensemble d'objets de même type. En plus, il peut être intéressant d'utiliser un système qui fonctionne quelque soit l'objet que nous développons. Les conteneurs séquentiels permettent effectivement de stocker des objets en séquence, c'est-à-dire les uns à la suite des autres, ce qui permettra ensuite de parcourir le conteneur dans un ordre particulier.

Il existe plusieurs conteneurs séquentiels :

vector : cette classe est un vecteur qui représente un tableau de haut niveau (les cases sont consécutives). Avec cette classe, il est possible d'atteindre n'importe quelle élément du tableau facilement grâce à l'indexation « *[ ]* ». Nous pouvons insérer de nouveaux éléments, en supprimer, etc.

list : cette classe implémente une liste doublement chaînée. Avec cette classe, il est plus facile de supprimer un élément particulier par rapport au vecteur (en effet pour un vecteur, si nous supprimons une case, il est nécessaire de décaler les cases suivantes vers le bas). Par contre, cette liste utilise systématiquement deux pointeurs pour parcourir la séquence ce qui prend plus de place en mémoire.

deque : cette classe est très peu utilisée et offre le même comportement mais plus spécialisé que la classe vecteur. Sa spécialisation consiste à pouvoir facilement ajouter ou retirer le premier élément. Cette classe est une abstraction d'une file pour laquelle le premier élément est retiré chaque fois.

Existe t – il des adaptateurs pour les conteneurs séquentiels ?

La bibliothèque standard dispose de trois patrons particuliers qui s'ajoutent aux conteneurs séquentiels en modifiant leurs comportements classiques.

Généralement, il s'agit d'une restriction et d'une adaptation à des fonctionnalités données :

stack : ce patron est destiné à la gestion des piles *LIFO* (*L*ast *I*n, *F*irst *O*ut).

queue : ce patron est destiné à la gestion des piles de type *FIFO* (*F*irst *I*n, *F*irst *O*ut).

priority_queue : un tel conteneur ressemble à une file d'attente, dans laquelle on introduit toujours des éléments en fin.

Qu'est ce qu'un patron, ici ?

(a completer)

Les conteneurs associatifs :

Les éléments d'un conteneur associatif ne sont plus placés dans un ordre particulier. Pour retrouver une entité, nous ferons appel dans ce cas là à une clé qui nous orientera vers la valeur recherchée.

map : ce conteneur représente une correspondance entre deux entités sous la forme d'une paire *clé/valeur* , la *clé* étant utilisée pour la recherche et la *valeur* contenant les données que l'on souhaite utiliser. Par exemple, un répertoire téléphonique.

multimap : ce conteneur représente une multicorrespondance, il peut donc stocker plusieurs occurrences d'une même clé.

set : ce conteneur représente la théorie des ensembles et contient une valeur de clé unique et supporte les requêtes concernant sa présence ou non. En effet, grâce à ce conteneur, nous pourrons indiquer si un élément fait parti de l'ensemble ou pas.

multiset : représente également la théorie des ensembles avec en plus la possibilité de comptabiliser le nombre de fois qu'un même élément apparaît dans l'ensemble. Ce conteneur autorise donc la présence de plusieurs éléments identiques, ce qui n'est pas le cas pour le conteneur *set* .

Les algorithmes génériques :

De même que la STL est composée de classes génériques, elle est également composée de fonctions génériques qui fournissent des opérations supplémentaires bien utiles sur les différents conteneurs étudiés précédemment.

Ces fonctions permettent d'effectuer un certain nombre de traitements différents, comme des insertions, des copies, des recherches, etc., dans une suite d'éléments d'un des conteneurs utilisé.

L'intérêt de ces fonctions génériques, que l'on appelle aussi algorithmes génériques, c'est qu'elles sont opérationnelles pour tous les types de conteneur, comme vector, list, map, etc.

La liste ci-dessous vous donnera une idée de quelques fonctions génériques intéressantes :

copy : copie d'une séquence dans une autre,

count : comptabilise le nombre d'élément présent dans une suite,

generate : génération de valeurs par une fonction,

find : recherche d'une valeur particulière,

max_element : recherche du maximum,

min_element : recherche du minimum,

replace : remplacement de valeurs,

rotate : permutation de valeurs,

remove : suppression de valeurs,

unique : suppression de doublons,

sort : tri d'une séquence,

merge : fusion de deux conteneurs.

reverse : inverse l'ordre des éléments dans un conteneur.

Quelles sont les classes les plus utilisées en développement d'application ?

Nous avons commencé cette étude en indiquant que le langage C++ ne possédait pas certains éléments qui sont indispensables à la programmation de haut niveau, comme les chaînes de caractères.

Par ailleurs, dans nos différentes études, nous avons en œuvre de toute pièce une classe qui représente les nombres complexes.

Il faut savoir qu'une telle classe fait partie de cette bibliothèque.

Voici une liste non exhaustive de classes qui me paraissent intéressante :

string : cette classe représente une chaîne de caractères et possède beaucoup de méthodes qui permettent tous les traitements possibles sur ses caractères. C'est la classe que l'on utilisera des que c'est possible.

complex : cette classe représente les nombres complexes.

bitset : cette classe représente les nombres binaires ou plus précisément un ensemble de bits et possèdent des méthodes associées à leurs traitements. Elle permet de faire des traitements divers sur les chiffres binaire, hexa, décimaux, etc...

Vector, list, map.

Maintenant, le mieux pour nous est de voir comment marche ces différentes classes tellement pratique.

La classe « string » :

Un objet de type « string » contient a un instant donné, une suite formée d'un nombre quelconque de caractères.

Sa taille peut évoluer dynamiquement au fil de l'exécution du programme.

En fait, cette chaîne réserve un bloc mémoire suffisant pour stocker un certain nombre de caractères.

Si la chaîne désirée est plus grand que cette zone réservée, la classe augmente automatiquement ce bloc en proposant une nouvelle allocation mémoire et en prenant la précaution d'avoir un bloc plus grand que nécessaire afin de répondre rapidement à une petite augmentation de la taille de la chaîne.

La notion de caractère de fin de chaîne n'existe plus pour cette classe, et ce caractère de code nul peut apparaître au sein de la chaîne, éventuellement à plusieurs reprises.

Comment peut-on utiliser cette fameuse classes ?

Pour utiliser cette classe, nous devons ne pas oublier de faire une directive (une inclusion) de compilation tout au début du programme.

Cette inclusion est : `#include <string>.`

Mais, aussi, nous devons utiliser l'espace de nom « standard » en inscrivant « `using namespace std ;` »

Description des différentes méthodes incluent dans la classe « string ».

1. append : ajoute une chaîne, à la fin d'une autre. Il s'agit d'une concaténation qui peut être également traitée par l'opérateur += .

2. assign : affecte à l'objet une nouvelle chaîne de caractères.

3. at : permet de lire ou de récupérer un caractère à la position indiquée. La première position est 0. Il est nécessaire de donner une position compatible et inférieure à la taille de la chaîne sinon une exception est levée. Cette méthode est similaire à la redéfinition de l'opérateur « [ ] ».

4. capacity : retourne la dimension du bloc mémoire réservé. Cette méthode fournit donc le nombre maximal de caractères qu'on pourra introduire, sans qu'il soit besoin de procéder à une nouvelle allocation mémoire. Les méthodes reserve et resize pouront être utilisée pour agir directement sur la capacité de ce bloc mémoire. La valeur retournée est toujours plus grande ou égale à la valeur que retourne la méthode size.

5. clear : vide entièrement la chaîne de caractères.

6. compare : cette méthode gère l'ordre alphabétique et retourne une valeur numérique négative ou positive suivant le placement de la chaîne par rapport à celle qui est passée en argument. Une valeur négative indique que la chaîne se trouve avant celle qui est passée en argument. Une valeur positive dans le cas contraire, et une valeur nulle dans le cas où les deux chaînes sont rigoureusement identiques. La plupart du temps, il sera préférable d'utiliser les opérateurs relationnels « <, <=, ==, !=, >, >= » pour gérer ce genre de problème.

7. c_str : cette méthode permet de passer d'une chaîne de type « string » vers une chaîne classique du C++ (const char *).

Attention, il est possible de récupérer cette chaîne sans toutefois pouvoir la modifier puisque une constante est déclarée.

8. empty : retourne true si la chaîne est vide (sans aucun caractère), sinon retourne false .

9. erase : efface une partie de la chaîne ou un caractère spécifié en argument.

10. find, rfind, find_first_of, find_last_of, find_first_not_of, find_last_not_of : effectuent des recherches sur une partie de la chaîne ou sur un caractère spécifié en argument.

11. insert : permet d'insérer une autre chaîne ou bien un ou plusieurs caractères donnés.

12. length : retourne la longueur de la chaîne de caractères. Similaire à la méthode size.

13. replace : remplace une partie de chaîne.

14. reserve : réserve un bloc mémoire dont la taille est fixé par l'argument. Cette méthode doit être rarement utilisé, juste dans le cas où la performance en terme de rapidité est primordiale ou alors, éventuellement, dans le cas où nous sommes très limité dans la capacité de la mémoire.

15. resize : donne une nouvelle dimension à votre chaîne de caractères. Attention ! à utiliser avec beaucoup de précaution.

16. size : retourne la longueur d'une chaîne de caractères. Similaire à length.

17. substr : retourne une partie de chaîne.

18. swap : assure la permutation de deux chaînes de caractères.

19. begin et end : ces opérations retournent des itérateurs au début et à la fin de la chaîne. Un itérateur est une abstraction d'un pointeur de classe générique, fourni par la bibliothèque standard. Ce sujet sera traité ultérieurement lorsque nous utiliserons la classe « vector ».

20. getline(istream &, string, char délimiteur) : cette fonction est très utile lorsque nous devons saisir tout un texte à partir du clavier. En effet, lorsque nous réalisons une saisie classique, nous sommes obligé de le faire ligne par ligne. Cette fonction permet justement de saisir un texte qui comporte plusieurs lignes. Il est alors nécessaire de choisir un caractère (délimiteur) qui servira de caractère de fin de texte.

A la suite de tout cela, vous devez vous en douter qu'il faut redéfinir certains opérateurs pour nous permettre par la suite d'avoir une écriture plus simple.

Attention, certains opérateurs disposent de méthodes équivalente.

= : affecte une chaîne à une autre. Cet opérateur est équivalent à la méthode *assign*.

[ ] : joue le même rôle que pour une chaîne de caractères classique du C++. Est similaire à la méthode *at*.

+= : ajoute une chaîne à la fin d'une autre. Cet opérateur joue le même rôle que la méthode *append*.

+ : cet opérateur assure la concaténation de deux chaînes de caractères pour former une troisième chaîne.

==, !=, <, >, <=, >= : ces opérateurs renvoient *true* ou *false* suivant la comparaison qui est faite sur deux chaînes de caractères. Les différentes comparaisons évaluent en fait l'ordre alphabétique.

>>,<< : Il est aussi possible d'afficher ou de saisir des chaînes de caractères de type « string » en utilisant la classe « iostream ».

Pouvons-nous faire de la concaténation de deux « string » ?

Oui, c'est tout a fait possible grâce à l'opérateur « + » qui à été redéfini pour permettre la concaténation. Cependant, il faut respecter ces deux règles qui suivent pour que cela marche bien.

de deux objets de type «string»,

d'un objet de type «string» avec une chaîne usuelle (char *) ou avec un caractère, et ceci dans n'importe quel ordre.

De même pour l'opérateur « += » a également été redéfini pour la concaténation.

Pouvons nous, ou bien avons-nous la possibilitée de rechercher une chaîne de caractères parmi un texte ?

Oui, cela est tout a fait possible, car dans cette classe se trouve des méthodes qui permettent de trouver ce que l'on cherche.

Ces méthodes permettent de retrouver la première ou la dernière occurrence d'une chaîne ou d'un caractère donnés, d'un caractère appartenant à une suite de caractères donnés, d'un caractère n'appartenant pas à une suite de caractères donnés. Ces méthodes retournent l'indice correspondant au premier caractère concerné.

Si la recherche n'aboutit pas, on obtient une valeur d'indice en dehors des limites permises pour la chaîne, ce qui rend quelque peu difficile l'examen de sa valeur.

Heureusement, dans la classe *string*, il existe un attribut constant public et statique appelé *npos* (qui veut dire : no position) qui généralement est initialisé à la valeur -1. Lorsque vous utilisez une des méthodes de recherche, il serait souhaitable de tester la valeur retournée avec cette constante statique *npos* de string afin de savoir si votre recherche a aboutie.

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4  //------------------------------------------
5  int main(  )
6  {
7    string message = "Bonjour";
8    int position = message.find('n');
9    if (position != string::npos)
10     cout << "La position est : " << position;
11   return 0;
12 }
```

Résultat --> La position est : 2

D'après le tableau de la page 44 du cour, nous pouvons voir qu'il existe des méthode spécialement faite pour cela (par exemple le numéro 10).

Passons en détail ces méthodes de recherche.

Méthode « find () » :

La méthode find permet de rechercher, dans une chaîne donnée, la première occurrence :

1. *d'une autre chaîne (on parle alors de sous-chaîne) fournie en argument,*
2. *d'une autre chaîne usuelle, soit d'un caractère donné.*

Méthode « rfind() » :

De manière semblable, la méthode rfind permet de rechercher la dernière occurrence d'une autre chaîne ou d'un caractère.

Autres méthodes de recherche :

La méthode find_first_of recherche la première occurrence de l'un des caractères d'une autre chaîne (string ou usuelle), tandis que find_last_of en recherche la dernière occurrence.

La méthode find_first_not_of recherche la première occurrence d'un caractère n'appartenant pas à une autre chaîne, tandis que find_last_not_of en recherche la dernière.

Pour les autres méthodes, telle que l'insertion, la suppression, le remplacement, je pense que vous avez compris. Sinon, veuillez me contacter. Merci.

La Classe « bitset » :

Dans la STL, une telle classe existe, elle est représentée par la classe générique « bitset » et permet également de manipuler efficacement des suites de bits dont la taille est spécifiée en paramètre du modèle. L'affectation n'est donc possible qu'entre suites de même taille.

Comment ce passe la phase construction ?

Il existe quatre constructeurs :

1. sans argument : on obtient une suite de bits nuls,
2. à partir d'un *unsigned long* : on obtient la suite correspondant au motif binaire contenu dans l'argument,
3. à partir d'une chaîne de caractères « *string* » (attention toutefois, il s'agit d'une construction de type « *explicit* »).
4. la construction par copie est implémentée et donc possible.

```
1  template <unsigned long bits>
2  class bitset
3  {
4  public:
5    bitset();                          // constructeur par défaut
6    bitset(unsigned long);
7    explicit bitset(const string&);    ATTENTION ! Construction explicite
8    bitset(const bitset<bits>&);       // constructeur de copie
9    ...
10 };
```

Ici, que signifie le terme « explicit » ?

explicit : Par défaut, les constructeurs permettent de réaliser des conversions implicites lorsque cela est nécessaire. Il peut arriver que dans certaines situations cette conversion automatique soit gênante. Nous pouvons alors bloquer le comportement par défaut du constructeur en demandant que l'argument passer au constructeur au moment de la création de l'objet soit rigoureusement du type attendu. Pour offrir cette alternative, il est nécessaire de préfixer le constructeur du mot réservé « explicit ». Lorsque nous avons un constructeur avec un seul paramètre, il est possible d'utiliser l'opérateur « = ». Si vous déclarez un constructeur de type « *explicit* », cette opportunité n'est plus possible (« = » --> création implicite de l'objet). Il est alors nécessaire d'utiliser systématiquement les parenthèses.

Faut – il redéfinir les opérateurs pour effectuer une opération binaire en utilisant cette classe ?

Nous disposons des opérateurs classiques de manipulation globale des bits « &, |, ~, ^, <<, >>, &=, |=, ~=, ^=, <<=, >>=, ==, != » qui fonctionnent de la même façon que les mêmes opérateurs appliqués à des entiers.

Nous pouvons accéder à un bit de la suite à l'aide de l'opérateur « [ ] » ; il déclenche une exception « out_of_range » si son opérande n'est pas dans les limites permises (les exceptions seront traitées ultérieurement).

>>, << : il est également possible d'afficher ou de saisir des « bitset » en utilisant les opérateurs des classes « iostream ».

Quelles sont les méthodes qui se trouvent dans la classe « Bitset » ?

La classe « bitset » dispose de méthodes supplémentaires qui, associées à l'opérateur « [ ] », permettent de satisfaire les programmeurs lorsqu'il s'agit de manipuler efficacement une information binaire.

any( ) : existe-t-il au moins un bit dans le nombre binaire qui est à 1 ?

none( ) : inverse de la précédente, tous les bits sont-ils à 0 ?

count( ) : détermine le nombre de bits mis à 1.

size( ) : indique la capacité (nombre de bits) du nombre binaire.

set( ) : tous les bits du nombre sont mis à 1.

set(position) : le bit désigné par position est mis à 1.

reset( ) : tous les bits sont mis à 0.

reset(position) : Le bit désigné par position est mis à 0.

flip( ) : inverse tous les bits du nombre.

flip(position) : inverse le bit désigné par position.

test(position) : teste si le bit désigné par position est à 1. La méthode renvoie *true*, et *false* suivant le résultat du test.

to_string( ) : retourne la valeur binaire sous forme de « *string* ».

to_ulong( ) : retourne la valeur binaire sous forme de valeur entière de type « *unsigned long* ».

Utilisation de la classe « bitset » :

Cette inclusion doit être faite pour l'utilisation de celle-ci :

```
#include <bitset>
```

La classe « vector » :

La classe vector sera presque systématiquement utilisée pour implémenter les tableaux. Notre première approche sera justement dans ce sens.

Toutefois, la classe vector représente bien plus que cela. Elle fait également partie de l'ensemble des conteneurs, et notamment des conteneurs de type séquentiel.

Notre deuxième approche sera donc liée à cette notion, et nous en profiterons pour généraliser le concept de conteneur.

Pour finir, nous utiliserons les algorithmes génériques afin de résoudre un certain nombre de critères qui ne sont pas spécialement intégrés dans cette classe.

ATTENTION : Ne pas oublier de rajouter la directive de compilation correspondante.

Cette inclusion est : #include <vector>.

La classe vector remplace aisément les tableaux classiques en offrant des manipulations simples et intuitives. Ainsi, il est possible de construire des tableaux de type quelconque, en indiquant le nombre de cases requis. Il est également possible d'initialiser un tableau avec une valeur particulière pour toutes les cases du tableau ou même de spécifier une valeur d'initialisation différente pour chacune des cases du tableau.

Avec ce tableau, un certain nombre d'opérations peuvent être réalisées simplement, comme :

= : l'affectation est possible entre deux tableaux de même type (Attention, il faut aussi qu'ils comportent le même nombre de cases).

[ ] : l'opérateur d'indexation à bien évidemment été redéfini pour supporter le comportement classique d'un tableau, puisque cet opérateur à été spécialement créé pour les tableaux.

==, !=, <, <=, >, >= : Il est de plus possible de comparer le contenu de deux tableaux entre eux en utilisant les opérateurs classiques de comparaison.

Vu la simplicité d'utilisation, il est impératif d'utiliser cette classe pour implémenter les tableaux.

La classe « vector » engendre des conteneurs vecteurs de différents types.

Quelles sont les propriétées communes au conteneur, vector, list, deque ?

J'ai déjà donné une brève description de ces trois types de conteneurs, je ne vais donc pas m'y étendre. Ce qui nous intéresse, ici, c'est de voir le comportement commun qui donne une certaine homogénéité dans la STL.

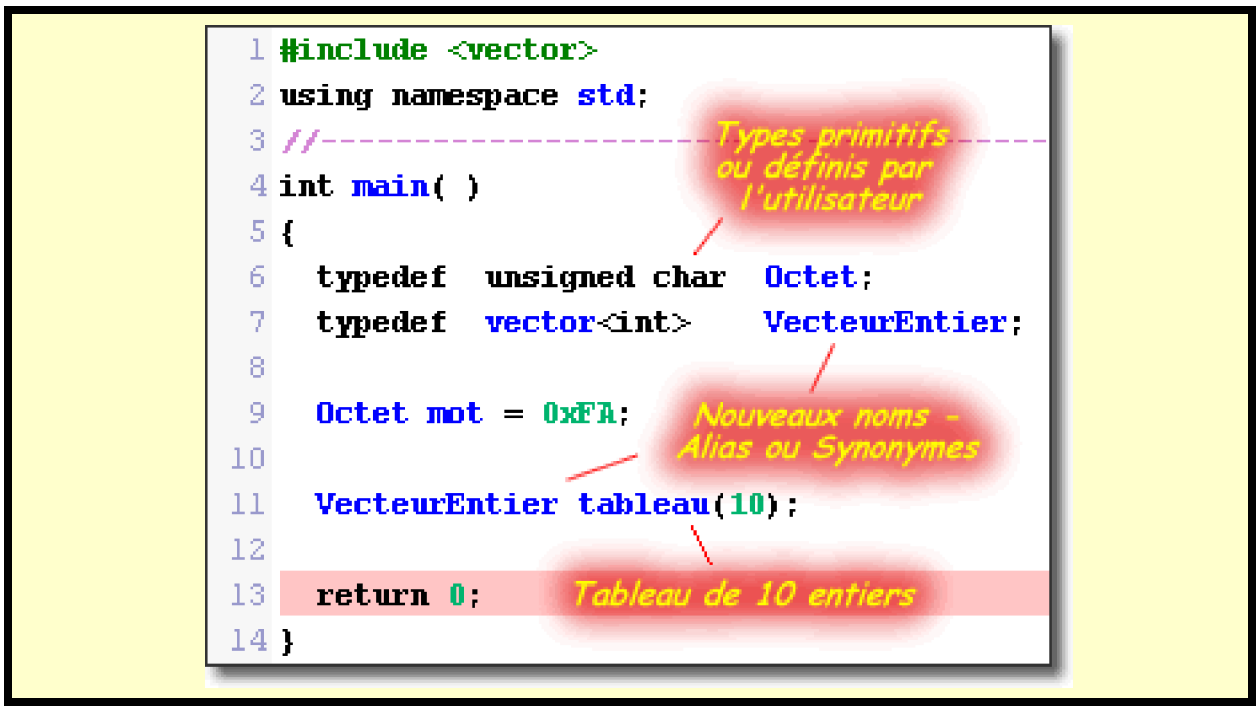
Directive de compilation « typedef » :

Le compilateur offre des mécanismes fort intéressants pour simplifier le travail du programmeur. Le préprocesseur, grâce à la directive typedef permet de donner un synonyme à un type de donnée standard ou défini par l'utilisateur.

C'est comme si nous définissions un nouveau type alors qu'il s'agit en fait d'un simple changement de texte durant la phase de précompilation.

Une définition par typedef commence avec le mot clé typedef, suivi du type de donnée et ensuite de l'identificateur.

L'identificateur, ou le nom typedef, n'introduit pas un nouveau type mais plutôt un synonyme pour le type de donnée existant. Un nom typedef peut apparaître n'importe où dans le programme, là ou un nom de type peut apparaître.



Dans tous les cas, pour parcourir un conteneur, nous serons amené à utiliser des *itérateurs*.

Qu'est ce qu'un itérateur ?

Un itérateur fournit un moyen général pour accéder successivement à chaque élément à l'intérieur de n'importe quel type de conteneur.

Un itérateur correspond à un pointeur qui, comme tous les pointeurs, permet d'utiliser l'incrémentation ou la décrémentation.

Ainsi, il est possible de consulter dans le sens direct ou en sens inverse une suite d'éléments faisant partie de la séquence.

! Tout cela est bien joli, mais voyons maintenant ce que c'est le sens direct et le sens inverse. !

Sens direct de parcours d'un conteneur :

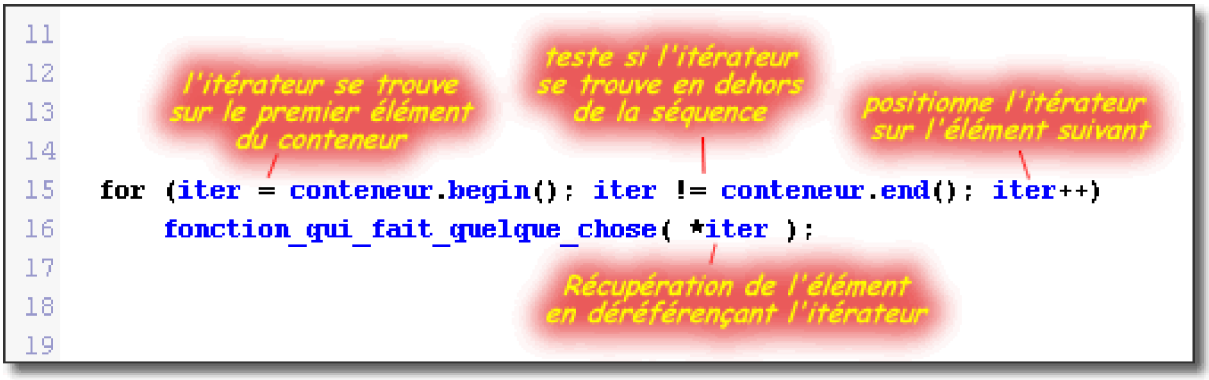
Grâce à un itérateur, nous pouvons naviguer avec une facilitée déconcertante. Pour cela, il vous ait possible d'incrémenter cet itérateur. Mais aussi, de le dé-référencer, pour obtenir la valeur correspondante au chiffre itérée et donc à la valeur de l'élément de la séquence.

Chaque type de conteneur fournit deux méthodes :

begin() : retourne un itérateur qui adresse le premier élément du conteneur,

end() : retourne un itérateur qui adresse un élément après le dernier élément du conteneur.

Exemple de fonctionnement d'un itérateur pour le sens direct :



Sens inverse de parcours d'un conteneur :

Et oui, il est également possible de parcourir un conteneur en sens inverse.

Attention, il faut pouvoir démarrer sur le dernier élément, ce que ne fait pas la méthode end() , puisqu'elle se trouve après le dernier élément.

D'autres méthodes ont donc été implémentées pour résoudre cette situation :

rbegin() : retourne un itérateur qui adresse le dernier élément du conteneur,

rend() : retourne un itérateur qui adresse un élément juste avant le premier élément du conteneur.

Comment déclarer un itérateur ?

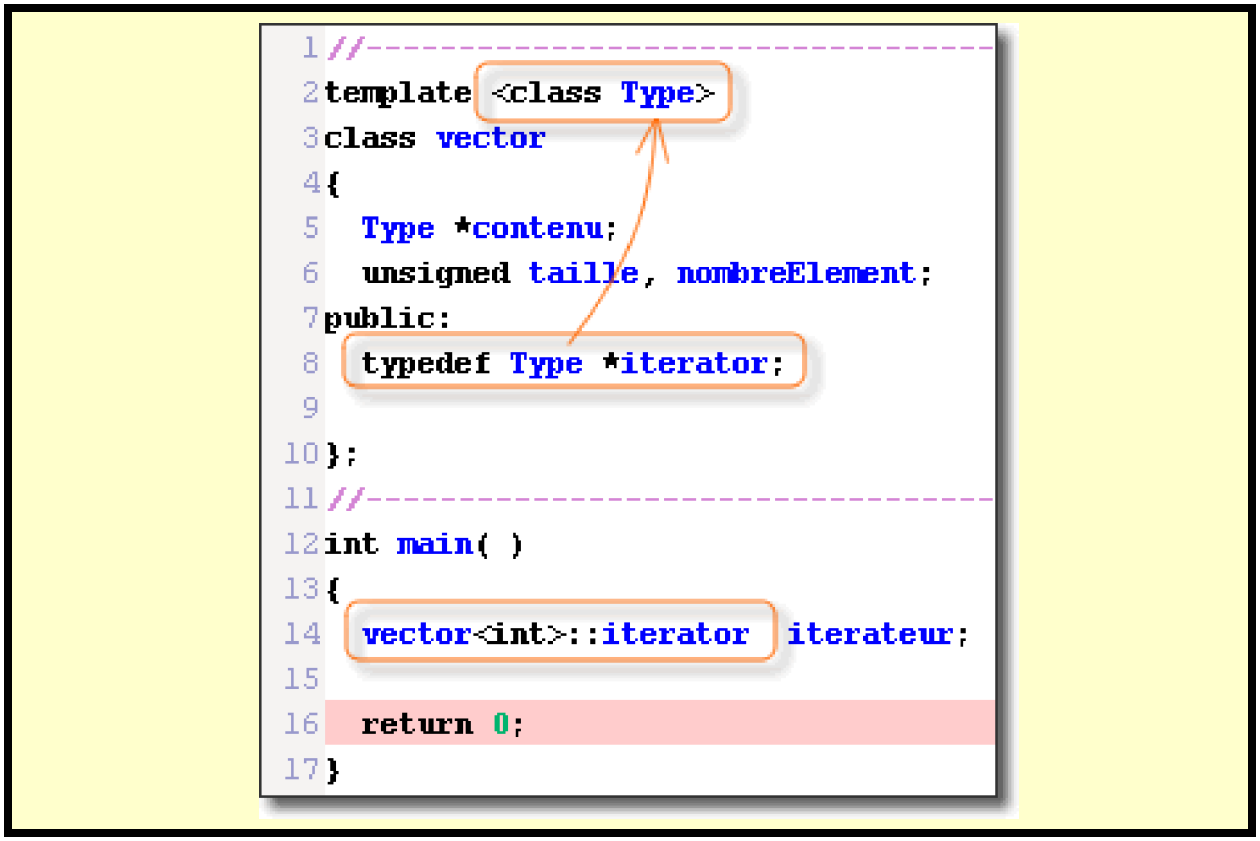
L'avantage de ce mécanisme, c'est qu'il fonctionne quelque soit le type de conteneur utiliser alors que la structure interne de chacun d'entre eux peut être totalement différente.

Il faut savoir que, pour les vecteurs, les éléments sont stockés sur des cases mémoires contiguës, alors que pour la liste ce n'est pas du tout le cas. La seule solution pour résoudre ces difficultés est de prendre le mécanisme des pointeurs.

Il faut bien comprendre également que ces différents conteneurs stockent des données de type quelconque (int , double , string , etc.).

Ceci dit pour réaliser ces différents parcours, il est bien évidemment nécessaire de déclarer la variable qui représente l'itérateur. Souvenez-vous que, pour que l'incrémentation d'un pointeur se fasse dans les bonnes conditions, il est impératif de connaître le type de l'élément faisant parti du conteneur.

Illustration :



```
1  //--------------------------------
2  template <class Type>
3  class vector
4  {
5    Type *contenu;
6    unsigned taille, nombreElement;
7  public:
8    typedef Type *iterator;
9
10 };
11 //--------------------------------
12 int main( )
13 {
14   vector<int>::iterator iterateur;
15
16   return 0;
17 }
```

La solution retenue a donc été de proposer un attribut public, présent sur tous les conteneurs, qui s'appelle *iterator* .

Cet attribut est en fait un pointeur sur le type passé en paramètre du modèle de la classe conteneur. Cette démarche est possible grâce à la directive de compilation typedef. Cette astuce est géniale malgré la petite lourdeur de la déclaration. Vous avez ci-dessous un exemple d'un parcours dans le sens direct d'une liste d'entiers.

Il existe également un deuxième itérateur spécialisé pour parcourir le conteneur en sens inverse, qui s'appelle *reverse_iterator*.

Par ailleurs, bien que rarement utilisé, la décrémentation peut être utilisée par les deux types d'itérateurs.

Illustration :

```cpp
#include <iostream>
#include <list>
using namespace std;
//--------------------------------------------------
int main( )
{
  list<int> liste;
  list<int>::iterator it;

  liste.push_back(12);
  liste.push_back(-23);
  liste.push_back(45);

  for (it = liste.begin(); it != liste.end(); it++)
    cout << *it << endl;
  return 0;
}
```

Liste d'entiers vide

Itérateur vers une liste d'entiers

Insertion successive de trois entiers en fin de liste

Parcours de la liste dans le sens direct et affichage des valeurs successivement : 12, -23, 45

```cpp
#include <iostream>
#include <list>
using namespace std;
//--------------------------------------------------
int main( )
{
  list<int> liste;
  list<int>::reverse_iterator it;

  liste.push_back(12);
  liste.push_back(-23);
  liste.push_back(45);

  for (it = liste.rbegin(); it != liste.rend(); it++)
    cout << *it << endl;
  return 0;
}
```

Liste d'entiers vide

Itérateur vers une liste d'entiers

Insertion successive de trois entiers en fin de liste

Parcours de la liste dans le sens inverse et affichage des valeurs successivement : 45, -23, 12

Nous avons de la chance, ici, car les différentes classes possède différents constructeur pouvant nous faciliter le travail par la suite.

Quels sont les différents constructeurs et comment se passe la construction au niveau des conteneurs ?

Construction d'un conteneur vide : L'appel d'un constructeur sans argument construit un conteneur vide, c'est-à-dire ne comportant aucun élément.

Construction avec un nombre donné d'éléments : De façon comparable à ce qui se passe avec la déclaration d'un tableau classique, l'appel d'un constructeur avec un seul argument entier « *n* » construit un conteneur comprenant « *n* » éléments. L'initialisation de ces éléments n'est correctement gérée que dans le cas ou les éléments sont des objets. En effet, ces derniers disposent d'un constructeur par défaut. Dans le cas des types primitifs, les valeurs sont aléatoires.

Construction avec un nombre d'éléments initialisés avec une valeur précise : Le premier argument fourni le nombre d'éléments alors que le second fixe la valeur d'initialisation.

Construction à partir d'une séquence : Nous pouvons construire un conteneur à partir d'une séquence d'éléments de même type. Dans ce cas, nous fournissons simplement au constructeur deux arguments représentant les bornes de l'intervalle correspondant.

Construction par recopie : Chaque type de conteneur dispose de son propre constructeur de copie. Attention, il est nécessaire d'utiliser des conteneurs rigoureusement identiques (même conteneur et même type d'éléments).

! Vous pouvez remarquer que tout ce qui s'applique à la classe « vector » peut s'appliqué à des conteneurs séquentiels. !

Comment marche l'affectation ou bien la comparaison dans ces cas là ?

Il est possible d'affecter un conteneur d'un type donné à un autre conteneur de même type, c'est-à-dire ayant le même nom de patron et le même type d'éléments.

Bien entendu, il n'est nullement nécessaire que le nombre d'éléments de chacun des conteneurs soit identique.

Les opérateurs relationnels « `==, !=, <, <=, >, >=` », eux aussi, ont été redéfinis pour supporter tous les types de comparaison, quelque soit le conteneur utilisé.

Jusqu'à présent, nous venons de voir certaines méthodes, très pratique et très simple d'utilisation.

Maintenant, je vais vous en montrer d'autres, qui vont vous semblez commune à certaines classes.

assign(début , fin) : alors que l'affectation n'est possible qu'entre conteneurs de même type, la méthode « *assign* » permet d'affecter, à un conteneur existant, les éléments d'une autre séquence définie par un intervalle (*début* , *fin*), à condition que les éléments des deux séquences soient de même type.

assign(nombreDeFois , valeur) : il existe également une version permettant d'affecter à un conteneur, un nombre donné d'éléments ayant une valeur imposée.

clear() : vide le conteneur de son contenu.

empty() : teste si le conteneur est vide et renvoie *true* si c'est le cas, et *false* ans le cas contraire.

swap() : permet d'échanger le contenu de deux conteneurs de même type.

insert(position , valeur) : insère une valeur avant l'élément pointé par la position.

insert(position , nombreDeFois , valeur) : insère un certain nombre de fois une valeur avant l'élément pointé par la position.

insert(début , fin , position) : insère les valeurs de l'intervalle (*début* , *fin*) avant l'élément pointé par la position.

push_back(valeur) : cette méthode est spécialisée pour insérer une valeur en fin de conteneur à la manière d'une pile.

erase(position) : supprime l'élément désigné par la position

erase(début , fin) : supprime les valeurs de l'intervalle « *début (compris) , fin (non compris)* ».

pop_back() : cette méthode est spécialisée pour supprimer la dernière valeur du conteneur à la manière d'une pile.

size() : détermine le nombre d'éléments que contient le conteneur.

Toutes ces ressemblances entre ces différentes classes, au niveau des méthodes, nous allons appeler cela :

« *Les algorithmes génériques* ».

LES ALGORITHMES GENERIQUES

Les conteneurs ont en commun beaucoup de méthodes. Chaque conteneur dispose également de méthodes supplémentaires qui font leur spécificité. Cependant, une fois que nous avons choisi un conteneur, il peut être intéressant de rajouter d'autres fonctionnalités non intégrées par le conteneur.

Dans ce cas là, nous avons besoin des fonctions génériques. Rappelons que les fonctions génériques sont opérationnelles quelque soit le conteneur utilisé.

Nous remarquons, par exemple, que la recherche d'un élément au sein d'un conteneur ne fait pas parti des méthodes communes, alors que c'est un comportement qui est souvent souhaitable.

Nous allons d'ailleurs détailler un certain nombre de fonctions qui sont généralement assez utiles. Il faudra quand même vérifier que votre conteneur ne dispose pas déjà de telles fonctions internes (méthodes).

Voici quelques fonctions génériques intéressantes :

copy (conteneur1début , conteneur1fin , conteneur2début) : copie d'une séquence dans une autre. Il suffit de préciser l'intervalle désiré en donnant les itérateurs du premier conteneur. Cet intervalle est ensuite copié à partir de l'itérateur donné par le second conteneur.

count (iterateurdébut , iterateurfin , valeurRecherchée) : comptabilise le nombre de fois qu'un élément est présent dans un conteneur,

iterateur find (iterateurdébut , iterateurfin , valeurRecherchée) : recherche d'une valeur particulière par rapport à l'intervalle d'une séquence. La fonction retourne l'itérateur correspondant à l'endroit où se situe la valeur recherchée. Si la recherche n'a pas aboutie, la fonction renvoie un itérateur sur la fin de la séquence - *conteneur.end()*.

iterateur max_element (iterateurdébut , iterateurfin) : recherche la valeur maximale d'une séquence. La fonction retourne l'itérateur correspondant à l'endroit où se situe la valeur recherchée.

iterateur min_element (iterateurdébut , iterateurfin) : recherche la valeur minimale d'une séquence. La fonction retourne l'itérateur correspondant à l'endroit où se situe la valeur recherchée.

replace (iterateurdébut , iterateurfin , ancienneValeur , nouvelleValeur) : remplace toutes les instances d'une valeur particulière par une nouvelle valeur,

remove (iterateurdébut , iterateurfin , valeurASupprimer) : supprime toutes les instances d'une valeur particulière par rapport à l'intervalle proposé.

sort (iterateurdébut , iterateurfin) : tri de la séquence. Reclasse les éléments de l'intervalle proposé dans l'ordre croissant.

reverse (iterateurdébut , iterateurfin) : inverse l'ordre des éléments dans un conteneur.

! Il se pourrait que c'est le moment le plus important pour se plonger dans les différentes spécificités de la classe « vector ». !

Specificités du conteneur « vector »

Nous connaissons déjà la classe « vector » en tant que tableau et par ailleurs nous connaissons un certain nombre de méthodes qui sont communes à tous les conteneurs. Nous allons découvrir, ici, d'autres méthodes qui sont propre à la classe « vector ». Nous en profiterons pour déterminer les avantages et les inconvénients de ce type de conteneur.

Ce conteneur représente bien un tableau, c'est-à-dire que les éléments qui constituent la séquence, sont placés dans des cases mémoires contiguës.

Ce conteneur « vector » est relativement performant, puisqu'il s'agit en fait d'un tableau dynamique, c'est-à-dire, que suivant le besoin, le nombre de cases qui composent le tableau peut augmenter en cours d'utilisation.

Si le tableau est plein, lorsque nous essayons d'introduire une nouvelle valeur, la gestion interne du vecteur prévoit de réserver un nouvel emplacement mémoire dont la capacité est le double de la précédente, ensuite une copie des anciennes valeurs est effectuée vers ce nouvel emplacement. La copie d'anciennes valeurs prend du temps, c'est pour cette raison que la nouvelle capacité est doublée.

Dans cette situation, nous avons un bon compromis entre la capacité du tableau (utiliser le moins de place possible) et le temps de réponse en général (répondre le plus rapidement possible à une requête).

Du coup, il faut noter que la capacité du tableau peut être plus grande que le nombre d'éléments déjà introduits dans le conteneur. Cette classe propose un certain nombre de méthodes qui permet de contrôler cette situation. Ci-dessous, se trouve l'ensemble des méthodes spécifiques à la classe « vector » :

back() : récupère la valeur du dernier élément sans toutefois l'enlever du conteneur comme c'est le cas avec la méthode *pop_back()* .

front() : récupère la valeur du premier élément sans l'enlever du conteneur.

size() : cette méthode n'est pas spécifique à « *vector* », toutefois, je rappelle que cette méthode retourne le nombre d'éléments présents dans le conteneur.

capacity() : retourne la capacité du tableau (nombre de cases allouées) au moment de la consultation. « *capacity() >= size()* » .

reserve(taille) : permet d'imposer une capacité.

Quel est l'intérêt d'utiliser la classe « vector » ?

ATTENTION : NE PAS OUBLIER D'ECRIRE « #include <vector>.

! Cette inclusion est obligatoire si l'on souhaite utilisé des méthodes de cette classe dans de bonnes conditions. !

Le conteneur vector présente deux caractéristiques essentielles :

Comme les cases mémoires relatives aux éléments sont contiguës, nous savons que l'élément qui suit un autre se trouve sur la case d'après, ainsi, il n'est pas nécessaire de rajouter de pointeurs supplémentaires pour parcourir un conteneur, comme c'est le cas avec une liste. Ainsi, la taille mémoire que prend ce type de conteneur est relativement réduite.

Il s'agit en fait d'un tableau, et grâce à l'opérateur d'indexation « *[ ]* », nous pouvons accéder directement à un élément du conteneur sans être obligé de parcourir toute la séquence. L'accès aléatoire est donc très rapide. Nous pouvons même nous passer de l'écriture explicite d'un « *iterator* ». Généralement, nous utilisons plutôt la syntaxe des itérateurs pour conserver une certaine homogénéité, plus tard, il sera alors plus facile de changer de type de conteneur si le besoin s'en fait sentir. (Comme le vecteur est un tableau, l'itérateur correspond, dans ce cas, là à l'adresse d'une des cases du tableau).

Ce type de conteneur est également très bien adapté à l'insertion de nouveaux éléments en fin de conteneur, ce que fait très bien la méthode push_back() .

Conteneur deque : Le conteneur qui lui ressemble beaucoup, c'est le conteneur « deque » qui offre la possibilité d'insérer « push_front() » et de supprimer « pop_front() » de nouveau éléments en début de conteneur, ce que ne permet pas le conteneur « vector ». « deque », par contre, ne possède pas la gestion de la capacité du tableau.

Quel est l'intérêt d'utiliser la classe « list » ?

ATTENTION : NE PAS OUBLIER D'ECRIRE « #include <list>.

Cette inclusion est obligatoire si l'on souhaite utilisé des méthodes de cette classe dans de bonnes conditions.

Si vous avez justement besoin de gérer beaucoup d'insertions et de suppressions, la liste est totalement adaptée à ce genre de situation. Elle dispose également de méthodes fort intéressantes qui évitent d'utiliser les fonctions génériques. Elle dispose, par exemple, de sa propre méthode de tri, ce que ne permet pas le conteneur « *vector* ».

Ce conteneur est implémenté par une liste en double chaînage, ce qui fait que chaque élément est constitué de deux pointeurs supplémentaires.

Du coup, la taille mémoire est plus conséquente que pour le conteneur « *vector* ».

Les désavantage de ce conteneur, c'est qu'il ne gère pas l'accès direct (accès aléatoire).

Pour atteindre un élément, vous êtes obligé de parcourir jusqu'à l'élément désiré afin de pouvoir l'atteindre. Le temps de réponse pour accéder à un élément peut donc être très important. Dans ces conditions, à vous de choisir le conteneur qui offre le plus de souplesse possible pour votre application, et ceci sans trop de contraintes.

Ci-dessous, se trouve l'ensemble des méthodes spécifiques à la classe « list » :

remove(valeur) : supprime tous les éléments égaux à valeur. Cette méthode remplace essaiment la fonction générique équivalente, ainsi que la méthode *erase()* commune à tous les conteneurs.

sort() : tri la liste. Reclasse les éléments dans l'ordre croissant. Dans le cas d'un tri d'une liste, c'est cette méthode qu'il faut utiliser. Il ne faut pas prendre la fonction générique algorithmique équivalente.

unique() : permet d'éliminer les éléments en double, à condition de la faire porter sur une liste préalablement triée.

merge(liste) : fusionne *liste* avec la liste concernée. Attention, à la fin de cette opération, *liste* est vide.

splice(position , liste_origine) :

splice(position , liste_origine , position_origine) :

splice(position , liste_origine , début_origine , fin_origine) : permet de déplacer des éléments d'une autre liste dans la liste concernée. Attention, comme avec *merge()* , les éléments déplacés sont supprimés de la liste d'origine et pas seulement copiés.

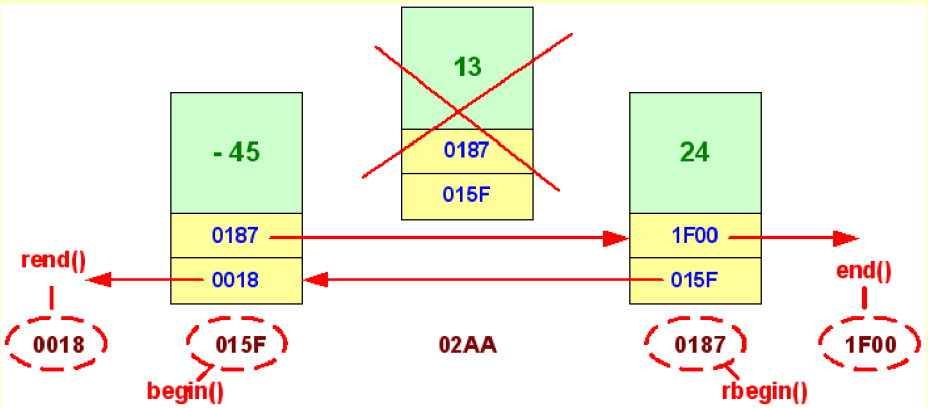
Maintenant, nous allons approfondir la suppression d'un élément dans un conteneur « list ».

Comment est réalisé cette suppression et de quoi elle en découle ?

Voici la réponse avec une petite illustration à l'appui.

Lorsque vous désirez supprimer une valeur de la liste, il suffit alors de redéfinir un des pointeurs de l'élément suivant ainsi qu'un des pointeurs de l'élément précédent.

```
1  #include <list>
2  #include <iostream>
3  using namespace std;
4  //------------------------------------------------------------
5  int main( )
6  {
7    list<int> liste;
8    liste.push_back(-45);
9    liste.push_back(13);
10   liste.push_back(24);
11
12   liste.remove(13);
13   list<int>::iterator itl;
14   for (itl = liste.begin(); itl != liste.end(); itl++)
15     cout << *itl << ' '; // affiche successivement -45 24
16   return 0;
17 }
```



Comment est composé la bibliothèque standard?

Nous avons encore des multiples choses à voir comme les adaptateurs de conteneur.

Qu'est ce qu'un adaptateur de conteneur ?

La bibliothèque standard dispose de deux patrons particuliers stack et queue dits adaptateurs de conteneurs.

Il s'agit de classes génériques construites sur un conteneur séquentiel qui en modifie l'interface, à la fois en la restreignant et en l'adaptant à leurs propres fonctionnalités.

Ils disposent tous d'un seul constructeur qui est le constructeur par défaut.

! D'après notre professeur de programmation, « stack » et « deque » ne sont pas à savoir par cœur en BTS IRIS. Il faut simplement y jeter un oeuil. !

! Cela étant dit, pour ceux qui veulent aller plus loin, il n'y a pas de soucis à avoir car je vais continuer à expliquer. !

L'adaptateur « stack » :

La classe générique stack est destinée à la gestion des piles de type LIFO (Last In, First Out) ; il peut être construit à partir de l'un des trois conteneurs séquentiels vector, deque, list.

Ci-dessous, nous pouvons voir comment créer cet adaptateur :

```
stack<int, vector<int> > pile; // pile de int, utilisant un conteneur vector
stack<int, list<int> > pile;   // pile de int, utilisant un conteneur list
stack<int, deque<int> > pile;  // pile de int, utilisant un conteneur deque
```

Dans un tel conteneur, on ne peut qu'introduire push() des informations qui s'empile les unes sur les autres et que l'on recueille, à raison d'une seule à la fois, en extrayant la dernière introduite.

Nous trouvons uniquement les méthodes suivantes :

empty() : renvoie ***true*** si la pile est vide.

size() : fournit le nombre d'éléments de la pile.

top() : accès à l'information située au sommet de la pile que nous pouvons consulter ou même modifier (sans la supprimer).

push(valeur) : place ***valeur*** sur le sommet de la pile.

pop() : suppression de l'élément situé au sommet de la pile (ne retourne aucune valeur).

Illustration :

```cpp
#include <stack>
#include <vector>
#include <iostream>
using namespace std;
//-------------------------------------------------------
int main(  )
{
  stack<int, vector<int> > pile;
  cout << "Taille initiale : " << pile.size() << endl;
  for (int i=0; i<10; i++) pile.push(i*i);
  cout << "Taille après for : " << pile.size() << endl;
  cout << "Sommet de la pile : " << pile.top() << endl;
  pile.top() = 99; // modification de sommet de la pile
  cout << "On dépile : ";
  for (int i=0; i<10; i++) {
    cout << pile.top() << ' ';
    pile.pop();
  }
  return 0;
}
```

Résultat obtenu par le morceau de code ci-dessus :

```
Taille initiale : 0
Taille après for : 10
Sommet de la pile : 81
On dépile : 99 64 49 36 25 16 9 4 1 0
```

L'adaptateur « deque » :

La classe générique queue est destinée à la gestion de piles de files d'attentes de type FIFO (First In, First Out).

Dans ce cas là, les informations sont placées à la fin du conteneur et peuvent être ensuite récupérées au début.

Un tel conteneur peut être construit à partir de l'un des deux conteneurs séquentiels deque, list.

Le conteneur vector n'est pas approprié puisqu'il ne dispose pas d'insertions efficaces au début.

empty() : renvoie true si la pile est vide.

size() : fournit le nombre d'éléments de la pile.

front() : accès à l'information située en tête de la file que nous pouvons consulter ou même modifier (sans la supprimer).

back() : accès à l'information située à la fin de la file que nous pouvons consulter ou même modifier (sans la supprimer).

push(valeur) : place valeur dans la file.

pop() : suppression de l'élément situé en tête de la file (ne retourne aucune valeur).

Illustration :

```cpp
#include <queue>
#include <deque>
#include <iostream>
using namespace std;
//-----------------------------------------------------------
int main(  )
{
  queue<int, deque<int> > file;
  for (int i=0; i<10; i++) file.push(i*i);
  cout << "Tête de la file : " << file.front() << endl;
  cout << "Queue de la file : " << file.back() << endl;
  file.front() = 99;  // modification de la tête de la file
  file.back() = -99;  // modification de la queue de la file
  cout << "Vidage de la file : ";
  for (int i=0; i<10; i++) {
    cout << file.front() << ' ';
    file.pop();
  }
  return 0;
}
```

```
Tête de la file : 0
Queue de la file : 81
Vidage de la file : 99 1 4 9 16 25 36 49 64 -99
```

Nous obtenons ce résultat après avoir écrit ce bout de code.

CONTENEURS ASSOCIATIFS

Les conteneurs se classent en deux catégories :

->les conteneurs séquentiels

->les conteneurs associatifs

Nous venons de le voir, les conteneurs séquentiels sont ordonnés suivant un ordre imposé explicitement par le programme lui-même ; nous accédons à un des éléments en tenant compte cet ordre, que nous utilisions un indice ou un itérateur.

Les conteneurs associatifs ont pour principale vocation de retrouver une information, non plus en fonction de sa place dans le conteneur, mais en fonction de sa valeur nommée « clé ».

Par exemple, un répertoire téléphonique, dans lequel on retrouve le numéro de téléphone à partir de la clé formée du nom de la personne concernée. Malgré tout, pour de simple questions d'efficacité, un conteneur associatif se trouve ordonné intrinsèquement en permanence, en se fondant sur une relation (par défaut « < ») choisie à la construction.

Les deux conteneurs associatifs les plus importants sont *map* et *multimap*. Ils correspondent pleinement au concept de conteneur associatif, en associant une clé et une valeur.

Mais, alors que *map* impose l'unicité des clés, autrement dit l'absence de deux éléments ayant la même clé, *multimap* ne l'impose pas et nous pourrons trouver plusieurs éléments d'une même clé qui apparaîtront alors consécutivement.

Si nous reprenons l'exemple du répertoire téléphonique, *multimap* autorise la présence de plusieurs numéros pour une même personne, tandis que *map* ne l'autorise pas.

Comment est régie le conteneur « map », comment l'utiliser à bonne escient ?

Ce conteneur offre une grande souplesse d'emploi.

Il permet effectivement d'intégrer ou de rechercher des éléments à l'aide de l'opérateur d'indexation [ ] sans se préoccuper spécialement de l'endroit où le conteneur stocke sa donnée.

Bien entendu, un certain nombre de méthodes sont proposés pour faciliter l'utilisation de ce type de conteneur.

empty() : renvoie ***true*** si le conteneur « ***map*** » est vide.

size() : fournit le nombre d'éléments du conteneur.

clear() : vide le conteneur de tout élément.

Objetmap[clé] = valeur : place le couple (***clé*** , ***valeur***) dans le conteneur ***Objetmap*** .

valeur = Objetmap[clé] : recherche la valeur associée à ***clé*** dans le conteneur ***Objetmap*** et la retourne à ***valeur*** .

erase(clé) : supprime un élément du conteneur référencé par ***clé*** .

begin(), end(), rbebin(), rend() : il est possible de parcourir le conteneur afin, par exemple, de recenser l'ensemble des éléments. Chacune de ces méthodes retourne un itérateur.

iterator , reverse_iterator : Pour permettre ce parcours, comme tous les autres conteneurs, nous nous servons donc itérateurs.

first, second : il existe deux attributs qui représentent respectivement la ***clé*** et la ***valeur*** d'un élément du conteneur.

insert(élément) : bien que nous ayons pour ce conteneur l'opérateur d'indexation qui permet d'introduire de nouveaux éléments de façon intuitive, nous pouvons également utiliser cette méthode. En paramètre, il est nécessaire de passer l'élément en entier, c'est-à-dire le couple (***clé*** , ***valeur***). Pour cela, « ***élément*** » sera fabriqué à l'aide de la fonction « ***make_pair*** » définie ci-dessous.

make_pair(clé , valeur) : cette fonction générique permet de fabriquer un élément du conteneur « ***map*** » constitué d'une ***clé*** et d'une ***valeur*** .

Itérateur find(clé) : bien que nous ayons pour ce conteneur l'opérateur d'indexation qui permet de rechercher une ***valeur*** , il est également possible d'utiliser cette méthode qui retourne un itérateur qui identifie l'emplacement de l'élément référencé par la ***clé*** passée en paramètre. Si aucun élément n'est retrouvé, la méthode renvoie l'itérateur relatif à « ***end()*** ».

Illustration de ce que l'on vient de voir :

```cpp
#include <map>
#include <iostream>
using namespace std;
//------------------------------------------------------------------
int main( )
{
  map<char, int> stock;
  map<char, int>::iterator it;
  stock['a'] = 28;
  stock['x'] = -89;
  stock.insert(make_pair('w', 45));
  for (it = stock.begin(); it != stock.end(); it++)
    cout << '(' << it->first << ", " << it->second << ')' << endl;
  cout << stock['a'] << " -- " << stock.find('a')->second << endl;
  stock.erase('x');
  it = stock.find('x');
  if (it != stock.end()) cout << it->second << endl;
  if (!stock.empty()) cout << stock.size();
  return 0;
}
```

Voici le résultat obtenu grâce au code ci-dessus.

```
(a, 28)
(w, 45)
(x, -89)
28 -- 28
2
```

Comment est régie le conteneur « multimap », comment l'utiliser à bonne escient ?

Le conteneur multimap est un conteneur map qui accepte en plus d'avoir plusieurs valeurs pour une même clé.

Nous retrouverons donc les mêmes méthodes, sauf pour l'opérateur d'indexation puisque, dans ce cas là, l'utilisation d'un tel opérateur ne conviendrait pas.

Il faut donc impérativement utiliser la méthode « insert(élément) » associée à la fonction générique « make_pair(clé , valeur) » pour palier ce manque.

La méthode « erase(clé) » cette fois ci permet d'effacer toutes les valeurs associées à une même clé.

Par contre, il faut pouvoir parcourir l'ensemble des valeurs associées à une même clé. D'autres méthodes qui existées déjà dans le conteneur map prennent maintenant toute leur utilité.

count(clé) : retourne le nombre d'éléments associés à une ***clé***.

Itérateur lower_bound(clé) : retourne un itérateur sur le premier élément associé à ***clé***.

Itérateur upper_bound(clé) : retourne un itérateur juste après le dernier élément associé à ***clé***.

Illustration de ce que l'on vient de voir :

```cpp
#include <map>
#include <iostream>
using namespace std;
//------------------------------------------------------------------------
int main(  )
{
  multimap<char, int> stockage;
  multimap<char, int>::iterator it;

  stockage.insert(make_pair('c', 10));
  stockage.insert(make_pair('f', 20));
  stockage.insert(make_pair('f', 30));
  stockage.insert(make_pair('p', 10));

  cout << "Nombre d'éléments :" << stockage.size() << endl;
  cout << "Nombre de f : " << stockage.count('f') << endl;

  cout << "Ensemble de tous les éléments" << endl;
  for (it = stockage.begin(); it != stockage.end(); it++)
    cout << '(' << it->first << ", " << it->second << ')' << endl;

  cout << "Ensemble de tous les f" << endl;
  for (it = stockage.lower_bound('f'); it != stockage.upper_bound('f'); it++)
    cout << '(' << it->first << ", " << it->second << ')' << endl;

  return 0;
}
```

```
Nombre d'éléments :4
Nombre de f : 2
Ensemble de tous les éléments
(c, 10)
(f, 20)
(f, 30)
(p, 10)
Ensemble de tous les f
(f, 20)
(f, 30)
```

LES FLUX ET LES FICHIERS

Depuis le temps, nous connaissons bien les objets cin et cout. Du moins nous semblons les connaître. Cette étude nous permettra de découvrir un certain nombre de méthodes associées qui peuvent rendre de grands services.

Par ailleurs, nous allons étendre plus généralement nos connaissances sur les classes qui représentent l'ensemble des flots.

Ainsi, nous pourrons travailler aussi bien sur les entrées/sorties standards que sur des fichiers ou que sur les flots en mémoire pour la gestion et le formatage des chaînes de caractères.

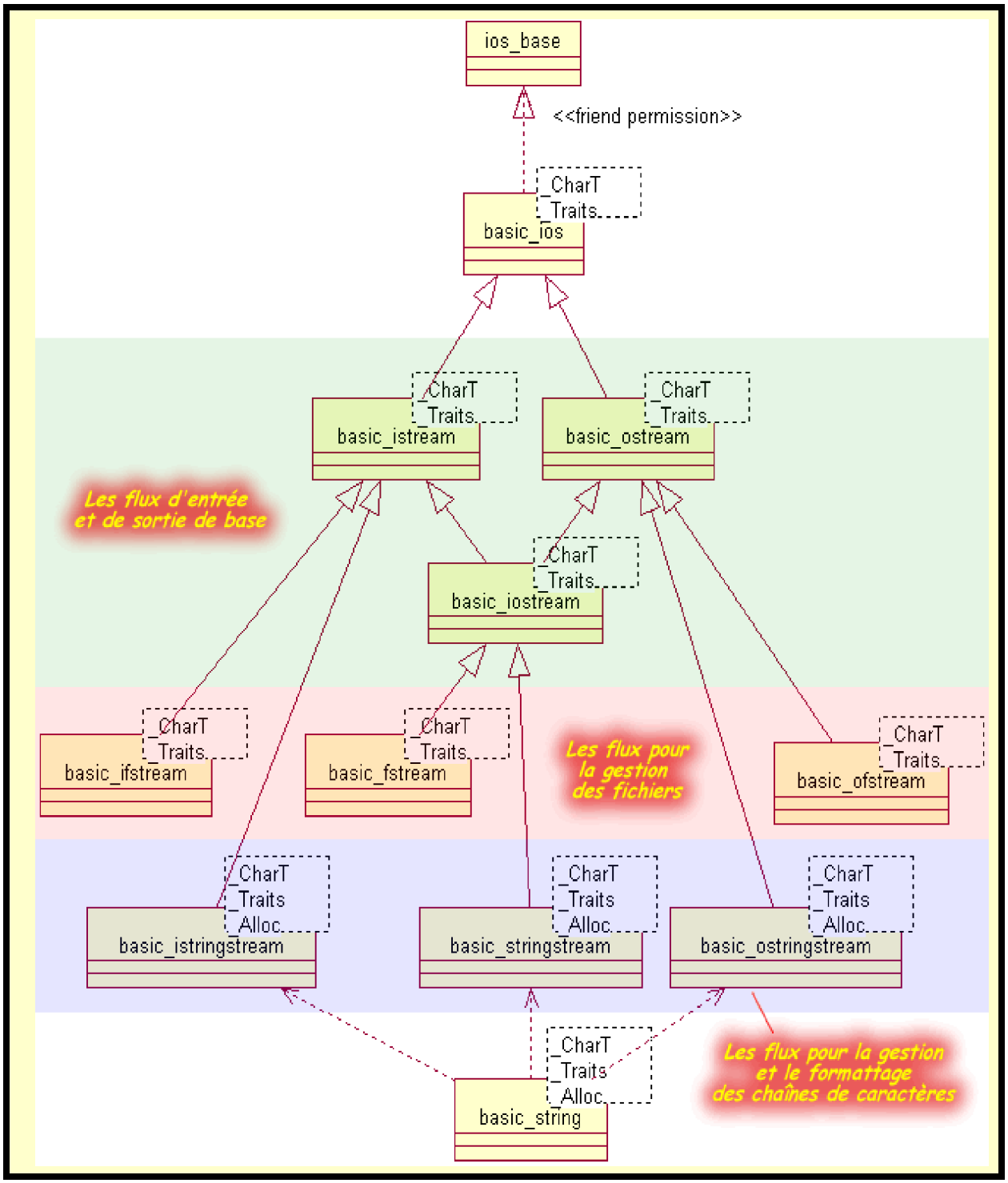
Nous pourrons même modifier le comportement standard de ces flots afin de permettre à nos propres classes de pouvoir s'intégrer aux classes représentatives de ces flots.

! Une question s'impose maintenant, comment est organisée ces classes très utiles. Existe − t'il une hiérarchie de ces classes représentant les flots et les fichier. Et bien oui, nous y allons arriver. !

Hiérarchie des classes représentant les flots :

Il existe un certain nombre de classes qui s'occupent de la gestion des flots. De plus, chaque classe est spécialisée afin de répondre parfaitement à l'adéquation recherchée. Ainsi, vous avez des classes réservées pour la gestion des fichiers, d'autres pour le traitement des flots en mémoire sous forme de chaînes de caractères, etc. par ailleurs, ces classes ne sont pas disposées n'importe comment, mais elles font toutes parties d'une même famille et profitent pleinement du polymorphisme dont les conséquences seront pleinement justifiées lorsque nous aborderons la fin de notre étude. Vous en avez une vue dans le diagramme UML ci dessous.

Sur le diagramme UML ci-dessous, nous pouvons voir qu'il y a une hiérarchie très compliqué qui existe en C++.



! Jusqu'à présent nous utilisions que très peu de classes ici présente. !

Les classes qui nous servirons presque tout le temps seront les suivantes :

- è classe ostream, pour les sorties.
- è classe istream, pour les entrées.
- è classe iostream, pour les deux sens.

Que signifie l'expression « flots d'entrée/sortie ?

D'une manière générale, un flot peut être considéré comme un « canal » qui permet une communication avec un périphérique ou un fichier ou même une partie de la mémoire (formatée comme une structure de fichier).

Il existe trois types de flots :

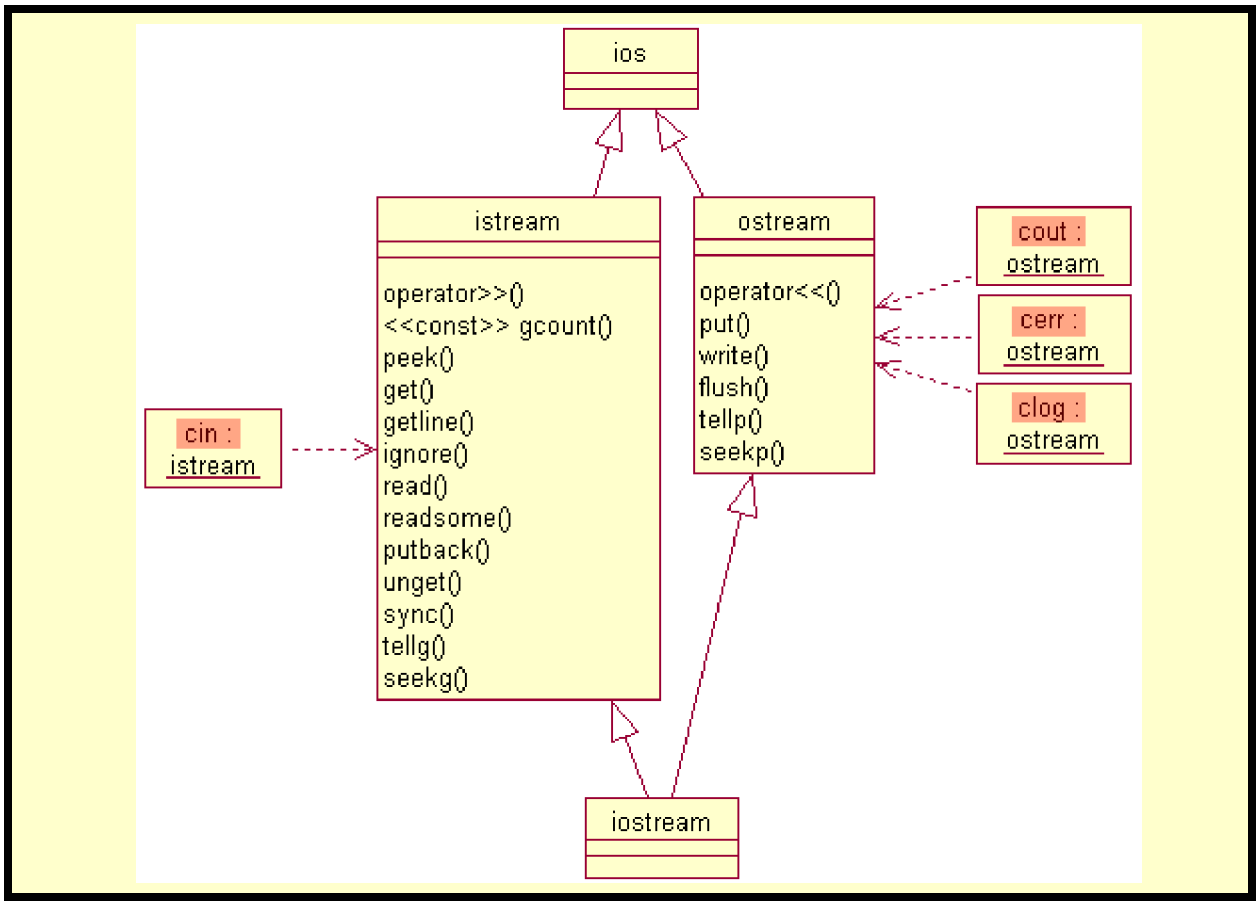
Recevant de l'information – flot de sortie.

Fournissant de l'information – flot d'entrée.

Recevant et fournissant de l'information – flot d'entrée et de sortie.

Toutes les opérations d'entrées et de sorties sont fournies par les classes istream (flot d'entrée), ostream (flot de sortie) et iostream (classe dérivée des deux premières qui permet donc la bidirectionnalité).

Ainsi, l'étude que nous ferons sur ces trois classes particulières sera également utile pour toutes les autres qui dérivent de celles-ci.



Il existe quatre objets prédéfinis qui permet de gérer les flux standard, savoir :

cin : objet de la classe ***istream*** représentant l'entrée standard. En général, ***cin*** permet de lire les données depuis le terminal de l'ordinateur, c'est-à-dire le clavier.

cout : objet de la classe ***ostream*** représentant la sortie standard. En général, ***cout*** permet d'écrire des données pour le terminal de l'ordinateur, c'est-à-dire l'écran.

cerr : objet également de la classe ***ostream*** représentant les erreurs standard. ***cerr*** est l'emplacement vers lequel diriger les messages d'erreur du programme.

clog : objet supplémentaire qui gère les erreurs qui est donc similaire à ***cerr***. Il dispose en plus d'un tampon intermédiaire.

! Dans nos programme, nous utilisons beaucoup les deux premières fonctions « cin » et « cout », n'est ce pas ?. !

Les classes istream et ostream sont prépondérantes puisque c'est à partir de ces classes que nous pouvons réellement communiquer. Elles disposent d'ailleurs d'un certain nombre de méthodes que nous allons détailler. Elles ont tellement d'importances qu'il a été décidé d'utiliser les opérateurs de redirection pour permettre une utilisation rapide et pratique. Le sens de redirection n'est pas choisi au hasard.

En effet, le sens proposé indique la direction de l'envoi des données. Ainsi, dans le cas général :

```
>> x ; // insère des données dans x.

<< x ; // extrait des données de x.
```

En prenant l'exemple des flots que nous connaissons :

```
cin >> x ; // insère les données saisies au clavier dans x.

cout << x ; // extrait des données de x et l'affiche à l'écran.
```

Dans le modèle UML, les deux opérateurs ne sont écrits qu'une seule fois, alors qu'ils sont surdéfinis pour permettre la communication avec différents types de variables.

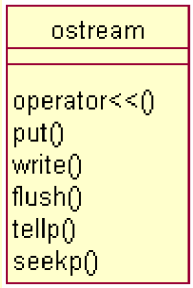
Ainsi, nous serons capable de saisir et d'afficher aussi bien des entiers que des réels, des chaînes de caractères, etc. En fait, tous les types primitifs ainsi que ceux faisant parti de la STL pourront être utilisés directement par les flots.

Seuls les types définis par l'utilisateur ne sont pas intégrés. Il sera alors nécessaire de proposer de nouvelles surdéfinitions pour élargir les compétences de ces flots, ce que nous ferons à l'issu de ce cour.

Détaillions maintenant les différentes classes qui vont nous intéresser.

! Je sens que vous êtes poussé par la curiosité d'en savoir plus, moi je suis heureux de vous montrer tout cela, encore merci de prêter attention à tout ce que je fais. J'arrête tout cela pour passer à la suite de mon cours. !

La classe « ostream » :



En écoutant mon professeur de programmation, j'ai compris que la classe « ostream » contenait quelques méthodes supplémentaire.

Voici ces méthodes, qui sont très importante.

ostream& operator<< (type) ;

Transfère une information d'un type prédéfini sur le flot de sortie. Comme l'opérateur renvoie un ostream, il est possible de proposer un enchaînement d'opérations.

```
int x = 5 ;
double y =-6.3 ;
cout << x << y ; // envoie la valeur 5 et la valeur -6.3 à l'écran.
```

ostream& put (char) ;

Cette méthode transmet un seul caractère dans le flot donné par l'argument. Cette méthode est souvent associée à la méthode get de istream.

Comme cette méthode renvoie un ostream, il est possible de proposer un enchaînement d'appels successifs au même titre que l'opérateur « << ».

```
cout.put('A') ; // envoie le caractère 'A' à l'écran, équivalent
à : cout << 'A' ;
cout.put('A').put('B').put('C') ; // envoie les trois
caractères « ABC » à l'écran.
```

ostream& write (const char*, int);

Cette méthode fournit une alternative à l'opérateur « << » pour transmettre un tableau de caractères. Elle transmet en sortie une certaine longueur de caractères (quels que soient ces caractères). La méthode write ne fait donc pas intervenir de caractères de fin de chaîne ; si un tel caractère apparaît dans la longueur prévue, il sera transmis, comme les autres, au flot de sortie. Son comportement est donc totalement différent de l'opérateur « << » puisque ce dernier affiche justement tous les caractères jusqu'à ce qu'il rencontre le caractère de fin de chaîne.

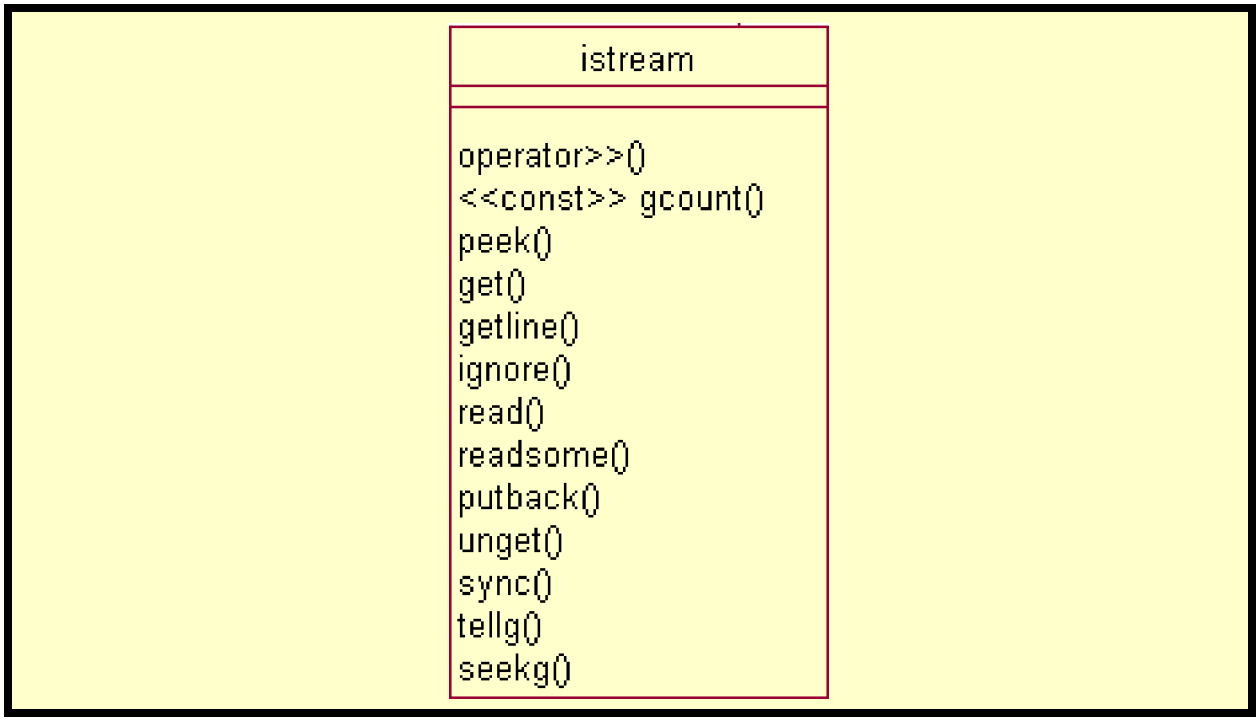
```
char message[] = « Bonjour » ;
cout.write(message, 4) ; // affiche les quatre premiers
caractères de message à l'écran.
```

Cette méthode n'est pas très utile pour un écran, mais elle le deviendra lorsque nous traiterons des informations brutes sans aucun formatage particulier (informations binaires) directement dans un fichier.

ostream& flush () ;

Cette méthode vide explicitement la mémoire tampon.

La classe « istream » :



La classe « istream » hérite de la classe « IOS », grâce à cette héritage, nous pouvons tester le flot de donnée grâce à l'opérator () .

```
(flot)   // ou flot représente n'importe qu'elle classe de la hiérarchie des flots.
```

Le résultat est :

true : si aucun des bits d'erreur n'est activé.

false : dans le cas contraire.

Lorsque nous écrivons :

```
!flot ou ( !flot)   // ou flot représente n'importe qu'elle classe de la hiérarchie des flots.
```

Le résultat est :

false *: si aucun des bits d'erreur n'est activé.*

true *: dans le cas contraire.*

Nous allons utiliser cette caractéristique dans la classe « *istream* » puisqu'elle hérite de tout le comportement de la classe « *ios* ».

! De plus, nous allons voir tout ce qui faut retenir de cette classe (les opérateurs, les méthodes qui sont les plus souvent utilisé à notre niveau).!

istream& operator>> (type) ;

Son rôle consiste à extraire du flot concerné les caractères nécessaires pour former une valeur du type voulu en réalisant une opération inverse du formatage opéré par l'opérateur « << ».

```
int x ;
double y ;
cin >> x >> y ; // saisie à partir du clavier et transfert des valeurs vers les variables x et y.
```

Ici dans cet exemple, il est nécessaire de se servir de séparateurs pour faire la différence entre la valeur prévue pour *x* et celle prévu pour *y* .

Comment peut-on séparer x et y ?

Pour cela, nous avons le choix parmi les délimiteurs qui suivent :

espace « ' ' »,

tabulation horizontale « \t »,

tabulation verticale « \v »,

fin de ligne « \n »,

changement de page « \f ».

Par conséquence, les délimiteurs ne peuvent pas être lus en tant que caractères.

Ainsi, après la déclaration suivante :

```
char message[50] ;
```

Or, lorsque que l'on saisie au clavier la chaîne suivante : « bonjour à tout le monde », seule la valeur « bonjour » est récupérée dans la variable message puisque l'espace est considéré comme un délimiteur.

Il faudra donc utiliser une autre méthode pour récupérer la totalité de la chaîne saisie, notamment la méthode getline .

Soit l'écriture suivante :

```
vector<int> ivec ;
int ival ;
while (cin >> ival) ivec.push_back(ival) ;
```

L'expression : while (cin >> ival) lit une séquence de valeurs depuis l'entrée standard jusqu'à ce que cin soit évalué à false.

Deux conditions générales sont à l'origine de l'évaluation de istream à false :

1. La lecture d'une fin de fichier (auquel cas, toutes les valeurs contenues dans le fichier ont été correctement lues), ou la détection d'une valeur invalide – tel 3.14159 (cette valeur est un réel, et c'est un entier qui est attendu).

2. Dans le cas de la lecture d'une valeur invalide, l'objet cin est placé en état d'erreur et la lecture des valeurs s'interrompt.

istream& getline (char * chaîne , int taille , char délimiteur = '\n') ;

Cette méthode facilite la lecture des chaînes de caractères (non « string »), ou plus généralement d'une suite de caractères quelconques (espace ou caractères de contrôle compris), terminé par un caractère qui sert de délimiteur et qui n'est donc pas utiliser par la chaîne à récupérer.

Cette méthode doit donc être utiliser à la place de l'opérateur de redirection « >> » pour saisir des chaînes comportant plusieurs mots (séparés par des espaces) ou même tout un texte écrit sur plusieurs lignes, auquel cas, il ne faut pas prendre le caractère proposé par défaut « \n ».

Cette méthode lit des caractères sur le flot l'ayant appelé et les places dans l'emplacement désigné par chaîne. Elle s'interrompt lorsqu'une des deux conditions suivantes est respectée :

le caractère qui sert de délimiteur a été trouvé : dans ce cas ce caractère n'est pas recopié en mémoire ;

taille-1 caractères ont été lus.

Dans tous les cas, cette méthode ajoute le caractère nul de terminaison de chaîne à la suite des caractères lus.

```
cin.getline(chaine, 50)
```

Juste au dessus, nous pouvons voir comment utiliser cette méthodes à bonne escient.

Avec :

-> chaine :représentant la variable où va être stockée ce que l'opérateur va taper sur le clavier.

-> 50 : Nombre de caractères stockés dans cette chaine.

istream& read (char * chaîne , int taille) ;

Cette méthode permet de lire sur le flot d'entrée considéré une suite de caractères (octets) en spécifiant la longueur voulue.

```
char chaine[20];
cin.read(chaine, 5) ; // récupère uniquement 5 caractères du tampon du clavier même si il en possède plus.
```

Ici encore cette méthode peut sembler faire double emploi, soit avec la lecture d'une chaîne avec l'opérateur « >> », soit avec la méthode getline.

Toutefois, read ne nécessite ni séparateur de fin de chaîne, ni délimiteur particulier.

Cette méthode est plus couramment utilisée, comme la méthode write, lorsque nous souhaiterons accéder à des fichiers sous forme binaire, c'est-à-dire en recopiant en mémoire les informations telles qu'elles figurent dans le fichier.

Quelle est le formatage de l'information employé sur un flot de données ?

Chaque objet flot conserve en permanence un ensemble d'indicateurs spécifiant quel est, à un moment donné, son statut de formatage.

Ces indicateurs servent à contrôler, par exemple, l'affichage des valeurs entières suivant la base désirée (décimal, octal, hexadécimal), ou bien encore, permet de contrôler la précision des nombres à virgule flottante.

Bien d'autres possibilités de formatage sont offertes.

Ces indicateurs sont positionnés avec des valeurs par défaut, ce qui permet à l'utilisateur d'ignorer totalement cet aspect, tant qu'il se contente de l'affichage par défaut. Un des avantages de ce système est de permettre à celui qui le souhaite, de définir, une fois pour toutes, un format approprié à une application donnée et de plus avoir à s'en soucier par la suite.

Le programmeur dispose de manipulateurs prédéfinis pour modifier l'état de format d'un objet flot. Un manipulateur s'applique à l'objet flux comme s'il s'agissait d'une donnée. Toutefois, au lieu de déclencher la lecture ou l'écriture des données, le manipulateur modifie en fait l'état interne de l'objet flux.

Pour vous montrez comment nous utilisons les flots, nous allons voir deux syntaxes :

1. *Flot << manipulateur* pour un flot de sortie.
2. *Flot >> manipulateur* pour un flot d'entrée.

! Pour les exemples, je vous invite à aller consulter mon site Web, pour récupérer les TPs. !

Voyons maintenant, quelques classes très utiles dans nos programmes.

Inclusion de la classe « iostream.h » :

#include <iostream> // Cette inclusion est obligatoire pour utiliser les manipulateurs ci-dessous	
dec / hex / oct	Base de numération pour les valeurs entières, respectivement : décimal, hexadécimal, octal.
ends	Insère le caractère de fin de chaîne nul, puis vide le tampon.
endl	Insère une nouvelle ligne, puis vide le tampon.
left	Ajoute des caractères de remplissage à droite de la valeur.
right	Ajoute des caractères de remplissage à gauche de la valeur.
boolalpha / noboolalpha	Représente true et false sous forme de chaînes / Représente true et false au format 0, 1
showbase / noshowbase	Génère (ou pas) un préfixe indiquant une base numérique
showpoint / noshowpoint	Affichage du point décimal lorsqu'on utilise des réels et qu'il n'y a pas de parties décimales.
uppercase / nouppercase	Affichage des caractères hexadécimaux en majuscule.
showpos / noshowpos	Affichage des nombres positifs précédés du signe +
skipws / noskipws	Saute (ou pas) l'espace avec les opérateurs d'entrée.
scientific / fixed	Notation scientifique des nombres réels : 1.5 e+01 , ou « point fixe » pour les nombres réels : 10.5
ws	« Mange » l'espace

Avec ce type de manipulateur, nous n'avons pas besoin de repréciser le même traitement pour les variables qui suivent. Une fois que l'on place un manipulateur, il reste actif. Si vous désirez réobtenir le comportement par défaut, vous devez appliquez le manipulateur adéquat.

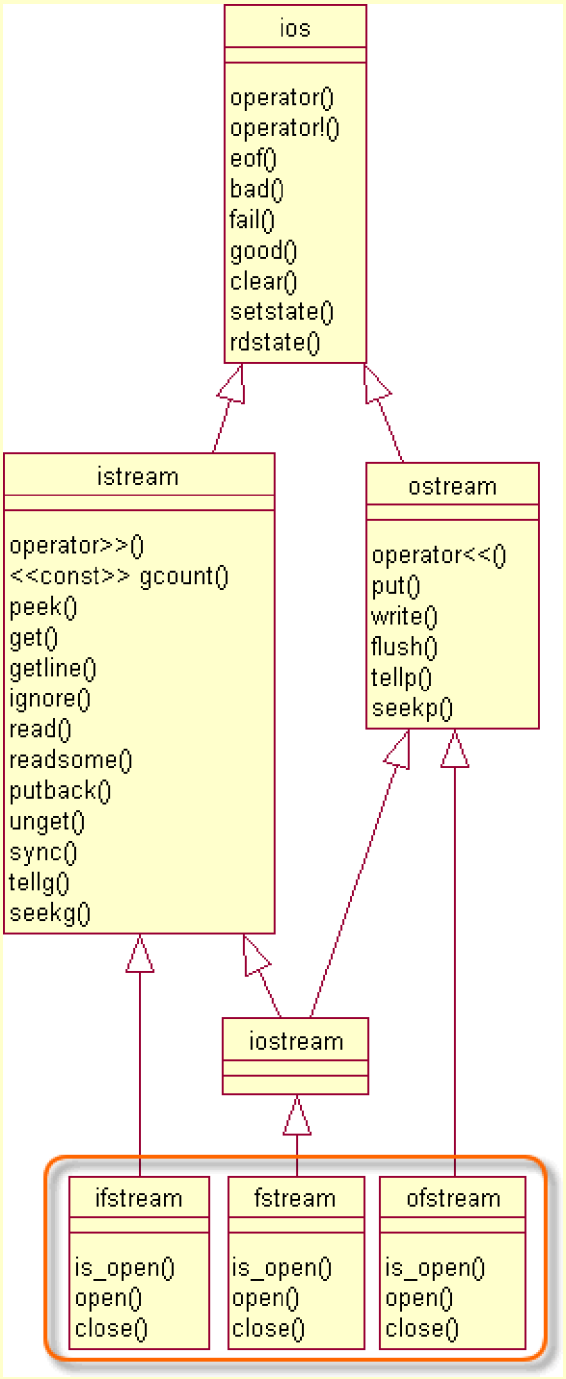
Il existe également des manipulateurs paramétriques qui représente des valeurs numériques et non plus une information tout ou rien. Le principe est similaire aux manipulateurs précédents, toutefois, ils comportent en plus un paramètre qui récupère la valeur spécifiée. Ces manipulateurs vous sont présentés dans la page suivante.

Inclusion de la classe « iomanip.h » :

#include <iomanip.h> // Cette inclusion est obligatoire pour utiliser les manipulateurs paramétrés ci-dessous	
setw (nombre)	**Définit le gabarit de la variable à afficher avec une justification à droite par défaut. Si la valeur à afficher est plus importante que le gabarit, cette valeur ne sera pas tronquée et sera donc affichée de façon conventionnelle. Le manipulateur setw doit être utilisé pour chacune des informations à afficher.**
setfill (caractère)	**Définit le caractère de remplissage lorsqu'on utilise un affichage avec la gestion de gabarit. Par défaut, le caractère de remplissage est l'espace.**
setprecision (nombre)	**Permet de définir le nombre de chiffres significatifs pour les nombres réels. C'est uniquement pour l'affichage, la variable garde sa précision, et la valeur affichée est arrondie. Lorsque l'on utilise au préalable le manipulateur fixed, le manipulateur setprecision permet d'indiquer le nombre de chiffres significatifs après la virgule.**
setbase(base)	**Spécifie la base d'affichage sur les nombres entiers.**

En ce qui concerne setw, sachez que ce manipulateur définit uniquement le gabarit de la prochaine information à écrire. Si l'on ne fait pas de nouveau appel à setw pour les informations suivantes, celles-ci seront écrites suivant les conventions habituelles, à savoir en utilisant l'emplacement minimal nécessaire pour les écrire.

Je vais vous montrer la hiérarchie des classes concernant les fichiers. Au niveau de l'écriture et de la lecture, le tout pour que vous arrivez à comprendre ce qu'il y a derrière.


ios
operator()
operator!()
eof()
bad()
fail()
good()
clear()
setstate()
rdstate()
istream
operator>>()
<<const>> gcount()
peek()
get()
getline()
ignore()
read()
readsome()
putback()
unget()
sync()
tellg()
seekg()
ostream
operator<<()
put()
write()
flush()
tellp()
seekp()
iostream
ifstream
is_open()
open()
close()
fstream
is_open()
open()
close()
ofstream
is_open()
open()
close()

Les classes que l'on utilisera le plus souvent sont celle entourer de rouge, à savoir :

1. ofstream : flot de sortie associé à un fichier. Permet d'écrire des informations de type quelconque dans un fichier.
2. ifstream : flot d'entrée associé à un fichier. Permet de lire des informations de type quelconque issues du fichier.
3. fstream : flot bidirectionnel associé à un fichier. Permet d'écrire ou de lire dans un fichier.

! Une seule question doit vous venir à l'esprit de curiosité. !

Existe-t-il des modes d'ouvertures de fichiers ?

En voici un tableau, regroupant ces différents modes.

Bits d'ouverture de fichier dans le mot d'état *open_mode*	
in	Ouverture en lecture. Le fichier doit exister.
out	Ouverture en écriture. Ecrase l'ancien contenu. Si le fichier n'existe pas, il est automatiquement créé.
app	Ouverture en ajout de données (écriture en fin de fichier).
trunc	Si le fichier existe, son contenu est définitivement perdu.
binary	Pour les précédents modes, l'information été systématiquement transformée en une suite de caractère. Dans l'exemple précédent, nous avons sauvegardé un certain nombre de valeurs de type différent. Au moment du transfert vers le fichier, ces valeurs subissent une transformation pour devenir une suite de caractères. Il est alors possible de contrôler le contenu du fichier avec un simple éditeur de texte. Toutefois, vous pouvez désirer conserver le type original et donc demander à avoir un stockage sous forme binaire. C'est ce que permet ce mode d'ouverture.

Signatures des méthodes et choix du mode d'ouverture d'un fichier

ifstream

- *ifstream (); : constructeur par défaut*
- *ifstream (const char *nomfichier, ios_base::open_mode mode = ios_base::in) ; : Constructeur. Les arguments transmis au constructeur spécifient, tour à tour, le nom du fichier à ouvrir et le mode d'ouverture. Par défaut, pour cette classe, le fichier est ouvert en lecture.*
- *bool is_open () const ; : détermine si le fichier représenté par l'objet est ouvert.*
- *void open (const char *nomfichier, ios_base::open_mode mode = ios_base::in) ; : Ouvre le fichier représenté par l'objet en mode lecture (par défaut).*
- *void close () ; : ferme le fichier représenté par l'objet.*

ofstream

- *ofstream (); : constructeur par défaut*
- *ofstream (const char *nomfichier, ios_base::open_mode mode = ios_base::out) ; : Constructeur. Les arguments transmis au constructeur spécifient, tour à tour, le nom du fichier à ouvrir et le mode d'ouverture. Par défaut, pour cette classe, le fichier est ouvert en écriture.*
- *bool is_open () const ; : détermine si le fichier représenté par l'objet est ouvert.*
- *void open (const char *nomfichier, ios_base::open_mode mode = ios_base::out) ; : Ouvre le fichier représenté par l'objet en mode écriture (par défaut).*
- *void close () ; : ferme le fichier représenté par l'objet.*

fstream

- *fstream (); : constructeur par défaut*
- *fstream (const char *nomfichier, ios_base::open_mode mode = ios_base::in | ios_base::out) ; : Constructeur. Les arguments transmis au constructeur spécifient, tour à tour, le nom du fichier à ouvrir et le mode d'ouverture. Par défaut, pour cette classe, le fichier est ouvert à la fois en lecture et en écriture.*
- *bool is_open () const ; : détermine si le fichier représenté par l'objet est ouvert.*
- *void open (const char *nomfichier, ios_base::open_mode mode = ios_base::in | ios_base::out) ; : Ouvre le fichier représenté par l'objet en mode lecture et écriture (par défaut).*
- *void close () ;* : ferme le fichier représenté par l'objet.

Dans ces classes, les modes d'ouvertures sont positionnés avec des paramètres par défaut, ce qui convient dans la plupart des cas.

Il est toutefois possible de proposer un autre comportement. Nous avons, par exemple, souvent besoin d'ouvrir un fichier en mode ajout.

Il faut donc prendre la classe ofstream qui permet d'écrire dans un fichier et changer le mode par défaut. Ainsi :

```
ofstream fichier("nom du fichier", ios::app) ; ou ofstream fichier("nom du fichier", ofstream::app) ;
```

Et si nous voulons fabriquer un fichier pour écrire des valeurs enregistrées sous forme brute :

```
ofstream fichier("nom du fichier", ios::out | ios ::binary) ;
```

Accès direct à une position absolue dans le fichier :

Le terme flot indique bien que nous soutirons l'information à la volée sous forme séquentielle.

Toutefois, il est possible d'accéder à un endroit précis du fichier pour prendre juste la valeur désirée.

L'accès direct est implémenté sous la forme d'un pointeur de fichier, c'est-à-dire un nombre précisant le rang du prochain « *octet* » à lire ou à écrire.

Après chaque opération de lecture ou d'écriture, ce pointeur est incrémenté du nombre d'octets transférés. Ainsi, lorsque nous n'agissons pas explicitement sur ce pointeur, nous réalisons en fait un accès séquentiel classique; c'est d'ailleurs ce que nous avons fait jusqu'à présent.

Attention, l'accès direct n'est possible qu'en considérant le fichier que sous forme d'une suite d'informations binaires.

Finalement, les possibilités d'accès direct se résument donc aux possibilités d'action sur ce pointeur ou à la détermination de sa valeur.

Des méthodes héritées respectivement de istream et de ostream permettent de se déplacer à une adresse absolue à l'intérieur du fichier ou à une distance en octets depuis une position donnée.

1. La méthode seek permet de positionner le pointeur de fichier à un endroit précis.
2. La méthode tell permet de donner la position actuelle du pointeur de fichier.

En fait, le nom de ces méthodes possède une lettre supplémentaire suivant qu'elles dérivent de istream ou de ostream.

Dans le premier cas, les méthodes issues de istream possèdent le suffixe g (pour get).

Dans le deuxième cas, les méthodes issues de ostream possèdent le suffixe p (pour put).

Par ailleurs, des constantes ont été spécialement conçues pour indiquer respectivement, le début du fichier, la fin du fichier, ou la position courante du fichier.

Ce qui donne dans les différentes classes :

istream

- *istream& seekg (int pos_courante) ;* Le pointeur de fichier pointe sur la position absolue *pos_courante*.
- *istream& seekg (int pos_relative, ios_base::seekdir origine) ;* Le pointeur de fichier pointe relativement à la position *pos_relative* par rapport à *l'origine* fixée par le deuxième argument.
- *int tellg () ;* renvoie la position courante du pointeur de fichier.

ostream

- *ostream& seekp (int pos_courante) ;* Le pointeur de fichier pointe sur la position absolue *pos_courante*.
- *ostream& seekp (int pos_relative, ios_base ::seekdir origine) ;* Le pointeur de fichier pointe relativement à la position *pos_relative* par rapport à *l'origine* fixée par le deuxième argument.
- *int tellp () ;* renvoie la position courante du pointeur de fichier.

ios_base

- *ios_base ::beg* : début du fichier.
- *ios_base ::cur* : position courante du fichier.
- *ios_base ::end* : fin du fichier.
- Ces trois constantes sont à utiliser en corrélation avec ios_base ::seekdir .

Pour l'accès direct, puisque nous travaillons octet par octet, il peut être utile de connaître la dimension exacte en octets des variables que nous utilisons pour envoyer ou recevoir des valeurs stockées dans les fichiers.

Il existe l'opérateur sizeof qui réalise ce calcul et qui peut être utilisé de trois façons différentes :

```
sizeof ( type )
sizeof ( objet )
sizeof objet  ;
```

Choix du type de fichier:

Ce que nous avons proposé comme nom de fichier jusqu'à présent, correspondait à un fichier sur le disque dur.

Il est toutefois possible de communiquer avec d'autres ressources.

Si vous tester le programme ci-dessous, vous allez vous apercevoir que les objets clavier et ecran remplacent cin et cout.

Ex :

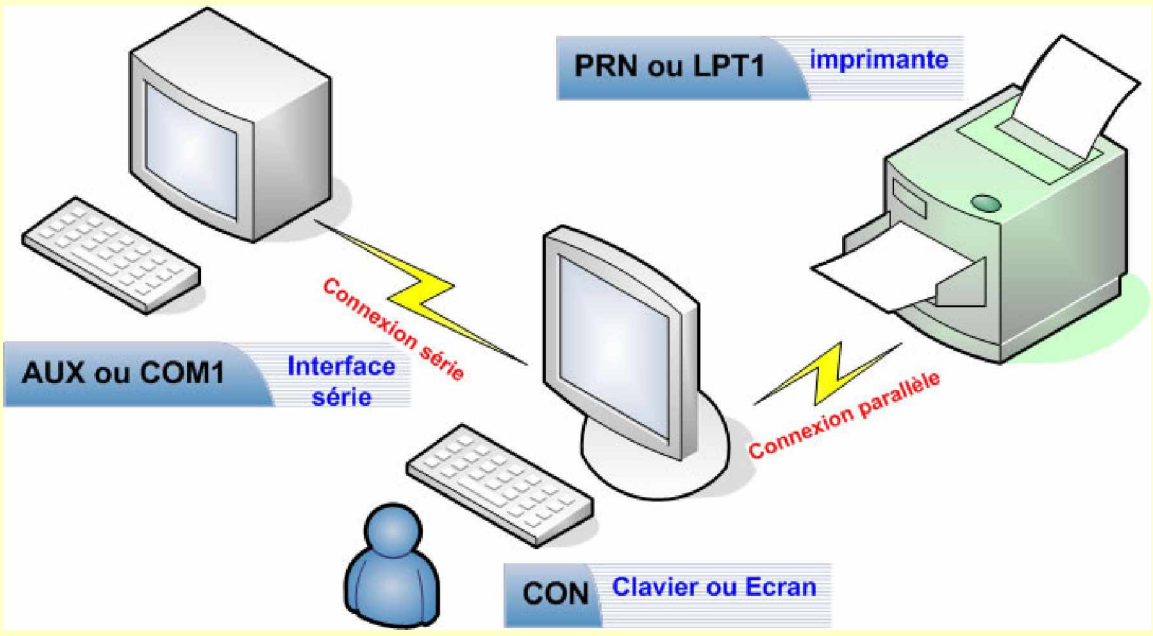
```
1  #include <fstream>
2  using namespace std;
3  //-------------------------------------------------
4  int main()
5  {
6    ifstream clavier("CON");
7    int valeur;
8    clavier >> valeur;
9    ofstream ecran("CON", ofstream::app);
10   ecran << valeur << endl;
11   return 0;
12 }
```

Le flux d'entrée est redirigé vers le clavier

Le flux de sortie est redirigé vers l'écran en mode ajout

En réalité, le nom de fichier « CON » est un mot réservé du système d'exploitation qui correspond à la « console » de l'ordinateur, c'est-à-dire, en entrée effectivement le clavier, et en sortie l'écran.

Il existe d'autres mots réservés du système d'exploitation, comme le montre le schéma ci-dessous :



Si vous travaillez sous Linux, Unix où autre OS, voici un exemple :

```
ofstream fenetre("/dev/tty3", ios::app) ;
```

Les flux de chaine:

Plutôt que d'être en communication avec un périphérique quelconque, il est également possible de gérer les flux directement en mémoire centrale, tout en utilisant les méthodes et les constantes que nous venons de mettre en œuvre. Ce stockage en mémoire se fait sous forme d'une chaîne de caractères. Il sera ensuite possible de récupérer cette chaîne dans un objet de type string. Nous avons vu que dans les flux, grâce à la redéfinition des opérateurs de redirection, il est possible de stocker et de récupérer des valeurs de type quelconque. Finalement, ce stockage en mémoire sera surtout utiliser pour transformer des valeurs de type quelconque, comme des entiers, des réels, etc. vers une chaîne de caractères ou l'inverse.

Voici 3 classes qui sont spécialisé dans ce cas :

1. ***ostringstream*** : flot de sortie associé à une chaîne de caractères.(hérite de ostream)
2. ***istringstream*** : flot d'entrée associé à une chaîne de caractères. (hérite de istream)
3. ***stringstream*** : flot bidirectionnel associé à une chaîne de caractères. (hérite de iostream)

De même, voici des méthodes associées à ces classes de gestion de chaînes de caractères.

Tout le mécanisme interne du traitement de chaîne de caractères est totalement caché (encapsulé) comme d'ailleurs pour la gestion des fichiers. Nous avons juste une méthode pour retourner la valeur de la chaîne de caractères et les constructeurs qui disposent chacun de paramètres par défaut pour permettre une utilisation la plus simple possible et pour correspondre aux cas les plus fréquents. Malgré tout, il existe des constructeurs qui permettent de construire le flot à partir d'une chaîne de caractères afin de proposer un formatage de cette chaîne vers d'autres types par la suite.

1. *istringstream (ios_base::open_mode mode = ios_base::in) ;* : Constructeur en mode lecture par défaut.
2. *istringstream (string chaîne, ios_base::open_mode mode = ios_base::in) ;* : Constructeur en mode lecture par défaut.
3. *ostringstream (ios_base::open_mode mode = ios_base::out) ;* : Constructeur en mode écriture par défaut.
4. *ostringstream (string chaîne, ios_base::open_mode mode = ios_base::out) ;* : Constructeur en mode écriture par défaut.
5. *stringstream (ios_base::open_mode mode = ios_base::in | ios_base::out) ;* : Constructeur en mode lecture et écriture par défaut.

6. *stringstream (string chaîne, ios_base::open_mode mode = ios_base::in | ios_base::out) ;* : Constructeur en mode lecture et écriture par défaut.
7. *string str () ;* : renvoie la chaîne de caractère stockée dans le flux.

En conclusion, nous disposons, grâce à ces classes, de tout un système de formatage pour passer d'un type quelconque vers une chaîne de caractère et vice versa.

Changement de comportement par défaut :

Le comportement par défaut de toute cette hiérarchie de la gestion des flux est déjà remarquable. Toutefois, ce serait encore mieux si nous pouvions avoir, par exemple, un affichage automatique sur les classes que nous créons. Il suffirait alors de proposer notre objet directement derrière l'opérateur de redirection pour que, effectivement, il s'affiche suivant notre désir.

En fait, il suffit de redéfinir l'opérateur « << » pour que ce fonctionnement s'applique. Attention toutefois, notre nouvelle classe ne faisant pas partie de la hiérarchie, nous sommes obligés de redéfinir cet opérateur en tant que fonction et non pas en tant que méthode. Il faudra donc proposer une relation d'amitié, à moins que vous ne disposiez de toutes les méthodes requises pour la lecture des attributs. (Revoir l'étude de la redéfinition des opérateurs pour comprendre ces problèmes).

Ce que je viens de dire pour une communication vers l'extérieur s'applique, bien entendu, pour une lecture. Ainsi, il sera également possible de redéfinir l'opérateur « >> », et donc de prévoir, par exemple, une saisie clavier adaptée à la classe étudiée.

Cette hiérarchie de classe intègre le polymorphisme. Donc, lorsque nous redéfinirons les différents opérateurs, ils seront alors utiles aussi bien pour la console (le clavier et l'écran) que pour l'écriture ou la lecture dans un fichier (disque dur, imprimante, interface série, etc.) ou pour transformer notre classe en une chaîne de caractères et vice versa.

Rappelons qu'à droite d'un opérateur de redirection nous avons la classe à traiter, ensuite à gauche le flot concerné, et qu'une fois que l'opération s'est déroulée correctement l'opérateur renvoie également un objet flot, ce qui permet notamment les enchaînement des opérateurs de redirection.

Nous aurons donc les gabarits suivants :

```
ostream& operator << (ostream&, const NouvelleClasse&) ;
istream& operator >> (istream&, NouvelleClasse&) ;
```

Gestions des répertoires

Il peut être utile d'avoir des renseignements sur le contenu d'un répertoire afin de pouvoir contrôler l'existence d'un fichier. Malheureusement, cette possibilité n'a pas été intégrée directement dans les flux standard. Par contre, nous pouvons faire référence à un certain nombre de fonctions qui s'occupent de ce genre de problème et qui existe depuis le début du langage C. Nous allons en recenser quelques unes.

- *int chdir (char * répertoire) ;* Changement du répertoire courant. Retourne *0* si l'opération a réussi.
- *int mkdir (char * répertoire) ;* Création d'un nouveau répertoire. Retourne *0* si l'opération a réussi.
- *int rmdir (char * répertoire) ;* Suppression du répertoire. Retourne *0* si l'opération a réussi.
- *int getcurdir (int unité , char * répertoire) ;* Spécifie le nom du répertoire courant en précisant l'unité de disque (*0 : disque courant, 1 : unité A, 3 : unité C, ...*). Retourne *0* si l'opération a réussi.
- *int getcwd (char * répertoire , int taille) ;* Spécifie le nom du répertoire courant avec en plus le nom de l'unité. Il faut, par contre préciser la dimension de la chaîne de caractères. La constante *MAXDIR* contient le nombre de caractères à réserver en toute sécurité.
- *int getdisk (void) ;* Retourne l'unité par défaut.
- *int setdisk (int unité) ;* Changement de l'unité courante. Retourne *0* si l'opération a réussi.
- *int findfirst (char * répertoire , struct ffblk * info , int attribut) ;* Cherche le premier fichier du répertoire courant en spécifiant les filtres désirés. Toutes les informations relatives à un fichier sont stockées dans la structure de type « *ffblk* ». Il est possible grâce à *attribut* de spécifier le type de fichier attendu (voir plus loin). Retourne *0* si l'opération a réussi.
- *int findnext (struct ffblk * info) ;* Trouve le fichier suivant. Cette fonction s'utilise à la suite de la fonction *findfirst* et se sert de la structure initialisée par cette dernière. Retourne *0* si le fichier a été trouvé, sinon, elle retourne *-1* .

Structure ffblk qui donne toutes les informations sur l'en-tête de fichier et la structure ftime <io.h> qui permet de récupérer la date et l'heure issues de ffblk

```
60 struct  ffblk   {        Structure d'un en-tête de fichier
61     long            ff_reserved;
62     long            ff_fsize;     taille
63     unsigned long   ff_attrib;
64     unsigned short  ff_ftime;     Date et heure
65     unsigned short  ff_fdate;
66     char            ff_name[MAXPATH];   nom
67 };
```

```
43 struct  ftime   {
44     unsigned    ft_tsec  : 5;
45     unsigned    ft_min   : 6;
46     unsigned    ft_hour  : 5;
47     unsigned    ft_day   : 5;
48     unsigned    ft_month : 4;
49     unsigned    ft_year  : 7;
50 };
```

Attributs d'un fichier <dir.h>

- *FA_NORMAL* : pour obtenir les fichiers normaux,
- *FA_RONLY* : pour obtenir les fichiers à lecture seule,
- *FA_HIDDEN* : les fichiers cachés,
- *FA_LABEL* : l'étiquette de volume (soit le disque, soit la partition),
- *FA_DIREC* : les répertoires,
- *FA_ARCH* : les fichiers marqués à archiver.

Pour terminer sur les flux et les fichiers, voici une petite illustration.

```cpp
#include <iostream.h>
#include <dir.h>
#include <io.h>
//---------------------------------------------------------------------------
int main( )
{
  char chaine[MAXDIR];
  ffblk info;
  ftime &temps = (ftime &) info.ff_ftime;

  chdir("..\\..");
  getcurdir(0, chaine);
  cout << "Répertoire courant : " << chaine << endl;
  cout << "Unité de disque : " << char('A'+getdisk()) << endl;
  getcwd(chaine, MAXDIR);
  cout << "Répertoire courant : " << chaine << endl;

  if (findfirst("*.*", &info, FA_NORMAL | FA_DIREC))
    cout << "Aucun fichier dans ce répertoire" << endl;
  else do {
    cout << info.ff_name << '\t';
    if (info.ff_attrib == FA_DIREC) cout << "<REP>\t";
    cout << temps.ft_day << '/' << temps.ft_month << '/' << 1980+temps.ft_year;
    cout << ' ' << temps.ft_hour << ':' << temps.ft_min << endl;
  }
  while (!findnext(&info));

  return 0;
}
```

```
Répertoire courant : Documents and Settings\Emmanuel REMY\cbproject\Test
Unité de disque : C
Répertoire courant : C:\Documents and Settings\Emmanuel REMY\cbproject\Test
.       <REP>   29/12/2004 16:15
..      <REP>   29/12/2004 16:15
bak     <REP>   29/12/2004 16:15
Test.cbx        29/12/2004 9:42
Test.cbx.local  29/12/2004 11:53
Test.cpp        29/12/2004 16:15
windows <REP>   8/12/2004 17:5
```

LE GESTION DES EXCEPTIONS

Définition :

Même lorsqu'un programme est au point, certaines circonstances exceptionnelles peuvent compromettre la poursuite de son exécution ; il peut s'agir par exemple de données incorrectes ou de la rencontre d'une fin de fichier prématurée (alors que nous avons besoin d'informations supplémentaires pour continuer le traitement).

Les exceptions sont donc des anomalies qu'un programme détecte en cours d'exécution, telles des divisions par 0, un accès à l'extérieur des bornes d'un tableau ou l'épuisement de la mémoire.

De telles exceptions sortent du fonctionnement normal du programme et requièrent de sa part une gestion immédiate.

Le langage C++ dispose d'un mécanisme très souple nommé gestion d'exception , qui permet à la fois :

- de dissocier la détection d'une anomalie de son traitement,
- de séparer la gestion des anomalies du reste du code, donc de contribuer à la lisibilité des programmes.

Ce mécanisme s'effectue toujours en deux temps.

1. Nous avons d'abord, la détection de l'anomalie. Le développeur doit alors avertir du disfonctionnement en provoquant une rupture de séquence par le déclenchement d'une exception correspondant à l'anomalie. Cette phase s'appelle souvent lever ou lancer une exception. Nous lançons une exception par la directive throw.
2. Ensuite, si nous le désirons, nous pouvons nous occuper de ce qui s'appelle la gestion d'exception, qui consiste à proposer un certain nombre d'actions pour gérer le problème lié à une ou plusieurs anomalies. Généralement, nous tentons d'abord d'effectuer le traitement prévu, et si cela se passe mal, nous capturons l'exception lancée par la directive throw. Nous proposons alors une alternative correspondant au type de l'exception. En fait, et plus précisément, chaque exception est caractérisée par un type, et le choix du bon gestionnaire se fait en fonction de la nature de l'expression mentionnée à throw. Cette phase est réalisée par les directives try – catch (essayer et capturer).

Généralement, ces deux phases sont traitées par deux développeurs différents.

En effet, le premier construit les classes. Celui-ci doit alors prévoir tous les cas où la classe peut être mal utilisée. Il doit donc proposer un ensemble d'exceptions correspondant aux disfonctionnements possibles.

Le second est celui qui utilise les classes. Celui-ci doit gérer les exceptions suivant l'utilisation qu'il fait de ces classes.

Ainsi, la gestion d'exception peut être totalement différente suivant l'utilisateur et surtout suivant le programme à traiter.

Le fait d'avoir deux phases permet de simplifier considérablement la situation, chaque programmeur s'occupe de son propre problème.

Détection des anomalies dans un fonctionnement d'un programme correct.

La première démarche, et ce n'est pas toujours la plus facile, consiste à recenser toutes les anomalies possibles au sein d'une classe, dues généralement, à une mauvaise manipulation de la part du programmeur qui l'utilise.

Pour illustrer ces propos, je vous propose de revenir sur l'étude d'un tableau d'entier.

Deux cas sont en réalité à évité :

1. D'une part, une mauvaise proposition d'indice qui nous envoie en dehors des limites du tableau.
2. D'autre part, une taille de tableau négative.

Quand ce genre de problème arrive, il est préférable de tout arrêter plutôt que de faire n'importe quoi et d'accéder à une partie de la mémoire qui n'est pas prévue.

! Cette méthode mène tout droit vers une écran bleu !

Pourrons nous levée une exception ?

Qu'est ce qu'une exception ?

Eventuellement, celui qui construit la classe pourrait envisager de proposer une solution dans le cas, par exemple, où l'utilisateur tente d'accéder à une case du tableau au delà des limites prévues.

Mais alors, que choisir comme indice. Le concepteur de la classe ne sait pas ce que l'utilisateur désire réellement faire. Il est préférable que le concepteur de la classe laisse l'initiative à l'utilisateur et juste le prévenir qu'il y a un problème.

Pour prévenir l'utilisateur, il faut lever une exception qui correspond à l'anomalie. Pour cela, nous devons utiliser l'instruction throw suivi d'une valeur d'un type quelconque.

Nous pouvons, par exemple, proposer une valeur numérique entière qui indique l'erreur correspondant à l'anomalie, comme -1 pour le problème de construction, et -2 pour le problème lié à l'indice.

Ceci dit, cette démarche n'est pas très élégante.

Que se passe-t-il lorsqu'une exception est levée ?

En fait, tout dépend si nous gérons l'exception ou pas (à l'aide du bloc try – catch).

Si ce n'est pas le cas, l'exception provoque l'arrêt pur et simple du programme.

De toute façon, ce n'est pas la peine d'aller plus loin, puisque si une exception est levée sans être gérée, nous nous trouvons alors dans une situation plutôt catastrophique.

Voici les trois règles à suivre pour avoir un programme qui sont au point face aux multiples situation engendrées par l'opérateur.

1. Le développeur qui fabrique les classes doit s'occuper de recenser l'ensemble des anomalies possibles et lance des exceptions correspondant à ces disfonctionnements.
2. Pour cela, il fabrique une classe d'erreur par type d'anomalie.
3. Ensuite, c'est tout, son travail est terminé.

Interception et gestion des exceptions :

Lorsqu'une exception est levée, plutôt que d'avoir un programme qui se termine de façon abrupte, il serait souhaitable de maîtriser la situation et de proposer une alternative de fonctionnement.

Pour cela, il faut mettre en œuvre ce que l'on appelle une gestion d'exception qui se déroule finalement en trois phases :

1. Tentative d'exécution d'un ensemble d'instructions,
2. Capture de l'exception, si un problème est rencontré durant cette tentative,
3. Gestion de l'exception en proposant une nouvelle suite d'instructions.

Qu'est ce qu'une « tentative » ?

Pour intercepter et gérer les exceptions possibles, vous devez d'abord entourer les instructions qui sont susceptibles de lever des exceptions par un bloc try.

Un bloc try commence par le mot clé try suivi d'une séquence d'instructions entourées d'accolades.

Le bloc try est suivi d'une liste de gestionnaires appelés clauses catch.

En fait, le bloc try regroupe un ensemble d'instructions et leur associe un ensemble de gestionnaires pour gérer les exceptions que peuvent lever les instructions.

Si aucune exception ne survient, l'ensemble du code à l'intérieur du bloc try est exécuté et les gestionnaires associés au bloc try sont ignorés.

Le programme exécute ensuite les instructions qui sont placées à la suite des clauses catch.

Si une exception est levée à l'intérieur d'un bloc try, les instructions qui suivent l'instruction lançant l'exception ne sont pas exécutées.

L'exécution du programme reprend dans la clause catch gérant l'exception.

Ici, la gestion des exceptions sont décrite dans le bloc catch.

Quand une exception est levée depuis des instructions dans un bloc try, la liste des clauses catch qui suit le bloc try est recherchée afin d'y trouver une clause catch qui soit capable de gérer l'exception.

Une clause catch se compose de trois parties :

1. le mot clé *catch*,
2. la déclaration d'un type unique ou d'un objet unique entre parenthèses (appelée déclaration d'exception),
3. et un ensemble d'instructions dans une instruction composée (accolades).

Si la clause catch est sélectionnée pour gérer une exception, l'instruction composée est exécutée.

Dès qu'une clause catch a terminée son travail, l'exécution du programme continue sur l'instruction qui suit la dernière clause catch de la liste.

Le mécanisme de gestion des exceptions du C++ est dit sans reprise ; une fois l'exception gérée, l'exécution du programme ne reprend pas là où l'exception a été levée.

Comment peut se dérouler une clause « catch » ?

La recherche d'une clause catch pour gérer une exception levée se déroule ainsi. Si l'expression throw se trouve dans un bloc try, les clauses catch associées à ce bloc sont examinées pour voir si l'une d'elles peut gérer l'exception.

Si une clause catch est détectée, l'exception est gérée. Si aucune clause catch n'est détectée, la recherche se poursuit dans le bloc try-catch de niveau supérieur (celui qui englobe le try-catch imbriqué).

Si une clause catch est trouvée dans ce nouveau bloc, l'exception est gérée sinon la recherche se poursuit à un niveau encore supérieur. Ce processus se poursuit en remontant l'imbrication des blocs try-catch jusqu'à ce qu'une clause catch pour l'exception soit trouvée. Dès qu'une clause catch pouvant gérer l'exception est rencontrée, on entre dans la clause catch et l'exécution du programme continue dans ce gestionnaire.

Si aucun gestionnaire n'est trouvé, le programme appelle la fonction terminate() définie dans la bibliothèque du C++ standard. Cette fonction propose un comportement par défaut, qui appelle notamment la fonction abort() qui elle-même indique que le programme se termine anormalement « Abnormal program termination ».

Comment propager une exception ?

Il est possible qu'une clause unique ne puisse pas gérer une exception complètement.

Après quelques actions correctives, une clause catch peut décider que l'exception sera gérée par un bloc try-catch de niveau supérieur.

Il suffit pour cela de propager l'exception.

Dans un gestionnaire, l'instruction throw (sans expression) retransmet (propage) l'exception au niveau englobant.

```
catch (ErreurIndice) {
  cerr << "Attention, votre indice n'est pas correct" << endl;
  correcte = false;
  throw;
}
```

Propagation vers le niveau englobant par redéclenchement de l'exception

Si nous utilisons cette technique, il faut, bien entendu, que le bloc supérieur soit capable de capturer ce type d'exception et qu'il dispose donc du même gestionnaire.

Existe-t-il un gestionnaire pour toutes les exceptions ?

Au lieu de proposer un gestionnaire par type d'anomalies possibles, vous pouvez capturer toutes les exceptions dans une seule clause catch.

Cette clause catch possède une déclaration d'exception de la forme (...), où les trois points sont une ellipse.

```
try {
   Tableau tab(10);
   tab[5] = 3;
   Tableau faux(-12);
   tab[-5] = tab[15];
 }
 catch (ErreurCreation) { cerr << "Problème à la création" << endl; }
 catch (ErreurIndice)   { cerr << "Mauvais indice" << endl; }
 catch (...)            { cerr << "Autre problèmes" << end; }
```

ATTENTION
Ne pas y mettre en 1er. !!!!!!!!!!

Vous pouvez aussi combiner les exceptions en gérant quelques unes plus précisément et les autres de façon globale en utilisant alors la clause avec l'ellipse.

Cela implique qui si un catch (...) est combiné avec d'autres clauses catch, il sera toujours placé en dernier de la liste des gestionnaires d'exception.

En effet, les clauses catch sont examinées à tour de rôle, dans l'ordre où elles apparaissent à la suite du bloc try.

Si les clauses catch particulières se trouvaient après la clause catch comportant l'ellipse, elles ne seraient jamais atteintes.

CONCLUSION :

Le développeur qui utilise les classes ne s'occupe pas du tout de recenser les anomalies possibles.

Il doit juste proposer un certain nombre d'alternatives en gérant les exceptions qui peuvent être levées suivant les tentatives qu'il propose.

Vous remarquez que par cette disposition, chacun s'occupe de son propre domaine, ce qui simplifie notablement le travail.

Quelles sont les spécifications d'exception ?

La spécification d'exception offre une solution pour lister les exceptions qu'une méthode peut lever en même temps que la déclaration de la méthode. Elle assure (vérifié par le compilateur) que la méthode ne lance aucun autre type d'exception.

Une spécification d'exception suit la liste des paramètres de la méthode. Elle est déclarée avec le mot clé throw, suivi d'une liste des types d'exception entourée de parenthèses.

Est –ce tout cela peut s'appliquée aux objets ?

Nous avons vus que la déclaration d'exception d'une clause catch peut être soit une déclaration de type, soit une déclaration d'objet.

Quand la déclaration d'exception dans une clause catch déclare-t-elle un objet ?

Un objet sera déclaré lorsque nous devons obtenir la valeur ou manipuler l'objet exception créé par l'expression throw.

En effet, jusqu'à présent, les classes d'exception que nous avons créées étaient réduites à leurs plus simples expressions. Mais il s'agit de classes à part entière, comme les autres, et rien n'empêche de les créer de façon beaucoup plus sophistiquées avec un certain nombre d'attributs et de méthodes. Il est même possible de structurer tout une hiérarchie de classes d'erreur.

Il peut être utile de fabriquer des classes d'erreur plus complètes afin de stocker, par exemple, la valeur qui a provoqué l'erreur ainsi que les valeurs limites qu'impose le bon fonctionnement des classes normales.

La déclaration d'exception au niveau des clauses catch ressemble à un paramètre d'une méthode. Pour prévenir les copies inutiles d'objet de classes de grande taille, il est préférable que les déclarations d'exception soient déclarées en tant que référence.

Existe-t-il une hiérarchie dans les exception ?

Nous allons continuer nos investigations en proposant, cette fois-ci, une hiérarchie de classes polymorphiques, juste pour montrer toutes les possibilités et la souplesse du langage C++.

Dans l'exemple qui suit, nous construisons une classe de base abstraite où il sera nécessaire de redéfinir la méthode getMessage qui délivrera le message correspondant à l'objet levé.

```
6  class ErreurTableau
7  {
8    int valeurIntroduite;
9  public:
10   ErreurTableau(int valeur)  { valeurIntroduite = valeur; }
11   int getValeurIntroduite()   { return valeurIntroduite; }
12   virtual string getMessage() = 0;
13 };
14 //---------------------------------------------------------
15 class ErreurCreation : public ErreurTableau
16 {
17 public:
18   ErreurCreation(int valeur) : ErreurTableau(valeur) { }
19   string getMessage() {
20     ostringstream message;
21     message << "La dimension du tableau doit être positive" << endl;
22     message << "La valeur que vous avez passé est la suivante : ";
23     message << getValeurIntroduite() << endl;
24     return message.str();
25   }
26 };
27 //---------------------------------------------------------
28 class ErreurIndice : public ErreurTableau
29 {
30   int limite;
31 public:
32   ErreurIndice(int valeur, int taille) : ErreurTableau(valeur) {
33     limite = taille;
34   }
35   int getLimite() { return limite; }
36   string getMessage() {
37     ostringstream message;
38     message << "Attention, votre valeur " << getValeurIntroduite();
39     message << " n'est pas correcte. Elle doit être comprise";
40     message << " entre 0 et " << limite << endl;
41     return message.str();
42   }
43 };
```

Classe abstraite

Redéfinition de la méthode adaptée à la situation

Redéfinition de la méthode adaptée à la situation

Pour lever une exception, rien ne change, il suffit de créer l'objet relatif à la classe qui correspond au défaut détecté.

Pour la capture, cela peut être, finalement, beaucoup plus simple. Il suffit, en effet, de faire une capture par rapport à une référence sur la classe de base uniquement.

Le mécanisme du polymorphisme permettra de récupérer le bon objet exception.

```
63 int main( int argc, char * argv[] )
64 {
65   try {
66     Tableau tab(10);
67     tab[5] = 3;
68     Tableau faux(-12);
69     tab[-5] = tab[15];
70   }
71   catch (ErreurTableau &erreur) { cerr << erreur.getMessage(); }
72   catch (...)   { cerr << "Pas assez de mémoire" << endl; }
73   cout << "Le programme se termine";
74   return 0;
75 }
```

Cette fois-ci, nous pouvons globaliser les erreurs issues du tableau par une référence à la classe de base. C'est l'objet levé par l'exception qui déterminera le type d'erreur.

REMERCIEMENT

Je remercie M. Remy Emmanuel, pour m'avoir donner l'autorisation de reprendre son cour, de le simplifier et de le publier sur l'Internet, plus particulièrement sur mon site.