LA LIBRAIRIE STANDARD DU C++

OURS BLANC DES CARPATHES™

ISIMA–1999

Table des matières

1. RAPPELS HISTORIQUES 4

2. LES FONDAMENTAUX 5

2.1 ESPACES DE NOMMAGE 5
2.1.1 MOTIVATION 5
2.1.2 PROPRIÉTÉS 5
2.1.3 MISE EN ŒUVRE DES ESPACES DE NOMMAGE 6
2.1.4 UN EXEMPLE PLUS COMPLET 11
2.1.5 LA STL ET LES ESPACES DE NOMMAGE 13
2.2 QUELQUES POINTS À CONNAÎTRE 13
2.2.1 LA PORTABILITÉ DE LA STL 13
2.2.2 LES FICHIERS D'ENTÊTE UTILISÉS PAR LA STL 13
2.3 LES CHAÎNES DE CARACTÈRES 14
2.3.1 CONSTRUCTION DE VARIABLES DU TYPE `STRING`, CONVERSION AVEC `CHAR *` 14
2.3.2 LES OPÉRATIONS DE ROUTINE 15
2.4 LES EXCEPTIONS DE LA STL 17
2.4.1 LA CLASSE `EXCEPTION` 17
2.4.2 LA HIÉRARCHIE DES EXCEPTIONS DE LA STL 17
2.4.3 EXEMPLE D'UTILISATION DES EXCEPTIONS DE LA STL 19
2.4.4 AUX UTILISATEURS DE C++ BUILDER 22
2.5 LE SUPPORT DE LA RTTI 22
2.5.1 QU'EST CE QUE LE RTTI 22
2.5.2 LA CLASSE `TYPE_INFO` ET L'OPÉRATEUR `TYPEID` 23
2.5.3 EXEMPLE D'UTILISATION DU RTTI 23

3. LES CLASSES CONTENEUR DE LA STL 26

3.1 PHILOSOPHIE GÉNÉRALE 26
3.2 PRÉSENTATION DES CONTENEURS 26
3.2.1 PRÉSENTATION DES SÉQUENCES ÉLÉMENTAIRES 26
1.1.2 LES OPÉRATIONS DISPONIBLES SUR L'ENSEMBLE DES SÉQUENCES ÉLÉMENTAIRES 28
1.1.3 OPÉRATIONS SPÉCIFIQUES 29
1.1.4 ITÉRATEURS ET SÉQUENCES ÉLÉMENTAIRES 41
1.1.5 LES UTILISATIONS SPÉCIALISÉES DES SÉQUENCES ÉLÉMENTAIRES 41
1.1.6 LES CONTENEURS ASSOCIATIFS 46
1.3 LES ITÉRATEURS 59
1.3.1 DÉFINITION ET PREMIER CONTACT 59
1.3.2 LES DIFFÉRENTS TYPES D'ITÉRATEURS 60
1.3.3 COMMENT OBTENIR UN ITÉRATEUR ? 61
1.3.4 LES ITÉRATEURS DE FLUX 62

4. ANNEXE : UN EXEMPLE DE `MAKEFILE` 63

Tables des illustrations

Figure 2.1 Espaces de nommage avec noms conflictuels *8*
Figure 2.2 Hiérarchie des exceptions de la STL *18*
Figure 2.3 Sortie écran du programme précédent *21*
Figure 2.4 Résultat écran du programme précédent *25*
Figure 3.1 Résultat du programme d'essai sur `vector` *33*
Figure 3.2 Sortie écran de la démonstration de `list` *38*
Figure 3.3 Sortie écran du programme de test piles / files *44*
Figure 3.4 Sortie écran du programme d'exemple de `map` *56*
Figure 3.5 Résultat d'exécution du programme de test sur les `multimap` *59*

Programme 2.1 déclaration et définition d'un espace de nommage *6*
Programme 2.2 Exemple de collision entraînée par l'utilisation de la cause `using namespace` *8*
Programme 2.3 Exemple d'utilisation de la clause `using` *et des dangers inhérents* *9*
Programme 2.4 Exemple plus complet d'utilisation des alias *11*
Programme 2.5 Exemple complet d'utilisation des espaces de nommage *13*
Programme 2.6 Mini exemple de constructeurs de la classe `string` *14*
Programme 2.7 Déclaration de la classe std::exception *17*
Programme 2.8 exemple complet d'utilisation des exceptions de la STL *21*
Programme 2.9 utilisation des `type_info` *et de* `typeid` *24*
Programme 3.1 La classe `Entier` *31*
Programme 3.2 Manipulations sur la taille des vecteurs *33*
Programme 3.3 Démonstration du type `list` *38*
Programme 3.4 Les classes de générateurs `GenAlea` *et* `GenLin` *39*
Programme 3.5 Programme de test piles et files *44*
Programme 3.6 Utilisation d'une file à priorité *46*
Programme 3.7 Définition du type `pair` *48*
Programme 3.8 Définition de la fonction `make_pair` *49*
Programme 3.9 Utilisation de `make_pair` *49*
Programme 3.10 signature des opérations ensemblistes *52*
Programme 3.11 Utilisation de `map` *55*
Programme 3.12 Utilisation d'une `multimap` *pour la gestion d'une base de code postaux* *58*
Programme 3.13 Essai des itérateurs de flux en entrée *63*
Programme 4.1 `Makefile` *générique pour le C++* *64*

Tableau 2.1 méthodes agissant sur la taille des chaînes de caractères *16*
Tableau 2.2 les méthodes générales d'ajout dans les séquences *17*
Tableau 2.3 Spécificités des différentes exceptions de la STL *19*
Tableau 3.1 Les opérations disponibles sur toutes les séquences élémentaires *29*
Tableau 3.2 Méthodes spécifiques à `vector` *30*
Tableau 3.3 Opérations spéficiques à `list` *36*
Tableau 3.4 Opérations spécifiques au type `deque` *40*
Tableau 3.5 Opérations disponibles sur une file *42*
Tableau 3.6 Opérations disponibles sur les piles *43*
Tableau 3.7 Opérations disponibles sur une file à priorité *45*
Tableau 3.8 Opérations identiques sur les deux conteneurs `set` *et* `map` *51*
Tableau 3.9 Les fonctions membres spécifiques à `set` *51*
Tableau 3.10 Fonctions globales s'appliquant au type `set` *52*
Tableau 3.11 Opérations spécifiques à `map` *53*
Tableau 3.12 Les différents types d'itérateurs *61*

1. Rappels historiques

La librairie standard du C++ est née de la volonté d'apporter aux programmeurs C++ un canevas de programmation efficace, générique et simple à utiliser. Celui-ci comprend notamment :

- Des classes conteneur génériques et des algorithmes associés
- Une classe **`string`** paramétrée par le type de caractères qu'elle contient
- Une nouvelle famille de classes flux (**`iostream`**).

Autrefois connue sous le nom de STL (Standard Template Library) – avant l'adjonction de la nouvelle classe chaînes et des bibliothèques de flux étendues–, la Librairie Standard du C++ reprend nombre des caractéristiques de son ancêtre.

Le présent essai n'a pas pour objectif – loin de là – d'être exhaustif sur le domaine mais uniquement de présenter les concepts fondamentaux et quelques exemples (orientés Recherche Opérationnelle) d'utilisation de la STL.

Pour clôturer cette introduction, je voudrais insister sur l'intérêt évident de généraliser l'utilisation de la STL par quelques critères directement issus du génie logiciel.

Fiabilité : les classes de la STL sont particulièrement sûres, et, dès qu'une erreur est découverte, elle est rapidement corrigée[1]. En outre, la plupart des composants a été écrite par des spécialistes, en particulier Andrew Koenig, l'un des premiers et plus fervents concepteurs / utilisateurs du C++.

Réutilisabilité : la STL étant dorénavant totalement intégrée au standard du C++, son utilisation est sensée garantir la portabilité du logiciel. Je souhaite néanmoins modérer dès maintenant cette dernière affirmation. En effet, certaines versions de la STL ne sont pas tout à fait « au standard » et ne garantissent donc pas la portabilité.

Compréhensibilité et maintenabilité : la standardisation de la STL garantit que tout utilisateur sera capable d'appréhender favorablement du code reposant sur elle.

Efficacité (au sens classique ☺) : les composants ayant été écrits par des maîtres, il y a fort à parier que leur utilisation conduit à du code plus efficace que celui reposant sur des bibliothèques « maison »

Présenter de manière cohérente la STL est assez difficile. En effet, il est impossible de séparer l'étude des conteneurs de celle des itérateurs ou des algorithmes tant ces notions sont mêlées. Aussi, cela contraindra le lecteur à des aller retours fréquents dans le texte.

---

[1] En effet, il est bien connu que plus un composant est utilisé, plus il devient sûr

2. Les fondamentaux

2.1 Espaces de nommage

Cette section décrit l'une des plus récentes additions au standard du C++ : les espaces de nommage.

2.1.1 Motivation

L'une des notions qui manque assurément le plus au C++ est celle de *paquetage*. Ceci est d'autant plus désastreux que même l'ADA (et, plus récemment, Java), langage pourtant non orienté objet à l'origine l'a toujours proposé. En effet, il est désolant de ne pouvoir *rassembler* dans un élément du langage un ensemble de classes, de fonctions, de constantes, de types et de variables traitant du même *concept*.

En outre, un paquetage est habituellement *nommé* et fournit un niveau d'abstraction supplémentaire sur les noms des entités qu'il contient. En effet, pour accéder à une entité quelconque d'un paquetage, il convient de *préfixer* son nom par celui du paquetage ou bien d'utiliser une clause spéciale offrant la visibilité sur l'ensemble des déclarations du paquetage.

De surcroît, l'utilisation de tels préfixes permet de résoudre l'un des problèmes les plus courants lorsque l'on utilise différentes bibliothèques de composants logiciels : les *collisions* de noms. Les préfixes de noms permettent non seulement au programmeur d'éviter la collision mais également à l'éditeur de liens de s'y retrouver.

Les *espaces de nommage* du C++ pallient partiellement au manque de paquetage. En effet, ils permettent de créer – comme leur nom l'indique – un groupe d'entités rassemblées autour d'un *préfixe de nommage*.

2.1.2 Propriétés

Les espaces de nommage sont pourvus – entre autres – des propriétés remarquables suivantes :

Imbrication : il est possible d'imbriquer les espaces de nommage pour découper un concept en sous parties. On peut ainsi organiser les espaces de nommage du C++ à la manière de l'arborescence des paquetages de Java

Définition incrémentale : l'ensemble des déclarations et des définitions d'un espace de nommage peut s'étaler sur plusieurs fichiers source. En outre, tout utilisateur est autorisé à rajouter ses propres déclarations et définitions dans un espace de nommage auquel il a accès. Cette faculté pouvant alternativement être vue comme un avantage ou un inconvénient.

Pas de modification d'accès : contrairement à Java ou ADA, il n'y a pas de notion de visibilité de paquetage associé aux espaces de nommage. En effet, considérons 2 classes A et B appartenant au même espace de nommage. Si elles n'ont pas de relation privilégiée (héritage ou amitié), les méthodes de A

n'ont pas de visibilité particulière sur les membres de B et réciproquement. Ce dernier point ne peut que laisser un arrière goût amer dans la bouche tant il aurait été intéressant de combler cette lacune du C++. Peut-être dans la prochaine norme ?

2.1.3 Mise en œuvre des espaces de nommage

Les prochaines sections décrivent succinctement l'essentiel de la syntaxe liée aux espaces de nommage. Après avoir étudié la déclaration d'un espace de nommage, nous verrons comment accéder aux entités qu'il contient.

2.1.3.1 Déclaration et définition

La déclaration d'un espace de nommage se fait par l'utilisation du mot clef **namespace**. Comme d'habitude avec le C++, il est possible de faire une déclaration de type « forward » :

```
namespace identificateur;
```

La syntaxe de définition d'un espace reprend celle d'un bloc du langage :

```
namespace identificateur
{
  // liste des entités regroupées dans l'espace de nommage
};
```

L'exemple suivant montre un espace de nommage regroupant deux classes et une constante :

```
namespace azedzed
{
 class A
 {
   // définition de la classe A
 };

 const double PI=3.1415927;

  class B
  {
    // définition de la classe B
  };
}
```

Programme 2.1 déclaration et définition d'un espace de nommage

2.1.3.2 Accès aux membres

Tout d'abord, il faut savoir que chaque membre d'un même espace de nommage a une visibilité directe sur les identificateurs de tout l'espace. Dorénavant, nous ne nous intéresserons qu'à traiter les règles de visibilité depuis l'extérieur de l'espace.

2.1.3.2.1 Les noms qualifiés

La première manière d'accéder à un membre d'un espace de nommage consiste à préfixer son nom par celui du préfixe suivi de l'opérateur de résolution de portée **::** on parle alors de *nom qualifié*. Par exemple, pour déclarer un objet **obA** de la classe **ClasseA** appartenant à l'espace de nommage **EspNom1**, il faudra utiliser une déclaration de la forme :

```
EspNom1::ClasseA *obA;
```

Considérons désormais la classe **ClasseB** déclarée dans **EspNomInt** lui même imbriqué dans **EspNomExt**. Pour accéder à **ClasseB**, il faut spécifier tout le chemin de nommage, soit :

```
EspNomExt::EspNomInt::ClasseB *obB;
```

Un espace de nommage non imbriqué est dit global ou *top-level*.

2.1.3.2.2 La clause **using namespace**

Il est parfois fastidieux d'utiliser des noms qualifiés qui peuvent s'avérer très long. La clause **using namespace** permet d'obtenir la visibilité directe sur l'ensemble des entités déclarées dans un espace de nommage. La syntaxe en est la suivante :

```
using namespace identificateur;
```

Il est important de noter que dans le cas où des espaces de nommage sont imbriqués, la clause **using namespace** ne donne qu'un seul niveau de visibilité. En outre, l'utiliser peut de nouveau faire surgir des problèmes de collision de noms, auquel cas, le compilateur vous en avertira.

En effet, considérez le cas suivant :

```
namespace A
{
  class Erreur // ou tout autre nom courant
  {
  };
}

namespace B
{
  class Erreur // ou taut autre nom courant
  {
  };
}


class Erreur
{
};

int main(int,char**)
{
  Erreur e1;    // Pas de probleme quoique ::Erreur e1 eut ete preferable
```

```
  A::Erreur e2; // Pas de probleme classe Erreur de l'espace A
  B::Erreur e3; // Pas de probleme classe Erreur de l'espace B

  using namespace A;

  Erreur    e4; // Ambiguite entre ::Erreur et A::Erreur

  using namespace B;

  Erreur    e5;    // ambiguite entre ::Erreur, A::Erreur et B::Erreur

  return 0;
}
```

Programme 2.2 Exemple de collision entraînée par l'utilisation de la cause `using namespace`

Autre cas vicieux. Considérez trois espaces de nommage avec les relations indiquées par la figure suivante :

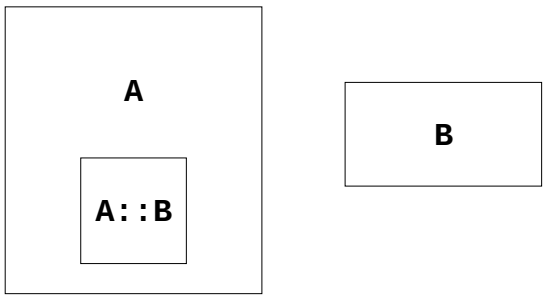

Figure 2.1 Espaces de nommage avec noms conflictuels

Il existe un espace de nommage **A**, un espace imbriqué **A::B** et un espace **B**. Considérez le fragment de code suivant :

```
using namespace A; // Ok, délivre la visibilité sur les éléments de A
using namespace B; // ??? est-ce A::B ou bien B?
```

Le résultat de la dernière est laissé à vérifier en exercice. Il était possible de lever l'ambiguïté en précisant, soit **using namespace A::B** soit **using namespace ::B** pour l'espace de nommage non imbriqué.

2.1.3.2.3 La clause **using identificateur**

Il est également possible d'utiliser la clause **using** (notez l'absence de **namespace** après **using**) sur un identificateur *unique* d'un espace de nommage pour le rendre accessible directement. Si un autre identificateur de même nom existait avant la clause, il est masqué par celui de la clause **using**, comme l'illustre l'exemple suivant :

```
#include <iostream>

namespace BASE
{
```

```
  class ErreurBase
  {
    public:
      virtual void show(void)=0;
  };
}

namespace A
{
  class Erreur : public BASE::ErreurBase
  {
    public:
      virtual void show(void)
      {
        cout << " A :: Erreur " << endl;
      }
  };
}


class Erreur : public BASE::ErreurBase
{
  public:
    virtual void show(void)
    {
      cout << "  :: Erreur " << endl;
    }
};

int main(int, char**)
{

  Erreur    e1; // ::Erreur
  A::Erreur e2; // A::Erreur

  using A::Erreur;
  Erreur    e3; // A::Erreur masque ::Erreur


  e1.show();
  e2.show();
  e3.show();

  return 0;
}
```

Programme 2.3 Exemple d'utilisation de la clause `using` **et des dangers inhérents**

2.1.3.2.4 Conclusion sur l'accès aux membres et alias d'espaces de nommage

Pour conclure, je ne conseille pas l'utilisation de la clause **`using`** sous quelque forme que ce soit car cela nuit considérablement à la clarté du code en supprimant l'un des avantages les plus importants de la notion d'espace de nommage : la non ambiguïté d'origine avec la possibilité de réintroduire des collisions de nommage.

Certains argueront probablement que les noms d'espace de nommage sont parfois très longs (notamment dans le cas d'imbrications multiples). Ce à quoi je répondrai par la possibilité de créer des noms *d'alias*. Voici la syntaxe :

```
namespace nomAlias = nomEspaceOriginel ;
```

Par exemple :

```
namespace local = EspaceExterne::EspaceInterne::EspacePlusInterne ;
```

ou encore :

```
#include <iostream>

namespace BASE
{
  class ErreurBase
  {
    public:
      virtual void show(void)=0;
  };
}

namespace NomTropLong
{
  class Erreur : public BASE::ErreurBase
  {
    public:
      virtual void show(void)
      {
        cout << "NomTropLong :: Erreur " << endl;
      }
  };

  namespace EncorePlusLong
  {
    class Erreur : public BASE::ErreurBase
    {
      public:
      virtual void show(void)
      {
        cout << "NomTropLong :: EncorePlusLong  :: Erreur " << endl;
      }
    };
  }
}

class Erreur : public BASE::ErreurBase
{
  public:
    virtual void show(void)
    {
      cout << "  :: Erreur " << endl;
    }
};

int main(int, char**)
{

  namespace A = NomTropLong;
  namespace B = A::EncorePlusLong;

  // Etant declares a l'interieur de main les noms d'alias
  // sont locaux et donc non exportes !

  Erreur                              e1; // ::Erreur
```

```
  NomTropLong::Erreur                  e2;
  NomTropLong::EncorePlusLong::Erreur e3;
  A::Erreur                            e4; // ni ambigue ni fastidieux !
  B::Erreur                            e5; // idem

  e1.show();
  e2.show();
  e3.show();
  e4.show();
  e5.show();

  return 0;
}
```

Programme 2.4 Exemple plus complet d'utilisation des alias

2.1.4 Un exemple plus complet

L'exemple de code suivant (incluant plusieurs fichiers source) illustre certaines fonctionnalités des espaces de nommage :

- Définition incrémentale
- Imbrication des espaces de nommage
- Utilisation des alias

```
#ifndef __NameRO_H__
#define __NameRO_H__

// Fichier des déclarations
// Déclare un espace de nommage externe (RO)
// et deux espaces de nommage internes RO::GRAPHES et
// RO::PROGRAMMATION_DYNAMIQUE
// Ainsi que des classes

namespace RO
{
  namespace GRAPHES
  {
    class Graphe
    {
      // code omis
      public:
        void afficher(void);
    };

    class AlgorithmePCC
    {
      // code omis
      public:
        void resoudre(void);
    };
  }

  namespace PROGRAMMATION_DYNAMIQUE
  {
    class SacADos
    {
      // code omis
      public:
```

```
        void resoudre(void);
    };
  }
}

#endif

// Deuxieme fichier : implemente l'une des classes
// Notez que l'on est oblige de recreer l'imbrication des namespace

#include "NameRO.hxx"

namespace RO
{
  namespace GRAPHES
  {
    void Graphe::afficher(void)
    {
    }
  }
}

// Troisieme fichier : suite de l'implementation des classes
// du package RO::GRAPHES
// Notez que l'on est oblige de recreer l'imbrication des namespace

#include "NameRO.hxx"

namespace RO
{
  namespace GRAPHES
  {
    void AlgorithmePCC::resoudre(void)
    {
    }
  }
}

// Quatrieme fichier : implementation de la classe
// du package RO::PROGRAMMATION_DYNAMIQUE
// Notez que l'on est oblige de recreer l'imbrication des namespace


#include "NameRO.hxx"

namespace RO
{
  namespace PROGRAMMATION_DYNAMIQUE
  {
    void SacADos::resoudre(void)
    {
    }
  }
}

// Dernier fichier : main utilisant des alias de namespace

#include "NameRO.hxx"

int main(int, char **)
{
  namespace GR=RO::GRAPHES;
  namespace DY=RO::PROGRAMMATION_DYNAMIQUE;

  GR::Graphe        g;
```

```
  GR::AlgorithmePCC a;
  DY::SacADos       s;

  g.afficher();
  a.resoudre();
  s.resoudre();

  return 0;
}
```

Programme 2.5 Exemple complet d'utilisation des espaces de nommage

2.1.5 La STL et les espaces de nommage

Hormis les noms *d'exceptions*, tous les identificateurs de la STL sont regroupés dans un espace nommé **std**. Ceci est particulièrement judicieux car l'on y retrouve des noms aussi courants que **vector, deque** ou **string**. Vous noterez que c'est un nom particulièrement court qui ne justifie aucunement l'utilisation de la clause **using namespace**.

L'utilisation de clauses type **#ifdef** permet néanmoins d'utiliser la STL sur les compilateurs non pourvus du support des espaces de nommage.

2.2 Quelques points à connaître

2.2.1 La portabilité de la STL

Si les fonctionnalités présentes dans la STL sont normalisées, toute la partie implémentation est laissée libre. Par exemple, le nom des fichiers d'entête peut légèrement varier d'une implémentation à l'autre, rendant nécessaire l'utilisation de clauses **#ifdef**. Toutefois, les différences sont toujours mineures et n'entraînent que très peu de modifications du code source.

Il est toutefois un point particulier, relatif à l'utilisation massive des **template** pour la description des classes conteneur, qu'il est nécessaire de souligner. En effet, de nombreuses classes utilisent des paramètres **template** avec une valeur par défaut, fonctionnalité non reconnue par de nombreux compilateurs C++ populaires, en particulier les compilateurs Borland et GNU.

2.2.2 Les fichiers d'entête utilisés par la STL

Selon les implémentations, les fichiers d'entête de la STL adoptent l'extension **.h** ou, aucune extension. En outre, par convention, l'on utilise jamais d'extension pour invoquer l'entête. En effet, l'utilisation de la classe `string`, par exemple, repose sur la directive **#include <string>** que les déclaration associées soient dans le fichier **string.h** ou le fichier **string**. Cette extension des compilateurs C++ s'avère bien pratique car elle permet de réaliser des inclusions sans connaître à l'avance la convention adoptée par le fournisseur de la STL que vous aurez à utiliser.

2.3 Les chaînes de caractères

La STL propose des fonctionnalités évoluées sur les chaînes de caractères. Celles-ci sont basées sur la classe **basic_string**, classe **template** paramétrable par le type de caractères contenus dans la chaîne. Deux instances correspondant respectivement aux types caractère 8 bits et caractère 16 bits sont proposées par défaut : **string** et **wstring**. Le fichier d'entête correspondant est habituellement nommé **string** ou **string.h**. La classe `string` s'obtenant par simple instanciation de **basic_string**, nous nous concentrerons sur son étude dans la suite de cet exposé.

Le nombre d'opérations fournies sur les chaînes est pour le moins impressionnant. Nous ne donnerons pas ici une liste exhaustive des possibilités mais seulement quelques opérations particulièrement utiles.

2.3.1 Construction de variables du type `string`, conversion avec `char *`

Il ne faut surtout pas oublier que le type **string** est une classe et donc, non totalement homomorphe à **char *** bien qu'il existe des opérations de conversion entre les deux.

Les opérations disponibles pour construire une variable de type `string` sont les suivantes :

```
string s1;                      // Chaîne vide s1
string s2("Clermont-Ferrand")   // Initialisation avec un char *
char   tab[]="Auvergne";
string s3(tab,4);               // s3=="Auve"
string s4(s2);                  // constructeur par recopie
string s5(s2,8);                // s5=="Ferrand"
string s6(s2,16);               // s6=="Clermont-Ferrand"
string s7(s2,18);               // s7=="Clermont-Ferrand"
string s8(4,'c');               // s8=="cccc"
```

Programme 2.6 Mini exemple de constructeurs de la classe `string`

A la lecture de cet exemple, plusieurs commentaires s'imposent :

- Il est possible de créer des **string** à partir des classiques tableaux de caractères en spécifiant, ou non, une taille maximale de recopie. Cette conversion peut même se faire de façon implicite avec certains compilateurs. Si le second paramètre dépasse la longueur de la chaîne, seuls les caractères avant le zéro sont copiés.

- Il n'y a pas de constructeur prenant simplement une taille. Le constructeur par défaut alloue (dans la plupart des implémentations) 20 caractères. Cet inconvénient est pallié par l'existence d'un constructeur créant une chaîne constituée de la répétition de **n** fois le même caractère. Il suffit de spécifier **'\0'** comme caractère de remplissage pour obtenir une chaîne vide avec capacité choisie par l'utilisateur.

- Le constructeur par recopie admet 2 paramètres supplémentaires permettant respectivement de spécifier :
  - La position de début de recopie – le premier caractère est à la position 0.
  - Le nombre de caractères à recopier. Si la recopie devait se faire après la fin de la chaîne, seuls les caractères réellement présents dans l'original sont effectivement recopiés.

La méthode de stockage effectif des chaînes de caractères peut varier d'une implémentation à l'autre, mais dans la plupart des cas, elle repose sur l'utilisation d'un tableau de caractères.

Il n'existe pas d'opérateur de conversion implicite depuis **`string`** vers **`char *`**, car cela permettrait d'effectuer des opérations malheureuses conduisant à des aberrations au niveau de la représentation mémoire. Au lieu de cela, la classe **`string`** possède deux méthodes équivalentes nommées **`c_str`** et **`data`** et renvoyant un **`const char *.`** En raison des particularités de stockage des chaînes de caractères sous certains implémentations, il est fortement déconseillé d'effectuer des modifications sur le tableau renvoyé après un **`cast`** supprimant son caractères constant.

La destruction des chaînes de caractères se fait via le destructeur de la classe **`basic_string`**.

2.3.2 Les opérations de routine

2.3.2.1 Accès aux caractères

En plus des méthodes **`c_str`** et **`data`**, et comme l'on peut légitimement s'y attendre il est tout à fait possible d'accéder aux différents caractères composant une chaîne à l'aide de l'opérateur d'indexation **`[]`**.

En outre, il existe la méthode **`at(int position)`** qui permet d'effectuer grosso modo la même opération que l'opérateur d'indexation. La principale différence tient à la gestion des dépassements de range.

En effet, l'opérateur **`[]`** renvoie une constante prédéfinie alors que **`at`** lance une exception de type **`range_error`**.

2.3.2.2 Longueur d'une chaîne

Il convient ici de bien faire la différence entre la longueur logique de la chaîne et sa capacité en caractères. La seconde est physique et rend compte de l'espace alloué dans le conteneur de la chaîne alors que la première témoigne du nombre de caractères réellement présents.

Pour établir un parallèle, en C la longueur logique serait renvoyée par **`strlen`** alors que la capacité serait le nombre de caractères fourni à **`malloc`** ou placé entre les crochets lors de l'allocation statique d'une chaîne

`size_t length()`	Renvoie la longueur logique d'une chaîne de caractères
`size_t size()`	Synonyme pour **`length`**
`size_t capacity()`	Capacité d'une chaîne
`size_t max_capacity()`	Capacité maximale d'une chaîne : renvoie un fait la quantité de mémoire disponible sur le système
`bool empty()`	Vrai si la chaîne ne contient aucun caractère assimilable au test **`(length==0)`**
`void reserve(size_t n)`	Porte la capacité de la chaîne (si possible !) à **`n`**.
`void resize(size_t n)`	Affecte à **`n`** la longueur de la chaîne Si (**`n < length`**), il y a troncature Si (**`length < n < capacity`**) il y a ajout d'espaces Si (**`n > capacity`**) la capacité est d'abord portée à **`n`** avant ajout d'espaces
`void resize(size_t n, char z)`	Comportement similaire à **`resize`** à la différence que le caractère de remplissage est spécifié

Tableau 2.1 méthodes agissant sur la taille des chaînes de caractères

2.3.2.3 Affectations, ajouts, échanges de caractères

La classe **`basic_string`** est équipée d'un opérateur d'affectation dont l'implémentation a tendance à varier en fonction du fournisseur. Il agit toujours par recopie du membre de droite vers le membre gauche mais, dans certains cas, il ne modifie la capacité de la chaîne cible que lorsque cela s'avère nécessaire.

A mon avis, de nombreux implémenteurs ont codé cet opérateur en utilisant des compteurs de référence et des recopies tardives : seul un pointeur est recopié lors de l'affectation, la structure de données sous-jacente n'est recopiée que si l'une des deux chaînes est modifiée. Cette astuce assure des gains de performance non négligeables dans de nombreuses situations et est encouragée par, excusez du peu, Scott Meyers et Steve Coplien … ce qui est bon pour les dieux doit bien l'être pour nous autres, pauvres mortels !

La classe **`basic_string`** dispose de méthodes très intéressantes et d'ordre très général permettant d'ajouter des séquences dans une chaîne. Nous n'en donnons ici qu'une description assez sommaire.

`append`	permet d'ajouter tout ou partie d'une chaîne argument au bout de la chaîne cible
`assign`	affecte tout ou partie de la chaîne argument à la chaîne cible
`insert`	insère tout ou partie d'une chaîne argument à une position spécifiée en

	paramètre dans la chaîne cible
swap	échange le contenu de la chaîne cible avec celui d'une chaîne passée en paramètre

Tableau 2.2 les méthodes générales d'ajout dans les séquences

2.4 Les exceptions de la STL

La STL fournit une panoplie d'exceptions très complète permettant d'établir des diagnostiques assez détaillés aux programmeurs. L'ensemble repose sur la classe de base **exception** et sur deux classes dérivées **logic_error** et **runtime_error**.

La classe **logic_error** et ses dérivées sont utilisées pour signaler des exceptions dues à la logique interne d'un programme, c'est à dire finalement à des problèmes de robustesse du logiciel. En général, de telles erreurs sont prévisibles et peuvent être facilement évitées. Elles sont donc surtout utilisées lors de la mise au point de programmes.

En revanche les exceptions issues de **runtime_error** notifient au programmeur (ou à l'utilisateur final si elles ne sont pas traitées) des erreurs liées à l'environnement du programme. Ces erreurs sont rarement évitables car parfois non prévisibles car elle témoignent de phénomènes divers tels que des problèmes de dépassement de précision inférieur ou supérieur ou des défaillances du matériel.

2.4.1 La classe exception

C'est la classe de base de toutes les exceptions. Elle est définie dans le fichier d'entête exception. Bien que ce ne soit pas une classe virtuelle pure, elle est pourvue de la méthode **what** dont le code doit être le suivant :

```
class exception
{

  // code omis
  virtual const char * what () const throw ()
  {
    return << pointeur vers une chaîne quelconque >> ;
  }
};
```

Programme 2.7 Déclaration de la classe std::exception

Cette méthode est clairement destinée à être redéfinie par les sous classes d'**exception** afin de fournir un message intelligible à l'utilisateur et le renseignant sur les causes de l'exception.

2.4.2 La hiérarchie des exceptions de la STL

La figure suivante indique les relations d'héritage entre les classes d'exception de la STL.

Les classes **runtime_error** et **logic_error** sont identiques dans leur déclaration : elles ne sont différenciées que dans le but de faire un traitement discriminant sur ces deux grandes catégories. A l'inverse de leur ancêtre commun, leur constructeur prend un **const string &** en paramètre (voir à ce sujet les sections concernant le type **string**) qui permet de rajouter un message explicitant, par exemple, les circonstances présidant à l'erreur.

Vous noterez au passage que les exceptions directement issues de **exception** utilisent un message sous forme de **const char *** alors que les erreurs « plus classieuses » utilisent des **const & string.**

Dans certaines implémentations de la STL, l'inclusion de **<exception>** n'est pas suffisante. Il est également nécessaire d'inclure un autre fichier d'entêtes nommé **<stdexcept>.**

Ces différentes classes peuvent, bien entendu, être dérivées afin de créer de nouvelles exceptions adaptées à chaque cas.

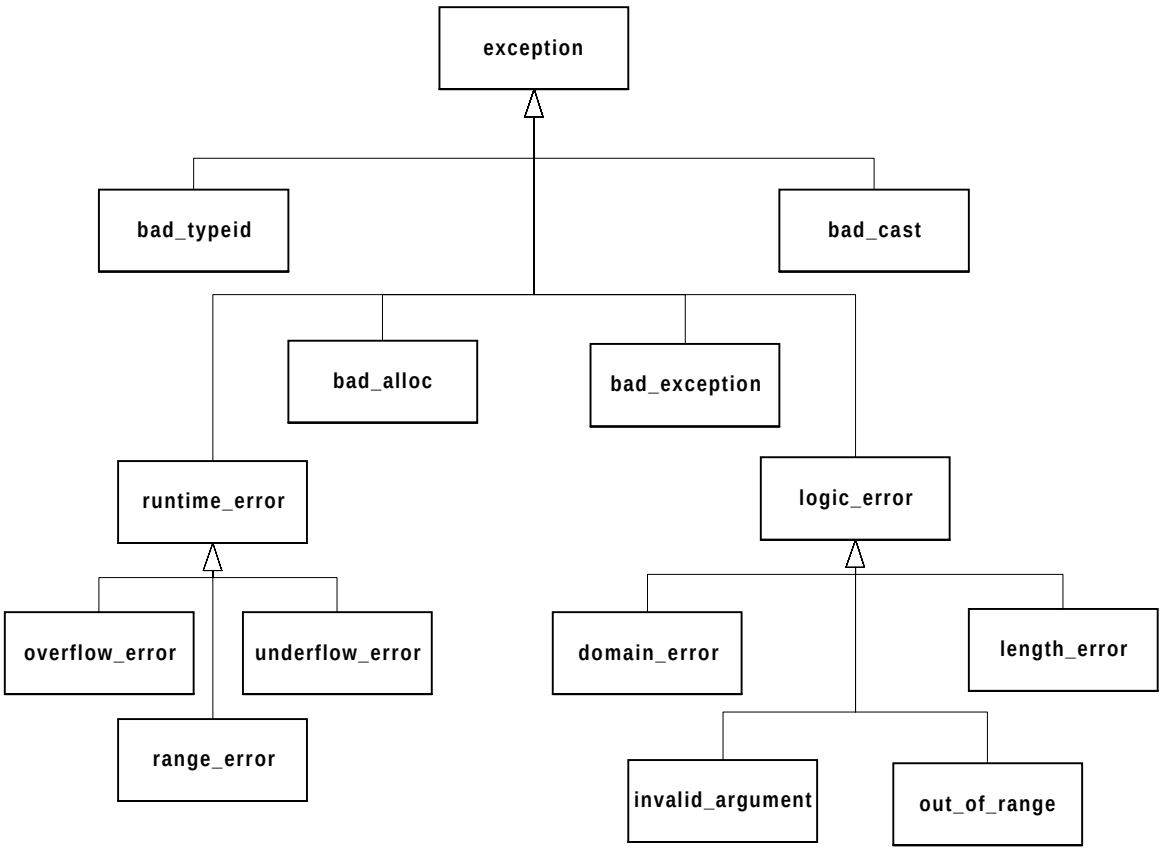

Figure 2.2 Hiérarchie des exceptions de la STL

Le tableau suivant reprend ces diverses exceptions en explicitant leur utilisation.

Exceptions directement issues de **exception**

bad_exception	Dénote un problème de spécification d'exception. Typiquement, une méthode ou une fonction a lancé une exception qui n'était pas prévue dans sa déclaration. En théorie le compilateur aurait du s'en apercevoir … Autre cas : **bad_exception** est lancée par la fonction standard **unexpected**.
bad_alloc	Déclarée dans le fichier d'entête new, l'exception **bad_alloc** rapporte un problème d'allocation survenu lors de l'exécution de **operator new** ou **operator [] new**.
bad_cast **bad_typeid**	Déclarées dans le fichier d'entête **typeinfo**, ces exceptions sont respectivement levées lors d'une mauvaise utilisation de **dynamic_cast** ou la recherche de **typeid(*NULL).** A ce dernier sujet, on consultera avec profit la section consacrée à l'étude du RTTI.
Exceptions issues de **runtime_error**	
overflow_error **underflow_error**	Dénotent des problèmes de précision dans les calculs arithmétiques
range_error	A comparer à l'exception de logique de programme **out_of_range**, **range_error** stipule plus une erreur interne au calcul d'un argument qu'une erreur liée au programmeur.
Exceptions issues de **logic_error**	
length_error	A rapprocher de **bad_alloc**, cette exception indique une tentative de création d'un objet de trop grande taille
out_of_range	Typiquement un accès en dehors des limites d'index d'un tableau
domain_error **invalid_argument**	Très semblables, ces exceptions traduisent le passage d'un argument de type ou de domaine inattendu.

Tableau 2.3 Spécificités des différentes exceptions de la STL

2.4.3 Exemple d'utilisation des exceptions de la STL

L'exemple suivant montre l'utilisation de certaines des exceptions de la STL. Les commentaires sont placés dans le code.

```
#include <stdexcept>
#include <iostream>
#include <string>
#include <new>
#include <typeinfo>

class Rationnel
{
```

```
  protected:
    int num;
    int den;
  public:
    explicit Rationnel (int n=0,int d=1) : num(n), den(d) {};
    void setNum(int n)
    {
      num=n;
    }
    void setDen(int d) throw (std::invalid_argument)
    {
      if (d == 0)
        throw std::invalid_argument("Rationnel : passage de denominateur
nul");
     den=d;
    }
    double asDouble () throw (std::range_error)
    {
      if (den == 0)
        throw std::range_error("Rationnel : division par 0");
      return static_cast<double>(num)/den;
    }
};


int main(int, char**)
{
  const unsigned long  exageremenGrand=10000000;
  int                  nouveauDen;
  Rationnel            r1;
  Rationnel           *poolRationnel;


  try
  {
      Rationnel r2(2,0);
      cout << r2.asDouble();
  }
  catch (std::range_error &e)
  {
    cerr << "Exception range_error levee : " << e.what() << endl;
  }
  catch (std::exception &e)
  {
    cerr << "Exception stl levee : " << e.what() << endl;
  }
  catch (...)
  {
    cerr << "Exception inconnue levee : abandon du programme" << endl;
    exit(1);
  }


  do
  {
    cout << "Saisissez un nouveau denominateur" << endl << flush;
    cin >> nouveauDen;
    try
    {
      r1.setDen(nouveauDen);
    }
    catch (std::invalid_argument)
    {
      cerr << "Le denominateur ne doit pas etre nul, recommencez !" <<
endl;
```

```
      continue;
    }
    break;
  } while (1);

  try
  {
    poolRationnel = new Rationnel[exagerementGrand];
  }
  catch (std::bad_alloc &e)
  {
    cerr << "Vous avez demande trop de memoire !" << endl;
    poolRationnel=0;
  }
  catch (...)
  {
    cerr << "Erreur inconnue !"<< endl;
    poolRationnel=0;
  }
  delete [] poolRationnel;

  return 0;
}
```

Programme 2.8 exemple complet d'utilisation des exceptions de la STL

```
1) Exception range_error levee : Rationnel : division par 0
Saisissez un nouveau denominateur
0
2) Le denominateur ne doit pas etre nul, recommencez !
Saisissez un nouveau denominateur
1
3) Vous avez demande trop de memoire !
```

Figure 2.3 Sortie écran du programme précédent

Commentons quelque peu les résultats de ce programme :

1. La création d'un `Rationnel` avec dénominateur nul lève une exception de type `range_error` : tout ce qu'il y a de plus normal ! Vous noterez dans le code que nous sommes allés du gestionnaire le plus restrictif au plus général.

2. Nous utilisons ici une exception à l'intérieur d'une boucle. Tant que l'exception est levée, on ne peut pas sortir de la boucle. C'est un système de programmation assez général.

3. L'exception `bad_alloc` sanctionne un `new` qui ne s'est pas terminé correctement.

   Vous allez me dire « il est toujours possible de vérifier si le pointeur demandé n'est pas nul, il n'était pas nécessaire d'utiliser un mécanisme aussi lourd qu'une exception pour cela ». Exact. Cependant, combien d'entre vous le font systématiquement ? D'accord, les exceptions ralentissent un programme et sont lourdes à utiliser, toutefois, elles ne peuvent être ignorées et permettent parfois de garantir le fonctionnement normal d'un programme.

2.4.4 Aux utilisateurs de C++ Builder

Inprise propose aux utilisateurs de C++ Builder sa propre bibliothèque d'exceptions au sein de la VCL (Visual Components Library) dont la philosophie est totalement différentes : les exceptions de la STL sont d'abord destinées à être levées par l'utilisateur, celles de la VCL sont utilisées par les composants pour signaler les erreurs à l'utilisateur.

Notons également que les arborescences des exceptions sont totalement différentes. En effet, celle de la VCL rajoute à des exceptions très générales tout ce qui est axé sur la programmation événementielle. Essayer d'utiliser conjointement les deux types d'exception n'est pas exclu mais peut s'avérer franchement délicat, il faut, par exemple, retenir que l'accès en dehors des bornes d'un vecteur de la STL génère l'exception **`std::out_of_range`** alors que l'accès en dehors des bornes d'un vecteur de la VCL (tel que, par exemple, celui des éléments d'une boîte de liste) se traduit par une exceptions **`ErangeError`** et qu'il sera souvent délicat d'établir un gestionnaire commun.

Un autre point différenciant les exceptions VCL et STL réside dans le type des chaînes de caractères impliquées. En effet, dans la STL, c'est, bien entendu, le type **`std::string`** qui est utilisé alors que la VCL privilégie le type natif de Windows : **`AnsiString`**.

2.5 Le Support de la RTTI

Avant toute chose précisons tout de suite que les comités de normalisation n'ont pas encore décidé si le RTTI faisait partie du langage C++ lui même ou bien de sa librairie standard.

Sans vouloir entrer dans cette polémique – de faible intérêt – je me contenterai de présenter ici le RTTI sans me soucier de savoir si c'est un composant de librairie ou bien un élément du langage ☺.

2.5.1 Qu'est ce que le RTTI

RTTI signifie littéralement *Run Time Type Information* soit en Français : Informations de Type disponibles à l'Exécution. Ceci peut sembler incongru dans un langage orienté objet dont le but est précisément de se délester sur les fonctions virtuelles des problèmes liés aux différents types des objets recensés dans un programme. D'ailleurs Coplien écrivait : « il est absolument impensable que les programmeurs puissent savoir au cours de l'exécution à quels types d'objets ils ont affaire ».

Aussi, les informations de type ne seront utilisées qu'à des fins de diagnostic ou pour éviter des erreurs de logique dans les programmes. En outre, ces informations vont permettre d'utiliser en toute sécurité l'une des fonctionnalités les plus décriées de la programmation orientée objet : les promotions de pointeurs sur des objets également appelées downcast en permettant de vérifier que l'opération que l'on va effectuer est bien légitime.

2.5.2 La classe `type_info` et l'opérateur `typeid`

Le support du RTTI est indissociable de la classe **`type_info`**, qui, comme son nom le suggère est dédiée à l'exploitation des informations de type contenues dans les classes. L'utilisation de cette classe nécessite l'inclusion de `typeinfo`.

La classe **`type_info`** possède la particularité de ne pas posséder de constructeur public. En effet, il existe une et une seule instance de **`type_info`** pour chaque classe présente dans un exécutable. Ces instances sont générées lors du lancement de l'exécutable, et le seul moyen d'obtenir une référence (et encore constante !) passe par l'utilisation de l'opérateur **`typeid`**. Celui-ci admet deux syntaxes :

```
const type_info &typeid( expression )
const type_info &typeid( nom-type )
```

Dans le premier cas, on obtient des informations sur le type d'une expression, par exemple, une variable, une constante ou, plus généralement, toute expression dotée d'un type.

Dans le second cas, on demande une information de type en passant à **`typeid`** le nom du type. Ceci est destiné à établir des comparaisons entre le type d'une expression et une série de références.

Les méthodes les plus utiles de la classe **`type_info`** sont les suivantes :

`operator==` bizarrement codé comme une méthode, cet opérateur teste l'égalité entre deux informations de type.

`operator!=` également (et toujours aussi bizarrement ☺) codé comme une fonction membre, cet opérateur teste la différence entre deux informations de type. Il paraît redondant d'avec le précédent et n'est pas généré automatiquement comme **`!operator==`**.

`before` est assurément l'opérateur (booléen) le plus intéressant, en effet **`a.before(b)`** renvoie **`true`** si et seulement si **`a`** correspond à une super classe de **`b`**. C'est précisément cette information qui est utilisée par **`dynamic_cast`** pour vérifier l'intégrité d'un **`downcast`**.

`name` renvoie une chaîne de caractères (de type **`char *`** ☺) indiquant en clair le nom de la classe associée au **`type_info`**.

2.5.3 Exemple d'utilisation du RTTI

L'exemple suivant créée des classes, les instancie et teste l'opérateur `==` ainsi que les méthode `before` et `name` du RTTI.

Important ! A l'heure où ces lignes sont tapées, `name` fonctionne correctement avec C++ Builder et `before` avec g++ mais pas l'inverse ! Espérons que tout ceci soit

bien vite rectifié ! Le résultat écran fourni est donc une compilation idyllique des résultats fournis dans ces deux environnements.

```
#include <iostream>
#include <typeinfo>

class A
{
  public:
    A() {}; // ce constructeur par défaut est nécessaire
            // sinon l'on obtient une erreur uninitialized constant
            // pour la variable aa dans main
};
class B {};
class C : public A {};
class D : public A {};
class E : public C {};

int main(int, char **)
{
  A         a;
  const A   aa;
  A        *pa = new A;
  B         b;
  C         c;
  D         d;
  E         e;

  cout << "Affichage du nom d'une classe "
       << typeid(A).name() << endl;
  cout << "Affichage du nom de la classe d'un objet "
       << typeid(a).name() << endl;
  cout << "Affichage du nom de la classe d'un pointeur "
       << typeid(pa).name() << endl;
  cout << "Affichage du nom de la classe d'une variable via pointeur "
       << typeid(*pa).name() << endl;

  cout << "Arithmetique sur les types " << endl;

  cout << "Variable et *pointeur de meme type "
       << (typeid(a) == typeid(*pa)) << endl;
  cout << "Variable et const Variable de meme type "
       << (typeid(a) == typeid(aa)) << endl;

  cout << "Variables de types differents "
       << (typeid(a) == typeid(b)) << endl;

  cout << "Essai de before " << endl;

  cout << "Variables de classes differentes "
       << typeid(a).before(typeid(b)) << endl;
  cout << "Classe meme et classe Fille "
       << typeid(a).before(typeid(c)) << endl;
  cout << "Classe grand meme et classe petite fille "
       << typeid(a).before(typeid(e)) << endl;
  cout << "Classe tante et classe niece "
       << typeid(d).before(typeid(e)) << endl;

  return 0;
}
```

Programme 2.9 utilisation des `type_info` **et de** `typeid`

```
Affichage du nom d'une classe A
Affichage du nom de la classe d'un objet A
Affichage du nom de la classe d'un pointeur A *
Affichage du nom de la classe d'une variable via pointeur A
Arithmetique sur les types
Variable et *pointeur de meme type 1
Variable et const Variable de meme type 1
Variables de types differents 0
Essai de before
Variables de classes differentes 0
Classe meme et classe Fille 1
Classe grand meme et classe petite fille 1
Classe tante et classe niece 0
```

Figure 2.4 Résultat écran du programme précédent

Quelques commentaires :

- Il n'y a pas grand chose à dire sur le résultat de `name` sinon que, lorsque cela fonctionne, c'est assez génial !
- Les résultats fournis par == sont conformes à ce que l'on pourrait attendre à une surprise près : `const A` et `A` sont ici considérés comme des types identiques.
- La sortie écran de `before` est la plus intéressante et donne elle aussi des résultats sans surprise :
  - Bien entendu la comparaison de deux classes sans parenté renvoie faux
  - Une classe grand mère est bien annoncée comme ancêtre de sa petite fille
  - Une classe tante n'est pas reconnue comme ancêtre de sa nièce

3. Les classes conteneur de la STL

De toutes les fonctionnalités présentes dans la STL, la plus connue (et celle qui a assuré sa popularité) repose sur les classes conteneur. Celles ci fournissent des implémentations de grande qualité pour les structures de données les plus courantes ainsi que des algorithmes génériques permettant de travailler avec elles.

3.1 Philosophie générale

En guise de préambule, il est très important de noter que la STL est issue de concepts non orientés objet. Pour être précis, ces précurseurs ont vus le jour en Ada 83. C'est pourquoi sa structure peut paraître hérétique à des gens pénétrés de culture Objet.

En effet, l'architecture de la STL prône une séparation très forte entre la notion de conteneur et celle d'algorithme. A l'opposé des concepts habituellement retenus, les algorithmes classiques (tri, échange de données, recopie de séquences) ne sont pas des méthodes des conteneurs mais des fonctions externes ou des objets foncteurs qui interagissent avec les conteneurs via les itérateurs, ces derniers n'étant ni plus ni moins que des objets permettant de baliser et / ou de parcourir des conteneurs.

3.2 Présentation des conteneurs

D'après la littérature proposée sur la STL, on peut ranger les conteneurs en trois catégories :

Les séquences élémentaires (quelquefois appelés stockages élémentaires)

Les utilisations spécialisées des séquences élémentaires (parfois et improprement appelées type abstrait de données)

Les conteneurs associatifs basés sur la notion de paire (clef, valeur)

Revenons brièvement sur chacun d'eux.

3.2.1 Présentation des séquences élémentaires

Les séquences élémentaires sont les structures de données élémentaires retenues par le comité de standardisation de la STL pour servir de base à des structures de données plus élaborées. Remarquons toutefois qu'elles sont néanmoins disponibles à l'utilisateur final !

On y retrouve : les vecteurs, les listes, les listes à accès préférentiel premier dernier ou DQ (deques en rosbif ☺).

3.2.1.1 Les vecteurs

Les vecteurs de la STL sont tout à fait semblables aux tableaux que l'on a l'habitude de manipuler en C++. Ils sont néanmoins encapsulés dans des classes pour leur fournir des capacités supplémentaires, en particulier, la croissance dynamique.

Leurs principales caractéristiques sont les suivantes :

- Accès indexé aux éléments en O(1)
- Ajout ou suppression d'un élément à la fin du vecteur sans redimensionnement en O(1)
- Ajout ou suppression d'un élément à la fin du vecteur avec redimensionnement en O(n)

  ☞ Ajout ou suppression d'un élément à la fin du vecteur en O(1) amorti
- Ajout ou suppression d'un élément au milieu du vecteur en O(n) où n est la taille du vecteur

Il convient de mettre un bémol concernant l'ajout d'un élément à la fin d'un vecteur. En effet, les vecteurs sont à croissance dynamique. Autrement dit, si vous décidez d'ajouter un élément dans un vecteur déjà plein à l'aide de la méthode **`push_back`** (détaillée plus bas), celui-ci sera agrandi, ce qui nécessite trois opérations :

1. Allocation d'un nouveau vecteur
2. Recopie des éléments depuis l'ancien vecteur vers le nouveau
3. Suppression de l'ancien vecteur

Or, ces opérations peuvent être coûteuses, au pire en O(n). Se pose également la question du choix de la future taille. Andrew Koenig nous explique que la technique la plus efficace consiste à utiliser un redimensionnement exponentiel *i.e.* à chaque réallocation, la taille du vecteur est multipliée par un facteur $(1 + \alpha)$ où α est un nombre strictement positif. Ainsi, plus le vecteur devient grand, plus il s'agrandira rapidement.

Cette stratégie a l'avantage de rendre les redimensionnement de moins en moins fréquents au détriment d'un encombrement parfois exagéré de la mémoire.

Du fait de la fréquence faible des redimensionnements, on peut considérer que la complexité asymptotique (ou amortie) de l'opération d'ajout d'un élément en bout de tableau est en O(1).

Notons immédiatement que l'accès, que ce soit en lecture ou en écriture, d'un élément en dehors des bornes actuelles d'un tableau se traduit immédiatement par la levée d'une exception **`out_of_range`** (voir la Figure 2.2 pour une vue d'ensemble des exceptions de la STL).

3.2.1.2 Les listes

Elles modélisent le type liste chaînée habituel. Contrairement aux vecteurs, elle ne possèdent pas d'opérateur d'indexation mais permettent d'ajouter ou de supprimer un élément à n'importe quelle position en O(1).

Bien qu'il n'y ait jamais de place morte dans une liste mais, du fait de la structure de chaîne, l'occupation en mémoire d'une liste est le plus souvent supérieure à celle d'un vecteur pour le même nombre d'éléments stockés.

3.2.1.3 Les DQ

Les DQ sont la séquence élémentaire la plus délicate à cerner car Intermédiaire entre une liste et un vecteur.

- Elle est très efficace dès lors qu'il s'agit d'ajouter ou de retirer une valeur à l'une de ses extrémités.
- Elle possède un opérateur d'indexation de complexité o(log(n)) amortie
- Les insertions en milieu de séquence sont plus rapides que sur un vecteur sans pour autant atteindre le niveau de performances des listes.

3.2.2 Les opérations disponibles sur l'ensemble des séquences élémentaires

Les méthodes suivants sont disponibles sur l'ensemble des conteneurs de base de la STL.

Gestion de la taille du conteneur	
`size_type size () const;`	Renvoie le nombre d'éléments présents dans le conteneur
`size_type max_size () const;`	Renvoie le nombre maximal d'éléments que peut receler un conteneur.
`bool empty () const;`	Renvoie **`true`** si le conteneur est vide
`void resize (size_type,T c = T())`	Redimensionne le conteneur. Si celui-ci est raccourci, les éléments terminaux sont supprimés sans autre forme de procès. En revanche, si le conteneur est agrandi, de nouveaux éléments sont ajoutés à la fin conformément au second paramètre
Accès aux éléments	
`reference front ();` `const_reference front () const;`	Renvoie une référence sur le premier élément. Existe en version **`const`** et non **`const`**
`reference back ();` `const_reference back () const;`	Renvoie une référence sur le dernier élément. Existe en version **`const`** et non **`const`**

Insertion d'éléments	
`void push_back (const T&);`	Ajouter l'élément spécifié par référence au bout de la collection. Appelle pour cela le constructeur par recopie
`void pop_back ();`	Retire l'élément en fin de collection

`iterator insert (iterator, const T& = T());`
 	
`void insert (iterator, size_type, const T& = T());`
	Insère un élément (éventuellement construit par le constructeur par défaut) à l'emplacement spécifié par l'itérateur passé en premier argument
	La deuxième version permet d'ajouter un facteur de répétition
	Seule la liste garantit un fonctionnement uniforme de ces méthodes en o(1)

`template <class InputIterator>` `void insert (iterator position, InputIterator debut, InputIterator fin);`	Insère à la position `position` les éléments contenus dans l'intervalle `[debut fin[`. Les itérateurs `debut` et `fin` doivent se trouver sur la même collection. Seule la liste garantit un fonctionnement uniforme de ces méthodes en o(1)
Suppression d'éléments	
`void erase (iterator);`	Supprime l'élément placé sous l'itérateur
`void erase (iterator debut, iterator fin);`	Supprime les éléments placés dans l'intervalle `[debut fin[`
Echange d'éléments	
`void swap (séquence<T>&);`	Où séquence est à remplacer par `vector`, `list` ou `deque` ! Echange les éléments de **`this`** avec ceux du conteneur passé en paramètre Le codage « méthode » ne fonctionne que sur des séquences de type rigoureusement identique. Pour échanger les éléments de séquences différentes, il faut recourir à la version « fonction » orientée itérateurs

Tableau 3.1 Les opérations disponibles sur toutes les séquences élémentaires

3.2.3 Opérations spécifiques

3.2.3.1 Opérations spécifiques sur les vecteurs

Gestion de la capacité	
`size_type capacity () const;`	Renvoie la capacité c'est à dire le nombre

	d'éléments que peut contenir le vecteur avant redimensionnement
`void reserve (size_type);`	Ajoute de la capacité à un vecteur
Accès indexé aux éléments	

`Reference operator[] (size_type);`
`const_reference operator[] (size_type) const;`
`Reference at (size_type n);`
`const_reference at (size_type n) const;`
	Opérateur d'indexation, existe en version **`const`** et non **`const`**.
	Renvoyer une référence permet d'utiliser le retour en tant que *lvalue* pour la version non **`const`**.
	La méthode `at` effectue exactement la même opération que l'opérateur crochets

Tableau 3.2 Méthodes spécifiques à `vector`

En outre, il faut savoir que si vous créez un vecteur avec une dimension initiale supérieure à 0, le constructeur par défaut du type élément est appelé pour créer un premier élément et le tableau est ensuite rempli par des appels au constructeur de recopie.

Afin de bien comprendre les mécanismes de création d'objets, que ce soit par le constructeur par défaut ou par appel au constructeur de recopie, nous utilisons la classe suivante qui encapsule les entiers :

```
// Fichier d'entete Entier.hxx
#ifndef __Entier__
#define __Entier__

class Entier
{
  private:
    static unsigned NbEntiers_;
    static unsigned NbRecopies_;

    int             val_;
  public:
    explicit Entier(int valeur=0) : val_(valeur)
    {
      NbEntiers_++;
    }

    Entier (const Entier &a)
    {
      val_=a.val_;
      NbRecopies_++;
    }

    int val(void) const
    {
      return val_;
    }

    static int NbEntiers(void)
    {
      return NbEntiers_;
    }
```

```
    static int NbRecopies(void)
    {
      return NbRecopies_;
    }

};

class ostream;

ostream &operator<<(ostream &a,const Entier &b);
bool operator==(const Entier &a, const Entier &b);
bool operator<(const Entier &a, const Entier &b);

#endif

// fichier Entier.cpp

#include "Entier.hxx"
#include <iostream>

ostream &operator<<(ostream &a,const Entier &b)
{
  a << b.val();
  return a;
}

bool operator<(const Entier &a, const Entier &b)
{
  return a.val() < b.val();
}

bool operator==(const Entier &a, const Entier &b)
{
  return a.val() == b.val();
}

unsigned int Entier::NbEntiers_=0;
unsigned int Entier::NbRecopies_=0;
```

Programme 3.1 La classe `Entier`

Le point intéressant réside dans l'utilisation des attributs de classe `NbEntiers_` et `NbRecopies_` comptant respectivement le nombre d'appels au constructeur « de création » et le nombre d'appels au constructeur de recopie. Nous pourrons donc suivre à la trace ces opérations.

En outre, cela nous permet de voir les pré requis nécessaires à l'utilisation d'une classe par la STL :

- Constructeur par défaut
- Constructeur par recopie. Ici, le constructeur par recopie généré automatiquement était suffisant, nous ne l'avons construit nous même que pour utiliser le comptage des appels
- Opérateur d'affectation, ici celui fourni automatiquement par le compilateur
- Opérateur de sortie sur flux **<<**

- Opérateur de comparaisons **<** et **==**, les autres étant automatiquement construits par **template**.

Notez également que le constructeur est déclaré **explicit** pour éviter toute conversion implicite avec un entier. Ici cela n'était toutefois pas dangereux. De même nous aurions très bien pu définir un opérateur de conversion implicite vers les entiers, sous la forme :

```
operator int (void)
{
  return val_;
}
```

Toutefois, il me paraît préférable de laisser le soin à l'utilisateur de réaliser les conversions de façon explicite lorsqu'il en a lui même besoin !

Il est important de noter que le redimensionnement du tableau – que ce soit par un appel direct à `reserve` où automatiquement lorsque l'on arrive en limite de capacité n'implique pas de création d'objet mais uniquement de la réservation de mémoire. En outre, ces opérations n'affectent que la capacité et en aucun cas la taille.

En revanche, `resize`, en mode accroissement, conjugue réservation de mémoire et création d'objets comme en témoigne les augmentations de taille (`size`)

```
#include <vector>
#include <iostream>
#include "Entier.hxx"

// Permet d'afficher les caracs du vecteur
// et le nombre d'entiers
template <class T>
void affichage(const T &a)
{
  static int numeroLigne=1;
  cout << '(' << numeroLigne++ << ") Capacite " << a.capacity();
  cout << " Taille " << a.size();
  cout << " Nb entiers " << Entier::NbEntiers() << endl << flush;
}

int main(int, char**)
{
  typedef std::vector<Entier> VecInt;
  VecInt  v1;               // Creation d'un vecteur vide

  affichage(v1);           // (1)
  v1.push_back(Entier(1)); // insertion d'un element
  affichage(v1);           // (2)
  v1.reserve(100);         // la capacite est portee a 100
  affichage(v1);           // (3)
  v1.resize(200);          // capacite et taille portees a 200
  affichage(v1);           // (4)

  VecInt v2;               // Nouveau vecteur vide

  for (int i=0;i<20;i++)   // Remplissage avec redimensionnements
  {
    v2.push_back(Entier(i));
    affichage(v2); // (5 -> 24)
  }
```

```
  for (int i=0;i<20;i++)   // Affichage par indexation (25)
    cout << v2[i] << " ";
  cout << endl;

  // Affichage avec parcours d'iterateur (26)
  VecInt::iterator courant=v2.begin();
  while (courant != v2.end())
  {
    cout << *courant << " ";
    ++ courant; // Je me souviens du cours de C++ avance
                // utiliser si possible l'operateur prefixe !
  }
  cout << endl;

  // Affichage avec iterateur de flux (27)
  copy(v2.begin(),
       v2.end(),
       ostream_iterator<Entier>(cout," "));
  cout << endl;

  return 0;
}
```

Programme 3.2 Manipulations sur la taille des vecteurs

Le résultat d'exécution est le suivant :

```
(1) Capacite 0 Taille 0 Nb entiers 0 Nb recopies 0
(2) Capacite 1 Taille 1 Nb entiers 1 Nb recopies 1
(3) Capacite 100 Taille 1 Nb entiers 1 Nb recopies 2
(4) Capacite 200 Taille 200 Nb entiers 2 Nb recopies 202
(5) Capacite 1 Taille 1 Nb entiers 3 Nb recopies 203
(6) Capacite 2 Taille 2 Nb entiers 4 Nb recopies 205
(7) Capacite 4 Taille 3 Nb entiers 5 Nb recopies 208
(8) Capacite 4 Taille 4 Nb entiers 6 Nb recopies 209
(9) Capacite 8 Taille 5 Nb entiers 7 Nb recopies 214
(10) Capacite 8 Taille 6 Nb entiers 8 Nb recopies 215
(11) Capacite 8 Taille 7 Nb entiers 9 Nb recopies 216
(12) Capacite 8 Taille 8 Nb entiers 10 Nb recopies 217
(13) Capacite 16 Taille 9 Nb entiers 11 Nb recopies 226
(14) Capacite 16 Taille 10 Nb entiers 12 Nb recopies 227
(15) Capacite 16 Taille 11 Nb entiers 13 Nb recopies 228
(16) Capacite 16 Taille 12 Nb entiers 14 Nb recopies 229
(17) Capacite 16 Taille 13 Nb entiers 15 Nb recopies 230
(18) Capacite 16 Taille 14 Nb entiers 16 Nb recopies 231
(19) Capacite 16 Taille 15 Nb entiers 17 Nb recopies 232
(20) Capacite 16 Taille 16 Nb entiers 18 Nb recopies 233
(21) Capacite 32 Taille 17 Nb entiers 19 Nb recopies 250
(22) Capacite 32 Taille 18 Nb entiers 20 Nb recopies 251
(23) Capacite 32 Taille 19 Nb entiers 21 Nb recopies 252
(24) Capacite 32 Taille 20 Nb entiers 22 Nb recopies 253
(25) 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
(26) 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
(27) 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Figure 3.1 Résultat du programme d'essai sur vector

Commentons une par une ces quelques lignes de sortie. Les commentaires du code font référence aux lignes de sortie.

(1) Le tableau est vide, sa capacité et sa taille sont nulles : prévisible

(2) On ajoute un élément, la capacité et la taille passent à 1, et l'on a alloué un entier : toujours prévisible.

Le nombre de recopies augmente d'une unité : ceci signifie que l'opération de recopie de l'ancien vecteur lors d'un redimensionnement se fait par appel à l'opérateur de construction par recopie.

(3) La capacité passe à 100 alors que la taille ne bouge pas, c'est typique du fonctionnement de `reserve`. Le fait que le nombre de créations d'entier n'ait pas bougé atteste du fait que `reserve` se contente d'allouer de la mémoire sans créer un seul objet. Le nombre de recopies augmente d'une unité pour la raison évoquée au point précédent.

(4) La capacité passe à 200 ainsi que la taille, c'est le rôle de `resize` : amener la taille à la valeur spécifiée en augmentant si nécessaire la capacité. En revanche, le point intéressant concerne le nombre d'entiers : un seul nouvel entier a été créé par le constructeur par défaut, les autres ont été générés par recopie comme l'atteste la brusque montée du nombre de recopies !

(5) Les lignes 5 à 24 permettent de bien voir le mécanisme de redimensionnement à partir d'un tableau vide. Dans ce cas (gcc), le redimensionnement est double à chaque étape. Eussions nous utilisé la *Rogue Wave* que la capacité serait passée de 0 à 256 avant d'être doublée à chaque étape.

Le nombre de recopies atteste que ce mécanisme est utilisé :

- D'une part lors des redimensionnements (voir point 2)
- Par chaque opération `push_back` : l'élément passé à `push_back` (par référence constante pour éviter une recopie de plus) est recopié à son emplacement dans le vecteur.

(6) Finalement, nous procédons à l'affichage des valeurs contenues dans le vecteur par trois méthodes différentes :

- Ligne 25 : accès avec l'opérateur d'indexation (méthode classique)
- Ligne 26 : parcours avec itérateurs
- Ligne 27 : utilisation d'un itérateur de flux
- Vous remarquerez que nous obtenons trois fois de suite la même solution, ce qui est très rassurant !

3.2.3.2 Opérations spécifiques sur les listes

Les opérations spécifiques sur les listes concernent essentiellement l'insertion et la suppression en tête de liste, le transfert d'éléments entre listes avec une méthode spécifique (`splice`), ainsi que le réordonnement de la collection.

Insertions et retraits en têtes de liste	
`void push_front (const T& x);`	Insère un élément en tête de liste
`void pop_front ();`	Retire l'élément en tête de liste
Transfert d'éléments entre listes	
Permettez moi un petit blah pipo (merci Pookie ☺) avant d'entrer dans le lard (oups, on dit le vif…) du sujet Toutes ces opérations sont réalisées en O(1) grâce à la structure chaînée spécifique de la liste. Elles sont donc à préférer aux algorithmes génériques pour des raisons de performance. En outre, toutes les insertions se font *avant* la position indiquée par un itérateur nommé `position`. En effet, cela permet d'ajouter à la fin d'une séquence en spécifiant `end()` pour `position`.	
`void splice (iterator position, list<T>& x);`	Insère tout le contenu de la liste `x` dans la liste courante avant l'itérateur `position`. A la fin de cette opération `x` est vide.
`void splice (iterator position, list<T>& x, iterator i);`	Insère dans la liste courante et avant `position`, l'élément de `x` pointé par `i`
`void splice (iterator position, list<T>& x, iterator debut, iterator fin);`	Insère dans la liste courante et avant `position`, l'ensemble des éléments de `x` compris dans l'intervalle `[debut fin[`
Suppressions d'éléments	
`void remove (const T& value);`	Supprime tous les éléments de valeur value. L'ordre des éléments non supprimés n'est pas modifié (ouf ☺)
`template <class Predicate> void remove_if (Predicate pred);`	Supprime tous les éléments de la liste vérifiant le prédicat `pred`. L'ordre des éléments non supprimés n'est pas modifié
`void unique ();`	Compresse à une seule occurrence les valeurs successives identiques
`template <class BinaryPredicate> void unique (BinaryPredicate pred);`	Compresse à une seule occurrence les valeurs successives qui vérifient le prédicat binaire `pred`
Concaténation de listes	
`void merge (list<T>& x);`	Trie la liste courante puis lui ajoute les éléments de `x` en les intercalant. Ces opérations sont effectuées respectivement à l'opérateur **<**. Si deux éléments, l'un de **`this`** et l'un de `x` ont même valeur de clef, alors l'élément de

	this précède l'élément de x dans le résultat.

`template <class Compare>`
`void merge (list<T>& x,`
`Compare comp);`
	Trie la liste courante selon l'ordre indiqué par l'objet foncteur comp puis lui ajoute les éléments de x en les intercalant toujours en fonction de l'objet foncteur comp.
	Même remarque que précédemment.

	Tris
`void sort ();`	Trie les éléments selon l'opérateur **<** Le tri est stable, c'est à dire que l'ordre relatif des éléments de même clef n'est pas modifié
`template <class Compare>` `void sort (Compare comp);`	Trie les éléments selon l'ordre indiqué par l'objet foncteur comp. Le tri est stable
`void reverse();`	Renverse l'ordre des éléments dans la liste

Tableau 3.3 Opérations spéficiques à `list`

Suit un petit exemple montrant l'utilisation des éléments spécifiques au type `list`.

```
#include <list>
#include <iostream>
#include <algo.h>

#include "Entier.hxx"
#include "Genera.hxx"

// Predicat fonction
bool estPair(const Entier &a)
{
  return ((a.val() % 2) == 0);
}

// Predicat foncteur
class EstImpair
{
  public:
  bool operator()(const Entier &a)
  {
    return ((a.val() % 2) != 0);
  }
};


int main(int, char**)
{
  typedef std::list<Entier> ListEnt;
  typedef ListEnt::iterator ListEntIt;

  ListEnt l1;
  ostream_iterator<Entier> sortie(cout, " ");
```

```
cout << "Generation aleatoire d'une liste de 10 entiers" << endl;
l1.resize(10);
generate(l1.begin(), l1.end(), GenAlea());

cout << "Affichage de l'ordre initial" << endl;
copy(l1.begin(),l1.end(),sortie);
cout << endl;

cout << "Apres tri" << endl;
l1.sort();
copy(l1.begin(),l1.end(),sortie);
cout << endl;

cout << "Apres renversement" << endl;

l1.reverse();
copy(l1.begin(),l1.end(),sortie);
cout << endl;

cout << "Generation lineaire des 10 premiers entiers" << endl;

generate(l1.begin(), l1.end(), GenLin());
copy(l1.begin(),l1.end(),sortie);
cout << endl;

cout << "Obtention de deux iterateurs sur le 3 et sur le 8" << endl;
cout << "Construction d'une seconde liste puis splice " << endl;

// notez ici que nous sommes obliges d'utiliser Entier(3) pour chercher le 3
// cette conversion aurait ete implicite si le constructeur d'entier
// n'avait pas ete marque explicit
// je prefere toutefois controler ce que fait mon programme et faire l'effort
// de l'appel de constructeur moi meme
ListEntIt gauche=find(l1.begin(),l1.end(),Entier(3));
ListEntIt droite=find(l1.begin(),l1.end(),Entier(8));

ListEnt l2;
l2.splice(l2.begin(),l1,gauche,++droite);

cout << "Liste initiale : ";
copy(l1.begin(),l1.end(),sortie);
cout << endl;

cout << "Seconde liste : ";
copy(l2.begin(),l2.end(),sortie);
cout << endl;

cout << "Creation d'une liste des elements pairs par suppression des impairs";
cout << endl;

l1.resize(10);
generate(l1.begin(),l1.end(),GenLin());
l1.remove_if(EstImpair());
copy(l1.begin(),l1.end(),sortie);
cout << endl;

cout << "Creation d'une liste des elements impairs par suppression des pairs";
cout << endl;
l2.resize(10);
generate(l2.begin(),l2.end(),GenLin());
l2.remove_if(estPair);
copy(l2.begin(),l2.end(),sortie);
cout << endl;

cout << "Apres Merge" << endl;
```

```
  l1.merge(l2);
  copy(l1.begin(),l1.end(),sortie);
  cout << endl;

  return 0;
}
```

Programme 3.3 Démonstration du type list

```
Generation aleatoire d'une liste de 10 entiers
Affichage de l'ordre initial
73141 251022 20443 194604 20305 3546 159127 109848 4909 167910
Apres tri
3546 4909 20305 20443 73141 109848 159127 167910 194604 251022
Apres renversement
251022 194604 167910 159127 109848 73141 20443 20305 4909 3546
Generation lineaire des 10 premiers entiers
0 1 2 3 4 5 6 7 8 9
Obtention de deux iterateurs sur le 3 et sur le 8
Construction d'une seconde liste puis splice
Liste initiale : 0 1 2 9
Seconde liste : 3 4 5 6 7 8
Creation d'une liste des elements pairs par suppression des impairs
0 2 4 6 8
Creation d'une liste des elements impairs par suppression des pairs
1 3 5 7 9
Apres Merge
0 1 2 3 4 5 6 7 8 9
```

Figure 3.2 Sortie écran de la démonstration de list

Quelques commentaires s'imposent aussi bien sur le programme que sur son résultat.

- Tout d'abord, nous utilisons l'algorithme `generate` pour générer (on s'en doutait un peu ☺) des valeurs dans les listes.

  Les emplacements doivent déjà être réservés ! `generate` n'ajoutera pas d'éléments dans votre liste mais écrase les valeurs déjà présentes. D'ailleurs, `generate` utilise une paire d'itérateurs pour spécifier les éléments à remplir.

  L'algorithme `generate` utilise un objet foncteur dont l'opérateur fonction (`operator ()`) ne doit pas prendre de paramètre et renvoyer un objet compatible avec le type présent dans la liste. Ici nous utilisons deux classes d'objets foncteurs : `GenAlea` et `GenLin`. Le programme suivant fournit leur code :

```
#ifndef __Genera_hxx__
#define __Genera_hxx__

#include "Entier.hxx"

class GenAlea
{
  private:
    static const unsigned long m=259200UL;
    static const unsigned long a=5141UL;
```

```
    static const unsigned long c=73141UL;

    unsigned long x;

  public:
    explicit GenAlea(unsigned long t=0) : x(t) {};
    Entier operator()(void)
    {
      x=(x*a+c)%m;
      return Entier(x);
    }
};

class GenLin
{
  private:
    int prochain;

  public:
    explicit GenLin(int leProchain=0) : prochain(leProchain) {};
    Entier operator()(void)
    {
      return Entier(prochain++);
    }
};

#endif
```

Programme 3.4 Les classes de générateurs `GenAlea` et `GenLin`

Les deux classes proposées sont les suivantes :

`GenAlea` est un générateur de nombres pseudo aléatoires simpliste

`GenLin` crée des entiers dans l'ordre croissant à partir de la valeur passée à son constructeur et par défaut égale à 0.

- Autre point notable, l'utilisation de `splice`. Ici, nous utilisons l'algorithme `find` qui permet de recherche une valeur parmi les éléments contenus dans un intervalle spécifié par une paire d'itérateurs.

  Nous désirons transférer les éléments de 3 à 8 vers une autre liste. Nous recherchons donc ces deux éléments avec `find` de manière à obtenir des itérateurs respectivement nommés `gauche` et `droite`. Hors si nous avions passé la commande :

```
l2.splice(l2.begin(),l1,gauche,droite);
```

  Nous n'aurions transféré que les éléments [3, prédécesseur de 8] vers l2 car le second itérateur est une borne non incluse.

  Afin de transférer jusqu'à droite inclus, nous utilisons l'opérateur ++ sur l'itérateur droite, soit :

```
l2.splice(l2.begin(),l1,gauche,++droite);
```

- La méthode `remove_if` permet de supprimer des éléments d'une liste en fonction d'un critère passé sous forme de prédicat. Celui ci peut prendre deux formes :
  - Une fonction `bool`
  - Un objet foncteur dont l'opérateur fonction (`operator ()`) renvoie `bool`

  Notre exemple illustre ces deux possibilités en fournissant une fonction de test de parité et un foncteur qui test l'inverse !

- Finalement, nous utilisons `merge` pour rassembler deux listes

3.2.3.3 Opérations spécifiques sur les DQ

Les opérations spécifiques aux DQ concernent l'insertion et la suppression en tête de conteneur. Opérations qui sont ici particulièrement optimisées !

On note également la présence d'un opérateur d'accès indexé relativement performant.

Accès indexé aux éléments	

`reference operator[] (size_type);`
`const_reference operator[] (size_type) const;`
`reference at (size_type n);`
`const_reference at (size_type n) const;`
	Opérateur d'indexation, existe en version **`const`** et non **`const`**.
	Renvoyer une référence permet d'utiliser le retour en tant que *lvalue* pour la version non **`const`**.
	La méthode `at` effectue exactement la même opération que l'opérateur crochets
	Ces opérations sont garanties de complexité amortie o(log(n))

Insertions et retraits en tête de DQ	
Ces opérations sont typiques de la DQ et donc particulièrement performantes	
`void push_front (const T& x);`	Insère un élément en tête de liste
`void pop_front ();`	Retire l'élément en tête de liste

Tableau 3.4 Opérations spécifiques au type deque

L'utilisation du type `deque` ne présente aucune difficulté, un exemple ne s'impose donc pas. En fait, vous utiliserez `deque` lorsque vous avez besoin à la fois de l'indexation et de l'accès aux éléments extrêmes d'une collection.

3.2.4 Itérateurs et séquences élémentaires

Les séquences élémentaires sont associées à des itérateurs bidirectionnels. On peut également toutes les doter d'itérateurs d'insertion avec une mention spéciale pour la liste.

En outre, `vector` et `deque` étant des conteneurs indexés, ils fournissent des itérateurs à accès aléatoire

3.2.5 Les utilisations spécialisées des séquences élémentaires

Les utilisations spécialisées des séquences élémentaires modélisent trois structures de données classiques : les piles, les files et les files à priorité. Il faut bien noter que ce sont des classes qui adaptent une séquence élémentaire à une structure de donnée en lui fournissant une nouvelle interface. Lorsque vous aurez besoin d'une de ces classes, vous devrez lui indiquer si elle doit être basée sur un vecteur, une liste ou une DQ.

3.2.5.1 Aspects généraux

A l'instar des autres conteneurs de la STL, les méthodes **`size`** et **`empty`** renvoient respectivement un entier indiquant le nombre d'éléments présents dans le conteneur et un booléen signalant si le conteneur est vide.

En outre, aucun de ces trois types ne supporte les itérateurs. Aussi, les algorithmes généraux (que nous verrons dans un prochain chapitres) seront ils généralement inutilisables sur ce genre de conteneurs ce qui n'est pas sans poser certains problèmes de compatibilité.

Les types files et piles ne posant pas de problèmes particuliers, nous ne donnerons pas ici d'exemple particulier. L'exemple final sur les graphes donnera deux utilisations sur les parcours de graphe en profondeur d'abord ou en largeur d'abord.

3.2.5.2 Les files (*queues*)

Les files sont associées au modèle d'accès premier arrivé, premier traité, à l'image, par exemple, d'une file d'attente à un guichet. Les opérations offertes sont très limitées car elles permettent d'ajouter un élément à la fin, consulter l'élément présent à chaque bout, retirer l'élément en tête de file.

Deux particularités notables sur les files :

- elles ne possèdent pas d'autre constructeur que celui par défaut (sans argument)
- afin de fonctionner correctement, les opérations **`==`** et **`<=`** sont requises sur le type `T` des éléments que l'on souhaite loger dans la file.

Les files effectuant des accès fréquents aux deux extrémités de la structure de donnée sous-jacent ne doivent pas être construites sur un vecteur (le retrait d'un élément en tête serait trop lent) mais plutôt sur une liste ou une DQ. Toutefois, il est

conseillé d'utiliser une DQ qui offre de meilleurs temps de réponse pour les opérations d'ajout et de suppression aux deux extrémités. D'ailleurs, si votre compilateur accepte les **template** avec valeur par défaut, la construction se fait automatiquement sur une DQ.

La déclaration d'une file obéit au prototype suivant :

```
template <class T, class Container = deque<T> >
class queue
{ // etc
```

L'on y retrouve le type T des éléments stockés dans la file ainsi que le type de la structure de données de base sous-jacente initialisée par défaut à une DQ sur T.

Les opérations suivantes sont disponibles sur une file :

`value_type& front ();` `const value_type& front () const;`	Accède à l'élément situé en tête de file sans le retirer, existe en version **const** et non **const**
`value_type& back ();` `const value_type& back () const;`	Accès à l'élément situé à la fin de la file sans le retirer, existe en version **const** et non **const**
`void pop ();`	Retire l'élément situé en tête de file
`void push (const value_type& x);`	Ajoute un élément à la fin de la file

Tableau 3.5 Opérations disponibles sur une file

3.2.5.3 Les piles (*stack*)

Les piles sont associées au modèle d'accès aux données dernier arrivé premier sorti. En d'autres termes, le premier élément à être entré dans la pile sera le premier utilisé. Le modèle tire son nom de l'analogie avec une pile d'assiette où l'on accède toujours au même élément : celui du dessus que ce soit par empilage ou dépilage.

Il est possible de baser une pile sur tous les types de structures élémentaires. On notera toutefois qu'utiliser une liste s'avère peu judicieux car l'on accède qu'à l'élément final. Aussi on se limitera aux vecteurs et aux DQ. Les « gourous » de la STL assurent qu'utiliser une DQ se révèle très légèrement plus performant (un fait que je n'ai pas pu établir personnellement, les différences de temps étant trop faibles pour être significatives) et, d'ailleurs, si votre compilateur accepte les **template** avec valeur par défaut, une DQ sera utilisée par défaut.

A l'instar des files, les piles n'ont qu'un constructeur par défaut, il n'est pas possible d'obtenir d'itérateurs sur les piles, et le type des éléments que vous souhaitez loger dans une pile doit supporter les opérations **==** et **<=**.

La déclaration d'une pile obéit au prototype suivant :

```
template <class T, class Container = deque<T> >
class stack
{ // etc
```

L'on y retrouve le type T des éléments stockés dans la pile ainsi que le type de la structure de données de base sous-jacente initialisée par défaut à une DQ sur T.

Les opérations disponibles sur une pile sont les suivantes :

`value_type& top ();` `const value_type& top () const;`	Accède à l'élément au sommet de la pile sans le retirer, existe en version **`const`** et non **`const`**
`void pop ();`	Retire l'élément au sommet de la pile
`void push (const value_type& x);`	Ajoute un élément au sommet de la pile

Tableau 3.6 Opérations disponibles sur les piles

3.2.5.4 Exemple sur les piles et les files

Le petit fragment de code suivant montre la différence de gestion des piles et des files.

```
#include <deque>
#include <queue>  // pour les files
#include <stack>  // pour les piles
#include <iostream>
#include <algo.h>

#include "Entier.hxx"
#include "Genera.hxx"

int main(int, char**)
{
  const int TAILLE=10;
  Entier    i;
  GenAlea   g;

  stack<Entier,deque<Entier> > pile;
  queue<Entier,deque<Entier> > file;

  cout << "Ordre d'entree : ";
  for (int j=0;j<TAILLE;j++)
  {
    i=g();
    cout << i << " ";
    pile.push(i);
    file.push(i);
  }

  cout << endl;
  cout << "Ordre de sortie pour la pile : ";
  while (!pile.empty())
  {
    cout << pile.top() << " ";
    pile.pop();
  }
```

```
  cout << endl;

  cout << "Ordre de sortie pour la file : ";
  while (!file.empty())
  {
    cout << file.front() << " ";
    file.pop();
  }
  cout << endl;

  return 0;
}
```

Programme 3.5 Programme de test piles et files

```
Ordre d'entree : 73141 251022 20443 194604 20305 3546 159127 109848 4909 167910
Ordre de sortie pour la pile : 167910 4909 109848 159127 3546 20305 194604 20443 251022 73141
Ordre de sortie pour la file : 73141 251022 20443 194604 20305 3546 159127 109848 4909 167910
```

Figure 3.3 Sortie écran du programme de test piles / files

Ce programme montre bien que les éléments de la file ressortent dans leur ordre d'arrivée alors qu'ils en ressortent dans l'ordre inverse pour une pile. La section sur les graphes montrera une autre application des piles et des files.

3.2.5.5 Les files à priorité

Les files à priorité (ou *tas*) sont des collections ordonnées d'éléments qui ne donnent accès qu'à un seul élément : le plus prioritaire. La priorité est déterminée en fonction d'un prédicat passé au constructeur de la structure.

Les utilisations des files à priorité sont multiples. Citons par exemple l'ordonnancement des processus dans un système d'exploitation. L'algorithme du tri en tas peut être divisé en deux phases : construction d'un tas puis extraction du meilleur élément tant que le tas n'est pas vide.

Les files à priorité doivent être basées sur une structure indexée … ce qui élimine les listes d'entrée de jeu ! Si l'occupation mémoire n'est pas votre soucis majeur, vous aurez intérêt à choisir une DQ qui autorise des opérations plus rapides. En outre, si votre compilateur autorise les valeurs par défaut pour les **`template`**, votre file à priorité sera construite par défaut sur une DQ.

Examinons d'un peu plus près la déclaration du type **`priority_queue`** :

```
template <class T,
          class Container = vector<T>,
          class Compare = less<Container::value_type> >
class priority_queue
{ // etc
```

En plus du type des éléments (`T`) et du type du conteneur de base (`Container` dont la valeur par défaut est une DQ basée sur `T`) on remarque la présence d'un troisième paramètre **`template`** nommé `Compare` : le type du prédicat de comparaison. Celui-ci prend comme valeur par défaut la comparaison inférieure ce qui fait de la `priority_queue` par défaut un tas min !

Contrairement aux types file et pile, la classe `priority_queue` dispose de deux constructeurs :

- Un constructeur par défaut de prototype :

```
explicit priority_queue (const Compare& x = Compare());
```

  Vous noterez que les concepteurs de la STL ont pris soin de déclarer ce constructeur à un paramètre **`explicit`** afin d'éviter d'éventuelles conversion implicites erronées depuis un prédicat vers un tas !

  Le paramètre par défaut de ce constructeur est un objet directement instancié à partir de la classe comparateur. Certains compilateurs ne supportent pas cette opération et exigent que vous construisiez vous même cet objet.

- Un constructeur permettant de construire un tas à partir d'une séquence d'éléments en provenance d'un conteneur et délimitée par une paire d'itérateurs :

```
template <class InputIterator>
priority_queue (InputIterator first,
                InputIterator last,
                const Compare& x = Compare());
```

  Notez qu'il s'agit ici d'un membre **`template`**, il vous faudra vérifier la conformité de votre compilateur avant d'utiliser une telle fonctionnalité.

Les opérations disponibles sur les files à priorité sont les suivantes :

`const value_type& top () const;`	Accède à l'élément le plus prioritaire. Notez qu'il n'existe qu'un accès **`const`**
`void pop()`	Retire l'élément le plus prioritaire
`void push (const value_type& x);`	Ajoute un élément dans la structure. Celui ci est stocké conformément à sa priorité

Tableau 3.7 Opérations disponibles sur une file à priorité

Pour finir, nous noterons l'existence de l'algorithme générique `make_heap` dont le but est de fournir une structure de tas à une structure de données indexée (essentiellement vecteur ou DQ). Les structures obtenues se manipulent à l'aide de deux fonctions spécialisées nommées `push_heap` et `pop_heap`. Bien que ces fonctionnalités puissent paraître attractives, il faut les utiliser avec parcimonie car leurs performances sont nettement inférieures à celles du type `priority_queue`.

Concluons l'étude des files à priorité par un petit exemple : nous souhaitons créer un petit programme qui génère `argv[1]` nombres pseudo aléatoires triés dans l'ordre décroissant. Une fois de plus, nous utilisons les classes `Entier` et `GenAlea`.

Le déroulement du programme est très simple :

1. Déclaration d'un type file à priorité sur le type `Entier`, basé sur une DQ d'`Entier` et utilisant la comparaison par valeur inférieure.

2. Remplissage de la file à priorité avec les valeurs issues du générateur de nombres pseudo aléatoires

3. Tant que la file à priorité n'est pas vide, affichage du meilleur élément puis extraction de cet élément.

```
#include <stdlib.h>
#include <iostream>
#include <queue>
#include <deque>

#include "Entier.hxx"
#include "Genera.hxx"

int main(int argc, char *argv[])
{
  typedef std::priority_queue<Entier,
                              deque<Entier>,
                              less<Entier>
                             > FilePrioInt;
  FilePrioInt fileAlea;
  GenAlea     generateur(time(0));
  int         nbAlea=((argc > 1) ? atoi(argv[1]) :  10);

  for (int compteurAlea=0;compteurAlea<nbAlea;compteurAlea++)
  {
    fileAlea.push(generateur());
  }

  while (!fileAlea.empty())
  {
    cout << fileAlea.top() << " ";
    fileAlea.pop();
  }

  cout << endl;
  return 0;
}
```

Programme 3.6 Utilisation d'une file à priorité

La sortie de ce programme est très simple : il s'agit tout simplement d'une ligne d'entiers triés par ordre décroissant.

3.2.6 Les conteneurs associatifs

Les conteneurs associatifs sont très particuliers. En effet leur structure repose sur la notion de paire (clef, valeur). Tout accès aux éléments contenus à l'intérieur des ces types associatifs se fait par l'intermédiaire de ces clefs, y compris les accès fournis par des itérateurs. En revanche, nous obtenons des méthodes de sélection des éléments très puissantes :

- Elément(s) associés à une valeur de clef particulière

- Liste d'éléments dont la clef est supérieur / inférieure à une valeur de référence

- Liste d'éléments dont la clef est comprise à l'intérieur d'une plage

Vous notez la présence d'un pluriel facultatif à la première clause de cette énumération. En effet, les conteneurs associatifs sont disponibles en deux grandes variantes :

- Les associations simples où l'on ne peut associer qu'une seule valeur à une même clef

- Les associations multiples où une clef peut permettre d'accéder à plusieurs valeurs.

Du fait de cette double particularité, la plupart des algorithmes génériques ne fonctionnent pas sur les conteneurs associatifs.

La STL propose deux types de conteneurs associatifs :

- Les ensembles : `set` pour un ensemble simple, `multiset` pour un ensemble multiple

- Les « map » : `map` pour une « map » simple, `multimap` pour une « map » multiple.)

Dans le cas des ensembles, la valeur et la clef sont confondues : l'ordonnancement se fait sur les valeurs des éléments. Dans le cas des « map », vous devez fournir le couple (clef, valeur) complet et les éléments sont stockés dans l'ordre des clefs.

Ces conteneurs étant des collections ordonnées, vous devrez fournir un prédicat lors de la construction de l'objet, le plus souvent sous la forme d'un objet foncteur, à moins que votre compilateur n'accepte les arguments par défaut des **`template`**, auquel cas, `less<type_clef>` est assumé par défaut.

- Les types associatifs exportent en public les types `value_type` et `key_type` respectivement associés au type des valeurs et au type des clefs.

  - Dans le cas de `set` ces deux types sont identiques

  - Dans le cas de `map`, `key_type` est associé au type de la clef (on s'en doutait un peu) et `value_type` est une paire `(const type_clef, type_valeur)` où `type_valeur` est le type des éléments à ordonner en fonction des clefs.

    Même si l'on connaît bien le type des éléments que l'on manipule, il est toujours préférable d'utiliser les **`typedef`** `type_associatif::value_type` dans le cas de s valeurs ou `type_associatif::key_type` lorsque l'on a besoin de référencer une clef.

- Les types associatifs exportent également le type `key_compare`, égal au type du prédicat passé en `template`.

  De la même manière qu'il vaut mieux utiliser le **`typedef`** `key_value`, préférez `key_compare` à l'argument du **`template`** de création

- Dans le même ordre d'idée, les conteneurs associatifs sont également pourvus du type `value_compare`.

  Dans le cas de `set`, celui ci est strictement identique à `key_compare` alors que ce traitement est différencié pour les « map ». En effet, dans ce cas la comparaison est à double critère.

  - En premier lieu sur les clefs

  - Si les clefs sont identiques, la comparaison porte alors sur les valeurs

Nombre d'opérations étant gérées par des paires, nous consacrons notre première rubrique à l'étude de ce type.

3.2.6.1 Le type pair

Accessible via `#include <utility>` le type pair est des plus précieux lors de la manipulation des types associatifs.

Il est rare que vous déclariez vous même un type pair. En effet, vous l'utiliserez plus souvent de façon implicite au retour d'une méthode de la STL ou via les **`typedef`** `value_type` des `map`.

Voici la définition du type `pair` (rappelons qu'en C++ une **`struct`** est une classe dont tous les membres sont publiques par défaut):

```
template <class T1, class T2>
struct pair
{
  T1 first;
  T2 second;
  pair (const T1& x, const T2& y);
};
```

Programme 3.7 Définition du type `pair`

Le type `pair` ne possède pas de fonction membre mais un constructeur ; l'accès aux deux membres respectivement nommés `first` et `second` se faisant de manière directe.

Les opérateurs **`==`** et **`<`** sont redéfinis pour les type pair avec les comportements habituels.

Notons également l'existence de la fonction `make_pair` permettant de construire une paire à partir de deux valeurs quelconques et définie par :

```
template <class T1, class T2>
pair<T1,T2> make_pair (const T1 &a, const T2 &b)
{
  return pair<T1,T2>(a,b);
}
```

Programme 3.8 Définition de la fonction `make_pair`

Cette fonction est vraiment pratique car elle permet de créer une paire sans avoir à spécifier explicitement le type de cette dernière (c'est la fonction qui s'en charge). Le (micro) fragment de programme suivant montre qu'il est vraiment facile de créer une paire quelconque à l'aide de cette fonction.

```
#include <utility>
#include <string>
#include <iostream>

int main(int, char**)
{
  int    i=3;
  string s("Chaîne");
  cout << make_pair(i,s).second << endl;
    // le retour de make_pair de type pair<int,string>
    // l'affichage concernera donc uniquement la chaine
  return 0;
}
```

Programme 3.9 Utilisation de `make_pair`

3.2.6.2 Itérateurs et conteneurs associatifs

Bien qu'ils soient inadaptés à la plupart des algorithmes génériques (du fait de leur structure particulière), les conteneurs associatifs ne sont pas allergiques, loin de la, à la notion d'itérateur.

Les conteneurs associatifs peuvent fournir des itérateurs bidirectionnels à l'aide des méthodes habituelles (`begin`, `end`, `rbegin`, `rend`).

En outre, les méthodes de recherche d'éléments, telles que, par exemple, `find` renvoient également ce genre d'itérateurs.

3.2.6.3 Méthodes et fonctions communes aux types `set` et `map`

Du fait de leurs grandes ressemblances, les opérations disponibles sur les types `set` et `map` sont très semblables. Collections ordonnées d'éléments, les types `set` et `map` n'autorisent que des opérations limitées sur les éléments :

- Insertion d'un ou plusieurs éléments
- Retrait d'un ou plusieurs éléments
- Recherche d'éléments

Toutes ses opérations sont prévues pour être exécutées avec une complexité au pire logarithmique !

Quelques petites différences nous empêchent de rassembler toutes les fonctionnalités dans un même tableau. Nous ne donnerons dans cette rubrique que les méthodes rigoureusement identiques à la signification de `value_type` et `key_type` près ! Nous rappelons également que dans le cas d'un `set`, les notions de clef et de valeur sont confondues.

Méthodes communes aux conteneurs	
`bool empty () const;`	Renvoie **`true`** si le conteneur est vide
`size_type size () const;`	Nombre d'éléments présents dans le conteneur
`size_type max_size () const;`	Nombre maximal d'éléments que peut détenir le conteneur
Accès à l'objet foncteur prédicat	
`key_compare key_comp () const;`	Renvoie l'objet foncteur prédicat du conteneur
Insertion d'éléments	
`pair<iterator, bool> insert (const value_type&);`	Insère un élément dans le conteneur et renvoie une paire constituée d'un itérateur positionné sur l'emplacement du nouvel élément et d'un booléen indiquant si l'opération a réussi ou non. Si le booléen est faux, la valeur de l'itérateur renvoyé n'a pas de signification
`template <class InputIterator> void insert (iterator, InputIterator, InputIterator);`	Insertion d'éléments compris entre deux itérateurs. Notez que c'est une fonction membre **`template`** et que votre compilateur doit être en mesure de les supporter
Suppression d'éléments	
`void erase (iterator);`	Supprime l'élément spécifié par l'itérateur
`size_type erase (const key_type&);`	Supprime l'élément spécifié par sa clef (ou sa valeur pour un `set`)
`void erase (iterator, iterator);`	Supprime les éléments compris entre les deux itérateurs
Accès aux éléments	
`size_type count (const key_type&) const;`	Compte le nombre d'objets associés à une clef particulière. Dans le cas d'un conteneur simple cela ne peut être que 0 ou 1 ; dans le cas d'un ensemble multiple, n'importe quelle valeur entière naturelle.

Attention ! les méthodes suivantes diffèrent par l'utilisation de **const** dans les deux conteneurs	

set :
`iterator find (const key_value&) const;`
map :
`iterator find (const key_value&);`
`const_iterator find (const key_value&) const;`
	Renvoie un itérateur sur l'élément (ou le premier élément dans le cas d'un conteneur multiple) dont la clef est passée en argument.
	Renvoie `this->end()` si aucun élément ne correspond au critère de recherche

set :
`iterator lower_bound (const key_type &x) const;`
`iterator upper_bound (const key_type &x) const;`
 	
map :
`iterator lower_bound (const key_type &x);`
`const_iterator lower_bound (const key_type &x) const;`
(idem **upper_bound**)
	Renvoient respectivement un itérateur sur :
	✓ Le premier élément de clef inférieure à x
	✓ Le dernier élément de clef supérieure à x

set :
`pair<iterator, iterator> equal_range (const key_type& x) const;`
map :
`pair<iterator,iterator> equal_range (const key_type& x)`
`pair<const_iterator,const_iterator> equal_range (const key_type& x) const;`
	Renvoie la paire d'itérateurs `<lower_bound(x),upper_bound(x)>`
	La méthode `equal_range` renvoie une paire d'itérateurs encadrant les éléments associés à une même valeur de clef.

Tableau 3.8 Opérations identiques sur les deux conteneurs `set` et `map`

Sont également fournis en standard pour ces deux types les opérateurs globaux de test d'égalité et d'infériorité (respectivement **==** et **<**).

3.2.6.4 Le type ensemble

Voici le descriptif des fonctions membres spécifiques à `set` :

`value_compare value_comp () const;`	Renvoie l'objet foncteur prédicat de l'ensemble, identique à `key_comp`
`void swap (set<Key, Compare>&);`	Echange les éléments de **this** avec ceux de l'autre ensemble de même type passé en paramètre

Tableau 3.9 Les fonctions membres spécifiques à `set`

En outre, en plus des habituels tests d'égalité et d'infériorité (**==** et **<**), la plupart des opérations sur les ensembles sont fournies en tant que fonctions externes. Toutes ces fonctions ont, à l'exception de `includes` qui renvoie un booléen, la même signature, laquelle se décline en deux versions :

```
itérateurSortie fonction(itérateur debutGauche, itérateur finGauche,
                         itérateur debutDroite, itérateur finDroite)

itérateurSortie fonction(itérateur debutGauche, itérateur finGauche,
                         itérateur debutDroite, itérateur finDroite,
                         prédicat comp)
```

Programme 3.10 signature des opérations ensemblistes

Bien entendu, il faut que les itérateurs soient positionnés sur des ensembles pour que cela fonctionne correctement.

Attention ! (piège à c... ☺), par défaut, tout ce beau monde travaille avec l'opérateur **<** quel que soit l'opérateur fourni pour la construction des **set**. Toutefois, il est possible de changer ce comportement en ajoutant un prédicat en dernier argument (deuxième forme).

Notez également que ces fonctions renvoient un itérateur de type output sur l'ensemble construit et non l'ensemble lui même ce qui peut paraître fâcheux.

La liste complète de ces fonctions est la suivante :

Includes	Renvoie vrai si le second ensemble est inclus dans le premier
set_union	Effectue l'union de deux ensembles en supprimant les doublons
set_intersection	Effectue l'intersection de deux ensembles c'est à dire la liste des éléments présents dans les 2 ensembles
set_difference	Recherche la liste des éléments présents dans A non présents dans B
set_symmetric_difference	Recherche la liste des éléments présents dans A non présents dans B à laquelle s'ajouter la liste des éléments présents dans B non présents dans A

Tableau 3.10 Fonctions globales s'appliquant au type set

3.2.6.5 Le type « map »

Le type map est un peu plus compliqué que set et possède donc plus de méthodes propres. En outre, nous avons pu remarquer dans les paragraphes précédents que la gestion de la constitude sur les accès aux éléments est plus compliquée que dans le cas de set. En effet, toutes les méthodes d'accès sont doublées selon qu'elles renvoient ou non des itérateurs **const**.

Plus fort encore ☺, le type map ***simple*** redéfinit l'opérateur crochet de manière à renvoyer la valeur d'un élément accédé par sa clef. En sus du côté éminemment pratique ce cet opérateur, le fonctionnement du conteneur devient par certains côtés similaire à celui d'un vecteur ou d'une DQ et les « map » deviennent donc utilisables par tous les algorithmes utilisant de l'indexation pure ! En particulier (ne le dites à

personne) l'auteur arrive à utiliser les « map » comme base pour des tas au prix d'une certaine gymnastique il est vrai !

Voici donc la liste des méthodes spécifiques au type map :

`value_compare value_comp () const;`	Renvoie un objet foncteur prédicat de comparaison des paires (clef, valeur)
`void swap (map<Key, Compare>&);`	Echange les éléments de **this** avec ceux de l'autre map de même type passé en paramètre
`T& operator[] (const key_type& x);`	Le fameux opérateur d'indexation. Attention, il ne fonctionne que sur les « map » simples ! Utilisé en écriture, il ajoute un nouvel élément à la « map ». Si un élément de même clef est déjà présent, aucune action n'est entreprise. En lecture, son fonctionnement est quelque peu compliqué : ✓ Si un élément de clef x est présent, alors sa référence est renvoyée ✓ Sinon, une nouvelle entrée de clef x est créée dans la « map », la valeur associée est T() d'où la nécessité d'un constructeur par défaut pour le type valeur ! Pour conclure sur cet opérateur, je désapprouve son utilisation en lecture sans appel préalable à count !

Tableau 3.11 Opérations spécifiques à map

3.2.6.6 Un exemple de map

L'exemple suivant crée une map associant une chaîne à un entier. Il illustre l'utilisation de l'indexation, de find et de count. L'utilisation de equal_range, lower_bound et upper_bound est laissée à l'exemple suivant sur les multimap.

Les commentaires du programme sont affichés par le programme et donc directement inclus dans sa sortie écran, (figure suivant le programme)

```
#include <string>
#include <map>
#include <iostream>

typedef std::map<int, string, less<int> > MapIntString;
typedef MapIntString::iterator MapIntStringIt;
typedef MapIntString::value_type PairIntString;

ostream &operator<<(ostream &a, const PairIntString &b)
{
  a << "Clef : " << b.first << " Valeur : " << b.second;
  return a;
}
```

```
int main(int, char**)
{
  MapIntString   map1;
  MapIntStringIt itCourant;

  map1[1]="premier";
  map1[2]="second";
  map1[3]="troisieme";

  cout << "  Affichage avec acces indexe" << endl;
  for (int compteur=0;compteur<4;compteur++)
    cout << map1[compteur] << " zz ";

  cout << endl;

  cout << " Affichage avec une paire d'iterateurs " << endl;
  itCourant=map1.begin();
  while (itCourant != map1.end())
    cout << *(itCourant++) << " zz ";

  cout << endl;

  cout << " Affichage avec copy " << endl;
  std::copy(map1.begin(),
            map1.end(),
            ostream_iterator<PairIntString>(cout, " ZZ  "));

  cout << "Tentative d'acces a un element non present : index 17" << endl;
  cout << map1[17] << endl;

  cout << "Affichage : remarquez la presence d'un element fictif sur la
clef 17 insere pas l'operateur []" << endl;

  std::copy(map1.begin(),
            map1.end(),
            ostream_iterator<PairIntString>(cout, " ZZ  "));

  cout << endl;

  cout << "Recherche avec find des elements presents " << endl;

  cout << "Recherche de la clef 3 : pas de probleme ";
  MapIntString::iterator itRecherche=map1.find(3);

  if (itRecherche !=  map1.end())
  {
    cout << (*itRecherche).second << endl;
  }
  else
  {
    cout << "Pas trouve" << endl;
  }

  cout << "Recherche de la clef 13 non presente ";
  itRecherche=map1.find(13);

  if (itRecherche !=  map1.end())
  {
    cout << (*itRecherche).second << endl;
  }
  else
  {
    cout << "Pas trouve" << endl;
  }
```

```
  cout << "Autre technique : utilisez count" << endl;

  cout << "Nb de clefs 1  " << map1.count(1) << endl;
  cout << "Nb de clefs 13 " << map1.count(13) << endl;
  cout << "Plus interessant : Nb de clefs 17 " << map1.count(17) << endl;
  cout << "Constatation : le systeme a ajoute des valeurs nulles
(constructeur par defaut) aux emplacements vides" << endl;
  cout << "Moralite : toujours verifier par un appel a find ou count la
presence d'un element" << endl;

  cout << "Effacage de l'element fictif 17" << endl;

  map1.erase(17);

  cout << "Affichage pour verifier : " << endl;

  std::copy(map1.begin(),
            map1.end(),
            ostream_iterator<PairIntString>(cout, " zz "));

  cout << endl;
  cout << "Il a bien disparu !" << endl;


  cout << "Tentative d'insertion d'un element dont la clef est deja
presente " << endl;
  cout << "Tentative de remplacement de (2,second) par (2,zzDeux) " <<
endl;

  map1.insert(MapIntString::value_type(2,"zzDeux"));

  cout << "Affichage pour verifier" << endl;

  std::copy(map1.begin(),
            map1.end(),
            ostream_iterator<PairIntString>(cout, " ZZ  "));

  cout << endl;

  cout << "Comme vous pouvez le voir, ca ne change rien du tout :( " <<
endl;

  cout << "En revanche, il est possible d'ajouter ou de modifier une
valeur a l'aide de l'indexation" << endl;

  map1[2]="deuxieme";
  map1[5]="cinquieme";

  copy(map1.begin(), map1.end(),
       ostream_iterator<PairIntString>(cout, " ZZ  "));

  cout << endl;

  return 0;
}
```

Programme 3.11 Utilisation de `map`

Et voici la sortie écran :

```
Affichage avec acces indexe
 zz premier zz second zz troisieme zz
 Affichage avec une paire d'iterateurs
Clef : 0 Valeur :  zz Clef : 1 Valeur : premier zz Clef : 2 Valeur :
second zz Clef : 3 Valeur : troisieme zz
```

```
 Affichage avec copy
Clef : 0 Valeur :  ZZ  Clef : 1 Valeur : premier ZZ  Clef : 2 Valeur :
second ZZ  Clef : 3 Valeur : troisieme ZZ  Tentative d'acces a un element
non present : index 17

Affichage : remarquez la presence d'un element fictif sur la 17 insere pas
l'operateur []
Clef : 0 Valeur :  ZZ  Clef : 1 Valeur : premier ZZ  Clef : 2 Valeur :
second ZZ  Clef : 3 Valeur : troisieme ZZ  Clef : 17 Valeur :  ZZ
Recherche avec find des elements presents
Recherche de la clef 3 : pas de probleme troisieme
Recherche de la clef 13 non presente Pas trouve
Autre technique : utilisez count
Nb de clefs 1  1
Nb de clefs 13 0
Plus interessant : Nb de clefs 17 1
Constatation : le systeme a ajoute des valeurs nulles (constructeur par
defaut) aux emplacements vides
Moralite : toujours verifier par un appel a find ou count la presence d'un
element
Effacage de l'element fictif 17
Affichage pour verifier :
Clef : 0 Valeur :  zz Clef : 1 Valeur : premier zz Clef : 2 Valeur :
second zz Clef : 3 Valeur : troisieme zz
Il a bien disparu !
Tentative d'insertion d'un element dont la clef est deja presente
Tentative de remplacement de (2,second) par (2,zzDeux)
Affichage pour verifier
Clef : 0 Valeur :  ZZ  Clef : 1 Valeur : premier ZZ  Clef : 2 Valeur :
second ZZ  Clef : 3 Valeur : troisieme ZZ
Comme vous pouvez le voir, ca ne change rien du tout :(
En revanche, il est possible d'ajouter ou de modifier une valeur a l'aide
de l'indexation
Clef : 0 Valeur :  ZZ  Clef : 1 Valeur : premier ZZ  Clef : 2 Valeur :
deuxieme ZZ  Clef : 3 Valeur : troisieme ZZ  Clef : 5 Valeur : cinquieme
ZZ
```

Figure 3.4 Sortie écran du programme d'exemple de `map`

3.2.6.7 Un exemple de `multimap`

L'exemple suivant traite d'une base de données de communes indexées par le code postal. Elle montre comment utiliser `find`, `equal_range`, `lower_bound` et `upper_bound`. Selon l'implémentation de la STL dont vous disposez, il vous faudra peut être utiliser *`#include <multimap>`* en lieu et place de *`#include <map>`* pour cet exemple.

Notez que la première version de l'affichage de la `multimap` (par parcours de la paire d'itérateurs) n'envoie à l'écran que les chaînes de caractères et pas leurs index alors que l'utilisation de copy nous fournit les deux.

Lorsque vous cherchez des éléments avec `equal_range`, il est possible de savoir si la recherche a échoué en comparant le résultat obtenu avec la paire `(conteneur.end(), conteneur.end())`. Il est également possible de tester la présence d'un ou plusieurs éléments avec `count`.

```
#include <string>
#include <map>
#include <iostream>

```

```
// multi map associant plusieurs chaines (les noms des communes)
// à un entier (le code postal)
typedef std::multimap<int, std::string, std::less<int> > MultiMapIntString;
typedef MultiMapIntString::iterator MultiMapIntStringIt;

typedef MultiMapIntString::value_type PairIntString;
typedef std::pair<MultiMapIntStringIt, MultiMapIntStringIt> PaireIterateurs;


// Opérateur d'affichage d'une paire (entier, chaîne)
ostream &operator<<(ostream &a, const PairIntString &b)
{
  a << "Clef : " << b.first << " Valeur : " << b.second;
  return a;
}

// Opérateur d'affichage permettant de "dumper" l'intégralité des valeurs
// comprises entre deux itérateurs
ostream &operator<<(ostream &a, const PaireIterateurs &b)
{
  std::copy(b.first,
            b.second,
            ostream_iterator<PairIntString>(a,"  "));
  return a;
}

int main(int, char**)
{
  MultiMapIntString map1;
  MultiMapIntString::iterator itCourant;
  MultiMapIntString::iterator itFin;
  PaireIterateurs its;

  // Remplissage de la "base"
  map1.insert(MultiMapIntString::value_type(63000,"Clermont Ferrand"));
  map1.insert(MultiMapIntString::value_type(63100,"Montferrand"));
  map1.insert(MultiMapIntString::value_type(63100,"Les Vergnes"));
  map1.insert(MultiMapIntString::value_type(63100,"La Plaine"));
  map1.insert(MultiMapIntString::value_type(63400,"Chamalieres"));
  map1.insert(MultiMapIntString::value_type(63400,"Voie Romaine"));
  map1.insert(MultiMapIntString::value_type(63800,"Cournon d'Auvergne"));
  map1.insert(MultiMapIntString::value_type(63800,"Perignat sur Allier"));
  map1.insert(MultiMapIntString::value_type(63800,"Saint Georges sur Allier"));

  // Affichage de la multimap par une boucle sur les itérateurs
  itCourant=map1.begin();
  itFin=map1.end();

  while (itCourant != itFin)
  {
    cout << (*itCourant).second << " zz ";
    itCourant++;
  }

  cout << endl;

// Affichage de la multimap par une copie sur itérateur de flux en sortie

  std::copy (map1.begin(),
             map1.end(),
             ostream_iterator<PairIntString>(cout, " zz "));

  cout << endl;
```

```
  // Recherche du premier élément associé à clef 63800
  cout << "Resultat de find  sur la clef 63800" << *(map1.find(63800)) << endl;

  // Recherche de tous les éléments associés à la clef 63800
  cout << "Resultat de equal_range sur la clef 63800 " ;

  cout << map1.equal_range(63800) << endl;

  // Recherche d'un élément avec equal_range
  its=map1.equal_range(3000);

  if (its == PaireIterateurs(map1.end(),map1.end()))
    {
      cout << "Pas Trouve" << endl;
    }

  its=map1.equal_range(63100);
  if (its == PaireIterateurs(map1.end(),map1.end()))
    {
      cout << "Pas Trouve" << endl;
    }
  else
    {
      cout << its << endl;
    }

  // Utilisation de lower et upper bound
  cout << "Tentative de lower/upper bound" << endl;
  its.first=map1.lower_bound(63100);
  its.second=map1.upper_bound(63400);
  cout << its << endl;

  // (lower_bound(i), upper_bound(i)) == equal_range(i)
  cout << "Simulation de equal_range avec lower/upper bound" << endl;

  cout << "Equal range " << map1.equal_range(63400) << endl;
  cout << "Lower/upper " << PaireIterateurs(map1.lower_bound(63400),
                                            map1.upper_bound(63400)) << endl;
  return 0;
}
```

Programme 3.12 Utilisation d'une `multimap` pour la gestion d'une base de code postaux

Le résultat à l'écran est le suivant :

```
Clermont Ferrand zz Montferrand zz Les Vergnes zz La Plaine zz Chamalieres zz Voie
Romaine zz Cournon d'Auvergne zz Perignat sur Allier zz Saint Georges sur Allier zz

Clef : 63000 Valeur : Clermont Ferrand zz Clef : 63100 Valeur : Montferrand zz Clef :
63100 Valeur : Les Vergnes zz Clef : 63100 Valeur : La Plaine zz Clef : 63400 Valeur :
Chamalieres zz Clef : 63400 Valeur : Voie Romaine zz Clef : 63800 Valeur : Cournon
d'Auvergne zz Clef : 63800 Valeur : Perignat sur Allier zz Clef : 63800 Valeur : Saint
Georges sur Allier zz

Resultat de find  sur la clef 63800 : Clef : 63800 Valeur : Cournon d'Auvergne

Resultat de equal_range sur la clef 63800 : Clef : 63800 Valeur : Cournon d'Auvergne
Clef : 63800 Valeur : Perignat sur Allier  Clef : 63800 Valeur : Saint Georges sur
Allier  Clef : 63800 Valeur : Cournon d'Auvergne  Clef : 63800 Valeur : Perignat sur
Allier  Clef : 63800 Valeur : Saint Georges sur Allier

3000 : Pas Trouve
```

```
Clef : 63100 Valeur : Montferrand  Clef : 63100 Valeur : Les Vergnes  Clef : 63100
Valeur : La Plaine

Tentative de lower/upper bound

Clef : 63100 Valeur : Montferrand  Clef : 63100 Valeur : Les Vergnes  Clef : 63100
Valeur : La Plaine  Clef : 63400 Valeur : Chamalieres  Clef : 63400 Valeur : Voie
Romaine

Simulation de equal_range avec lower/upper bound

Equal range Clef : 63400 Valeur : Chamalieres  Clef : 63400 Valeur : Voie Romaine

Lower/upper Clef : 63400 Valeur : Chamalieres  Clef : 63400 Valeur : Voie Romaine
```

Figure 3.5 Résultat d'exécution du programme de test sur les `multimap`

3.3 Les itérateurs

Tout au long de la présentation des conteneurs, nous n'avons cessé d'utiliser des itérateurs. De même, tous les algorithmes génériques qui permettent d'utiliser les conteneurs reposent sur cette notion.

3.3.1 Définition et premier contact

Un itérateur est une variable qui repère la position d'un élément dans un conteneur. Il existe deux positions bien spécifiques :

- Le début d'une collection, c'est à dire son premier élément. Cet itérateur est renvoyé par la méthode `begin()`
- La fin d'une collection, c'est à dire l'élément positionné juste après le dernier. Cet itérateur est renvoyé par la méthode `end()`

L'élément le plus important à noter est le suivant :

Les itérateurs ont un fonctionnement identique à celui des pointeurs dans un tableau. En effet, considérez le code suivant :

```
int tab[10];

int *p;
int  i=0;
for (p=tab;i<10;i++,p++)
  cout << *p << " " << endl;
```

Le pointeur *p joue exactement le rôle d'un itérateur : référencer un élément. L'opérateur * permet d'accéder à l'élément placé sous l'itérateur ; l'opérateur ++ étant dédié à « l'avancement » de l'itérateur.

Dans la boucle précédente, le test d'arrêt porte sur la valeur d'un compteur entier. Supposons que nous décidions de faire tous les tests sur un pointeur, nous obtiendrions alors :

```
int tab[10];
```

```
int *p;
int *borne=tab+10;
for (p=tab;p<borne;p++)
  cout << *p << " " << endl;
```

Ce qui, en terme de STL s'écrira :

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

for (vector<int>::iterator p=tab.begin();p<tab.end();p++)
cout << *p << " " << endl;
```

et le tour est joué !

Vous voyez c'est pas compliqué d'utiliser un itérateur !

3.3.2 Les différents types d'itérateurs

Il existe différents types d'itérateurs liés à divers types d'opérations ainsi qu'aux conteneurs sur lesquels ils opèrent. Les opérations suivantes seront plus ou moins disponibles selon le type d'itérateur auquel vous serez confronté :

Déréférencement : accès à l'élément (soit en lecture, soit en écriture) contenu dans le conteneur à l'emplacement pointé par l'itérateur

Déplacement de l'itérateur : en avant, en arrière ou de façon aléatoire

Comparaison de deux itérateurs : afin de savoir s'ils font référence au même emplacement dans le conteneur

Les itérateurs sont regroupés selon deux grands critères :

- Le type d'accès aux éléments (lecture et / ou écriture)
- Le type de mouvement (en avant, en arrière, aléatoire)

En fonction de leurs spécificités, les collections seront à même de générer certains types d'itérateurs.

Le tableau suivant récapitule quels types d'itérateurs vous pourrez rencontrer selon le conteneur, sachant que les types `queue`, `stack` et `priority_queue` *ne fournissent jamais d'itérateur*

Type	Déplacement	Accès	Génération
`Input`	Aucun	Lecture	Classe `input_iterator`
`Output`	Aucun	Ecriture	Classes `output_iterator`, `font_inserter` et `back_inserter` (utilisés par les algorithmes génériques d'insertion)

Forward	Avant	Lecture / Ecriture	Tous
Bidirectionnel	Avant / Arrière	Lecture / Ecriture	`Vector, deque, list, set, multiset, map, multimap`
Aléatoire	Avant /Arrière/ Aléatoire	Lecture / Ecriture	`Vector, deque`

Tableau 3.12 Les différents types d'itérateurs

En outre, les itérateurs existent en version **`const`** et non **`const`**. Déréférencer un itérateur non **`const`** renvoie une référence sur un objet contenu dans le conteneur. Cette référence pourra être utilisée en tant que *lvalue*, c'est à dire à gauche d'un signe = ; en particulier, vous pourrez changer l'objet présent dans le conteneur à cet emplacement. Avec un itérateur **`const`**, non seulement vous ne pourrez pas utiliser le retour comme *lvalue*, mais vous pourrez seulement appliquer des méthodes **`const`** sur l'objet désigné !

Bien entendu, un conteneur **`const`** renverra systématiquement des itérateurs **`const`**.

3.3.3 Comment obtenir un itérateur ?

Il existe plusieurs manières d'obtenir un itérateur :

- Les méthodes `begin()` et `end()` renvoient respectivement un itérateur sur le début (premier élément) et la fin (élément après le dernier) d'un conteneur

  Notons immédiatement qu'il existe des itérateurs bizarres : les *reverse iterators*, qui reculent sous l'effet de l'opérateur ++. Ils s'obtiennent à l'aide des méthodes `rbegin()` et `rend()` et sont disponible en version **`const`** et non **`const`**.

  Ils s'utilisent essentiellement lorsque vous devez parcourir une collection à rebrousse poil, la plupart des algorithmes de parcours utilisant l'opérateur ++.

- Plusieurs méthodes ou algorithmes renvoient également des itérateurs

  - Tous les algorithmes de type `find`, `lower_bound` ou `upper_bound` qui recherchent des éléments et renvoient des itérateurs sur leurs emplacements. Toute recherche infructueuse se soldant par le renvoie de la borne supérieure de recherche ou de `conteneur.end()`.
  - Les fonctions d'opérations ensemblistes
  - `etc` …

- Construire directement un itérateur à l'aide de son constructeur. C'est notamment le cas pour les itérateurs de flux ou les itérateurs d'insertion

3.3.4 Les itérateurs de flux

Ils permettent de « dumper » une séquence d'éléments (en provenance d'un conteneur et encadrés par des itérateurs) vers un flux ou encore de lires des éléments depuis un flux vers un conteneur.

Les itérateurs de flux de sortie sont construits grâce au constructeur :

```
ostream_iterator<TypeAEcrire>(flux_de_sortie,chaineDeSeparateur)
```

Nous les avons régulièrement utilisés pour toutes les opérations de sortie dans les exemples sur les conteneurs. Toutefois, nous redonnons ici un exemple permettant d'afficher le contenu d'un vecteur à l'écran, chaque élément étant séparé par un saut de ligne. L'algorithme `copy` est utilisé dans la plupart des cas pour envoyer les éléments vers l'itérateur de sortie.

```
vector<int> v;
// code omis
copy (ostream_iterator<int>(cout,"\n"),v.begin(),v.end());
```

Les itérateurs de flux en entrée sont un peu plus compliqués. En effet, ils ont besoin de 2 paramètres **`template`** :

- Le type de données à lire
- Un second type que l'on doit fixé à `ptrdiff_t` sous peine d'ennuis

Appeler l'opérateur ++ sur un tel itérateur lit un objet dans le flux et le stocke dans une structure interne. L'opérateur * permet alors d'y accéder.

Le constructeur d'un itérateur en entrée de flux ne prend qu'un seul paramètre : le flux à lire. En l'absence de paramètre, le constructeur fournit un itérateur servant de comparaison et modélisant EOF. L'exemple suivant permet de clarifier les idées :

```
#include <iostream>
#include <fstream>
#include <deque>

int main(int, char**)
{
  ifstream    f("toto");
  deque<int> t;

  istream_iterator<int,ptrdiff_t> entree(f), eof;

  while (entree != eof)
    {
      t.push_back(*entree);
      entree++
    }

  copy(t.begin(),t.end(),ostream_iterator<int>(cout," "));
```

```
  cout << endl;
  f.close();
  return 0;
}
```

Programme 3.13 Essai des itérateurs de flux en entrée

Ce petit programme ne fait que lire un fichier jusqu'à la fin, stocker le contenu dans une DQ avant de le renvoyer à l'écran.

Le code suivant n'utiliser pas d'intermédiaire :

```
#include <iostream>
#include <fstream>
#include <deque>

int main(int, char**)
{
  ifstream    f("toto");

  istream_iterator<int,ptrdiff_t> entree(f), eof;


  copy(entree,eof,ostream_iterator<int>(cout," "));
  cout << endl;
  f.close();
  return 0;
}
```

4. Annexe : un exemple de `makefile`

Le programme suivant indique le `Makefile` utilisable pour générer l'exécutable associé à divers fichiers source dont l'extension est `.cpp`.

```
#On purge la liste des suffixes
.SUFFIXES:

#On ajoute simplement les extensions dont l'on a besoin
.SUFFIXES: .cpp .o

#Executable
EXEC=princi

#Liste des fichiers objet
OBJETS=Vehicule.o Depannage.o Princi.o RandomGenerator.o \
 MarsagliaGenerator.o

#Liste des fichiers source
SOURCES=Vehicule.cpp Depannage.cpp Princi.cpp RandomGenerator.cpp \
 MarsagliaGenerator.cpp

# avec GnuMake : SOURCES=${OBJETS:%.o=%.cpp}

#Compilateur et options de compilation
CCPP=g++
CCFLAGS=-g -Wall -ansi -pedantic

#Regle explicite de construction de l'executable
${EXEC}:${OBJETS} makefile
      ${CCPP} -o ${EXEC} ${OBJETS}
```

```
#Regle implicite de construction des fichiers objet à partir
#des fichiers .cpp
.cpp.o:
      ${CCPP} -c ${CCFLAGS} $< -o $@

# avec GnuMake
# %.o : %.cpp
#     ${CCPP} -c ${CCFLAGS} $< -o $@

#gestion des dépendances
depend:
      sed -e "/^#DEPENDANCIES/,$$ d" makefile > dependances
      echo "#DEPENDANCIES" >> dependances
      ${CCPP} -MM ${SOURCES} >> dependances
      cat dependances > makefile
      rm dependances

#DEPENDANCIES
Depannage.o: Depannage.cpp Depannage.hxx
MarsagliaGenerator.o: MarsagliaGenerator.cpp MarsagliaGenerator.hxx \
 RandomGenerator.hxx
Princi.o: Princi.cpp Voiture.hxx Vehicule.hxx Camion.hxx Parc.hxx \
 Adapter.hxx Helico.hxx MarsagliaGenerator.hxx RandomGenerator.hxx
RandomGenerator.o: RandomGenerator.cpp RandomGenerator.hxx
Vehicule.o: Vehicule.cpp Vehicule.hxx Depannage.hxx \
 RandomGenerator.hxx
```

Programme 4.1 `Makefile` générique pour le C++

Vous noterez les points importants suivant :

- `.SUFFIXES:` purge la liste des règles implicites. En particulier, la liste des extensions autorisées pour les règles implicites est vidée.

- `.SUFFIXES: .cpp .o` ajouter les extensions `.cpp` et `.o` à la liste des extensions valides pour les règles implicites. Ce comportement en ajout justifie la purge précédente qui permet de repartir d'une situation connue : le vide !

- La règle `.cpp.o:` constitue un bon exemple de règle implicite permettant de créer un fichier `.o` à partir d'un fichier `.cpp`. Notez l'utilisation des macros $< et $@ respectivement associées au membre gauche et au membre droit de la règle.

  Soit, dans notre exemple : $< ⇔ `fichier.cpp` et $@ ⇔ `fichier.o`

- La règle `depend` permet de créer automatiquement des règles de dépendances. Elle utilise pour cela l'option `–MM` de `gcc` et une petite instruction `sed` !

- `Gnu make` permet de rajouter des règles plus puissantes ainsi qu'une syntaxe plus intuitive pour les règles implicites.