

Les flux

Classes *ostream* et *istream*

Surdéfinition des opérateurs << et >>

Connexion à un fichier

Les flux

- Un flux peut être considéré comme un « canal »
 - Recevant de l'information (flux de sortie)
 - Fournissant de l'information (flux d'entrée)
- Les opérateurs << ou >> servent à assurer le transfert de l'information
- Un flux peut être connecté à un périphérique ou à un fichier

```
int nbr;  
cout << "bonjour ";  
cin >> nbr;
```



cout est un flux de sortie connecté à l'écran, et *cin* est un flux d'entrée connecté au clavier.

Classes ostream et istream

- Un flux est un objet d'une classe prédéfinie
 - *ostream* pour les flux de sortie
 - *istream* pour les flux d'entrée
- Ces classes surdéfinissent les opérateurs << et >> pour les types de base
- Leur emploi nécessite l'incorporation du fichier d'en-tête *iostream*



L'opérateur <<

- L'opérateur << est surdéfini au sein de la classe *ostream*



- *ostream & operator << (expression)*
- L'argument est une expression d'un type quelconque
- Transmet la valeur de l'expression en la formatant de façon appropriée
 - Les pointeurs sont acceptés
 - Pour le type *char **, la chaîne de caractère pointée est affichée
 - Pour les pointeurs d'un autre type que *char*, la valeur du pointeur est affichée (adresse)
 - Les tableaux sont acceptés, mais sont considérés comme des pointeurs
- Renvoie une référence sur le flux après écriture de l'information

La fonction write

- La fonction membre `write` permet de transmettre une suite d'octets
 - On peut utiliser `write` avec une chaîne de caractères
 - `out.write("bonjour ", 4)`
 - `write` ne fait pas intervenir de caractère de fin de chaîne
 - Si un tel caractère apparaît dans la longueur prévue, il est transmis comme les autres
 - `write` ne réalise aucun formatage
 - Indispensable lorsqu'on souhaite transmettre une information sous forme « brut »
 - Fichier sous forme « binaire » où les informations sont enregistrées comme en mémoire



```
int nbr = 15;  
out.write((char*)&nbr, sizeof(int));
```



Ici, l'adresse du premier octet en mémoire de la variable `nbr` est donnée, ainsi que le nombre d'octets d'un `int`. Le cast permet ici de transformer un `int*` en `char*` (mais l'adresse reste la même). La fonction `write` va donc envoyer, dans le flux, la valeur des différents octets qui représentent la variable `nbr` en mémoire.

L'opérateur >>

- L'opérateur >> est surdéfini au sein de la classe *istream*
 - *istream & operator >> (type_de_base &)*
 - L'argument est une lvalue d'un type de base quelconque
- Extrait les caractères nécessaires pour former une valeur du type voulu
- Renvoie une référence sur le flux après extraction de l'information
- Les « espaces blancs » servent de séparateurs
 - Espace, tabulation ('`\t`'), fin de ligne ('`\n`')

Types acceptés par >>



- Tous les types de base sont acceptés
- Parmi les pointeurs, seuls les *char** sont acceptés
 - L'information lue est complétée par un caractère nul de fin de chaîne ('`\0`')
 - Pour lire *k* caractères il faut prévoir un tableau de *k+1* caractères
 - On peut recourir au manipulateur `setw` pour limiter le nombre de caractères lus
 - Exemple: `char *tab = new char[21]; cin >> setw(20) >> tab;`
- Les tableaux ne sont pas acceptés, sauf ceux de caractères



Ici, `setw(20)` permet de restreindre le nombre de caractères lus à 20. Au plus 20 caractères seront lus, mais il est possible que moins de caractères soient lus si un séparateur est rencontré.

La fonction `get`

- `istream & get (char &)`

- Extrait un caractère et copie sa valeur dans la lvalue donnée en argument



- Contrairement à `>>`, la fonction `get` permet de lire n'importe quel caractère

- Comme par exemple les caractères d'échappement
 - Exemple: `char c; cin.get(c);`

- `int get()`

- Extrait un caractère et renvoie sa valeur sous la forme d'un entier
 - Fournit une valeur spéciale `EOF` lorsque la fin du fichier a été rencontrée
 - Exemple: `int i; i = cin.get();`

La fonction `getline`

- `istream & getline(char* ch, int taille, char delim = '\n')`
 - Lit des caractères sur le flux et les place dans l'emplacement d'adresse `ch`
 - La lecture s'interrompt lorsque
 - Le caractère `delim` a été trouvé
 - `taille-1` caractères ont été lus
 - La fonction ajoute un caractère de fin de chaîne à la suite des caractères lus
- `gcount()` donne le nombre de caractères lus au dernier appel de `getline`

```
char tab[20];
```

```
cin.getline(tab, 20);
```

```
cout >> "Vous avez écrit " >> tab >> " de taille " >> cin.gcount();
```

La fonction read

- ▶ Permet de lire une suite d'octets
 - ▶ On peut utiliser *read* pour une chaîne de caractères de longueur donnée
 - ▶ `char tab[20]; in.read(tab, 20);`
 - ▶ Aucun caractère de fin de chaîne n'est ajouté
 - ▶ Indispensable pour accéder à une information d'un fichier sous forme « brut »
 - ▶ Joue un rôle symétrique à la fonction *write*

```
int nbr;  
in.read((char *)&nbr, sizeof(int));
```



Ici, l'adresse du premier octet en mémoire de la variable *nbr* est donné, ainsi que le nombre d'octets que comprend un *int*. La fonction *read* va donc récupérer dans le flux, la valeur de suffisamment d'octets pour représenter une variable de type *int*, et mettre leurs valeurs à l'adresse en mémoire de *nbr*.

Statut d'erreur

- A chaque flux est associé un ensemble de bits formant le « statut d'erreur »
 - Permet de rendre compte du bon ou mauvais fonctionnement du flux
 - Positions des différents bits définies par des constantes déclarées dans *ios*
 - *ios::eofbit*: activé si la fin du fichier est atteinte
 - *ios::failbit*: activé lorsque la prochaine opération d'entrée-sortie ne peut aboutir
 - *ios::badbit*: activé lorsque le flux est dans un état irrécupérable
 - La constante *ios::goodbit* (0) correspond à la valeur du statut sans erreur
 - La prochaine opération d'entrée-sortie ne pourra aboutir que si le statut est *ios::goodbit*



Le statut d'erreur *ios::goodbit* correspond au cas où les bits *ios::eofbit*, *ios::failbit* et *ios::badbit* sont tous à 0. Notez que *ios::goodbit* correspond à une valeur que peut prendre les bits, et non à la position d'un bit (comme c'est le cas pour *ios::eofbit*).

Action concernant les bits d'erreur

- Certaines fonctions permettent de connaître le statut d'erreur
 - `eof()`, `bad()`, `fail()` renvoient vrai si le bit correspondant est activé
 - `good()` renvoie vrai si le flux est dans un statut sans erreur.
 - La fonction `rdstate()` renvoie une valeur entière correspondant au statut d'erreur
- Certaines fonctions modifient la valeur de certains bits d'erreur
 - `void clear(int i = ios::goodbit)`
 - Active les bits d'erreur correspondant à la valeur fournie en argument
 - Exemple: `cin.clear(ios::badbit);`
 - Appelée sans argument, elle met le flux dans un état sans erreur
 - A utiliser lors de la surdéfinition des opérateurs `<<` et `>>`

Surdéfinition des opérateurs () et !

- Il est possible de tester un flux en le considérant comme une valeur logique
 - L'opérateur () renvoie vrai si et seulement si aucun bit d'erreur n'est activé
 - L'opérateur ! renvoie faux si et seulement si aucun bit d'erreur n'est activé

```
int nbr; char c;
cin >> nbr;
if(cin)
    cout >> " Vous avez tapé " >> nbr;
else{
    cin.clear();
    cin >> c;
    cout >> " Le caractère " >> c >> " est invalide ";
}
```

Le test du *if* consiste à regarder si le flux *cin* est dans un état sans erreur. L'instruction (*cin*) est en fait équivalente à *cin.good()*. Le *else* correspond au cas où *!cin* est vrai. Ici, cela pourrait être le cas si l'utilisateur rentre un caractère qui n'est pas un chiffre. Dans ce cas, l'instruction *cin.clear()* remet le flux dans un état sans erreur (ce qui permettra de le réutiliser plus tard).



Surdéfinition des opérateurs << et >>

- L'opérateur << défini dans la classe *ostream* peut être surdéfini
 - Sous la forme d'une fonction indépendante ou amie de la classe concernée
 - `ostream & operator << (ostream &, expression_de_type_classe)`
 - Utilise les possibilités classiques de l'opérateur <<
- L'opérateur >> défini dans la classe *istream* peut être surdéfini
 - Sous la forme d'une fonction indépendante ou amie de la classe concernée
 - `istream & operator >> (istream &, type_classe &)`
 - Utilise les possibilités classiques de l'opérateur >>

Exemple



Les opérateurs << et >> sont redéfinis dans le cas où le second opérande est un *Point*. Dans le cas de l'opérateur >>, toutes les valeurs pour construire le *Point* sont extraites. Si le flux est dans un état d'erreur ou si le séparateur n'est pas ',', le flux est mis dans un état d'erreur par la fonction *clear*. *ios::badbit | in.rdstate()* correspond à l'ancien état du flux où le bit *badbit* est mis à 1.

```
class Point{
private:
    int x,y;
public:
    friend ostream& operator<<(ostream&, Point);
    friend istream& operator>>(istream&, Point&);
};

ostream& operator << (ostream& out, Point p){
    out << p.x << ',' << p.y;
    return out;
}
```

```
istream& operator >> (istream& in, Point &p){
    int temp_x, temp_y; char c;
    in >> temp_x >> c >> temp_y;
    if((in) && c == ','){
        p.x = temp_x;
        p.y = temp_y;
    }
    else
        in.clear(ios::badbit | in.rdstate());
    return in;
}
```

Connexion à un fichier (sortie)

- Il suffit de créer un objet de type *ofstream*, classe dérivant de *ostream*
- ⚠ ➤ Nécessite d'inclure un fichier d'en-tête nommé *fstream* en plus de *iostream*
- Le constructeur de la classe *ofstream* a deux arguments
 - Une chaîne de caractères correspondant au nom du fichier
 - Le mode d'ouverture (*ios::out*, *ios::app*, *ios::binary*)
- L'écriture dans le fichier se fait comme pour n'importe quel flux

```
ofstream out("nom_fichier", ios::out);
```

```
if(out)
```

```
    out << "écriture d'une chaîne de caractères";
```

Si l'ouverture du fichier s'est bien passée (test du *if*) il est possible d'écrire dans le fichier en utilisant l'opérateur *<<* comme pour *cout*.



Connexion à un fichier (entrée)

- Il suffit de créer un objet de type *ifstream*, classe dérivant de *istream*
- ⚠ ➤ Nécessite d'inclure un fichier d'en-tête nommé *fstream* en plus de *iostream*
- Le constructeur de la classe *ifstream* a deux arguments
 - Une chaîne de caractères correspondant au nom du fichier
 - Le mode d'ouverture (*ios::in*, *ios::binary*)
- La lecture dans le fichier se fait comme pour n'importe quel flux

```
ifstream in("nom_fichier", ios::in);
```

```
int nbr;
```

```
if(in)
```

```
    in >> nbr;
```

Accès direct

- La position courante dans le fichier est gérée par un « pointeur »
 - Nombre précisant le rang du prochain octet à lire ou écrire
- Il est possible d'agir sur ce pointeur par l'intermédiaire de fonctions
 - *seekg* (de *ifstream*) et *seekp* (de *ofstream*) changent la valeur ce pointeur
 - Le premier argument est un entier correspondant au déplacement du pointeur
 - Le second argument précise le point de départ du déplacement
 - *ios::beg* déplacement par rapport au début du fichier
 - *ios::cur* déplacement par rapport à la position actuelle
 - *ios::end* déplacement par rapport à la fin du fichier
 - *tellg* (de *ifstream*) et *tellp* (de *ofstream*) donnent la position du pointeur

```
ifstream in("nom_fichier", ios::in);  
in.seekg(10, ios::beg);
```



Le « pointeur » est initialement placé sur le 10^{eme} octet en partant du début du fichier.

Bibliothèque standard

Conteneurs, itérateurs et algorithmes

Fonctions, prédicats et objet fonctions

Notion de conteneur

- Conteneurs: classes représentant les structures de données les plus connues
 - Vecteurs, listes, ensembles, tableaux associatifs
 - Patrons de classes paramétrés par le type de leurs éléments

```
list<int> li;  
vector<Point> vp;
```

list et *vector* sont deux patrons de classe. Le paramètre correspond au type d'éléments contenu dans la liste ou le vecteur. *list<int>* correspond à une liste d'entier, et *vector<Point>* à une liste de Point.



Notion d'itérateur

- Itérateur: objet défini par le conteneur qui généralise la notion de pointeur
 - Un itérateur « pointe » vers un élément du conteneur
 - Peut être incrémenté par l'opérateur ++ pour pointer vers l'élément suivant
 - Peut être déréférencé en utilisant l'opérateur *
 - Deux itérateurs peuvent être comparés par les opérateurs == ou !=
- Tous les conteneurs fournissent un itérateur portant le nom *iterator*
 - *begin()* renvoie un itérateur qui pointe sur le premier élément
 - *end()* renvoie un itérateur qui pointe « juste après » le dernier élément

```
list<int> li;
```


```
for(list<int>::iterator it = li.begin(); it != li.end(); it++)
```

```
    cout << *it << '\n';
```



list<int>::iterator est la classe d'un itérateur sur des *list<int>*. L'objet *iterator* est obtenu par l'appel de la fonction membre *begin*. Tant que l'itérateur ne pointe pas sur la dernière case, on incrémente l'itérateur pour le faire pointer sur la case suivante.

Intervalles d'itérateur

- Certains itérateurs pourront posséder des propriétés supplémentaires
 - Décrémentation par l'opérateur -- (itérateur bidirectionnel)
 - Accès direct par l'opérateur []
 - Exemple: `vector<float> v(10); vector<float>::iterator it = v.begin(); it[5] = 5;`
- Intervalle d'itérateur: définie sous forme de deux valeurs d'itérateur
 - Exemple: `vector<float>::iterator it1 = v.begin(), it2 = v.end();`
 - `it2` accessible à partir de `it1` par l'intermédiaire de l'opérateur ++
 - Définit un intervalle allant de l'élément pointé par `it1` jusqu'à celui pointé par `it2`
 -  L'élément pointé par `it2` n'est pas compris dans l'intervalle

Notion d'algorithme

- Beaucoup d'opérations peuvent être appliquées par le biais d'un itérateur
 - Fournies sous forme de patrons de fonctions, paramétrés par le type des itérateurs
 - S'appliquent à une séquence définie par un intervalle d'itérateur

```
vector<int> ve(10);  
list<int> li(10);  
int n = count(li.begin(), li.end(), 0);  
copy(ve.begin(), ve.end(), li.begin());
```

La fonction *count* va compter le nombre d'occurrences de la valeur (ici 0), au sein de l'intervalle d'itérateur. La fonction *copy* va recopier le contenu donné par l'intervalle d'itérateur (ici le l'ensemble des éléments de *ve*) à partir de l'itérateur donné en troisième paramètre (ici le début de la liste *li*).



- Pour que *count()* fonctionne, il faut que l'opérateur `==` soit défini



Itérateurs et pointeurs

- Un algorithme peut utiliser tout objet ayant les propriétés d'un itérateur
 - C'est le cas des pointeurs usuels

```
int tab[6] = {0, 1, 2, 3, 4, 5};  
list<int> li(6);  
copy(tab, tab+6, li.begin());
```

Ici, la fonction `copy` est utilisée avec un tableau. `tab` est un pointeur vers la première case du tableau, et `tab+6` est un pointeur sur la case qui se trouve juste après la dernière du tableau. `tab` et `tab+6` forment donc un intervalle d'itérateur, et les éléments qui seront recopiés sont tous ceux du tableau. Ils seront recopiés dans la liste `li`.



Conteneurs et objets



- La construction d'un conteneur dont les éléments sont des objets entraîne
 - Soit l'appel d'un constructeur (sans argument) de la classe des objets
 - Soit l'appel d'un constructeur par copie de la classe des objets
- L'opérateur = du conteneur fera aussi appel à celui des objets contenus



```
vector<Point> v1(3);
```

```
vector<Point> v2 = v1;
```



- Il faudra prévoir les fonctions appropriées pour les champs dynamiques
 - Schéma de classe canonique

Fonctions

- Peut appliquer une fonction aux éléments d'une séquence
 - Fonction passée en argument de l'algorithme
- ⚠
 - La fonction doit posséder un argument du type des éléments
 - La fonction peut posséder une valeur de retour qui ne sera pas utilisée

```
void affiche(int i){  
    cout << i << '\n';  
}  
  
list<int> li;  
for_each(li.begin(), li.end(), affiche);
```

L'algorithme *for_each* va appliquer la fonction donnée en paramètre (ici *affiche*) à chaque élément qui se trouve dans l'intervalle d'itérateur (ici, les éléments de la liste *li*). Cette fonction est applicable ici car elle prend en paramètre un *int* et que les éléments de la liste sont des *int*.



Prédicats

- Prédicat: fonction qui renvoie une valeur de type *bool*
 - On rencontrera des prédicats unaires et des prédicats binaires
- Certains algorithmes nécessiteront des prédicats en argument

```
bool impair(int n){  
    return n%2 == 1;  
}  
  
list<int> li;  
list<int>::iterator it;  
it = find_if(li.begin(), li.end(), impair);
```

L'algorithme *find_if* va rechercher le premier élément dans l'intervalle d'itérateur (autrement dit dans la liste *li*) qui est impair (pour lequel la fonction *impair* renvoie vrai). La fonction *impair* s'applique à des *int* donc elle peut être utilisée pour une liste d'*int*.



Objets fonctions

- ▀ Objet fonction: objet dont la classe surdéfinit l'opérateur ()
 - ▀ Ces objets peuvent être utilisés comme des fonctions
- ▀ Les algorithmes font appel à des objets fonctions dans leurs patrons
- ⚠ ▀ On peut fournir indifféremment un objet fonction ou une fonction usuelle

```
class Affiche{  
    public:  
        void operator()(int);  
};  
void Affiche::operator()(int i){  
    cout << i << '\n';  
}  
  
list<int> li;  
for_each(li.begin(), li.end(), Affiche);
```

La classe *Affiche* redéfinit l'opérateur () pour un paramètre entier, et permet donc d'instancier des objets fonctions. Ces objets peuvent être utilisés comme des fonctions sur des entiers. Ici, l'algorithme *for_each* va appliquer l'opérateur () de la classe *Affiche* pour afficher chaque entier de la liste *li*.



Classes fonctions prédéfinies

➤ Le fichier d'en-tête *functional* définit des patrons de classes fonctions

➤ *equal_to* (*==*), *not_equal_to* (*!=*)

➤ *less* (*<*), *greater* (*>*) , *greater_equal* (*>=*), *less_equal* (*<=*)



➤ Les opérateurs correspondants doivent être définis dans la classe considérée

➤ Toutes ces classes fonctions possèdent un constructeur sans argument



➤ Peut être cité comme argument d'un algorithme

```
vector<int> ve;
```

```
sort(ve.begin(), ve.end(), greater<int>);
```



greater<int> renvoie un objet du patron de classe *greater* (de paramètre *int*). Cet objet fonction correspond à l'opérateur *>* (qui n'est pas une fonction). L'algorithme *sort* va ranger les éléments de l'intervalle d'itérateur en utilisant l'ordre associé à l'opérateur *>*.

Relations d'ordre

- Certains algorithmes nécessitent la connaissance d'une relation d'ordre
 - Conteneurs ordonnés, algorithmes de tri ...
- Pour les objets il faudra que l'opérateur $<$ soit surdéfini convenablement
- ⚠ ➤ Cette définition doit correspondre à une relation d'ordre faible stricte
 - Antiréflexive: $x < x$ est faux pour tout x
 - Transitive: $a < b$ et $b < c$ implique $a < c$
 - Asymétrique: $x < y$ vrai implique $y < x$ faux
 - $x \sim y$ et $y \sim z$ implique $x \sim z$
 - $x \sim y$ (l'incomparabilité) équivaut à ni $x < y$ ni $y < x$ vrai
- Définit une relation d'ordre sur les classes d'équivalences induites

Conteneurs séquentiels

Fonctions communes

Conteneurs *vector* et *deque*

Conteneur *list*

Adaptateurs *stack* et *queue*

Conteneurs séquentiels

- Ordonnés suivant un ordre imposé explicitement par le programme
- Trois conteneurs principaux
 - *vector*: généralise la notion de tableau
 - *list*: correspond à la notion de liste doublement chaînée
 - *deque*: classe intermédiaire
- ⚠ ➤ Chacun d'eux est défini dans un fichier d'en-tête portant son nom
- Ces conteneurs séquentiels ont des fonctionnalités communes

Constructeurs

- Sans argument qui construit un conteneur vide
 - Exemple: `list<float> li;`
- Un argument entier qui construit un conteneur avec autant d'éléments
 - ⚠ ➤ Pour des éléments « objets », initialisés par appel au constructeur sans argument
 - Exemple: `vector<Point> v(5);`
- Un argument entier et un argument du type des éléments
 - ⚠ ➤ Similaire mais les éléments sont des copies du second arguments
 - Exemple: `Point p(0, 0); vector<Point> v(5, p);`
- Un intervalle d'itérateur
 - Construire un conteneur avec un séquence d'éléments du même type
 - Exemple: `list<int> li; vector<int> ve(li.begin(), li.end());`
- Constructeur par recopie
 - Exemple: `list<int> li1; list<int> li2 = li1;`

Modification globale

- On peut affecter un conteneur à un autre du même type



- Même nom de patron et même type d'éléments

- Exemple: `vector<float> v1, v2; v1 = v2;`

- `assign(début, fin)` permet d'affecter les éléments d'une séquence d'itérateur



- Les éléments doivent être du même type

- Exemple: `vector<float> ve; list<float> li; ve.assign(li.begin(), li.end());`

- `assign(nbr_fois, val)` permet d'affecter une valeur plusieurs fois

- Exemple: `vector<float> ve(5); ve.assign(5, 0);`

- `clear()` vide le conteneur de son contenu

- Exemple: `vector<Point> ve; ve.clear();`

- `swap()` permet d'échanger le contenu de deux conteneurs de même type

- Exemple: `vector<float> v1, v2; v1.swap(v2)`

Comparaisons

- L'opérateur == est surdéfini pour les conteneurs séquentiels
 - Renvoie vrai si les conteneurs sont de même taille et leurs éléments sont égaux
 - ⚠ ➤ Repose sur la surdéfinition de l'opérateur == pour le type des éléments
- L'opérateur < est surdéfini pour les conteneurs séquentiels
 - Renvoie vrai si le 1^{er} opérande est lexicographiquement plus petit que le 2nd
 - ⚠ ➤ Repose sur la surdéfinition de l'opérateur < pour le type des éléments
- Les opérateurs !=, >, <= et >= sont surdéfinis de façon analogue

Insertion et suppression

- La fonction *insert* permet d'insérer des éléments
 - *insert(position, valeur)* insère *valeur* avant l'élément pointé par *position*
 - Exemple: `list<int> li; list<int>::iterator it = li.begin(); li.insert(it, 1);`
 - *insert(position, nbr_fois, valeur)* insère *nbr_fois* copies de *valeur*
 - Exemple: `list<int> li; li.insert(li.begin(), 10, 1);`
 - *insert(position, début, fin)* insère les éléments de la séquence *[début, fin)*
 - Exemple: `list<int> li; vector<int> ve; li.insert(li.begin(), ve.begin(), ve.end());`
- La fonction *erase* permet de supprimer des éléments
 - *erase(position)* supprime l'élément désigné par *position*
 - Exemple: `list<int> li; list<int>::iterator it = li.begin(); li.erase(it);`
 - *erase(début, fin)* supprime les éléments de la séquence *[début, fin)*
 - Exemple: `list<int> li; li.erase(li.begin(), li.end());`

Conteneur vector

- Reprend la notion usuelle de tableau



- Autorisant un accès direct à un élément en $O(1)$

- Accès à un élément peut se faire par l'opérateur []

- Exemple: `vector<int> v(5); v[4] = 0;`

- Accès par `at()` qui génère l'exception `out_of_range` si l'indice est incorrect



- Moins rapide que l'opérateur []

- Exemple: `vector<int> v(5); v.at(4) = 0;`

- Sa taille peut varier au fil de l'exécution

- La fonction `size()` permet de connaître le nombre d'éléments



- L'insertion et la suppression en $O(n)$

Gestion mémoire



- Certaines opérations invalident les itérateurs et références
 - En cas d'augmentation de la taille
 - En cas d'insertion d'un élément
 - En cas de suppression d'un élément
- `capacity()`: nombre d'éléments maximum sans réallocation de mémoire
- `reserve(taille)`: fixe la capacité à *taille*
- `max_size()`: taille maximum qu'on peut allouer au vecteur

```
vector<int> ve;  
vector<int>::iterator it = ve.begin();  
ve.reserve(ve.capacity()*5);
```



Ici, `ve.capacity()` renvoie le nombre de cases réservées pour le vecteur `ve`. `ve.reserve(ve.capacity()*5)` va réserver un nombre de cases 5 fois plus important pour le vecteur `ve`. Les anciennes cases vont être désallouées, et l'itérateur `it` ne pointera plus sur les cases du vecteur.

Conteneur deque

➤ Fonctionnalités voisines d'un vecteur



➤ Accès direct en $O(1)$ et insertion en $O(n)$

➤ Insertion et suppression en début et fin en $O(1)$



➤ Opération en $O(1)$ plus lentes que vecteur

➤ *front()* et *back()* pour accéder au premier et dernier élément

➤ *push_front(valeur)* et *push_back(valeur)* pour insérer en début et fin

➤ *pop_front()* et *pop_back()* pour supprimer le premier et dernier élément

```
deque<int> de;
```

```
de.push_front(4);
```

```
de.back() = 5;
```

```
de.pop_back();
```



L'instruction `de.push_front(4)` va ajouter un élément en début de `deque` dont la valeur est 4. L'instruction `de.back() = 5` va modifier la valeur de la dernière case (remplace par la valeur 5). L'instruction `de.pop_back()` va supprimer la dernière case du `deque`.

Conteneur list

- Correspond au concept de liste doublement chaînée
 - Itérateur bidirectionnel pour parcourir la liste à l'endroit (++) ou à l'envers (--)
- ⚠
 - Insertion et suppression en $O(1)$
 - Pas d'accès direct et pas de surdéfinition de l'opérateur []
- Les fonctions *front()*, *back()*, ... de *deque* sont également disponibles
- Fonctions de suppression conditionnelle
 - *remove(valeur)*: supprime tous les éléments égaux à *valeur*
 - ⚠ ➤ Repose sur l'opérateur == de la classe des éléments
 - *remove_if(prédicat)*: supprime tous les élément pour lequel *prédicat* est vrai

```
int tab[10] = {0, 1, 0, 2, 0, 3, 0, 4, 0, 10};  
list<int> li(tab, tab+10);  
li.remove(0);
```



Ici, la liste est construite à partir d'un intervalle d'itérateur (sur les éléments d'un tableau). Elle contient initialement les mêmes entiers que dans le tableau. La fonction membre *remove* est ensuite appelée sur cette liste, et supprime tous les cases à 0.

Opérations globales sur les listes

- La classe *list* dispose de sa propre fonction de tri
 - ⚠ ➤ S'appuie sur une relation d'ordre faible stricte
 - *sort()*: trie la liste en s'appuyant sur l'opérateur *<* de la classe des éléments
 - *sort(prédicat)*: trie la liste en s'appuyant sur le prédicat binaire *prédicat*
- Fonction permettant de supprimer les séquences d'éléments identiques
 - *unique()*: conserve un élément d'une suite de valeurs égales selon l'opérateur *==*
 - *unique(prédicat)*: idem mais valeurs égales selon le prédicat binaire *prédicat*

```
int tab[10] = {2, 3, 1, 2, 5, 4, 1, 2, 3, 4};
```

```
list<int> li(tab, tab+10);
```

```
li.sort();
```

```
li.unique();
```



Ici, la liste d'entiers est triée par la fonction membre *sort* selon l'opérateur *<* défini sur les entiers. Le résultat devrait être une liste contenant les éléments 1,1,2,2,2,3,3,4,4,5. La fonction *unique* va ensuite supprimer les éléments répétés, donnant la liste 1,2,3,4,5.

Opérations globales sur les listes

- La classe *list* dispose de sa propre fonction de fusion
 - *merge(liste)*: fusionne *liste* avec la liste appelante
 - ⚠
 - Vide *liste* dont le contenu est intégré dans la liste appelante
 - S'appuie sur l'opérateur *<* de la classe des éléments pour « ranger » les éléments
 - *merge(liste, prédicat)*: idem mais s'appuie sur le prédicat binaire *prédicat*
- Fonction permettant de transférer une partie d'une liste vers une autre
 - *splice(position, liste)*: déplace les éléments de *liste* à la position donnée
 - *splice(position, liste, position2)*: l'élément déplacé est celui à la position *position2*
 - *splice(position, début, fin)*: déplace les éléments dans l'intervalle *[début, fin)*

```
list<float> li1, li2;
```

```
list<float>::iterator it = li2.begin();
```

```
li1.splice(li1.begin(), li2, it);
```

```
li2.merge(li1);
```



La méthode *splice* va transférer le premier élément de la liste *li2* dans la liste *li1* (plus précisément en premier position de *li1*). La fonction *merge* va ensuite transférer les éléments de *li2* dans *li1* de manière ordonnée (si les éléments de *li1* et *li2* étaient ordonnées, alors les éléments de *li2* après *merge* seront ordonnés).

L'adaptateur stack

- Classe patron construite sur un conteneur d'un type donné
- Le patron *stack* est destiné à la gestion des piles (Last In, First Out)
- ⚠ ➤ Peut être construit à partir des trois conteneurs séquentiels (*vector*, *deque*, *list*)
 - Constructeur sans argument
- Possède uniquement les fonctions membres suivantes
 - *empty()*: fournit *true* si et seulement si la pile est vide
 - *size()*: fournit le nombre d'éléments dans la pile
 - *top()*: accès à l'élément au sommet de la pile
 - *push(valeur)*: insère *valeur* dans la pile
 - *pop()*: supprime l'élément au sommet de la pile

```
stack<int, vector<int>> st;  
st.push(0); st.push(1);  
cout << st.top() << '\n';  
st.pop();
```



La pile *st* contient des entiers et est construite à partir d'un vecteur d'entiers. Les entiers 0 et 1 sont insérés dans la pile par la fonction *push*. La fonction *top* renvoie la valeur en haut de la pile (ici 1) sans la supprimer. La fonction *pop* supprime la valeur en haut de la pile (1).

L'adaptateur queue

- Le patron *queue* est destiné à la gestion des files (First In, First Out)
 - Peut être construit à partir de *deque* ou *list*
 - Constructeur sans argument
- Possède uniquement les fonctions membres suivantes
 - *empty()*: fournit *true* ssi la file est vide
 - *size()*: fournit le nombre d'éléments dans la file
 - *front()*: accès à l'élément en tête de file
 - *back()*: accès à l'élément en fin de file
 - *push(valeur)*: insère *valeur* dans la file
 - *pop()*: supprime l'élément en tête de file

```
queue<int, list<int>> fi;  
fi.push(0); fi.push(1);  
cout << fi.front() << '\n';  
fi.pop();
```



La file *fi* contient des entiers et est construite à partir d'une liste d'entiers. Les entiers 0 et 1 sont insérés dans la file (à la fin) par la fonction *push*. La fonction *top* renvoie la valeur en début de file (ici 0) sans la supprimer. La fonction *pop* supprime la valeur en début de file (0).

Conteneurs associatifs

Conteneurs map et multimap

Conteneurs set et multiset

Conteneurs associatifs

- ▶ Permet de retrouver une information en fonction de sa valeur
 - ▶ Partie de sa valeur nommée clé
- ▶ Ordonnée intrinsèquement en se fondant sur une relation (par défaut $<$)
 - ▶ Cette relation doit être définie comme une relation d'ordre faible stricte
- ▶ Les deux conteneurs les plus importants sont *map* et *multimap*
 - ▶ Correspondent pleinement au concept en associant une clé et une valeur
- ▶ *map* impose l'unicité des clés
- ▶ *multimap* ne l'impose pas
 - ▶ On pourra avoir plusieurs éléments de mêmes clés qui apparaîtront consécutivement
- ▶ Chacun d'eux est défini dans un fichier d'en-tête portant son nom

Conteneur map

- Formé d'éléments composés de deux parties

- Une clé et une valeur



- Patron de classe *pair* paramétré par le type de la clé et celui de la valeur

- L'opérateur `[]` permet d'accéder à la valeur associée à une clé



- Efficacité en $O(\log(n))$

```
map<char, int> m;
```

```
m['S'] = 5;
```

```
cout << m['S'] << ' ' << m['X'];
```



m est un conteneur associatif dont les clés sont des *char* et dont les éléments stockés sont des *int*. L'instruction `m['S'] = 5` va insérer un élément dont la clé est 'S', et va mettre sa valeur associée à 5. L'instruction `m['S']` va permettre ensuite d'accéder à cette valeur. L'instruction `m['X']` va insérer dans le conteneur la clé 'X' associée à l'élément 0, et renvoyer sa valeur (0).



- Le simple fait de chercher `m['X']` créera l'élément correspondant

- Sa valeur est initialisée à 0

Iterateurs

- Peut parcourir les éléments d'un *map* avec un itérateur bidirectionnel
 - Les conteneurs associatifs sont ordonnés intrinsèquement
- ⚠ ➤ Les éléments pointés par l'itérateur sont de type *pair*
 - Disposent de deux membres publics
 - *first*: correspond à la clé
 - *second*: correspond à la valeur

```
map<char, int> m;  
map<char, int>::iterator it;  
for(it = m.begin(); it != m.end(); it++)  
    cout << " ( " << (*it).first << " , " << it->second << " )\n ";
```

Pour accéder aux membres *first* et *second* de la paire pointée par l'itérateur *it*, il est possible d'utiliser les notation *(*it)*. et *it->*.



Constructeurs

- Le patron de classe *pair* possède un constructeur à deux arguments
 - Exemple: `pair<char, int> p('X', 0);`
- Le patron de classe *map* possède trois constructeurs
 - Constructeur sans argument créant un conteneur vide
 - Constructeur par recopie
 - Exemple: `map<char, int> m1; map<char, int> m2 = m1;`
 - Constructeur à partir d'une séquence
 - Exemple: `list<pair<char, int>> li; map<char, int> m(li.begin(), li.end());`



Notez que les éléments de l'intervalle d'itérateur doivent être des objets de type *pair* dont les paramètres de type sont les mêmes que la *map*.

Accès et insertion

- *find(clé)* renvoie un itérateur sur un élément (*pair*) ayant la clé donnée
 - Renvoie la valeur de *end()* si la clé n'a pas été trouvée
 - Exemple: `map<char, int> m; map<char, int>::iterator it=m.find('S'); cout << it->second;`

- La fonction membre *insert* permet l'insertion



- Opération en $O(\log(n))$
- *insert(élément)*: insère la paire *élément*
 - Exemple: `map<char, int> m; m.insert(pair('X', 0));`

- *insert(début, fin)*: insère les paires de la séquence *[début, fin)*
 - Exemple: `map<char, int> m1, m2; m2.insert(m1.begin(), m1.end());`



- Une insertion n'aboutit pas si une clé équivalente est déjà présente
 - Renvoient une paire dont le champs *second* est *vrai* si l'insertion a lieu
 - `map<char, int> m; pair<map<char, int>::iterator, bool> p = m.insert(pair('X', 0));`

Suppression

- La fonction `erase` permet de supprimer un ou plusieurs éléments
 - `erase(position)`: supprime l'élément désigné par `position`
 - Exemple: `map<char, int> m; map<char, int>::iterator it = m.begin(); m.erase(it);`
 - `erase(début, fin)`: supprime les éléments de l'intervalle `[début, fin)`
 - Exemple: `map<char, int> m; m.erase(m.begin(), m.end());`
- ⚠ ➤ Aucune opération n'entraîne d'invalidation des itérateurs
 - Excepté pour les éléments supprimés qui ne sont plus accessibles

```
map<char, int> m;  
map<char, int>::iterator it=m.find('S');  
m.erase(it);
```

Conteneur multimap

- Dans un conteneur *multimap*, une même clé peut apparaître plusieurs fois
- ⚠ ➤ L'opérateur [] n'est plus applicable
- Hormis cette restriction, mêmes fonctions membres que *map*
 - S'il existe plusieurs clés équivalentes, *find* renvoie un itérateur vers un de ces éléments
 - La fonction membre *erase(clé)* supprime tous les éléments de clé voulue
- *count(clé)* renvoie le nombre d'éléments dont la clé est clé
- *lower_bound(clé)*: renvoie un itérateur sur le premier élément non inférieur à clé
- *upper_bound(clé)*: renvoie un itérateur sur le premier élément supérieur à clé

```
multimap<char, int> m; m.insert(pair('S', 1)); m.insert(pair('S', 2));  
multimap<char, int>::iterator it = m.find('S');  
m.erase(m.lower_bound('S'), m.upper_bound('S'));
```

Conteneur set

- Cas particulier de *map* où aucune valeur n'est associée à la clé
 - Un élément d'un *set* est une constante qui ne peut pas être modifiée

```
char tab[10] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};  
set<char> s(tab, tab+10);  
if(s.count('1') > 0)  
    cout << "l'ensemble contient 1";  
if(s.find('a') == s.end())  
    cout << << "l'ensemble ne contient pas a";  
s.erase('0');
```

Conteneur multiset

- Conteneur *set* dans lequel on autorise plusieurs éléments équivalents

```
char chaine[] = "bonjour monsieur "  
multiset<char> m(chaine, chaine+strlen(chaine));  
cout << "occurrences de o = " << m.count('o');  
m.erase(m.lower_bound('m'), m.upper_bound('n'));
```

Algorithmes

Catégories d'itérateurs

Différents types d'algorithmes

Algorithmes

- Les algorithmes standard se présentent sous forme de patrons de fonctions



- Définis dans le fichier d'en-tête *algorithm* (à inclure)



- La manipulation se fait toujours par l'intermédiaire d'un itérateur
 - Le type des éléments se déduit à partir du type de l'itérateur
- Il existe plusieurs catégories d'itérateurs
 - Unidirectionnels, bidirectionnels et à accès direct
 - Itérateur en entrée: unidirectionnel qui n'autorise que la consultation
 - Itérateur en sortie: unidirectionnel qui n'autorise que la modification

Itérateurs de flux de sortie

- Itérateur de sortie associé à un flux
 - Patron de classe `ostream_iterator` paramétré par le type des éléments concernés
 - Constructeur recevant un flux de sortie existant
 - Exemple: `ostream_iterator<char> oi(cout);`
 - L'instruction `*oi = 'x'` envoie le caractère 'x' sur le flux `cout`
 - L'incrémentation `oi++` est théoriquement possible mais sans effet

```
char tab[10] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};  
ofstream out("nom_fichier", ios::out);  
ostream_iterator<char> oi(out);  
copy(tab, tab+10, oi);
```



Ici, un `ofstream_iterator` est construit à partir du flux de sortie `out` (qui correspond à un fichier). La fonction `copy` ici va copier les `char` de l'intervalle d'itérateur (autrement dit du tableau) dans le fichier.

Itérateurs de flux d'entrée

- Itérateur d'entrée associé à un flux
 - Patron de classe *istream_iterator* paramétré par le type des éléments concernés
 - Constructeur recevant un flux d'entrée existant
 - Exemple: `istream_iterator<int> ii(cin);`
- ⚠ ➤ Nécessite la possibilité de détecter la fin
 - Par convention, le constructeur sans argument représente la fin

```
list<int> li(100);  
ifstream in("nom_fichier", ios::in);  
istream_iterator<int> ii(in), ie;  
copy(ii, ie, li.begin());
```



Ici, un *ifstream_iterator* est construit à partir du flux d'entrée *in* (qui correspond à un fichier). Un autre *ifstream_iterator* est construit à partir du constructeur sans argument (correspond à la fin du fichier). La fonction `copy` ici va lire les entiers du fichier et les copier dans la liste *li* (à partir du début)

Itérateur d'insertion

- Certains algorithmes modifient les éléments d'une séquence
 - ⚠
 - Les emplacements doivent déjà exister
 - Leur modification doit être autorisée
- Mécanisme transformant une succession de modifications en insertions
 - Patron de classe *insert_iterator* paramétré par le type de conteneur
 - Patrons de fonctions pour créer un itérateur d'insertion
 - *front_inserter(conteneur)*: crée un itérateur d'insertion en début de conteneur
 - *back_inserter(conteneur)*: crée un itérateur d'insertion en fin de conteneur
 - *inserter(conteneur, position)*: crée un itérateur d'insertion à *position*
 - ⚠ ➤ Le conteneur doit disposer d'une fonction membre *insert(valeur, position)*

```
vector<int> ve; list<int> li;  
insert_iterator<list<int>> ii = front_inserter(li);  
copy(ve.begin(), ve.end(), ii);
```



Le patron de fonction *front_inserter* crée un itérateur d'insertion pour insérer des éléments (*int* ici) en début de liste *li*. La fonction *copy* va donc ici insérer une copie des éléments du vecteur *ve* en début de liste *li*.

Algorithmes d'initialisation

- ▶ Permettent de donner des valeurs à des éléments existants
- ⚠ ▶ Peuvent être utilisés avec un itérateur d'insertion
- ▶ `copy(début, fin, position)`
 - ▶ Copie les éléments de la séquence `[début, fin)` à partir de `position`
- ▶ `generate(début, fin, fonction)`
 - ▶ Initialise la séquence `[début, fin)` en faisant appel à une fonction sans argument

```
list<int> li1(10), li2;  
generate(li1.begin(), li1.end(), rand);  
copy(li1.begin(), li1.end(), front_inserter(li2));
```



La fonction `generate` va ici initialiser les éléments de la liste `li1` en utilisant la fonction `rand` (qui génère des valeurs aléatoires). La fonction `copy`, utilisée avec un itérateur d'insertion, va ensuite insérer une copie des éléments de `li1` en fin de liste `li2`.

Algorithmes de recherche

- Ces algorithmes ne modifient pas la séquence
- ⚠ ➤ Fournissent tous un itérateur sur l'élément trouvé
 - L'itérateur sur la fin de séquence est renvoyé si l'élément n'est pas trouvé
- Algorithmes fondés sur l'égalité (opérateur ==)
 - *find(début, fin, valeur)*: renvoie un élément de *[début, fin)* égal à *valeur*
 - *find_first_of(début1, fin1, début2, fin2)*
 - Renvoie le premier élément de la séquence *[début2, fin2)* trouvé dans *[début1, fin1)*
- Algorithmes fondés sur l'inégalité (opérateur <)
 - *max_element(début, fin)*: renvoie le plus grand élément de *[début, fin)*
 - *min_element(début, fin)*: renvoie le plus petit élément de *[début, fin)*



```
int tab[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; int tab2[3] = {3, 5, 7};  
int *i = find_first_of(tab, tab+10, tab2, tab2+3);  
int *m = max_element(tab, tab+5);
```

Le patron de fonction *find_first_of* (et *max_element*) est appelé avec un intervalle d'itérateur sous forme de pointeurs sur des entiers, et renvoie donc un itérateur sous la forme d'un pointeur sur un entier.

Algorithme de transformation

- Modifient les valeurs d'une séquence sans en modifier le nombre
- ⚠ ➤ Disposent d'une version suffixée par `_copy` copiant le résultat dans une séquence
- `replace(début, fin, valeur1, valeur2)`
 - Remplace dans toute la séquence `[début, fin)` `valeur1` par `valeur2`
 - ⚠ ➤ Se fonde sur l'opérateur `==`
- `replace_if(début, fin, prédicat, valeur)`
 - Remplace dans `[début, fin)` tout élément pour lequel `prédicat` est vrai par `valeur`
- `rotate(début, position, fin)`
 - Décalage circulaire pour que l'élément à `position` devienne le premier
- `random_shuffle(début, fin)`: permutation aléatoire des éléments



```
vector<int> ve(10); generate(ve.begin(), ve.end(), rand); list<int> li(10);  
replace_copy(ve.begin(), ve.end(), li.begin(), 0, 1);  
rotate(ve.begin(), ve.begin()+5, ve.end());
```

La fonction `replace_copy` va insérer, en début de liste `li`, une copie des éléments de `ve` en remplaçant 0 par 1. La fonction `rotate` va ensuite décaler les éléments de `ve` de cinq cases.

Algorithmes de suppression

- Pour éliminer d'une séquence les éléments répondant à certains critères
- ⚠
 - Ne suppriment pas d'éléments mais regroupent au début ceux non concernés
 - Fournit en retour un itérateur sur le premier élément non conservé
- Version suffixée par `_copy` copiant le résultat dans une séquence
 - Permet de créer une nouvelle séquence si utilisé avec un itérateur d'insertion
- `remove(début, fin, valeur)`: élimine tous les éléments de `valeur == valeur`
- `remove_if(début, fin, prédicat)`: idem avec les éléments où `prédicat` est vrai
- `unique(début, fin)`: conserve la première valeur des séries de valeurs ==



```
list<int> li(10) , li2; generate(li.begin(), li.end(), rand);  
list<int>::iterator it remove(li.begin(), li.end(), 0);  
li.erase(it, li.end());  
unique_copy(li.begin(), li.end(), front_inserter(li2));
```

La fonction `remove` est appelée sur la liste `li` pour supprimer les occurrences de 0. Cette fonction regroupe ces occurrences à la fin de la liste et renvoie un itérateur sur le premier 0. La fonction `erase` va permettre de supprimer réellement les éléments à 0.

Algorithmes de tri

- S'appliquent lorsque l'opérateur `<` est défini ou à l'aide d'un prédicat binaire



- Nécessitent un itérateur à accès direct (*list* dispose de sa propre fonction `sort`)
- `sort(début, fin)`: trie les éléments de la séquence `[début, fin)`
 - Pas stable dans le sens où l'ordre des éléments équivalents n'est pas conservé
- `stable_sort(début, fin)`: version stable de `sort`
- `partial_sort(début, position, fin)`: effectue un tri des éléments jusqu'à `position`

```
vector<int> ve(10); generate(ve.begin(), ve.end(), rand); vector<int> ve2 = ve;  
sort(ve.begin(), ve.begin()+5);  
partial_sort(ve2.begin(), ve2.begin()+5, ve2.end());
```



La fonction `sort` va ici trier les éléments contenus dans les 5 premières cases de `ve` (sans changer l'ordre des 5 dernières). La fonction `partial_sort` va rechercher les 5 plus petits éléments de tout le tableau et les placer, de manière ordonnée, dans les 5 premières cases de `ve2`.

Algorithmes de recherche et fusion



- S'applique à une séquence ordonnée par une relation d'ordre faible stricte
 - Par la surdéfinition de l'opérateur $<$ ou par le même prédicat binaire
- `lower_bound(début, fin, valeur)`: itérateur sur la première position \geq *valeur*
- `upper_bound(début, fin, valeur)`: itérateur sur la première position $>$ *valeur*
- `merge(début1, fin1, début2, fin2, position)`
 - Fusionne les séquences $[début1, fin1)$ et $[début2, fin2)$ et mets le résultat à *position*



```
list<int> li(10); generate(li.begin(), li.end(), rand); li.sort();  
list<int> li2(10), li3; generate(li2.begin(), li2.end(), rand); li2.sort();  
list<int>::iterator it = lower_bound(li.begin(), li.end(), 5);  
merge(li.begin(), it, li2.begin(), li2.end(), front_inserter(li3));
```

La fonction *merge* va ici fusionner le contenu des liste *li* et *li2* et insérer le résultat en début de liste *li3*. Notez que *li3* va être ordonnée car la fonction *merge* produit une séquence ordonnée (à condition que *li* et *li2* le soit).

Algorithmes à caractères numériques

- Effectuent des opérations numériques fondées sur les opérateurs +, - ou *
- *accumulate(début, fin, valeur_init)*: somme des éléments + *valeur_init*
- *inner_product(début1, fin1, début2, valeur_init)*
 - Produit scalaire entre les séquences *[début1, fin1)* et *[début2, ...)* + *valeur_init*
- *partial_sum(début, fin, position)*
 - Cumul partiel de la séquence *[début1, fin1)* placé à *position*

```
int tab[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, tab2[10];
```

```
partial_sum(tab, tab+10, tab2);
```

```
cout << inner_product(tab, tab+10, tab2, 0);
```