Résumé des possibilités des collections de la bibliothèque standard (par C.G. et M.M)

1- Points communs à toutes les collections

Construction

```
C<...> c;                // size == 0
C<...> c(c2);            // À partir d'une autre collection de même type(CC)
C<...> c(itDebut,itFin); // Un intervalle d'une autre collection de type quelconque
```

Opérateur d'affectation

```
c1= c2; // Même type de collection
```

`operator==`

```
c1 == c2  // Vrai si taille égale et que les éléments (de même rang) sont égaux.
```

`operator<`

```
c1 < c2         // Comparaison lexicale élément par élément.
```

Autres `operator >, >=, <=, !=` (Tous les opérateurs relationnels sont des fonctions non membres)

Taille

```
c.size();     // Retourne le nombre d'élements contenus dans la collection
c.empty();    // Retourne true si le nombre d'éléments == 0
c.max_size(); // Permet de connaître la taille maximale qu'on peut allouer
```

Suppression

```
c.erase(it);            // Supprime le(s) élément(s) désigné(s)
c.erase(itDebut,itFin); // (bien sûr, itFin n'est pas compris)

c.clear(); // c.erase(c.begin(), c.end());
```

Échange

```
c1.swap(c2); // Permet d'échanger le contenu de deux collections de même type
```

Types imbriqués : Chaque collection standard définit des types comme membres de la façon qui lui est la plus appropriée.

```
C::value_type       // Type des éléments
C::size_type        // Type pour représenter le numéro des éléments
C::difference_type  // Nombre d'éléments entre 2 éléments

C::reference        // Type reference aux éléments (Type&)
C::const_reference  //                            (const Type&)
```

Ces différents membres vont permettre au programmeur d'écrire du code utilisant un collection sans savoir le type d'objets engagés. En particulier, ils vont lui permettre d'écrire du code qui fonctionnera avec n'importe quelle collection standard ou autre collection respectant les mêmes normes. Par exemple :

```
template <typename C>                    //C peut être un vector<int>, list<double>, etc.
typename C::value_type Somme(const C& c) //C::value_type sera alors int, double, etc.
    {
    typename C::value_type somme= C::value_type();  //int() vaut 0 etc.

    for (typename C::const_iterator it= c.begin(); it != c.end(); ++it)
        somme+= *it;

    return somme;
    }
```

Exemples d'utilisation :

```
vector<int> v;
...
int total= Somme(v);

list<complex<double> > l;
...
complex<double> total= Somme(l);

deque<ClArgent> d;  // ClArgent est style DCB
...
ClArgent richesse= Somme(d);
```

Itérateurs : Les types d'itérateurs suivants sont définis dans chaque collection :

```
C::iterator                // Les catégories (possibilités) dépendent du
C::const_iterator          //  type de la collection C
C::reverse_iterator
C::const_reverse_iterator
```

Et on a toujours les fonction suivantes :

```
C::iterator it= c.begin();          // const_iterator si c est const
it= c.end();

C::reverse_iterator itR= c.rbegin(); // const_reverse_iterator si c est const
itR= c.rend();                       // On aurait pu mettre le nom it...
```

N.B. La fonction membre `base()` des (`const_`)`reverse_iterator` renvoie un itérateur non *reverse* sur l'élément suivant :

```
it= itR.base();
--it;
```

met `it` sur le même élément que `itR`. L'itérateur ne pointe pas directement sur le même élément pour que les positions `rbegin()` et `rend()` soient conservées en fonction des itérateurs non renversés.

2- Points communs supplémentaires pour les collections séquentielles : `vector-list-deque`

Construction

```
C<TypeElement> c(n);        //size == n, valeur==TypeElement()
C<TypeElement> c(n,v);      //size == n  valeur==v
```

Réaffectation «similaire» à une affectation selon les possibilités des constructeurs.

```
c.assign(nbFois,valeur);
c.assign(itDebut, itFin);  // Intervalle d'une autre séquence de type quelconque
```

Insertion

```
c.insert(it, valeur);          // Renvoie la position de l'élément ajouté
c.insert(it, nbFois, valeur);  // Les autres insert ne renvoient rien
c.insert(it, itDebut, itFin);  // itDebut,itFin: peuvent provenir d'une autre type de séquence
c.push_back(valeur);           // c.insert(c.end(),valeur);
```

Suppression

```
c.pop_back();  // c.erase(c.rbegin());
```

Taille

```
c.resize(n, valeur); // Augmente ou diminue le nombre d'éléments.
c.resize(n);         // valeur == TypeElement()
```

Accès aux éléments

```
c.back()   // Retourne une référence à la valeur du dernier élément
c.front()  // (la référence est const si c est const)
```

3- Points communs supplémentaires pour les collections associatives : `map-multimap-set-multiset`

Les éléments sont ordonnés selon la clef, par défaut selon l'`operator <` de celle-ci, mais on peut aussi fournir une fonction de comparaison au constructeur (non montré ici). La clef est utilisée pour retrouver la valeur.

Types imbriqués

```
C::key_type  // Type de la clef
```

Insertion

```
c.insert(valeur);     // La valeur est une pair<>(clef,donnée) pour les map
c.insert(it,valeur);
```

Suppression

```
c.erase(clef);
```

Recherche

```
it= c.find(clef);  // Retourne c.end() si non trouvé
nb= c.count(clef);

it= c.lower_bound(clef);      // Renvoie position du premier élément >= clef
it= c.upper_bound(clef);      // Renvoie position du premier élément > clef
pairIt= c.equal_range(clef);  // une pair<> des deux précédents...
```

4- Exemples explicatifs montrant aussi le reste des possibilités des collections spécifiques

4.1 vector

```
// vector.cpp : document d'exemples et d'informations sur <vector>
//              CG et MM fév.98, rév. avril 99, jan.2000
#include <algorithm>
#include <vector>
#include <list>
#include <iostream>
#include <stdcpp>
using namespace std;

template <typename T>
void Afficher(const T& c)
    {
    typedef typename T::const_iterator ClIter;
    // N.B. Sans typename le compilateur doit penser que const_iterator est une donnée car il
    //      ne connaît pas T. VC5 accepte le code sans typename mais ne devrait pas...

    for (ClIter it= c.begin(); it != c.end(); ++it)
        cout << *it << ' ';

    cout << endl;
    }

/*
template<class T, class A = allocator<T> > class vector {...};

    allocator_type · assign · at · back · begin · capacity · clear · const_iterator ·
    const_reference · const_reverse_iterator · difference_type · empty · end · erase ·
    front · get_allocator · insert · iterator · max_size · operator[] · pop_back ·
    push_back · rbegin · reference · rend · reserve · resize · reverse_iterator ·
    size · size_type · swap · value_type · vector

ITÉRATEURS
    Les itérateurs sont de catégorie «random-access».

    N.B. Après des ajouts ou retraits d'éléments, la valeur des variables itérateurs
         devient théoriquement invalide... Les indices restent donc souvent utiles avec les vector !

FONCTIONS SUPPLÉMENTAIRES

    explicit vector(size_type nbFois, const T& valeur = T(), const A& al = A());

        Ex. vector<double> nombres(100, 123.45);
            vector<double> nombres(100);            // init. à double() c-à-d 0.0

    size_type capacity() const; // Nb d'éléments alloués (>=size())
    void reserve(size_type n);  // Réalloue au besoin : size() inchangé, capacity() ajustée

    reference operator[](size_type pos);        // Normalement, sont sans validation
    const_reference operator[](size_type pos);  //

    reference at(size_type pos);                // Validées, peuvent lancer un objet de
    const_reference at(size_type pos) const;    //  type out_of_range
*/

int main()
    {
    // OPÉRATIONS DE BASE
    ///////////////////////
    typedef vector<double> ClVecDouble;
    typedef ClVecDouble::iterator ClIter;
    typedef ClVecDouble::reverse_iterator ClIterRev;

    ClVecDouble v1(3);
    v1[0]= 1.1; v1[1]= 2.2; v1.at(2)= 3.3;

    for (ClIter it= v1.begin(); it != v1.end(); ++it)
        cout << *it << ' ';                             // 1.1 2.2 3.3

    cout << endl;

    for (ClIterRev itR= v1.rbegin(); itR != v1.rend(); ++itR)
        cout << *itR << ' ';                            // 3.3 2.2 1.1

    cout << endl;
```

```
// INSERTION-DESTRUCTION
//////////////////////
ClVecDouble v2;                         // v2.size()==0

v2.push_back(11.11); v2.push_back(22.22);
v2.push_back(77.77); v2.pop_back();
Afficher(v2);                           // v2[0]==11.11  v2[1]==22.22 (v2.size()==2)
v2.assign(4, 7.7);
Afficher(v2);                           // v2[0]==7.7 ... v2[3]==7.7  (v2.size()==4)
v2.assign(v1.begin(), v1.end()-1);
Afficher(v2);                           // v2[0]==1.1  v2[1]==2.2 (v2.size()==2)

v2.insert(v2.begin()+1, 1.6);                    // 1.1  1.6  2.2
v2.insert(v2.end(), 3, 4.4);                     // 1.1  1.6  2.2  4.4  4.4  4.4
v2.insert(v2.end()-1, v1.begin(), v1.begin()+1); // 1.1  1.6  2.2  4.4  4.4  1.1  4.4
         // ou v2.rbegin().base()-1...
Afficher(v2);

v2.erase(v2.end()-1);                           // 1.1  1.6  2.2  4.4  4.4  1.1
v2.erase(v2.begin(), v2.begin()+2);             // 2.2  4.4  4.4  1.1
Afficher(v2);

v2.clear(); // v2 devient vide, v1 contient toujours 1.1  2.2  3.3

// AUTRES FONCTIONS
////////////////////
v1.swap(v2);        // v1 est maintenant vide, v2 contient 1.1  2.2  3.3
v2.swap(v1);        // v2 est maintenant vide, v1 contient 1.1  2.2  3.3

if (v2 < v1) cout << "Une collection vide est «plus petite» qu'une non-vide.\n";

ClVecDouble v3(v1); // v3 contient  1.1  2.2  3.3

if (v1 == v3)
    cout << "Collections identiques, car on a utilisé le CC.\n";
else
    cout << "Erreur.\n";

v1[2]+= 1.0;                    // v1 contient  1.1  2.2  4.3

if (v1 < v3)
    cout << "Erreur.\n";
else
    cout << "Le premier élément différent est plus petit, donc v1 est «plus petit».\n";

// FONCTIONS PERMETTANT DE PASSER DES ÉLÉMENTS D'UNE COLLECTION QUELCONQUE
//////////////////////////////////////////////////////////////////////////
list<double> l;
l.push_back(99.5); l.push_back(99.9); l.push_front(99.1);// l contient 99.1 99.5 99.9
Afficher(l);

ClVecDouble v4(l.begin(), l.end());         // 99.1  99.5  99.9
v4.insert(v4.begin(), l.begin(), l.end());
Afficher(v4);                               // 99.1  99.5  99.9  99.1  99.5  99.9
v4.assign(l.begin(), l.end());
Afficher(v4);                               // 99.1  99.5  99.9

// UTILISATION DE FONCTIONS DE <algorithm>
//////////////////////////////////////////
v4.push_back(10.1); v4.push_back(5.5);
v4.insert(v4.begin()+1, l.begin(), l.end());
Afficher(v4);                           // 99.1 99.1 99.5 99.9 99.5 99.9 10.1 5.5
sort(v4.begin(), v4.end());
Afficher(v4);                           // 5.5 10.1 99.1 99.1 99.5 99.5 99.9 99.9

// EXERCICE : Créer un vector contenant les lettres de l'alphabet en ordre. Le faire
//           imprimer en ordre normal et en ordre inverse, sans utiliser d'indices.
////////////////////////////////////////////////////////////////////////////////////
vector<char> tabAlphabet;  // On ne crée pas un vecteur de 26 éléments initialisés à la
tabAlphabet.reserve(26);   //  mauvaise valeur mais, pour accélérer, on réserve les 26
                           //  char... Sinon réallouerait pendant les push_back...
for (char c= 'a'; c <= 'z'; ++c)
    tabAlphabet.push_back(c);

for (vector<char>::iterator it= tabAlphabet.begin(); it != tabAlphabet.end(); ++it)
    cout << *it;

cout << endl;

for (vector<char>::reverse_iterator it= tabAlphabet.rbegin();
     it != tabAlphabet.rend();
     ++it)
    cout << *it;

cout << endl;
}
```

4.2 list

```
// liste.cpp : document d'exemples et d'informations sur <list>
//             CG et MM fév.98, révision avril 99, jan.2000
#include <list>
#include <iostream>
#include <iomanip>
#include <functional>
#include <algorithm>
#include <string>
#include <stdcpp>
using namespace std;

template <typename T>
void Afficher(const T& c)
    {
    typedef typename T::const_iterator ClIter;

    if (c.empty())
        cout << "(vide)\n";
    else
        {
        for (ClIter it= c.begin(); it != c.end(); ++it)
            cout << fixed << setprecision(1) << *it << ' ';

        cout << endl;
        }
    }

// Fonctions diverses (prédicats)
inline bool EstPair(double p_n)
    {
    int n= static_cast<int>(p_n);

    return n == p_n && n%2 == 0;
    }

inline bool PlusQueSept(double p_v)
    {
    return p_v > 7.0;
    }

inline bool SontProches(double p_v1, double p_v2)
    {
    return abs(p_v1 - p_v2) < 1.0;
    }

inline bool SontEnOrdreDecroissant(const string& p_s1, const string& p_s2)
    {
    return p_s1 > p_s2;
    }

/*
template<class T, class A = allocator<T> > class list {...};

    allocator_type · assign · back · begin · clear · const_iterator ·
    const_reference · const_reverse_iterator · difference_type · empty ·
    end · erase · front · get_allocator · insert · iterator · list · max_size ·
    merge · pop_back · pop_front · push_back · push_front · rbegin · reference ·
    remove · remove_if · rend · resize · reverse · reverse_iterator · size ·
    size_type · sort · splice · swap · unique · value_type

ITÉRATEURS

    Les itérateurs sont bidirectionnels seulement.

FONCTIONS SUPPLÉMENTAIRES

    explicit list(size_type nbFois, const T& valeur = T(), const A& al = A());

    void push_front(const T& x);    // Ajoute au début
    void pop_front();               // Enlève le premier

    // DÉPLACEMENTS D'ÉLÉMENTS ENTRE DEUX LISTES
    // Les splices sont des «move», les éléments de x sont enlevés avant d'être réinsérés.
    void splice(iterator it, list& x);                  // Déplace les éléments de x avant it
    void splice(iterator it, list& x, iterator p);      // Déplace l'élément p de x avant it
    void splice(iterator it, list& x,                   // Déplace la séquence first-last
                iterator first, iterator last);         //  de x avant it
```

```
    // RETRAITS DE DONNÉES SELON UN CRITÈRE
    void remove(const T& x);                        // Enlève les éléments dont la donnée==x
    template<class Pred> void remove_if(Pred pr);   // Enlève les éléments pour lesquels
                                                    //  pr(donnée) est vrai. Normalement
                                                    //  pr est une fonction, une
                                                    //  fonction-objet ou un prédicat de la
                                                    //  librairie, on verra les deux derniers
                                                    //  plus tard...
    void unique();                                  // Enlève les éléments consécutifs dont
                                                    //  les données sont identiques
    template<class BinPred> void unique(BinPred pr);// Enlève les éléments consécutifs pour
                                                    //  lesquels pr(donnée1, donnée2) est vrai
                                                    //  (c'est le deuxième qui sera enlevé)
    // FUSION, DE TRI ET INVERSION
    // Il faut que les données soient triées (selon un certain critère ou operator< par
    // défaut) avant de faire un merge qui sera utile. Les merge sont des «move»...
    void merge(list& x);                // Ajoute les éléments de x dans this
    template <class Comp>               // Ajoute les éléments de x dans this
        void merge(list& x, Comp cmp);  //  cmp est le critère de tri (fonction de comparaison)
    void sort();                        // Trie les éléments
    template<class Comp>                // Trie les éléments en utilisant le
        void sort(Comp cmp);            //  critère fourni par cmp
    void reverse();                     // Renverse l'ordre des éléments.
*/

int main()
    {
    typedef list<double> ClListeDouble;
    typedef ClListeDouble::iterator ClIter;
    typedef ClListeDouble::reverse_iterator ClIterRev;

    ClListeDouble liste1(1);        // init. à 0.0
    liste1.push_back(1.1); liste1.push_back(2.2); liste1.push_back(3.3);

    Afficher(liste1);                                       // 0.0 1.1 2.2 3.3

    for (ClIterRev it= liste1.rbegin(); it!=liste1.rend(); ++it)
        cout << *it << ' ';                                 // 3.3 2.2 1.1 0.0

    cout << endl;

    liste1.resize(3);
    Afficher(liste1);                                       // 0.0 1.1 2.2
    liste1.resize(4);
    Afficher(liste1);                                       // 0.0 1.1 2.2 0.0

    // DÉPLACEMENTS D'ÉLÉMENTS ENTRE DEUX LISTES
    ////////////////////////////////////////////
    ClListeDouble liste2(3, 3.3);
    liste2.push_front(1.1); liste2.push_back(5.5);  // 1.1 3.3 3.3 3.3 5.5

    ClListeDouble liste3(5);                        // 0.0 0.0 0.0 0.0 0.0
    ClIter it= liste3.begin();                      //      ^
    ++it;                                           //      it
    *it= 9.9;                                       // 0.0 9.9 0.0 0.0 0.0
    liste3.splice(it, liste2, liste2.begin());
    Afficher(liste2);                               // 3.3 3.3 3.3 5.5
    Afficher(liste3);                               // 0.0 1.1 9.9 0.0 0.0 0.0
                                                    //              ^
    ++it;                                           //              it
    *it= 7.7;                                       // 0.0 1.1 9.9 7.7 0.0 0.0
    liste3.splice(liste3.begin(), liste3, it, liste3.end());
    Afficher(liste3);                               // 7.7 0.0 0.0 0.0 1.1 9.9
                                                    //      ^
    ++it;                                           //      it
    liste3.splice(it, liste2);
    Afficher(liste2);                               // (vide)
    Afficher(liste3);                               // 7.7 3.3 3.3 3.3 5.5 0.0 0.0 0.0 1.1 9.9

    // RETRAITS DE DONNÉES SELON UN CRITÈRE
    ///////////////////////////////////////
    liste3.push_front(5.9); liste3.push_front(2); liste3.push_front(5.5);
    liste3.remove(5.5); Afficher(liste3);           // 2.0 5.9 7.7 3.3 3.3 3.3 0.0 0.0 0.0 1.1 9.9

    liste3.remove_if(EstPair);
    liste3.remove_if(PlusQueSept);
    Afficher(liste3);                               // 5.9 3.3 3.3 3.3 1.1
    liste3.remove_if(bind2nd(less<double>(), 3.8)); // À voir... (fait comme remove_if(MoinsQue3Point8))
    Afficher(liste3);                               // 5.9

    liste3.insert(liste3.begin(), 3, 6.6);
    liste3.insert(liste3.end(), 2, 6.6);
    liste3.push_front(5.8); liste3.push_front(3.9);
    liste3.push_back(8.8);  liste3.push_back(8.0);
    Afficher(liste3);                               // 3.9 5.8 6.6 6.6 6.6 5.9 6.6 6.6 8.8 8.0
    liste3.unique(); Afficher(liste3);              // 3.9 5.8 6.6 5.9 6.6 8.8 8.0
    liste3.unique(SontProches); Afficher(liste3);   // 3.9 5.8 8.8
```

```
    // FUSION, TRI ET INVERSION
    ////////////////////////////
    list<string> femmes;
    femmes.push_front("Catherine"); femmes.push_front("Céline");

    list<string> hommes;
    hommes.push_front("Michel");    hommes.push_back("Omar");
    hommes.push_front("Michel");    hommes.push_back("Omar");
    hommes.push_front("Richard");   hommes.push_back("Robert");
    Afficher(hommes);

    list<string> h2(hommes);
    list<string> f2(femmes);

    femmes.sort();
    hommes.sort();

    list<string> profs;
    profs.merge(femmes);                    // ou .assign(femmes.begin(), femmes.end())
    profs.merge(hommes);

    Afficher(femmes);                       // (vide)
    Afficher(hommes);                       // (vide)
    Afficher(profs);                        // Catherine Céline Michel Omar Richard Robert

    f2.sort(SontEnOrdreDecroissant);
    h2.sort(greater<string>());  // À voir... (fait comme h2.sort(SontEnOrdreDecroissant);)

    profs.assign(f2.begin(), f2.end());
    profs.merge(h2, SontEnOrdreDecroissant);
    Afficher(f2);                           // Céline Catherine
    Afficher(h2);                           // (vide)
    Afficher(profs);                        // Robert Richard Omar Michel Céline Catherine

    profs.reverse();
    Afficher(profs);                        // Catherine Céline Michel Omar Richard Robert
    }
```

4.3 deque

```
// deque.cpp : document d'exemples et d'informations sur <deque>
//             CG et MM fév.98, révision avril 99, jan.2000
#include <deque>
#include <list>
#include <iostream>
#include <iomanip>
#include <stdcpp>
using namespace std;

template <typename T>
void Afficher(const T& c)
    {
    typedef typename T::const_iterator ClIter;

    if (c.empty())
        cout << "(vide)\n";
    else
        {
        for (ClIter it= c.begin(); it != c.end(); ++it)
            cout << fixed << setprecision(1) << *it << ' ';
        cout << endl;
        }
    }


/*
template<class T, class A = allocator<T> > class deque {...};

    allocator_type · assign · at · back · begin · clear · const_iterator ·
    const_reference · const_reverse_iterator · deque · difference_type · empty ·
    end · erase · front · get_allocator · insert · iterator · max_size · operator[] ·
    pop_back · pop_front · push_back · push_front · rbegin · reference · rend ·
    resize · reverse_iterator · size · size_type · swap · value_type

ITÉRATEURS

    Les itérateurs sont de catégorie «random access».

    N.B. Après des ajouts ou retraits d'éléments, la valeur des variables itérateurs
         devient théoriquement invalide...
```

```
FONCTIONS SUPPLÉMENTAIRES

    explicit deque(size_type nbFois, const T& valeur = T(), const A& al = A());

    reference operator[](size_type pos);        // Normalement, sont sans validation
    const_reference operator[](size_type pos);  //

    reference at(size_type pos);                // Validées, peuvent lancer un objet de
    const_reference at(size_type pos) const;    //  type out_of_range

    void push_front(const T& x);    // Ajoute au début
    void pop_front();               // Enlève le premier
*/

int main()
    {
    deque<double> deq(1);       // init. à 0.0
    deq.push_front(1.1); deq.push_back(2.2); deq.push_back(3.3);

    Afficher(deq);                                          // 1.1 0.0 2.2 3.3

    deq.resize(3);
    deq.resize(4);
    Afficher(deq);                                          // 1.1 0.0 2.2 0.0

    list<int> l;
    for (int i=0; i < 30; ++i)
        l.push_back(i);

    deque<int> d(l.begin(), l.end());
    Afficher(d);
    }
```

4.4 stack, queue et priority_queue

```
// adaptation.cpp : document d'exemples et d'informations sur stack, queue et priority_queue
//                  Par défaut, ce sont des adaptations de deque et vector.
//                  Donne une interface limitée, en particulier pas d'itérateurs.
//                  CG et MM fév.98, révision avril 99, jan.2000.
#include <stack>
#include <queue>
#include <list>
#include <iostream>
#include <iomanip>
#include <string>
#include <stdcpp>
using namespace std;

struct TypeClient
    {
    string nom;
    int age;

    TypeClient(string p_nom= "inconnu", int p_age= -1)
        : nom(p_nom), age(p_age)
        {}
    };

inline bool operator<(const TypeClient& p_c1, const TypeClient& p_c2)
    {
    return p_c1.nom < p_c2.nom;
    }

inline bool operator==(const TypeClient& p_c1, const TypeClient& p_c2)
    {
    return p_c1.nom == p_c2.nom;
    }

/* ************** S T A C K **************
template<class T, class Cont = deque<T> > class stack {...};

    allocator_type · value_type · size_type · stack · empty · size · get_allocator · top · push · pop

    explicit stack(const allocator_type& al=allocator_type());  // Le seul constructeur, donc la
                                                                //  pile est toujours vide au départ.
    bool empty() const;             // Trivial
    size_type size() const;         // Trivial

    value_type& top();              // Consultation ou modification
    const value_type& top() const;  // Consultation seulement, si pile const (rare)
    void pop();                     // Ne renvoie rien, il faut donc utiliser top avant...

    void push(const value_type& x); // Trivial
```

```
    On peut utiliser une collection autre que deque pour conserver les données du stack en
    autant qu'elle supporte les opérations suivantes : empty, size, get_allocator, back,
    push_back, pop_back.
*/

int main()
    {
    stack<int> pileD;

    for (int i= 0; pileD.size()<10; ++i)
        pileD.push(i);

    stack<int, vector<int> > pileV;

    while (!pileD.empty())
        {
        cout << pileD.top() << ' ';         // 9 8 7 6 5 4 3 2 1 0
        pileV.push(pileD.top());
        pileD.pop();
        }

    cout << endl;


/* ************** Q U E U E **************
template<class T, class Cont = deque<T> >  class queue {...};

    allocator_type · value_type · size_type · queue · empty · size · get_allocator ·
    push · pop · front · back

    explicit queue(const allocator_type& al = allocator_type());    // vide au départ...

    bool empty() const;                 // Triviaux...
    size_type size() const;
    value_type& front();
    const value_type& front() const;
    value_type& back();
    const value_type& back() const;

    void push(const value_type& x);     // À la queue... (back)
    void pop();                         // À la tête... (front)

    bool operator==(const queue<T, Cont>& x) const;
    bool operator!=(const queue<T, Cont>& x) const;
    bool operator<(const queue<T, Cont>& x) const;
    bool operator>(const queue<T, Cont>& x) const;
    bool operator<=(const queue<T, Cont>& x) const;
    bool operator>=(const queue<T, Cont>& x) const;

    On peut utiliser une collection autre que deque pour conserver les données de la queue
    en autant qu'elle supporte les opérations suivantes : empty, size, get_allocator,
    front, back, push_back, pop_front.
*/

    queue<int> q;

    while (!pileV.empty())
        {
        q.push(pileV.top());
        pileV.pop();
        }

    queue<int, list<int> > ql;

    while (!q.empty())              // 0 9 1 9 2 9 3 9 4 9 5 9 6 9 7 9 8 9 9 9
        {
        cout << q.front() << ' ' << q.back() << ' ';
        ql.push(q.front());
        q.pop();
        }

    cout << endl;


/* ************** P R I O R I T Y _ Q U E U E **************
template<class T, class Cont = vector<T>, class Comp = less<Cont::value_type> >
    class priority_queue {...};

    Les données sont conservées de façon à extraire la donnée ayant la plus «grande»
    priorité (selon le prédicat). Par défaut, les données les plus «grandes» sont les
    plus prioritaires. Pour remplacer le vector, il faudrait une collection qui permet
    empty, size, front, push_back, pop_back et des itérateurs «random».

    allocator_type · value_type · size_type · priority_queue · get_allocator ·
    empty · size · top · push
```

```
    explicit priority_queue(const Comp& x = Comp(),
                            const allocator_type& A = allocator_type());  // pq vide

    template <typename It>
    priority_queue(It deb, It fin, const Comp& x = Comp(),      // pq copié d'une autre
                   const allocator_type& A = allocator_type()); // séquence

    bool empty() const;
    size_type size() const;
    value_type& top();
    const value_type& top() const;

    void push(const value_type& x);
    void pop();
*/

    priority_queue<int> pq;

    pq.push(8); pq.push(10); pq.push(1); pq.push(4); pq.push(7);

    while (!pq.empty())
        {
        cout << pq.top() << ' ';
        pq.pop();
        }

    cout << endl;

    priority_queue<TypeClient> pqClients;

    pqClients.push(TypeClient("Céline", 1));
    pqClients.push(TypeClient("Catherine", 2));
    pqClients.push(TypeClient("Richard", 3));
    pqClients.push(TypeClient("Robert", 4));
    pqClients.push(TypeClient("Michel", 5));
    pqClients.push(TypeClient("Omar", 6));
    pqClients.push(TypeClient("Richard", 23));
    pqClients.push(TypeClient("Robert", 64));

    while (pqClients.size() > 0)
        {
        cout << pqClients.top().nom << ' ' << pqClients.top().age << endl;
        pqClients.pop();
        }
    // Robert 4
    // Robert 64
    // Richard 23
    // Richard 3
    // Omar 6
    // Michel 5
    // Céline 1
    // Catherine 2
    }
```

4.5 map et multimap

```
// tablasso.cpp : document d'exemples et d'informations sur <map> (et pair de <utility>)
//                Ce document décrit les map et multimap.
//                CG et MM fév.98, révision avril 99, jan.2000
#include <map>
#include <iostream>
#include <iomanip>
#include <functional>
#include <string>
#include <stdcpp>
using namespace std;

template <typename T>
void Afficher(const T& c)
    {
    typedef typename T::const_iterator Iterateur;

    if (c.empty())
        cout << "(vide)\n";
    else
        {
        for (Iterateur it= c.begin(); it != c.end(); ++it)
            cout << fixed << setprecision(1) << *it << ' ';
        cout << endl;
        }
    }
```

```
template <typename T1, typename T2>
ostream& operator<<(ostream& os, pair<T1,T2> p_pair)
    {
    os << '{' << p_pair.first << " # " << p_pair.second << '}';
    return os;
    }

/* ************** P A I R **************
   La classe (struct en fait) std::pair ressemble à :

template<class T, class U>
    struct pair
        {
        typedef T first_type;                   // Les types des deux données
        typedef U second_type                   //

        T first;                                // Les deux données
        U second;                               //

        pair();                                 // Init. par défaut
        pair(const T& x, const U& y);           // Avec init.

        template<class V, class W>
            pair(const pair<V, W>& pr);         // Constructeur permettant les conversions
        };                                      //  d'un autre type de pair dont les éléments
                                                //  sont compatibles.
*/

int main()
    {
    // Classique (et normal)
    struct TypeClient
        {
        int numero;
        string nom;

        TypeClient()
            : numero(0), nom("")
            {}

        TypeClient(int p_no, const string& p_nom)
            : numero(p_no), nom(p_nom)
            {}
        };

    TypeClient cc1; // Init par défaut
    TypeClient cc2(321, "Michel");
    cout << cc2.numero << ' ' << cc2.nom << endl;   // 321 Michel

    // Avec pair (à éviter normalement pour ce genre de traitement)
    typedef pair<int, string> PaireClient;

    PaireClient c1; // Init par défaut (int() et string())
    PaireClient c2(123, "Michel");
    cout << c2.first << ' ' << c2.second << endl;   // 123 Michel

    pair<double, char*> autre(11.99, "Céline");
    PaireClient c3(autre);                          // Conversion
    PaireClient c4(pair<char, char>('1', '!'));
    PaireClient c5(make_pair('2', '!'));            // make_pair est dans la bibliothèque
    cout << c3 << ' ' << c4 << ' ' << c5 << endl;       // {11 # Céline} {49 # !} {50 # !}

/* ************** M A P  E T  M U L T I M A P **************
template<class Key, class T, class Pred = less<Key>, class A = allocator<T> >
    class map {...};

template<class Key, class T, class Pred = less<Key>, class A = allocator<T> >
    class multimap {...};

    Le map est une table associative faisant des relations entre une clé (non
    modifiable) et une donnée (clés uniques). On a l'opérateur [] pour l'accès
    ordinaire et des iterators (bidirectionnels) pour le parcours de toutes
    les données.

    Le multimap est presque identique sauf qu'il permet d'avoir plusieurs données
    associées à la même clé. Il ne supporte pas l'opérateur [].

    allocator_type · begin · clear · const_iterator · const_reference ·
    const_reverse_iterator · count · difference_type · empty · end · equal_range ·
    erase · find · get_allocator · insert · iterator · key_comp · key_compare ·
    key_type · lower_bound · map · max_size · operator[] · rbegin · reference ·
    referent_type · rend · reverse_iterator · size · size_type · swap ·
    upper_bound · value_comp · value_compare · value_type
```

```
TYPES IMPORTANTS

  En plus des types usuels (iterator, size_type, etc.), il y a bien sûr le type
  des clés (key_type) et des données associées (mapped_type). En plus, on a la
  combinaison de ces deux éléments dans une pair<const key_type, mapped_type> pour le
  value_type (on peut modifier les données associées, mais pas la clé).

  Pour les maps, le mapped_type doit permettre la construction sans paramètre (car
  l'insertion avec [] met d'abord une donnée qui sera initialisée à la valeur par défaut).

FONCTIONS SUPPLÉMENTAIRES

    // CONSTRUCTEURS
    explicit map(const Pred& comp = Pred(), const A& al = A()); // Map vide
    template <class It>
        map(It first, It last,                                  // À partir d'une
            const Pred& comp = Pred(), const A& al = A());      //  séquence de paires

    Idem pour les multimaps...

    // MAP SEULEMENT : ACCÈS «DIRECT» AUX ÉLÉMENTS ET INSERTION
    mapped_type& operator[](const Key& key);    // Si la clé n'est pas trouvée, un élément
                                                //  (clé+donnée par défaut) est inséré.
                                                //  La référence à la donnée associée
                                                //  (pré-existante ou ajoutée) est renvoyée.
    pair<iterator, bool> insert(const value_type& x);   // Ajoute la paire clé+donnée si
                                                        //  la clé n'existe pas déjà.
                                                        //  Le bool de la paire renvoyée
                                                        //  indique si l'élément a été ajouté.
    // MULTIMAP SEULEMENT : INSERTION
    iterator insert(const value_type& x);   // Ajoute la paire clé+donnée (retourne
                                            //  la position de l'élément ajouté).

    // AUTRES INSERTIONS ET RETRAITS (MAP ET MULTIMAP)
    iterator insert(iterator it, const value_type& x);  // On passe un iterator où
                                                        //  débutera la recherche pour
                                                        //  la position d'insertion (faut
                                                        //  donner une valeur utile...)
    template <typename It>
        void insert(It first, It last);     // Insère une séquence de paires

    size_type erase(const Key& key);        // Renvoie le nombre d'éléments enlevés (0 ou 1
                                            //  pour les maps)

    // FONCTIONS RETOURNANT DES ITÉRATEURS
    begin, end, rbegin, rend renvoie des itérateurs sur des paires...

    // RECHERCHE (MAP ET MULTIMAP)
    iterator find(const Key& key);                  // Renvoie end() si non trouvé
    const_iterator find(const Key& key) const;

    // RECHERCHES STYLE MULTIMAP (MAP ET MULTIMAP)
    size_type count(const Key& key) const;              // Renvoie 0 ou 1 dans un map...

    iterator lower_bound(const Key& key);               // Renvoie la position du premier
    const_iterator lower_bound(const Key& key) const;   //  élément >= clé (ou end() si aucun)

    iterator upper_bound(const Key& key);               // Renvoie la position du premier
    const_iterator upper_bound(const Key& key) const;   //  élément > clé (ou end() si aucun)

    pair<iterator, iterator> equal_range(const Key& key);   // Renvoie la paire formée
    pair<const_iterator, const_iterator>                    //  du lower_bound et upper_bound
        equal_range(const Key& key) const;

    // AUTRES...
    key_compare key_comp() const;           // Retournent des fonction-objets permettant
    value_compare value_comp() const;       //  la comparaison des clés/des paires.
*/

    typedef pair<string, unsigned long> PaireInscription;   // unsigned long permet
    typedef map<string, unsigned long> ClBottin;            //  au moins 9 chiffres... (!)
    typedef multimap<string, unsigned long> ClListeTel;

    ClBottin b;

    b["Michel"]= 1234567;
    b["Céline"]= 2361;
    b["Robert"]= 2265;

    PaireInscription insc("Céline", 3475301);       // Ne sera pas changé car existe déjà

    if (b.insert(insc).second)
        cout << "Pas normal, Céline aurait dû être déjà inscrite.\n";

    b["Michel"]= 2545;                          // Remplace l'autre
```

```
    pair<ClBottin::iterator, bool> codeRetour= b.insert(make_pair(string("Catherine"), 2265));

    if (!codeRetour.second)
        cout << "Ça va pas ? Pourquoi ?? On veut Catherine !!!\n";

    ClBottin::iterator it= b.insert(b.begin(), PaireInscription("Omar", 2529));
    b.insert(it, PaireInscription("Omar", 0));      // Pas remplacé, on pourrait tester la
                                                    //  valeur de retour.second
    Afficher(b);
    // {Catherine # 2265} {Céline # 2361} {Michel # 2545} {Omar # 2529} {Robert # 2265}


    ClListeTel t;

    t.insert(make_pair(string("Michel"), 1114567));
    t.insert(make_pair(string("Céline"), 2224567));
    t.insert(make_pair(string("Robert"), 3334567));
    t.insert(make_pair(string("Catherine"), 2265));
    t.insert(make_pair(string("Michel"), 2545));
    ClListeTel::iterator it2= t.insert(t.begin(), make_pair(string("Céline"), 2361));
    t.insert(it2, PaireInscription("Omar", 2529));

    Afficher(t);
    // {Catherine # 2265} {Céline # 2224567} {Céline # 2361} {Michel # 1114567}
    // {Michel # 2545} {Omar # 2529} {Robert # 3334567}


    if (t.end() != t.find("Omar"))
        cout << "Il y a " << t.count("Omar") << " inscription(s) pour Omar.\n";

    pair<ClListeTel::iterator, ClListeTel::iterator> lesCelines= t.equal_range("Céline");

    lesCelines.first->second= 3334567;  // mais lesCelines.first->first est const...

    for (; lesCelines.first != lesCelines.second; ++lesCelines.first)
        cout << *lesCelines.first << endl;              // {Céline # 3334567} {Céline # 2361}

    cout << "On efface les " << t.erase("Michel") << " Michel.\n";  // On efface les 2 Michel
    Afficher(t);
    // {Catherine # 2265} {Céline # 3334567} {Céline # 2361} {Omar # 2529} {Robert # 3334567}
    }
```

4.6 set et multiset

```
// ensemble.cpp : document d'exemples et d'informations sur <set> (set et multiset)
//                CG et MM fév.98, révision avril 99
#include <set>
#include <iostream>
#include <iomanip>
#include <functional>
#include <string>
#include <stdcpp>
using namespace std;

template <typename T>
void Afficher(const T& c)
    {
    typedef typename T::const_iterator Iterateur;

    if (c.empty())
        cout << "(vide)\n";
    else
        {
        for (Iterateur it= c.begin(); it != c.end(); ++it)
            cout << fixed << setprecision(1) << *it << ' ';
        cout << endl;
        }
    }

/*
template<class Key, class Pred = less<Key>, class A = allocator<T> >
    class set {...};

template<class Key, class Pred = less<Key>, class A = allocator<T> >
    class multiset {...};

    Le set/multiset est un peu comme un map/multimap sans donnée associée à leurs clés.
    Les clés sont les valeurs conservées. On n'a pas d'opérateur [].

    allocator_type · begin · clear · const_iterator · const_reference ·
    const_reverse_iterator · count · difference_type · empty · end · equal_range ·
    erase · find · get_allocator · insert · iterator · key_comp · key_compare ·
    key_type · lower_bound · max_size · rbegin · reference · rend · reverse_iterator ·
    set · size · size_type · swap · upper_bound · value_comp · value_compare · value_type
```

```
TYPES IMPORTANTS

   Comme dans map, on a le type key_type et value_type, mais ils sont équivalents, car il n'y a pas de
   mapped_type. Les iterators sont comme les const_iterator, car on ne peut pas changer les données qui
   sont considérées comme des clés (si on veut en changer une partie, on peut retirer puis ajouter...).

CONSTRUCTEURS

   explicit set(const Pred& comp = Pred(), const A& al = A());  // set vide

   template <class It>
        set(It first, It last,                              // À partir d'une
            const Pred& comp = Pred(), const A& al = A());  //  séquence de données

   Idem pour multiset...

SET SEULEMENT : INSERTION VALIDÉ

    pair<iterator, bool> insert(const value_type& x);   // Ajoute la donnée (==clé) si
                                                        //  la clé n'existe pas déjà.
                                                        //  Le bool de la paire renvoyée
                                                        //  indique si élément ajouté.
MULTISET SEULEMENT : INSERTION

    iterator insert(const value_type& x);               // Ajoute la donnée (==clé)

AUTRES INSERTIONS (SET ET MULTISET)

    iterator insert(iterator it, const value_type& x);      // Mêmes principes que pour les
    template <typename It> void insert(It first, It last);  //  map/multimap.

RETRAITS (SET ET MULTISET)

    iterator erase(iterator it);                    // Voir autres collections
    iterator erase(iterator first, iterator last);  //  pour ceux-ci.

    size_type erase(const Key& key);        // Renvoie le nombre d'éléments enlevés (0 ou 1
                                            //  pour les sets)
    void clear();   // Trivial

RECHERCHE (SET ET MULTISET)

    iterator find(const Key& key);                  // Renvoie end() si non trouvé
    const_iterator find(const Key& key) const;

RECHERCHES STYLE MULTIMAP (SET ET MULTISET)

    size_type count(const Key& key) const;              // Renvoie 0 ou 1 dans un set...

    iterator lower_bound(const Key& key);               // Renvoie la position du premier
    const_iterator lower_bound(const Key& key) const;   //  élément >= clé (ou end() si aucun)

    iterator upper_bound(const Key& key);               // Renvoie la position du premier
    const_iterator upper_bound(const Key& key) const;   //  élément > clé (ou end() si aucun)

    pair<iterator, iterator> equal_range(const Key& key);   // Renvoie la paire formée
    pair<const_iterator, const_iterator>                    //  du lower_bound et upper_bound
        equal_range(const Key& key) const;
*/

int main()
    {
    typedef set<string> ClProfs;

    ClProfs profsInfo;

    profsInfo.insert("Céline");
    profsInfo.insert("Michel");
    profsInfo.insert("Catherine");

    if (profsInfo.insert("Michel").second) cout << "Pas normal, une seul Michel dans set\n";

    profsInfo.insert("Omar");
    profsInfo.insert("Richard");
    profsInfo.insert("Robert");

    Afficher(profsInfo);


    multiset<string> profs(profsInfo.begin(), profsInfo.end());

    profs.insert("Michel");

    Afficher(profs);
    }
```