Source: http://www.cplusplus.com/reference/stl/

			Sequence containers			Associative containers				
Headers			**<vector>**	**<deque>**	**<list>**	**<set>**		**<map>**		**<bitset>**
	Members	complex	**vector**	**deque**	**list**	**set**	**multiset**	**map**	**multimap**	**bitset**
	constructor	*	constructor	constructor	constructor	constructor	constructor	constructor	constructor	constructor
	destructor	O(n)	destructor	destructor	destructor	destructor	destructor	destructor	destructor	
	operator=	O(n)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operators
iterators	begin	O(1)	**begin**	**begin**	**begin**	**begin**	**begin**	**begin**	**begin**	
	end	O(1)	**end**	**end**	**end**	**end**	**end**	**end**	**end**	
	rbegin	O(1)	rbegin	rbegin	rbegin	rbegin	rbegin	rbegin	rbegin	
	rend	O(1)	rend	rend	rend	rend	rend	rend	rend	
capacity	size	*	**size**	**size**	**size**	**size**	**size**	**size**	**size**	**size**
	max_size	*	**max_size**	**max_size**	**max_size**	**max_size**	**max_size**	**max_size**	**max_size**	
	empty	O(1)	empty	empty	empty	empty	empty	empty	empty	
	resize	O(n)	resize	resize	resize					
element access	front	O(1)	front	front	front					
	back	O(1)	back	back	back					
	operator[]	*	**operator[]**	**operator[]**				**operator[]**		**operator[]**
	at	O(1)	**at**	**at**						
modifiers	assign	O(n)	assign	assign	assign					
	insert	*	**insert**	**insert**	**insert**	**insert**	**insert**	**insert**	**insert**	
	erase	*	**erase**	**erase**	**erase**	**erase**	**erase**	**erase**	**erase**	
	swap	O(1)	swap	swap	swap	swap	swap	swap	swap	
	clear	O(n)	clear	clear	clear	clear	clear	clear	clear	
	push_front	O(1)		**push_front**	**push_front**					
	pop_front	O(1)		**pop_front**	**pop_front**					
	push_back	O(1)	**push_back**	**push_back**	**push_back**					
	pop_back	O(1)	**pop_back**	**pop_back**	**pop_back**					
observers	key_comp	O(1)				key_comp	key_comp	key_comp	key_comp	
	value_comp	O(1)				value_comp	value_comp	value_comp	value_comp	
operations	find	O(log n)				**find**	**find**	**find**	**find**	
	count	O(log n)				**count**	**count**	**count**	**count**	**count**
	lower_bound	O(log n)				lower_bound	lower_bound	lower_bound	lower_bound	
	upper_bound	O(log n)				upper_bound	upper_bound	upper_bound	upper_bound	
	equal_range	O(log n)				equal_range	equal_range	equal_range	equal_range	
unique members			**capacity** **reserve**		splice **remove** **remove_if** unique merge **sort** **reverse**					set reset flip to_ulong to_string test any none

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

int main (){
        vector<int> myVector;

//value insertion: use push_back and pop_back, if possible
//attn: be careful with insert and erase, esp. when used with iterators
        for (int i=0; i<5; i++)
                myVector.push_back(3*i+17);

//sizes
        int size = myVector.size();
        cout << "My vector has "<< size << " elements,\n can grow up to " << myVector.max_size() << " elements, and \n
currently reserves memory for " << myVector.capacity() << " elements\n" << endl;

//Iterating all the members of the vector with an iterator
        cout <<"The vector contains: " <<endl;
        vector<int>::iterator it;
        for (it = myVector.begin(); it < myVector.end(); it++)
                cout << " " << *it;                         //observe that the iterator POINTS to an object
        cout << endl;

//observe the at() and [] operator, doing the same job
        cout << "Middle element is: " << myVector.at(size/2) << endl;
        cout << "Last element inserted: " << myVector[size-1] << endl;
        return 0;
}

/*
// reserve() alters the capacity of the vector
//try adding this early enough, and check the diagnostics on size
        myVector.reserve(12);

//resize alters the size:
//if new size is less than the existing, it kills elements; else it adds and you can tell it what to put in the new slots (here: value
155)
        myVector.resize(2);
        myVector.resize(30, 155);

*/
```

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

//to sort DESC -- by default sort sorts ASC
bool compareForDescendingSort(int a, int b){ if(a>b)return true; else return false;}

void print(list<int> & aList){
                list<int>::iterator it;
                for (it=aList.begin(); it != aList.end(); it++)  //ATTN: != myList.end() instead of < myList.end()
                        cout << " " << *it;
                cout << endl;
}

int main (){
        list<int> myList;
        list<int>::iterator it;
        myList.push_front(12);   myList.push_front(145);
        myList.push_back(10);    myList.push_front(148); myList.push_back(146);   //148 145 12 10 146

//sizes and printouts
        int size = myList.size();
        cout << "My list has "<< size << " elements,\n can grow up to " << myList.max_size() << " elements"<< endl;
        cout <<"The list contains: " <<endl;
        print(myList);

//NO possibility for at() and [] operator, got to DIY. Let's find who is at position 3.
        int position = 3; it=myList.begin();
        for (int i=0; i<position; i++)
                it++;
        cout <<"In position " << position << " we find element: " << *it << endl ;

//Now let's delete some stuff and insert some more
//kill the element pointed to by the iterator. ATTN: assignment (it = ...) is obligatory, else, you lose the iterator
        it = myList.erase(it);
        it--; it--;
//move back 2 positions: now the list is: 148 145 12 146 and we point at 145 with it
        myList.insert(it, 155);  //insert a new one between 148 and 145
//For you: check out remove() and remove_if() and merge()

        cout<< "\n... and the list is now\n";
        print(myList);

//sort
        cout<< "\n... and now I can reverse it\n";
        myList.reverse();
        print(myList);
        cout<< "\n... and if we sort, the list is now (luckily for types supporting <, we need no extra function)\n";
        myList.sort();
        print(myList);
        cout<< "\n... still, if we have to use our own function, e.g., to sort descending\n";
        myList.sort(compareForDescendingSort);
        print(myList);

        return 0;
}
```

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

void print(set<int> & aSet){
                set<int>::iterator it;
                for (it=aSet.begin(); it != aSet.end(); it++)
                        cout << " " << *it;
                cout << endl;
}

int main (){
        set<int> mySet;
        set<int>::iterator it;

//insert & delete
        for (int i=0; i<7; i++)
                mySet.insert(2*i+1);
        mySet.erase(7);

//sizes and printouts
        int size = mySet.size();
        cout << "My set has "<< size << " elements,\n can grow up to " << mySet.max_size() << " elements"<< endl;
        cout <<"The set contains: " <<endl;
        print(mySet);

//find stuff
        cout << "\n--------FINDERS---------------\n";
        int searchKey1 = 45; int searchKey2 = 3;
//0 if searchKey does not belong to the set, 1 if it does
        cout << "Num. occurences of " << searchKey1 << " is " << mySet.count(searchKey1) << endl;
        cout << "Num. occurences of " << searchKey2 << " is " << mySet.count(searchKey2) << endl;

        it=mySet.find(searchKey1);
        if (it == mySet.end())
                cout << "Could not find the searchKey\n\n";

//some more insert and delete
        it=mySet.begin();
        if ((it=mySet.find(searchKey2)) != mySet.end())
                mySet.erase (it);
        mySet.insert(12);
//does not matter where iterator is, a set has no positions
//try inserting sth that already exists in the set, e.g., 1 or 9 and see what happens
        cout <<"The set contains: " <<endl;
        print(mySet);

//inverse iteration, holds for all containers, use r(everse)begin/end
        set<int>::reverse_iterator rit;
        cout << "\nInverse iteration:";
        for ( rit=mySet.rbegin() ; rit != mySet.rend(); rit++ )
            cout << " " << *rit;
        cout << endl;

        return 0;
}
```

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

class Pebble{
 public:
        Pebble(const int & anId, const int &l, const int & h){id = anId; low = l; high = h;}
        int hasInt(const int & anInt){if ((low==anInt)||(high == anInt)) return 1; else return 0;}
        void showPebble(){cout <<"Pebble: " << id << "," << low << "," << high << endl;}
 private:
        int id;
        int low;
        int high;
};

class PebbleMgr{
        public:
                Pebble * PebbleMgr::findPebbleHavingInt(const int & anInt);
                void addPebble(const int & anId, const int &l, const int & h);
                int size(){return pebbles.size();}
                Pebble * at(const int & pos){if (pos<size()) return &pebbles[pos]; else return NULL;}
                void printPebbles();
        private:
                vector<Pebble> pebbles;
};

void PebbleMgr::addPebble(const int & anId, const int &l, const int & h){
        Pebble newPebble(anId, l, h);
        pebbles.push_back(newPebble);
}

Pebble * PebbleMgr::findPebbleHavingInt(const int & anInt){
        vector<Pebble>::iterator it;
        for (it = pebbles.begin(); it < pebbles.end(); it++)
                if ((*it).hasInt(anInt))
                        return &(*it);           //ATTN: here, it != &(*it)
        //the for loop stops whenever one good pebble is found. if it is not interrupted,
        //the next stmnt to fire returns NULL as an indication of not found
        //This is why we need to return Pebble * and not Pebble &
        return NULL;
}

void PebbleMgr::printPebbles(){
        vector<Pebble>::iterator it;
        for (it = pebbles.begin(); it != pebbles.end(); it++ )
                (*it).showPebble();
}

int main (){
        PebbleMgr engine;

        engine.addPebble(0,0,0); engine.addPebble(1,0,1); engine.addPebble(2,1,1); engine.addPebble(3,0,2);
        for (int i = 0; i< engine.size(); i++)   //equivalent: engine.printPebbles();
                engine.at(i)->showPebble();


        cout << endl<< "Gonna find the 1st pebble that includes the searchKey\n";
        int searchKey = 1;
        Pebble * ptr = engine.findPebbleHavingInt(searchKey);
        if (ptr !=NULL)
                ptr->showPebble();
        else
                cout << "Search key " << searchKey << " not found\n";
        return 0;
}
```

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

class Pebble{
 public:
        Pebble(const int & anId, const int &l, const int & h){id = anId; low = l; high = h;}
        int hasInt(const int & anInt){if ((low==anInt)||(high == anInt)) return 1; else return 0;}
        void showPebble(){cout <<"Pebble: " << id << "," << low << "," << high << endl;}
 private:
        int id;
        int low;
        int high;
};

class PebbleMgr{
        public:
                Pebble * PebbleMgr::findPebbleHavingInt(const int & anInt);
                void addPebble(const int & anId, const int &l, const int & h);
                int size(){return pebbles.size();}
                //Pebble * at(const int & pos){if (pos<size()) return &pebbles[pos]; else return NULL;}
                void printPebbles();
        private:
                list<Pebble> pebbles;
};

void PebbleMgr::addPebble(const int & anId, const int &l, const int & h){
        Pebble newPebble(anId, l, h);
        pebbles.push_back(newPebble);
}

Pebble * PebbleMgr::findPebbleHavingInt(const int & anInt){
        list<Pebble>::iterator it;
        for (it = pebbles.begin(); it != pebbles.end(); it++)
                if ((*it).hasInt(anInt))
                        return &(*it);           //ATTN: here, it != &(*it)
        //the for loop stops whenever one good pebble is found. if it is not interrupted,
        //the next stmnt to fire returns NULL as an indication of not found
        //This is why we need to return Pebble * and not Pebble &
        return NULL;
}

void PebbleMgr::printPebbles(){
        list<Pebble>::iterator it;
        for (it = pebbles.begin(); it != pebbles.end(); it++ )
                (*it).showPebble();
}

int main (){
        PebbleMgr engine;

        engine.addPebble(0,0,0); engine.addPebble(1,0,1); engine.addPebble(2,1,1); engine.addPebble(3,0,2);
        engine.printPebbles();

        cout << endl<< "Gonna find the 1st pebble that includes the searchKey\n";
        int searchKey = 1;
        Pebble * ptr = engine.findPebbleHavingInt(searchKey);
        if (ptr !=NULL)
                ptr->showPebble();
        else
                cout << "Search key " << searchKey << " not found\n";
        return 0;
}
```

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

class Pebble{
 public:
        Pebble(const int & anId, const int &l, const int & h){id = anId; low = l; high = h;}
        int hasInt(const int & anInt){if ((low==anInt)||(high == anInt)) return 1; else return 0;}
        void showPebble(){cout <<"Pebble: " << id << "," << low << "," << high << endl;}
 private:
        int id;
        int low;
        int high;
};

class PebbleMgr{
        public:
                Pebble * findPebbleHavingInt(const int & anInt);
                set<Pebble *> findAllPebblesHavingInt(const int & anInt);
                void addPebble(const int & anId, const int &l, const int & h);
                int size(){return pebbles.size();}
                void printPebbles();
        private:
                vector<Pebble> pebbles;
};

void PebbleMgr::addPebble(const int & anId, const int &l, const int & h){
        Pebble newPebble(anId, l, h);
        pebbles.push_back(newPebble);
}

Pebble * PebbleMgr::findPebbleHavingInt(const int & anInt){
        vector<Pebble>::iterator it;
        for (it = pebbles.begin(); it < pebbles.end(); it++)
                if ((*it).hasInt(anInt))
                        return &(*it);           //ATTN: here, it != &(*it)
        return NULL;
}

set<Pebble *> PebbleMgr::findAllPebblesHavingInt(const int & anInt){
        vector<Pebble>::iterator it;
        set<Pebble *> result;
        for (it = pebbles.begin(); it != pebbles.end(); it++)
                if ((*it).hasInt(anInt))
                        result.insert(&(*it));
        return result;
}

void PebbleMgr::printPebbles(){
        vector<Pebble>::iterator it;
        for (it = pebbles.begin(); it != pebbles.end(); it++ )
                (*it).showPebble();
}

int main (){
        PebbleMgr engine;

        engine.addPebble(0,0,0); engine.addPebble(1,0,1); engine.addPebble(2,1,1); engine.addPebble(3,0,2);
        engine.printPebbles();

        cout << endl<< "Gonna find the 1st pebble that includes the searchKey\n";
        int searchKey = 1;
        Pebble * ptr = engine.findPebbleHavingInt(searchKey);
        if (ptr !=NULL) ptr->showPebble();
        else cout << "Search key " << searchKey << " not found\n";

        cout << endl<< "Gonna find the ALL pebbles that includes the searchKey\n";
        set<Pebble *> allPAddresses = engine.findAllPebblesHavingInt(searchKey);
        if (allPAddresses.empty())
                cout << "Search key " << searchKey << " not found\n";
        else{
                set<Pebble *>::iterator itP;
                for (itP = allPAddresses.begin(); itP != allPAddresses.end(); itP++ )
                (*itP)->showPebble();
        }
        return 0;
}
```