C++ Basics

Structs and Class

In vanilla C, structs are collections of variables, originally created for custom typing. In C++, structs and classes are the same and have the same function as any other object oriented language. The only difference between the two is that in structs, variables and methods are by default public and in classes they are by default private. This basically means that for the purpose of competitive programming we almost always want to use structs.
The following is an example of a struct used to keep tract of a product on sale:

```
// this is a declaration of a struct
// it follows the general pattern of
// struct <struct_name> {
//     <member_type_1> <member_name_1>;
//     <member_type_2> <member_name_2>;
//     . . . . . .
//     <method_type_1> <method_name_1>(<params>) {<implementation>};
// };
struct product {
    string name;
    double price;
    bool avaliable;

    // since structs and classes are the same, we can define constructors
    product(string a_name, double a_price) {
        name = a_name;
        price = a_price;
        avaliable = true;
    };

    // we can also define our own comparison operators, default behavior
    // is very glitchy so it is recommended if you want to sort an array
    // of this type to define your own comparison
    bool operator < (const product &B) const {
        return price < B.price;
    };
};

// to create an element of the type product you can use a constructor
product apple("apple", 6.5);
// or in c++11 you can initialize any struct with list initialization
// by default the order of arguments provided is the order of struct
// members in the definition, but you can also specify
// omitted values are given the default value of the type
product banana = {"banana", 3.33, true}; // <- for our purposes use this
product orange = {.price = 3.33, .name = "orange", .avaliable = true};
```

Lambda Expression

Lambda expressions are inline anonymous functions especially helpful when using certain functions in `algorithms` when doing special comparison. The following example is an implementation of something similar to filter in cpp:

```
#define T 5

void filterAbove(vector<int> &v) {
    // the syntax for a lambda expression is
    // [<capture group>](<argument list>) -> <ret type> { <code> }
    // the return type along with -> are often omitted for simple code
    // because the compiler can guess the return type.
    // the capture group is what the lambda function ‘‘captures’’ from the
    // surrounding namespace, for competitive programming purposes leave
    // it empty and just use #define statements up top.
    transform(v.begin(), v.end(), [](double d) { return d > T ? d : 0 });
}
```

Pointers

Don't use them on purpose... Just know that `*p` dereferences a pointer and `p+1` accesses the address at `p+` the size of the type of pointer that `p` is.

C++ Containers

vectors

Constructors		
`vector<T>v;`	Make an empty vector.	O(1)
`vector<T>v(n);`	Make a vector with N elements.	O(n)
`vector<T>v(n, value);`	Make a vector with N elements, initialized to value.	O(n)
`vector<T>v(begin, end);`	Make a vector and copy the elements from begin to end.	O(n)
Accessors		
`v[i];`	Return (or set) the I'th element.	O(1)
`v.at(i);`	Return (or set) the I'th element, with bounds checking.	O(1)
`v.size();`	Return current number of elements.	O(1)
`v.empty();`	Return true if vector is empty.	O(1)
`v.begin();`	Return random access iterator to start.	O(1)
`v.end();`	Return random access iterator to end.	O(1)
`v.front();`	Return the first element.	O(1)
`v.back();`	Return the last element.	O(1)
`v.capacity();`	Return maximum number of elements.	O(1)
Modifiers		
`v.push_back(value);`	Add value to end.	O(1)*
`v.insert(iterator, value);`	Insert value at the position indexed by iterator.	O(n)
`v.pop_back();`	Remove value from end.	O(1)
`v.erase(iterator);`	Erase value indexed by iterator.	O(n)
`v.erase(begin, end);`	Erase the elements from begin to end.	O(n)

lists

Constructors		
`list<T>l;`	Make an empty list.	O(1)
`list<T>l(begin, end);`	Make a list and copy the values from begin to end.	O(n)
Accessors		
`l.size();`	Return current number of elements.	O(1)
`l.empty();`	Return true if list is empty.	O(1)
`l.begin();`	Return bidirectional iterator to start.	O(1)
`l.end();`	Return bidirectional iterator to end.	O(1)
`l.front();`	Return the first element.	O(1)
`l.back();`	Return the last element.	O(1)
Modifiers		
`l.push_front(value);`	Add value to front.	O(1)
`l.push_back(value);`	Add value to end.	O(1)
`l.insert(iterator, value);`	Insert value after position indexed by iterator.	O(1)
`l.pop_front();`	Remove value from front.	O(1)
`l.pop_back();`	Remove value from end.	O(1)
`l.erase(iterator);`	Erase value indexed by iterator.	O(1)
`l.erase(begin, end);`	Erase the elements from begin to end.	O(1)
`l.remove(value);`	Remove all occurrences of value.	O(n)
`l.remove_if(test);`	Remove all element that satisfy test.	O(n)
`l.reverse();`	Reverse the list.	O(n)
`l.sort();`	Sort the list.	O(n log n)
`l.sort(comparison);`	Sort with comparison function.	O(n log n)
`l.merge(l2);`	Merge sorted lists.	O(n)

deques

Constructors		
`deque<T>d;`	Make an empty deque.	O(1)
`deque<T>d(n);`	Make a deque with N elements.	O(n)
`deque<T>d(n, value);`	Make a deque with N elements, initialized to value.	O(n)
`deque<T>d(begin, end);`	Make a deque and copy the values from begin to end.	O(n)
Accessors		
`d[i];`	Return (or set) the I'th element.	O(1)
`d.at(i);`	Return (or set) the I'th element, with bounds checking.	O(1)

`d.size();`	Return current number of elements.	O(1)
`d.empty();`	Return true if deque is empty.	O(1)
`d.begin();`	Return random access iterator to start.	O(1)
`d.end();`	Return random access iterator to end.	O(1)
`d.front();`	Return the first element.	O(1)
`d.back();`	Return the last element.	O(1)
Modifiers		
`d.push_front(value);`	Add value to front.	O(1)*
`d.push_back(value);`	Add value to end.	O(1)*
`d.insert(iterator, value);`	Insert value at the position indexed by iterator.	O(n)
`d.pop_front();`	Remove value from front.	O(1)
`d.pop_back();`	Remove value from end.	O(1)
`d.erase(iterator);`	Erase value indexed by iterator.	O(n)
`d.erase(begin, end);`	Erase the elements from begin to end.	O(n)

stacks and queues

In C++ STL, stacks and queues are *container adaptors*, so they are created from another container (one of the ones listed above, default to vector).

Constructors		
`stack< container<T> > s;`	Make an empty stack.	O(1)
`queue< container<T> > q;`	Make an empty queue.	O(1)
Accessors		
`s.top(); q.front(); q.back()`	Returns the top/front/back element.	O(1)
`s.size(); q.size();`	Returns current number of elements	O(1)
`s.empty(); q.empty();`	Returns true if empty	O(1)
Modifiers		
`s.push(v); q.push(v)`	Push value to top/end	varies
`s.pop(); q.pop()`	Removes value from the top/front	O(1)

Note that the complexity of `push` varies depending on the underlying container. But remember that it is always equal to the complexity of `push_back` for the underlying container.

priority queues

This is also a *container adaptor*. Except the element on top is irrelevent of order of insertion. Instead the "biggest" element is on top. Biggest is determined by the comparison predicate you give the priority queue constructor.

- If that predicate is a "less than" type predicate, then biggest means largest.
- If it is a "greater than" type predicate, then biggest means smallest.

Again, the default container is a vector, the constructor is

```
priority_queue<T, container<T>, comparison<T> > q;
```

and the complexity for push and pop are both O(n log n)

sets and multisets

Constructors		
`set<T, compare> s;`	Make an empty set. compare should be a binary predicate for ordering the set. It's optional and will default to a function that uses `operator<`.	O(1)
`set<T, compare> s(begin, end);`	Make a set and copy the values from begin to end.	O(n log n)
Accessors		
`s.find(key)`	Return an iterator pointing to an occurrence of key in s, or s.end() if key is not in s.	O(log n)
`s.lower_bound(key)`	Return an iterator pointing to the first occurrence of an item in s not less than key, or s.end() if no such item is found.	O(log n)
`s.upper_bound(key)`	Return an iterator pointing to the first occurrence of an item greater than key in s, or s.end() if no such item is found.	O(log n)
`s.equal_range(key)`	Returns pair_bound(key), upper_bound(key)>.	O(log n)
`s.count(key)`	Returns the number of items equal to key in s.	O(log n)
`s.size();`	Return current number of elements.	O(1)
`s.empty();`	Return true if set is empty.	O(1)
`s.begin()`	Return an iterator pointing to the first element.	O(1)
`s.end()`	Return an iterator pointing one past the last element.	O(1)
Modifiers		
`s.insert(iterator, key)`	Inserts key into s. iterator is taken as a "hint" but key will go in the correct position no matter what. Returns an iterator pointing to where key went.	O(log n)
`s.insert(key)`	Inserts key into s and returns a pair<iterator, bool>, where iterator is where key went and bool is true if key was actually inserted, i.e., was not already in the set.	O(log n)

maps and multimaps

Constructors		
`map<key_type, value_type, key_compare> m;`	Make an empty map. key_compare should be a binary predicate for ordering the keys. It's optional and will default to a function that uses operator<.	O(1)
`map<key_type, value_type, key_compare> m(begin, end);`	Make a map and copy the values from begin to end.	O(n log n)
Accessors		
`m[key]`	Return the value stored for key. This adds a default value if key not in map.	O(log n)
`m.find(key)`	Return an iterator pointing to a key-value pair, or m.end() if key is not in map.	O(log n)
`m.lower_bound(key)`	Return an iterator pointing to the first pair containing key, or m.end() if key is not in map.	O(log n)
`m.upper_bound(key)`	Return an iterator pointing one past the last pair containing key, or m.end() if key is not in map.	O(log n)
`m.equal_range(key)`	Return a pair containing the lower and upper bounds for key. This may be more efficient than calling those functions separately.	O(log n)
`m.size();`	Return current number of elements.	O(1)
`m.empty();`	Return true if map is empty.	O(1)
`m.begin()`	Return an iterator pointing to the first pair.	O(1)
`m.end()`	Return an iterator pointing one past the last pair.	O(1)
Modifiers		
`m[key] = value;`	Store value under key in map.	O(log n)
`m.insert(pair)`	Inserts the <key, value>pair into the map. Equivalent to the above operation.	O(log n)

Adapted from http://www.cs.northwestern.edu/ riesbeck/programming/c++/stl-summary.html

Using Lambda Functions as Comparisons

Usually comparisons call for a class, but that's annoying, below is an example of how to use lambda function to get around it.

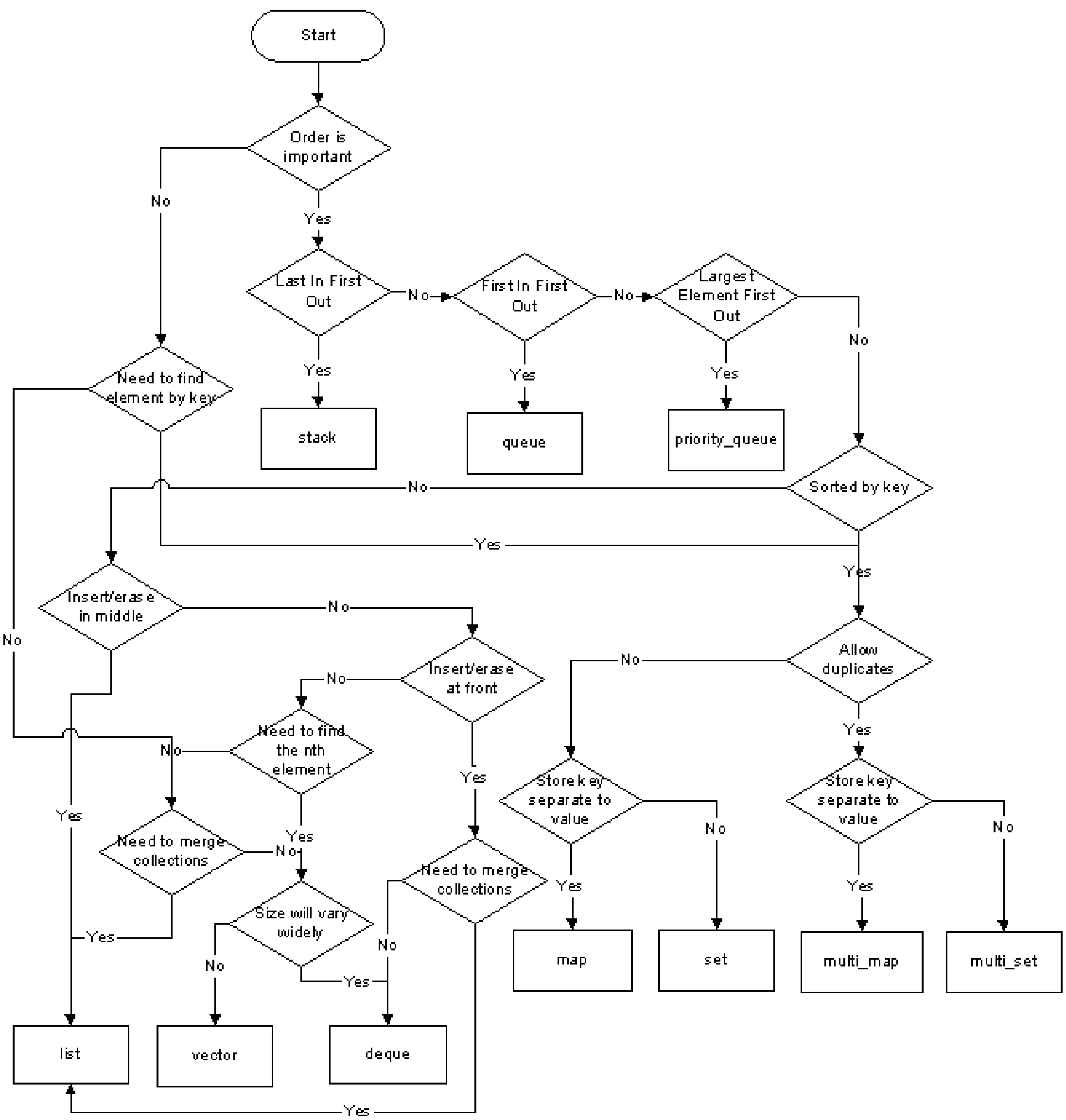
```
using namespace std;

auto comparison = [](int a, int b) {return a > b;};
typedef set<int, decltype(comparison)> sint;

int main(int argc, char *argv[]) {
    sint A(comparison);
    int N = 1;
```

```
    while (N) {
        cin >> N;
        auto print_bool = [](int b) {return b == 0 ? "False" : "True";};
        cout << print_bool(A.insert(N).second) << '\n';
    }
    return 0;
}
```

Choosing the Right Container



Algorithms

```
#include <algorithm>
// almost all of these modify on containers with c.begin(), c.end()
// let us first create a vector!
vector<int> v(10);
// fill - fills the vector, O(n)
// this fills the vector with the value 10
fill(v.begin(), v.end(), 10);
// this fills the length - 2 with the value 5
fill(v.begin(), v.end() - 2, 5);

// find - returns the iterator to the first object with value v
// returns .end() if none found
auto it = find(v.begin(), v.end(), 5);
// find_if - returns the iterator to the first object satisfying
// condition
// the following returns the iterator to the first even number
auto it = find(v.begin(), v.end(), [](int i) {return i % 2 == 0;});

// count - returns the number of objects w/ value
int N = count(v.begin(), v.end(), 5);
// count_if - returns the number of objects satisfying a condition
int N = count_if(v.begin(), v.end(), [](int i) {return i % 2 == 0;});

// search - searches a subsequence within a bigger sequence
// returns an iterator to the beginning of the subsequence
// O(mn) where m and n are the lengths of the target and source
int target[] = {5,5,5};
auto it = search(v.begin(), v.end(), target.begin(), target.end());

// sort - sorts the range according to comparison function
// if less than sorts lowest -> highest
// if more than sorts highest -> lowest
// the following sorts lowest -> highest
// O(N log N)
sort(v.begin(), v.end(), [](int a, int b) {return a < b};);

// partition - moves the content around such that elements satisfying
// the precondition comes before elements that don't
// O(N)
// the following puts the 10's in front of the 5's
partition(v.begin(), v.end(), [](int a) {return a > 10;});

// following are based on binary searches on a *sorted* container
// lower_bound - returns iterator to the lower bound of the value
// upper_bound - returns iterator to the upper bound of the value
// equal_range - return <lower_bound, upper_bound>
// binary_search - returns true if the value is found
// all of these are O(log N)
auto it_l = lower_bound(v.begin(), v.end(), 7);
auto it_u = uppere_bound(v.begin(), v.end(), 7);
auto range = equal_range(v.begin(), v.end(), 7);
bool contains7 = binary_search(v.begin(), v.end(), 7, less<int>);
```