Initial Self-Assessment Test

```
g++ -std=c++11 -Wall -O2 program.cc // Compile
```

C++11 New futures

```
// read a vector from input
vector<int> v;
int x;
while (cin >> x) v.push_back(x);

// write all elements of v to output
for (int y : v) cout << y << "endl";
// double all elements of v (note the reference to modify the elements!!!)
for (int& y : v) y *= 2;
// write all elements of v again
for (int y : v) cout << y << endl;

// make a set of sets of integers and write it
set<set<int>> S = {{2,3}, {5,1,5}, {}, {3}};
for (auto s : S) {
   cout << "{";
   for (auto x : s) cout << x << ",";
   cout << "}" << endl;
}
```

Reading the input line by line

```
#include <sstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
   string s;
   while (getline(cin, s)) ;
}
```

Increment and decrement a variable

```
i++ First assigns a value to expression and then increments the variable.
++i First increments then assigns a value to the expression.
```

Pairs

```
pair<value_type1, value_type2> var;
```

```
struct pair {
   value_type1 first;
   value_type2 second;
};

pair var;
```

```
var.first                //get first parameter of a pair
var.second               //get second parameter of a pair
```

Functions	Description	Cost
`bool empty() const;`	Test whether container is empty	$\Theta(1)$
`size_type size() const;`	Return size	$\Theta(1)$

Vector

```
#include <vector>

vector<value_type> var(size);
```

```
vector<value_type>(n, initial_value); // vector with initial value (Cost: Θ(n))
```

Functions	Description	Cost
`operator[]`	Return element at position inside [ ]	Θ(1)
`void push_back(const value_type& x);`	Insert element at the end	Θ(1)

```
#include <algoritm>              // Sort a vector

sort(v.begin(); v.end()); // Increasing sort
sort(v. begin (), v.end(), greater<int>()); // Decreasing sort
bool comp(int a, int b) {
   // Comparations
}
…
sort(v.begin(); v.end(), comp); // Custom sort
```

Matrix

```
vector< vector<value_type> > M(3, vector<value_type>(4)); // In a single line

typedef vector<value_type> Row;   // Elegant declaration
typedef vector<Row> Matrix;

Matrix M(3, Row(4));
```

Stacks

```
#include <stack>          //Last In First Out

stack<value_type> var;
```

Functions	Description	Cost
`value_type& top() const;`	Access the element of the top of stack	Θ(1)
`void push(const value_type& x);`	Insert element at the top of stack	Θ(1)
`void pop();`	Remove top element	Θ(1)

Queue

```
#include <queue>          //First In First Out

queue<value_type> var;
```

Functions	Description	Cost
`value_type& front() const;`	Access the first element of the queue	Θ(1)
`void push(const value_type& x);`	Insert element	Θ(1)
`void pop();`	Remove top element	Θ(1)

Priority queue → Heap Implementation

```
#include <queue>          //First In First Out

priority_queue<value_type> var;

// Priority queue with inverted order
priority_queue<value_type, vector<value_type>, greater<value_type>> var;
```

Priority queue with custom order

```
struct comp {
  bool operator() (const value_type& a, const value_type& b) {
    // Comparations
  }
};
…
priority_queue<value_type, vector<value_type>, comp> var;
```

Functions	Description	Cost
`value_type& front() const;`	Access the first element of the queue	$\Theta(1)$
`void push(const value_type& x);`	Insert element	$\Theta(\log n)$
`void pop();`	Remove top element	$\Theta(\log n)$

Iterators

An iterator is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators

```
CONTAINER<value_type>::iterator it; // Iterator that could change the container

// Iterator that prevents modifications in the container
CONTAINER<value_type>::const_iterator it;

it++ // Point to next element
it--; // Point to previous element
*it; // Get the element that the iterator is pointing
it->first // Get the first element of a pair or a struct
(*it).first // Get the first element of a pair or a struct
```

Functions	Description	Cost
`iterator begin();`	Return iterator to beginning	$\Theta(1)$
`iterator end();`	Return iterator to end	$\Theta(1)$

Sets → Dictionaries Implementation

```
#include <set>

set<value_type> var;

set<value_type, greater<value_type> > var; // Set with inverted order
```

Set with custom order

```
struct comp {
  bool operator() (const value_type& a, const value_type& b) const {
    // Comparations
  }
};
…
set<value_type, comp> var;
```

Functions	Description	Cost
`void clear();`	Clear content	$\Theta(n)$
`void insert(const value_type& x);`	Insert element	$\Theta(\log n)$
`void erase(const value_type& x);`	Erase element	$\Theta(\log n)$
`void erase(iterator position);`	Erase element that's pointing	$\Theta(1)$
`void erase(iterator first, iterator last);`	Erase elements between first and last iterators	$\Theta(n)$
`iterator find(const value_type& x) const;`	Get iterator to element	$\Theta(\log n)$
`iterator lower_bound(const value_type& x) const;`	Returns an iterator pointing to the first element in the container which is not considered to go before **x**	$\Theta(\log n)$
`iterator upper_bound(const value_type& x) const;`	Returns an iterator pointing to the first element in the container which is considered to go after **x**	$\Theta(\log n)$

Unordered sets → Hash tables Implementation

```
#include <unordered_set>

unordered_set<value_type> var;
```

Functions	Cost	
Same ones of the normal set (except `lower_bound` and `upper_bound`) but these ones down here change the cost		
`void insert(const value_type& x);`	$\Theta(1)$	$O(n)$
`void erase(const value_type& x);`	$\Theta(1)$	$O(n)$
`void erase(iterator position);`	$\Theta(1)$	$O(n)$
`void erase(iterator first, iterator last);`	$\Theta(n)$	
`iterator find(const value_type& x) const;`	$\Theta(1)$	$O(n)$

Maps → Dictionaries Implementation

```
#include <map>
```

```
map<value_type1, value_type2> var;
```

Functions	Description	Cost
`operator[]`	If the key match with an element in the container, the function returns a reference to its mapped value If the key doesn't match with any element in the container, the functions inserts a new element with that key and returns a reference to its mapped value	$\Theta(\log n)$
`void clear();`	Clear content	$\Theta(n)$
`void insert(const value_type& x);`	Insert element	$\Theta(\log n)$
`void erase(const value_type& x);`	Erase element	$\Theta(\log n)$
`void erase(iterator position);`	Erase element that's pointing	$\Theta(1)$
`void erase(iterator first, iterator last);`	Erase elements between first and last iterators	$\Theta(n)$
`iterator find(const value_type& x) const;`	Get iterator to element	$\Theta(\log n)$
`iterator lower_bound(const value_type& x) const;`	Returns an iterator pointing to the first element in the container which is not considered to go before **x**	$\Theta(\log n)$
`iterator upper_bound(const value_type& x) const;`	Returns an iterator pointing to the first element in the container which is considered to go after **x**	$\Theta(\log n)$

Unordered maps → Hash tables Implementation

```
#include <unordered_map>
```

```
unordered_map<value_type1, value_type2> var;
```

Functions	Cost	
Same ones of the normal map (except `lower_bound` and `upper_bound`) but these ones down here change the cost		
`operator[]`	$\Theta(n)$	
`void insert(const value_type& x);`	$\Theta(1)$	$O(n)$
`void erase(const value_type& x);`	$\Theta(1)$	$O(n)$
`void erase(iterator position);`	$\Theta(1)$	$O(n)$
`void erase(iterator first, iterator last);`	$\Theta(n)$	
`iterator find(const value_type& x) const;`	$\Theta(1)$	$O(n)$