

## 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
<code>bool empty() const;</code>	Test whether container is empty	$\Theta(1)$
<code>size_type size() const;</code>	Return size	$\Theta(1)$

## Vector

```
#include <vector>

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

Functions	Description	Cost
<code>operator[]</code>	Return element at position inside [ ]	$\Theta(1)$
<code>void push_back(const value_type&amp; x);</code>	Insert element at the end	$\Theta(1)$

```
#include <algorithm>           // 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
<code>value_type&amp; top() const;</code>	Access the element of the top of stack	$\Theta(1)$
<code>void push(const value_type&amp; x);</code>	Insert element at the top of stack	$\Theta(1)$
<code>void pop();</code>	Remove top element	$\Theta(1)$

## Queue

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

queue<value_type> var;
```

Functions	Description	Cost
<code>value_type&amp; front() const;</code>	Access the first element of the queue	$\Theta(1)$
<code>void push(const value_type&amp; x);</code>	Insert element	$\Theta(1)$
<code>void pop();</code>	Remove top element	$\Theta(1)$

## 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
<code>value_type&amp; front() const;</code>	Access the first element of the queue	$\Theta(1)$
<code>void push(const value_type&amp; x);</code>	Insert element	$\Theta(\log n)$
<code>void pop();</code>	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
<code>iterator begin();</code>	Return iterator to beginning	$\Theta(1)$
<code>iterator end();</code>	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 {
        // Comparisons
    }
};
```

```
...
```

```
set<value_type, comp> var;
```

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

**Unordered sets** → Hash tables Implementation

```
#include <unordered_set>
```

```
unordered_set<value_type> var;
```

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

## Maps → Dictionaries Implementation

```
#include <map>
```

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

Functions	Description	Cost
<code>operator[]</code>	<p>If the key match with an element in the container, the function returns a reference to its mapped value</p> <p>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</p>	$\Theta(\log n)$
<code>void clear();</code>	Clear content	$\Theta(n)$
<code>void insert(const value_type&amp; x);</code>	Insert element	$\Theta(\log n)$
<code>void erase(const value_type&amp; x);</code>	Erase element	$\Theta(\log n)$
<code>void erase(iterator position);</code>	Erase element that's pointing	$\Theta(1)$
<code>void erase(iterator first, iterator last);</code>	Erase elements between first and last iterators	$\Theta(n)$
<code>iterator find(const value_type&amp; x) const;</code>	Get iterator to element	$\Theta(\log n)$
<code>iterator lower_bound(const value_type&amp; x) const;</code>	Returns an iterator pointing to the first element in the container which is not considered to go before x	$\Theta(\log n)$
<code>iterator upper_bound(const value_type&amp; x) const;</code>	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 <code>lower_bound</code> and <code>upper_bound</code> ) but these ones down here change the cost		
<code>operator[]</code>	$\theta(n)$	
<code>void insert(const value_type&amp; x);</code>	$\theta(1)$	$O(n)$
<code>void erase(const value_type&amp; x);</code>	$\theta(1)$	$O(n)$
<code>void erase(iterator position);</code>	$\theta(1)$	$O(n)$
<code>void erase(iterator first, iterator last);</code>	$\theta(n)$	
<code>iterator find(const value_type&amp; x) const;</code>	$\theta(1)$	$O(n)$