Data Structures and Object Oriented Programming

Lecture 25

Dr. Naveed Anwar Bhatti

Webpage: naveedanwarbhatti.github.io

Templates (Continue …)

Templates and Inheritance

- We can use inheritance comfortably with templates or their specializations

- But we must follow one rule:

“Derived class must take at least as many template parameters as the base class requires for an instantiation”

Derivations of a Template

- Example

```
template<class T>
class base
{
public:
    T data;
    void print()
    {
        cout << data << endl;
    }

};


template<class T>
class derived : public base<T> {

};
```

```
int main()
{
    base<int> x;
    derived<float> y;

    x.data = 10;
    y.data = 11.5;

    x.print();
    y.print();
}
```

Templates and Friends

- Like inheritance, templates are compatible with friendship feature of C++
- A function will be a friend to all template class instantiations

```
template< class T >
class A
{
    friend void hello();
    …
};
```

One-to-Many

Templates & Static Members

- Each instantiation of a class template has its own copy of static members

…Templates & Static Members

```
template< class T >
class A {
public:
  static int data;
  ...
};
```

```
int main() {
  A< int > ia;
  A< char > ca;
  ia.data = 5;
  ca.data = 7;
  cout << “ia.data = ” << ia.data
         << endl
         <<   “ca.data = ” << ca.data;
  return 0;
}
```

- **Output**

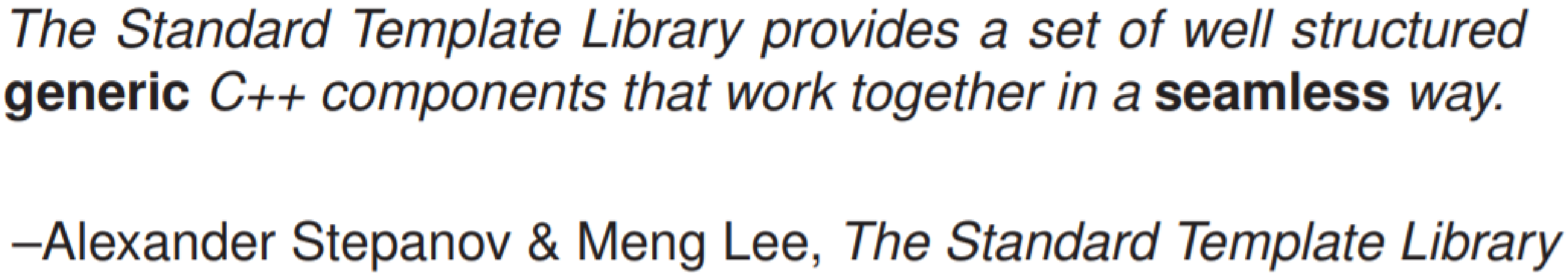
```
ia.data = 5
ca.data = 7
```

Templates – Conclusion

- Templates provide
  - Reusability
  - Readability
  - Writablity

Standard Template Library (STL)

What is STL?

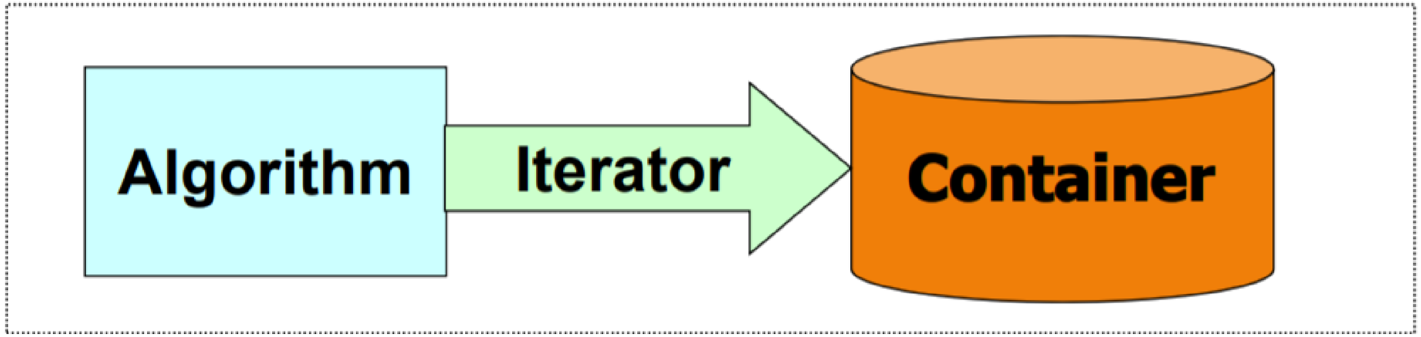


> *The Standard Template Library provides a set of well structured* **generic** *C++ components that work together in a* **seamless** *way.*
>
> –Alexander Stepanov & Meng Lee, *The Standard Template Library*

STL

- A major part of the C++ standard library
- Consists of three components:
  1. **Container**
  2. **Iterator**
  3. **Algorithm**
- Defined as template classes

- Relationship between STL components:



STL Containers

- **Containers:**
  - Object that contains other objects
  - Represent general **data structures** in computing

- Main features:
  - Template class
    - Can be used for built-in and user defined data types
  - All containers supports a set of general methods
    - **`size()`** : number of elements
    - **`empty()`** : is the container empty?
    - Etc
  - Specialized methods are defined for individual container classes

STL Containers

STL provides three kinds of containers

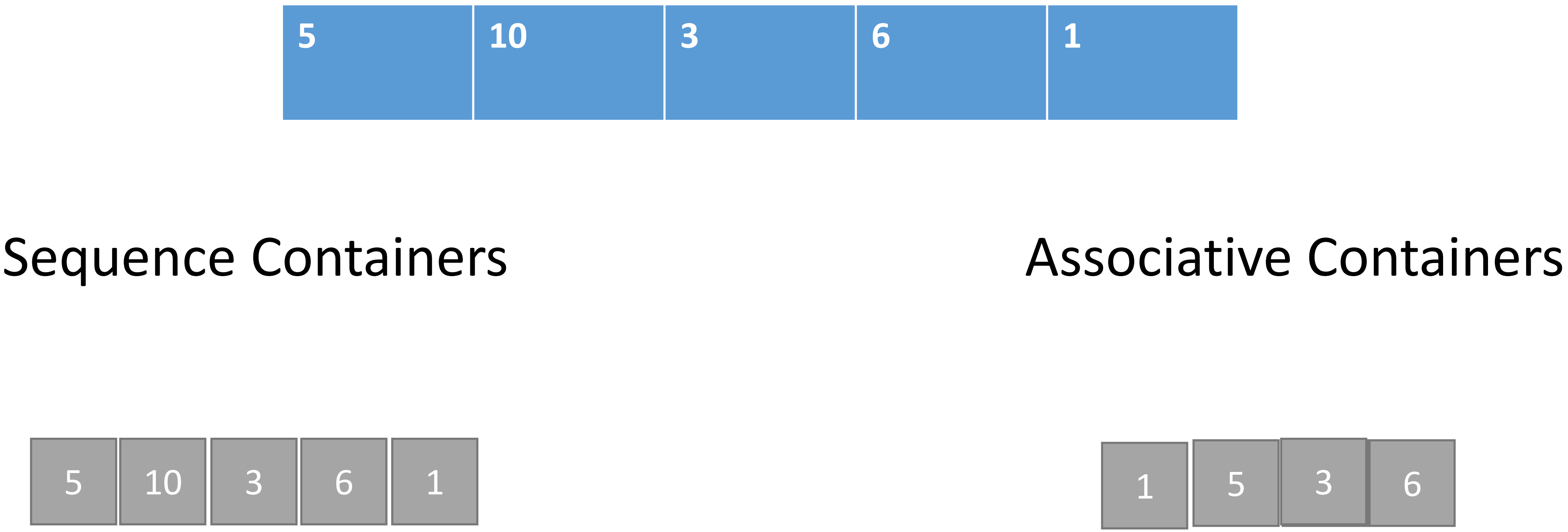
- Sequence Containers
  - Position depends on the time of insertion
  - Element order has nothing to do with their value

- Associative Containers
  - The value of the elements determine the position of the elements in the container
  - The order of insertion doesn't matter

- Container Adapters
  - Provide different ways to access sequential & associative containers

STL Containers

User Input Sequence



5	10	3	6	1

Sequence Containers

5	10	3	6	1

Associative Containers

1	5	3	6

STL Containers

- Sequence Containers
  - Arrays
  - Vectors
  - Deque
  - Singly Linked List
  - Doubly Linked-List

- Associative Containers
  - set
  - Multi-set
  - map
  - Multi-map

- Container Adapters
  - Stack
  - Queue

STL Vector

- Header File:

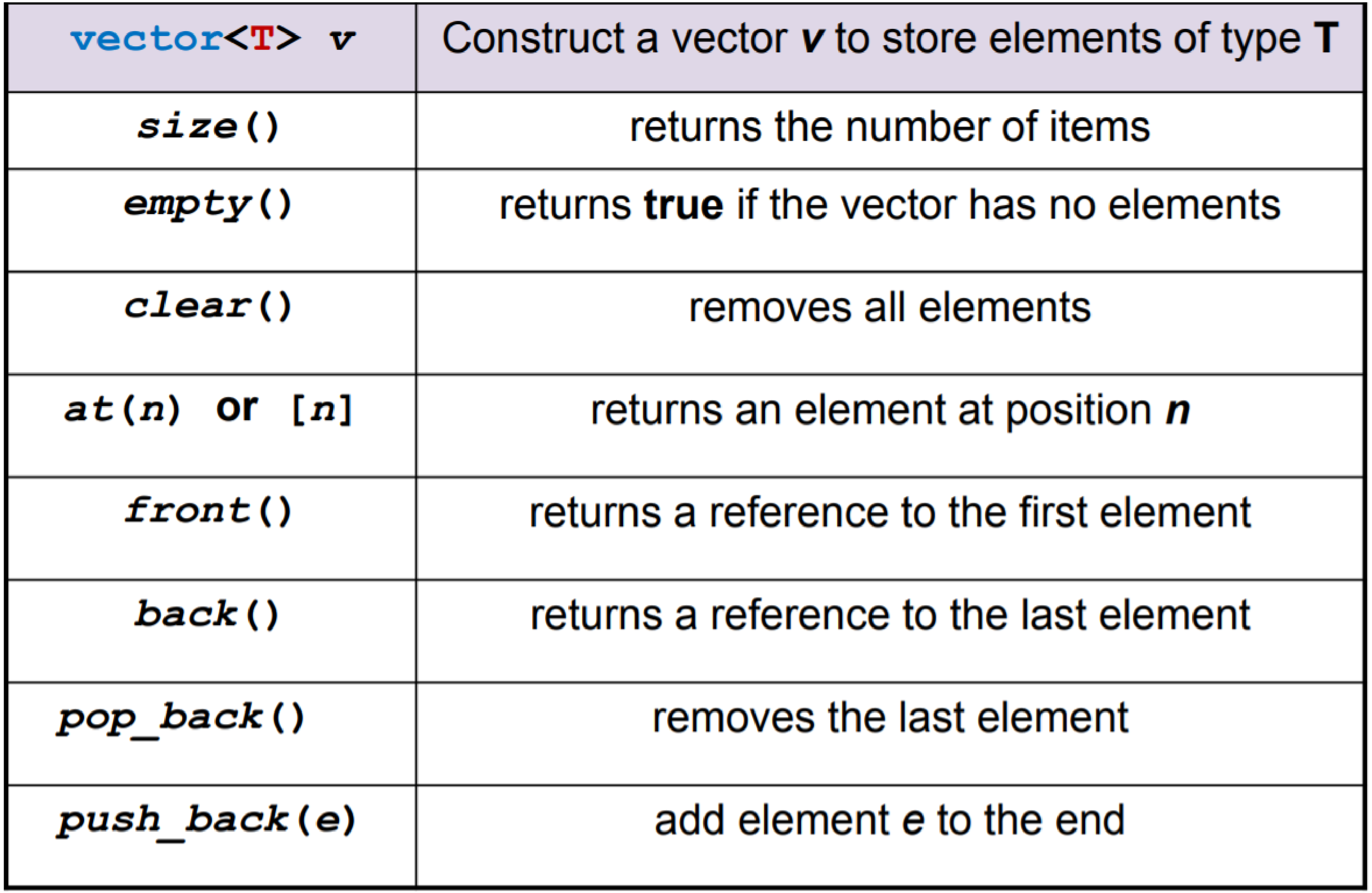
```
#include <vector>
```

- Defined as template class

```
vector<int> intVector;
```

- **Advantages:**
  - Fast insertion and removal of at the end of vector
  - Support dynamic number of elements
  - Automatic memory management
- Vector is the simplest STL container class, and in *some* cases the most efficient

STL Vector



`vector<T> v`	Construct a vector ***v*** to store elements of type **T**
`size()`	returns the number of items
`empty()`	returns **true** if the vector has no elements
`clear()`	removes all elements
`at(n)` or `[n]`	returns an element at position ***n***
`front()`	returns a reference to the first element
`back()`	returns a reference to the last element
`pop_back()`	removes the last element
`push_back(e)`	add element ***e*** to the end

Example – STL Vector

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> iv;
    int x, y;
    char ch;
    do
    {
        cout << "Enter the first integer:";
        cin >> x;
        cout << "Enter the second integer:";
        cin >> y;
        iv.push_back(x);
        iv.push_back(y);
        cout << "Current size of iv = " << iv.size() << endl;
        cout << "Do you want to continue?";
        cin >> ch;
    } while (ch == 'y');
}
```

Sample Output

```
Enter the first integer: 1

Enter the second integer: 2

Current size of iv = 2

Do you want to continue? y
```

…Sample Output

```
Enter the first integer: 3

Enter the second integer: 4

Current size of iv = 4

Do you want to continue? y
```

…Sample Output

```
Enter the first integer: 5

Enter the second integer: 6

Current size of iv = 6

Do you want to continue? n
```

…Sequence Containers

- **`vector`**
  - Rapid insertions and deletions at back end
  - Random access to elements

- **`deque`**
  - Rapid insertions and deletions at front or back
  - Random access to elements

Example – STL Deque

```
#include <deque>

int main()
{
   deque< int > dq;
   dq.push_front(3);
   dq.push_back(5);

   dq.pop_front();
   dq.pop_back();
   return 0;
}
```

…Sequence Containers

- **`vector`**
  - Rapid insertions and deletions at back end
  - Random access to elements

- **`deque`**
  - Rapid insertions and deletions at front or back
  - Random access to elements

- **`lists`**
  - Single (or Doubly) linked list
  - Rapid insertions and deletions anywhere

…Associative Containers

- `set`
  - No duplicates
- `multiset`
  - Duplicates allowed
- `map`
  - No duplicate keys
- `multimap`
  - Duplicate keys allowed

Example – STL Set

```
#include <set>
int main()
{
    set< char > cs;
    cout << "Size before insertions : " << cs.size() << endl;
    cs.insert('a');
    cs.insert('b');
    cs.insert('b');
    cout << "Size after insertions : " << cs.size();
    return 0;
}
```

Output

```
Size before insertions: 0

Size after insertions: 2
```

Example – STL Multi-Set

```
#include <set>
int main()
{
    multiset< char > cms;
    cout << "Size before insertions: " << cms.size() << endl;
    cms.insert('a');
    cms.insert('b');
    cms.insert('b');
    cout << "Size after insertions: " << cms.size();
    return 0;
}
```

Output

```
Size before insertions: 0

Size after insertions: 3
```

Example – STL Map

```
#include <map>
int main() {
    map< int, char > m;

    m.insert(pair < int, char>(1, 'a'));
    m.insert(pair < int, char>(2, 'b'));
    m.insert(pair < int, char>(3, 'c'));
    return 0;
}
```

Example – STL Map

```
#include <map>
int main() {
    multimap< int, char > m;

    m.insert(pair < int, char>(1, 'a'));
    m.insert(pair < int, char>(2, 'b'));
    m.insert(pair < int, char>(2, 'c'));
    return 0;
}
```

First-class Containers

- Sequence and associative containers are collectively referred to as the first-class containers

Container Adapters

- A container adapter is a constrained version of some first-class container

…Container Adapters

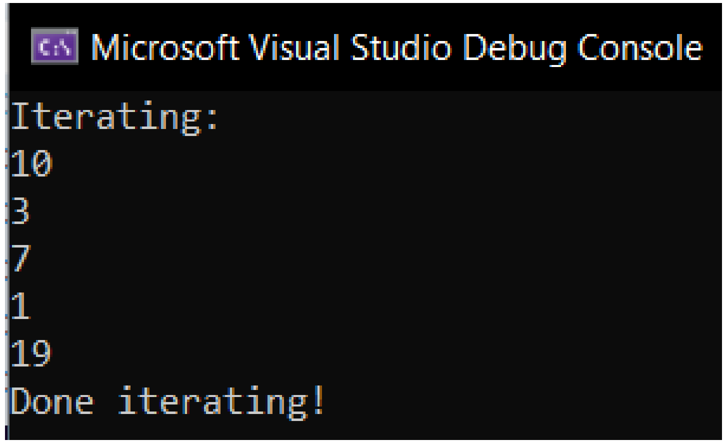
- **`stack`**
  - Last in first out (LIFO)
  - Can adapt **`vector`**, **`deque`** or **`list`**
- **`queue`**
  - First in first out (FIFO)
  - Can adapt **`deque`** or **`list`**

Iterator

- Each container class has an associated iterator class (*e.g.* `vector<int>::iterator`) used to iterate through elements of the container
  - Iterator range is from `begin` up to `end`
    - `end` is one past the last container element!
  - Some container iterators support more operations than others
    - All can be incremented (`++`), copied, copy-constructed
    - Some can be dereferenced
    - Some can be decremented (`--`)
    - Some support random access (`[]`, `+`, `-`, `+=`, `-=`, `<`, `>` operators)
  - http://www.cplusplus.com/reference/std/iterator/

Iterator - Example

```
#include <vector>

using namespace std;
int main() {

    vector<int> vec;

    vec.push_back(10);
    vec.push_back(3);
    vec.push_back(7);
    vec.push_back(1);
    vec.push_back(19);

    cout << "Iterating:" << endl;
    vector<int>::iterator it;
    for (it = vec.begin(); it < vec.end(); it++)
    {
        cout << *it << endl;
    }
    cout << "Done iterating!" << endl;
    return 0;
}
```



```
Iterating:
10
3
7
1
19
Done iterating!
```

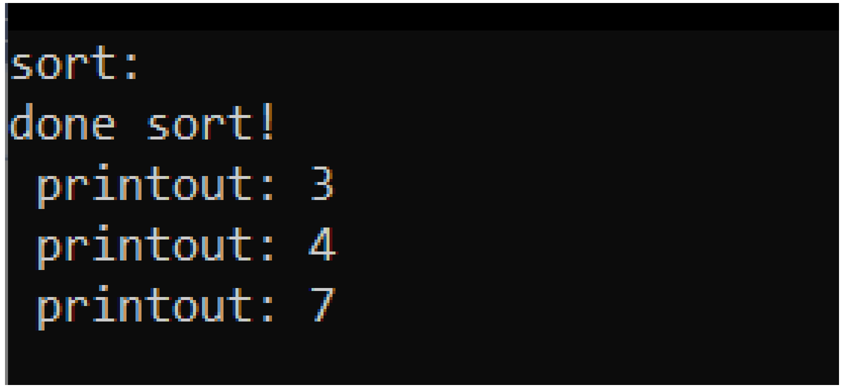
Algorithms

- A set of functions to be used on ranges of elements
  - Range: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
  - General form: `algorithm(begin, end, ...);`

- Algorithms operate directly on range *elements* rather than the containers they live in
  - Some do not modify elements
    - *e.g.* find, count, for_each, min_element, binary_search
  - Some do modify elements
    - *e.g.* sort, transform, copy, swap

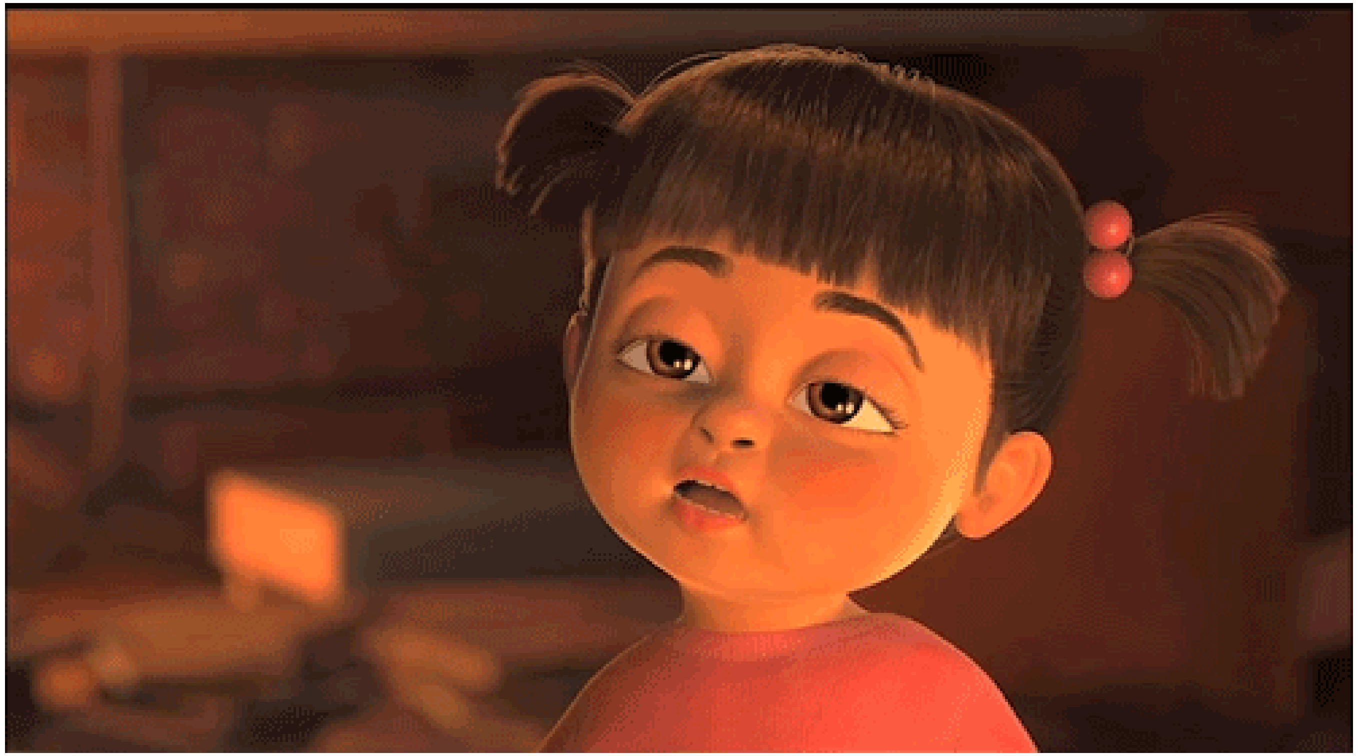
Algorithms - Example

```
#include <vector>
#include <algorithm>

void PrintOut(int  p)
{
    cout << " printout: " << p << endl;
}

int main()
{

    vector<int> vec;
    vec.push_back(3);
    vec.push_back(7);
    vec.push_back(4);
    cout << "sort:" << endl;
    sort(vec.begin(), vec.end());
    cout << "done sort!" << endl;
    for_each(vec.begin(), vec.end(), PrintOut);
    return 0;
}
```



```
sort:
done sort!
 printout: 3
 printout: 4
 printout: 7
```

Thanks a lot



If you are taking a Nap, **wake up**.......Lecture Over