Templates
Chapter 12

Objectives

- Study the implementation of a **Stack Template Class**
- Introduction to the **Standard Template Library** (**STL**)
- Understand how to use the **STL Vector Template** class

Review: Class for storing a piece of information (type int)

```
class Storage
{
private:
    int data;
public:
    int get_store() {
        return data;
    }
    void set_store(const int &item) {
        data = item;
    }
};
```

Review: **Template** class for storing a **generic** piece of information

```
template <typename T>
class Storage
{
private:
    T data;
public:
    T get_store() {
        return data;
    }
    void set_store(const T &item) {
        data = item;
    }
};
```

Review: **Declaring** objects of a **template** class

```
Storage <int>  intStore; // T is int

intStore.set_store(4);
cout << intStore.get_store() << endl;

Storage<string> strStore; // T is string

strStore.set_store("eddie");
cout << strStore.get_store() << endl;
```

Data Structure: Stack Class

Rule: You can only access items at the **top**.

To add an item to the top - **push()**
To remove an item at the top - **pop()**
To inspect an item at the top – **top()**
To check if stack is empty – **empty()**

We want our stack class to operate similarly to Shishkebob stacks

B.push("onion");

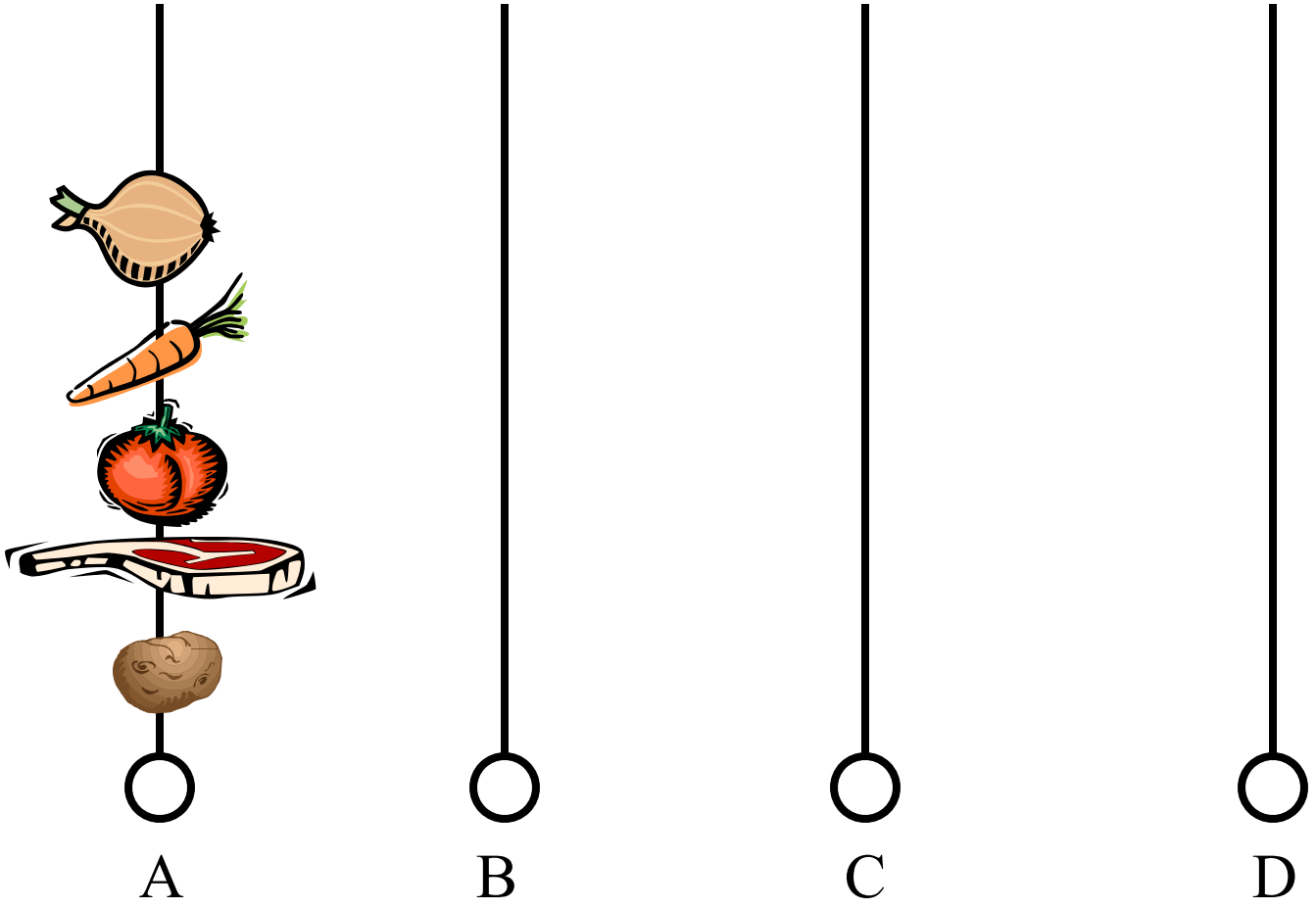
B.push("chop");

C.push("potato");

A.pop();

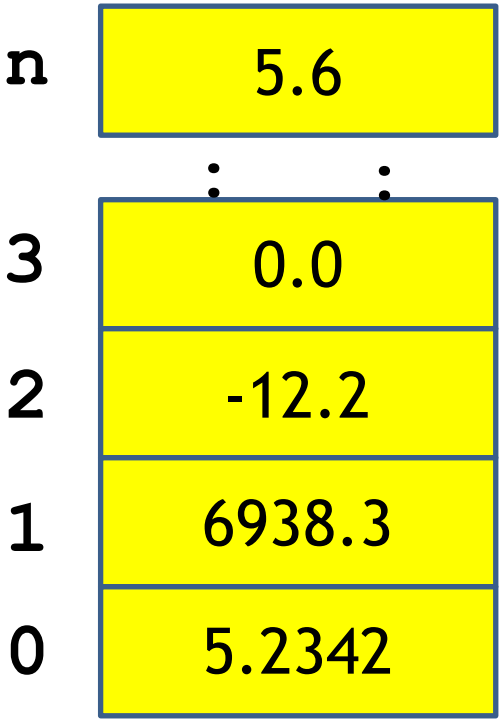
A.top();

D.push("carrot");

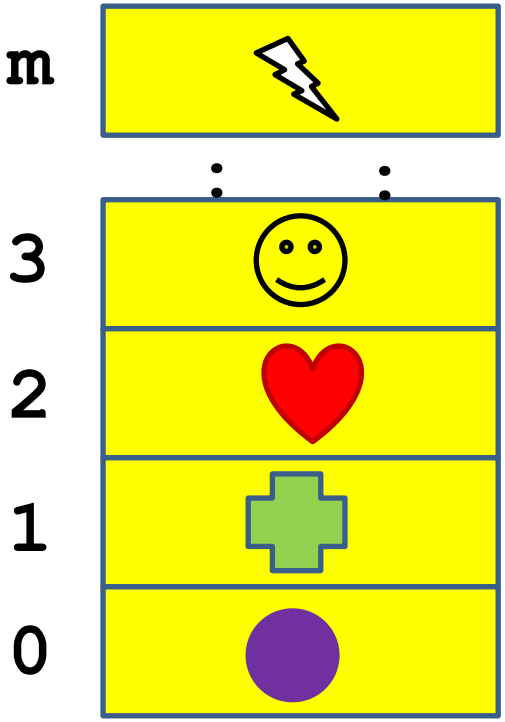


Stack is an ideal candidate for a **template class**:
Declaring stacks for store values of **different types**

```
stack<double> nums;
```




```
stack<shape> shapes;
```



Let's look at an application of a stack - reversing a string

```
int main()
{
    string b, a = "!desrever saw gnirts yM";
    stack<char> cstack;    // Make a stack that can hold chars

    // push all of the string's characters onto the stack
    for ( int i=0; i<a.size(); i++ )
           cstack.push( a[i] );

    // pop the characters off the stack and append to string b
    while ( ! cstack.empty() ) {
       b = b + cstack.top(); // append top of stack to string
        cstack.pop();
    }
    cout << b << endl;   // output string b
}
```

Other applications for Stacks

- Uno: pile that is played to
  - A card added to the pile must match the top card's color or number (or be a wild card)
  - `stack<Card> discardpile;`
  - ```
    if (mycard.getColor() == discardpile.top().getColor() ||
        mycard.getNumber() == discardpile.top().getNumber())
            //push card to stack
    ```

- Washing dishes: stack of plates to clean
  - `stack<plates> dirtyplates`
  - `dirtyplates.push(myplate);  //plate to be washed`
  - ```
    Plate p = wash(dirtyplates.top());
    dirtyplates.pop();
    ```

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

template< typename T >
class stack
{
private:
    T * elements;  // dynamic array of objects
    int num;       // number of objects in the
stack
    int capacity;  // current capacity of the
stack
public:
    stack() : num(0), capacity(16)
    { elements = new T[ capacity ];
    }

    ~stack() { delete [] elements; }

    void push( T value )
    {  ensureCapacity();
       elements[num++] = value;
    }

    void ensureCapacity()
    {    if ( num >= capacity )
        {    T *old = elements;
             capacity = 2 * num;
             elements = new T[ capacity ];
             for ( int i=0; i<num; i++ )
                 elements[i] = old[i];
             delete [] old;
        }
    }

    T top() { return elements[num-1]; }

    void pop() { --num; }

    bool empty() { return (num == 0); }
};
```

Lets look at the implementation of a generic stack class

This is that same doubleCapacity function from EX04_02!

Understanding the Stack Class…

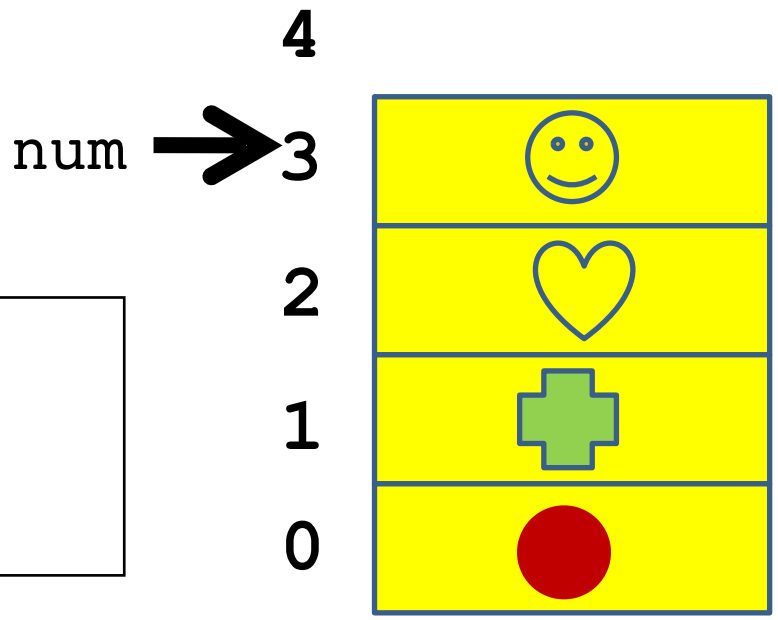
```
#include <iostream>
using namespace std;

template< typename T >
class stack
{
private:
    T * elements;// pointer to an array of objects of
type T
    int capacity;// current capacity of the array
    int num;      // number of objects in the stack
public:
    stack() {
    }

    ~stack() {
    }
```

Constructor

Destructor

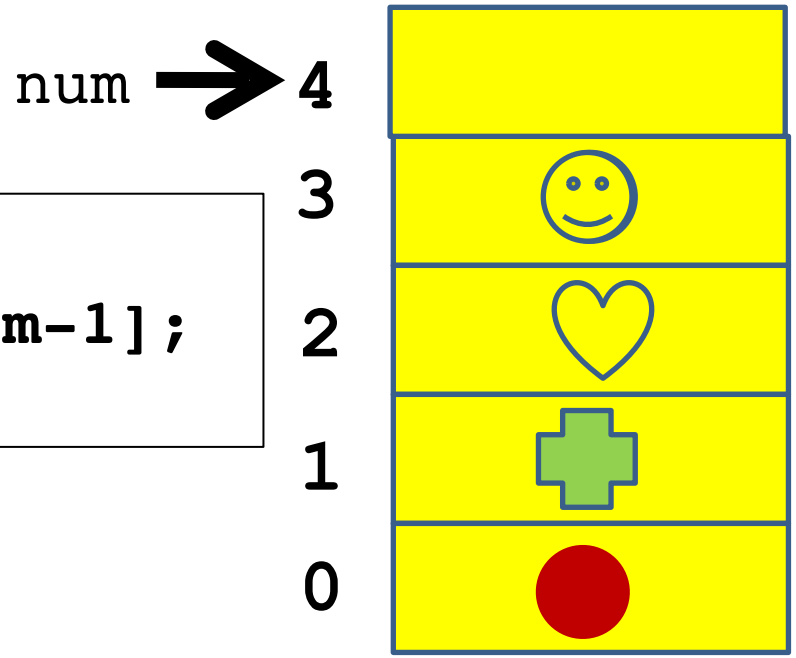
The **push( )** member function will place an item on the top of the stack, after making sure there is room to hold it. **num** will then increment.

```
void push( T value )
{
    elements[num++] = value;
}
```



The **top()** member function will return the item that's on the top of the stack

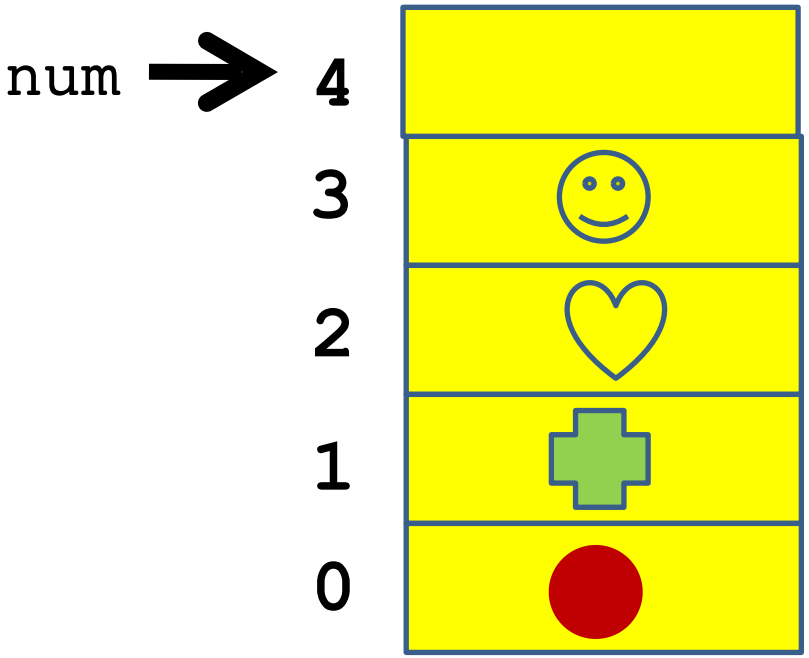
```
T top() {
    return elements[num-1];
}
```



pop( ) "removes" the top item from the stack, but in reality, it simply just changes **num** ...

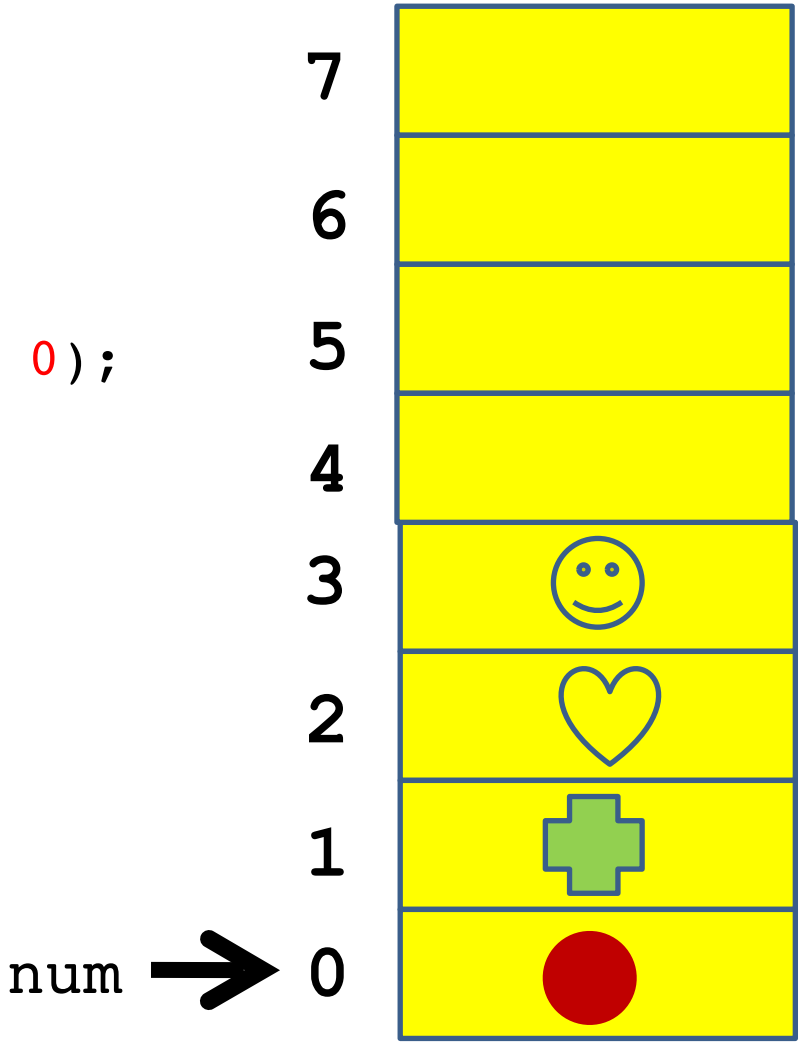
```
void pop() { --num; }
```

```
cstack.pop();
cstack.pop();
cstack.pop();
cstack.pop();
```



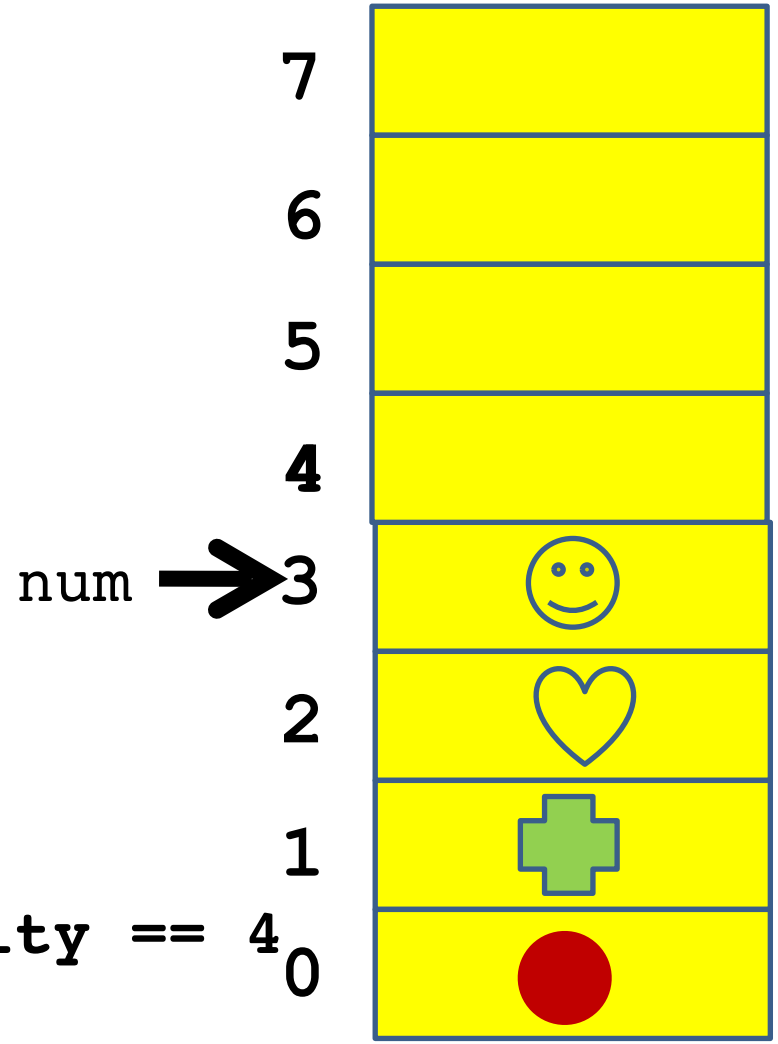
empty() will return true if the stack is empty

```
 bool empty() { return (num == 0);
}
```



If the **number of items equals** the **capacity** of the stack, it will **double** in capacity to allow for more items.

```
void doubleCapacity()
{
  if ( num == capacity ) {
       capacity = 2 * capacity;
       T *new_array = new
T[capacity];
       for ( int i=0; i<num; i++ )
           new_array[i] =
elements[i];
       delete [] elements;
       elements = new_array;
  }
}
```



The **Standard Template Library** (**STL**) is a **library** of **Template Class Data Structures** included in all C++ compilers.

Stack is a template class in STL!

```
#include <stack>
using namespace std;

stack<int> myIntStack;
```

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

vector<int> myIntVector;
```

Try it yourself!

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

int main()
{
    string b, a = "!desrever saw gnirts yM";
    stack<char> cstack;    // Make a stack that can hold chars

    // push all of the string's characters onto the stack
    for ( int i=0; i<a.size(); i++ )
           cstack.push( a[i] );

    // pop the characters off the stack and append to string b
    while ( ! cstack.empty() ) {
       b = b + cstack.top(); // append top of stack to string
        cstack.pop();
    }
    cout << b << endl;   // output string b
}
```

A **vector** behaves like an **array**.

```
#include <vector>

using namespace std;


vector<int> v1;
vector<double> v2;
vector<string> v3;
```

A vector is a template class

Vectors behave like arrays, EXCEPT, a vector is a **class object** that provides **useful member functions**!

```
vector<int> v1;
vector<double> v2;

v1.push_back(10);
v1.push_back(20);

v2.push_back(142.2);
```

You use the **push_back() method** to APPEND an item to the vector

Vector bonus: Vectors automatically increase size as needed!

size() - Find out how many elements are in the vector.

at() - Get item at a specific position in the vector

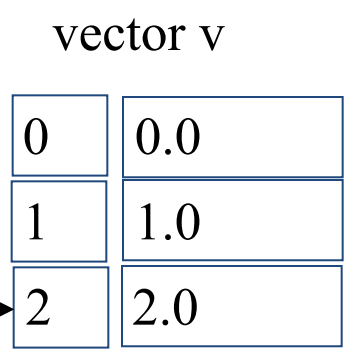
```
vector<int> grades;

grades.push_back(99);
grades.push_back(87);
for (int i=0; i < grades.size(); i++ ) {
      cout << grades.at(i) << endl;
}
```

Accessing vector Elements

Use the **[ ]** operator to **read** or **change** the value of a specific item in the vector.

You cannot **add** an item with **[ ] !!!**

```
vector<double> v;

v.push_back( 0.0 );
v.push_back( 1.0 );
v.push_back( 2.0 );

cout << "first " << v[0] << endl;
cout << "last  " << v[v.size()-1];
cout << endl;
```



You can also remove elements … or clear all elements … or check to see if it's empty… etc.

```
scores.pop_back();
```

pop_back() removes the last element from the vector.

```
scores.clear()
```

clear() will remove all elements from the scores vector.

```
while( ! scores.empty() )
{

}
```

This loop will execute as long as the vector contains some elements

You can provide arguments to a vector's **constructor** to specify **the initial size (and values)** stored in a vector

```
vector<int> scores;
```

Define a vector of integers (starts with 0 elements)

```
vector<int> scores(5);
```

Define an **int** vector with initial size 5 elements

```
vector<int> scores(20,0);
```

Define 20-element **int** vector and initialize all elements to 0

```
vector<int> scores(finals);
```

Define a vector initialized to size and contents of *another vector*

Useful Vector methods

Method	Description
`push_back(item)`	Appends the item to the vector
`pop_back()`	Removes the last element from this vector
`size()`	Returns the number of the elements in this vector
`at(int index)`	Returns the element at the specified index in this vector
`clear()`	Removes all elements from this vector
`swap(vector v2)`	Swaps the contents of this vector with vector v2
`resize(int n)`	Resizes a vector to n elements (n is greater than current size)
`resize(n, value)`	Same resize, except elements will be initialized to *value*

You can use a **vector** like any **data type**, e.g.

```
vector<int> extra_credit(vector<int> &grades)
{
    for (int i = 0; i < grades.size(); ++i) {
                ++grades[i];
    }
    return grades;
}
```