CS217 – Object Oriented Programming (OOP)

Week – 14

May 17-21, 2021

Instructor: **Basit Ali**

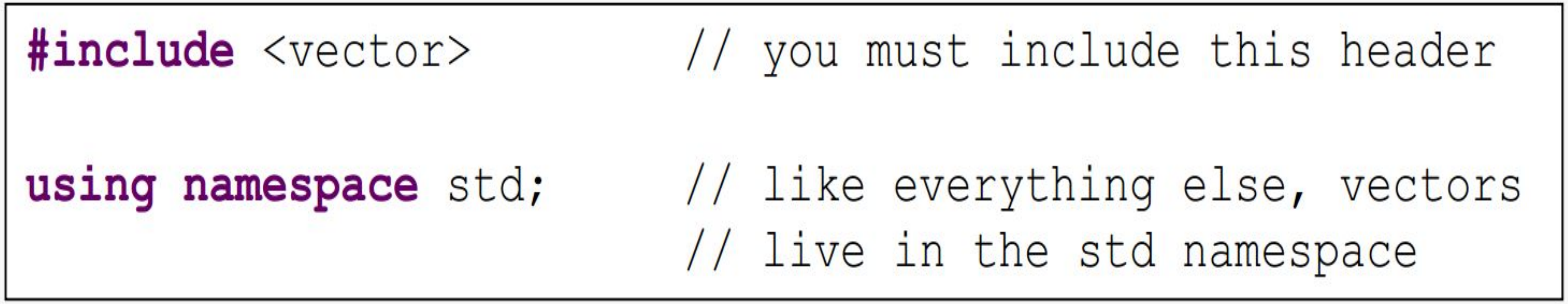
Vectors

- The vector class simply provides a dynamic array i.e. an array that can grow as needed
- Vectors allocates memory when needed
- Although a vector is dynamic, we can still use normal array syntax to access its elements

Advantages of Vectors

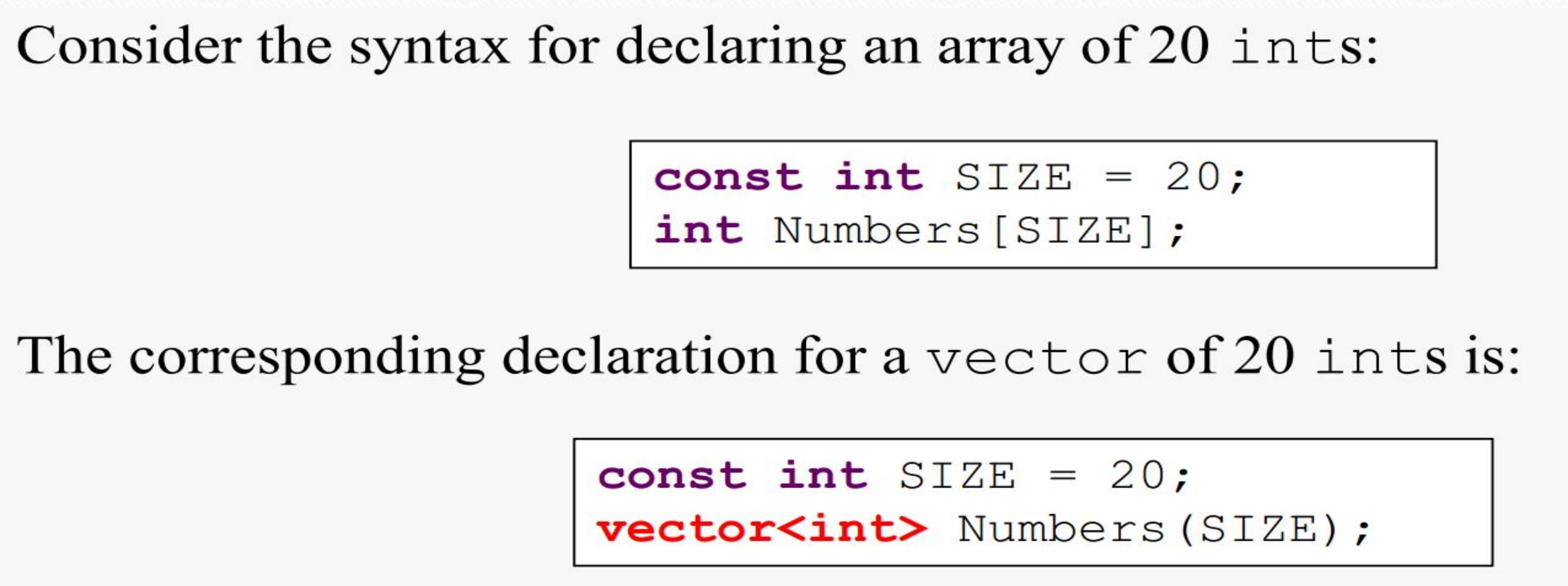
- The size of a vector does not have to be a fixed constant, and it can also grow or shrink during execution. If you want to read data from a file and store it in an vector, you can just keep adding new elements as you receive them, without having to know what the largest amount would be
- A vector always knows its own size, so passing one to a function does not require you to separately pass this information
- Elements can be accessed by position in a vector just as they are in an array, but you can also insert and remove elements anywhere in a vector
- Vectors can be returned from a function easily

Library



```
#include <vector>          // you must include this header

using namespace std;       // like everything else, vectors
                           // live in the std namespace
```

Declaring a Vector



Consider the syntax for declaring an array of 20 `ints`:

```
const int SIZE = 20;
int Numbers[SIZE];
```

The corresponding declaration for a `vector` of 20 `ints` is:

```
const int SIZE = 20;
vector<int> Numbers(SIZE);
```

Unlike an array, the elements of a vector *are* initialized with appropriate default values. This means 0 for `ints`, 0.0 for `doubles`, and “” for `strings`.

Initializing a Vector

```
const int SIZE = 5;
vector<int> Numbers(SIZE);
```

0	0	0	0	0

```
const int SIZE = 5;
vector<int> Numbers(SIZE, 18);
```

18	18	18	18	18

```
const int SIZE = 5;
vector<double> Numbers(SIZE, 3.5);
```

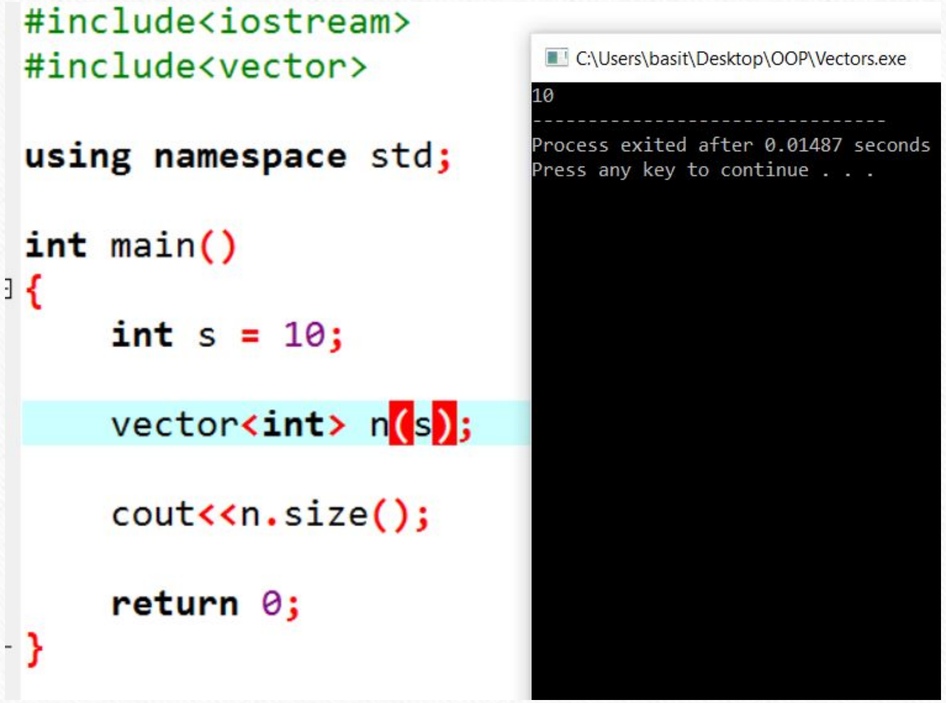
3.5	3.5	3.5	3.5	3.5

Some useful functions in Vector class

- Some of the most commonly used member functions are **size( ), begin( ), end( ), push_back( ), insert( ) erase( ), empty(), resize(), clear() and pop_back()**

- Many other functions and their overloaded versions are also available

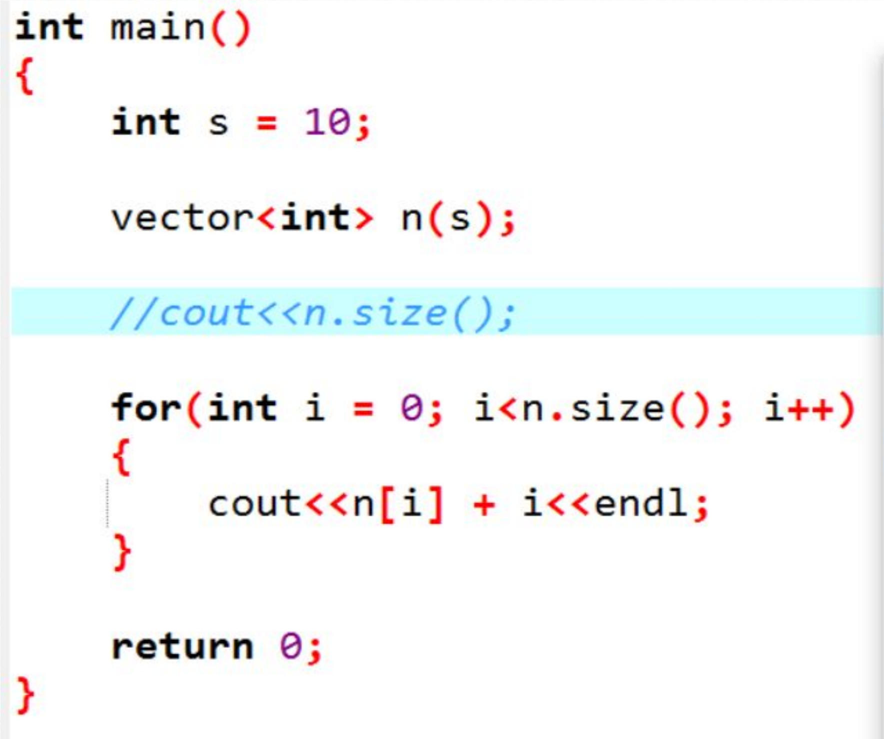
size() function



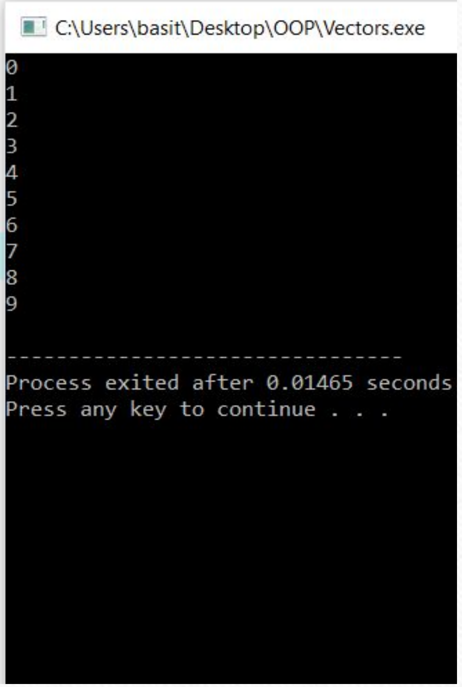
```cpp
#include<iostream>
#include<vector>

using namespace std;

int main()
{
    int s = 10;

    vector<int> n(s);

    cout<<n.size();

    return 0;
}
```

```
10
--------------------------------
Process exited after 0.01487 seconds
Press any key to continue . . .
```

Accessing a Vector

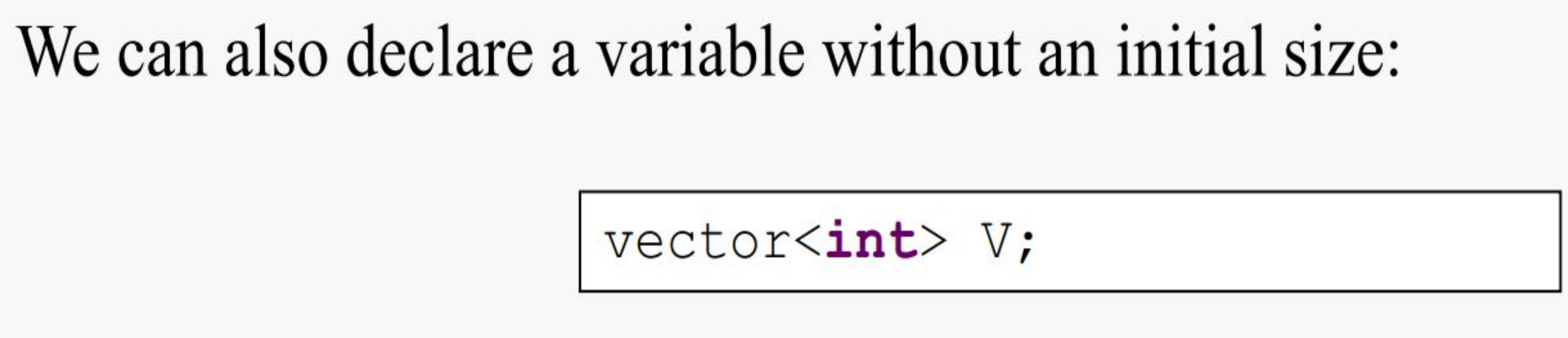


```
int main()
{
    int s = 10;

    vector<int> n(s);

    //cout<<n.size();

    for(int i = 0; i<n.size(); i++)
    {
        cout<<n[i] + i<<endl;
    }

    return 0;
}
```



```
0
1
2
3
4
5
6
7
8
9

--------------------------------
Process exited after 0.01465 seconds
Press any key to continue . . .
```

Vectors without an initial size

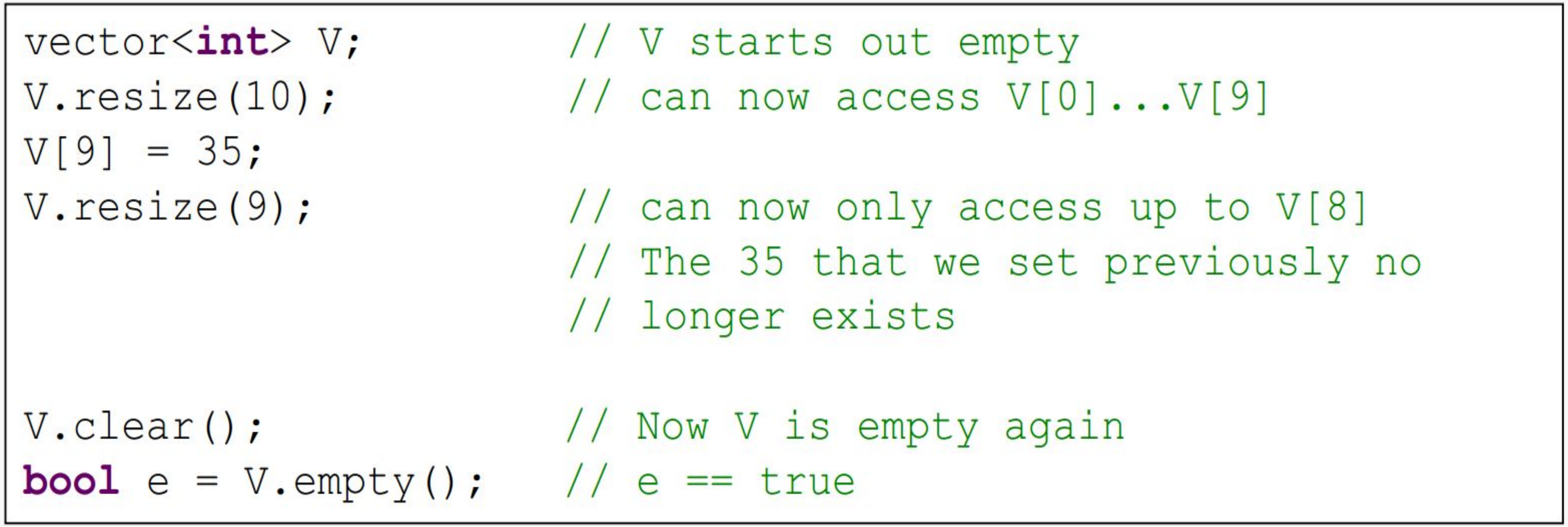


We can also declare a variable without an initial size:

```
vector<int> V;
```

When we do this, the vector is empty. It has no elements, so you can’t even access V[0]. Calling the size() method will return 0. There is also a method named empty() that returns true if the vector is empty or false if it is not.

resize() , clear() & empty() functions

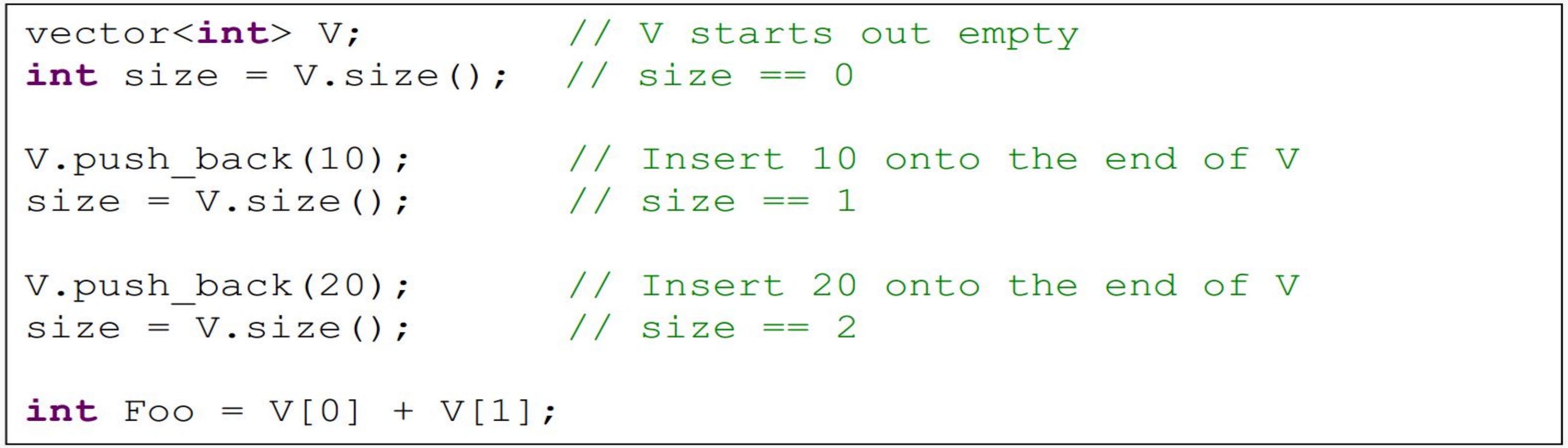


```
vector<int> V;        // V starts out empty
V.resize(10);         // can now access V[0]...V[9]
V[9] = 35;
V.resize(9);          // can now only access up to V[8]
                      // The 35 that we set previously no
                      // longer exists


V.clear();            // Now V is empty again
bool e = V.empty();   // e == true
```

Adding items to a Vector

`V.push_back(`*`Type`*` element)`

Adds an element to the end of the vector, increasing its size by 1. The Type of the argument is the element type used when declaring the vector.



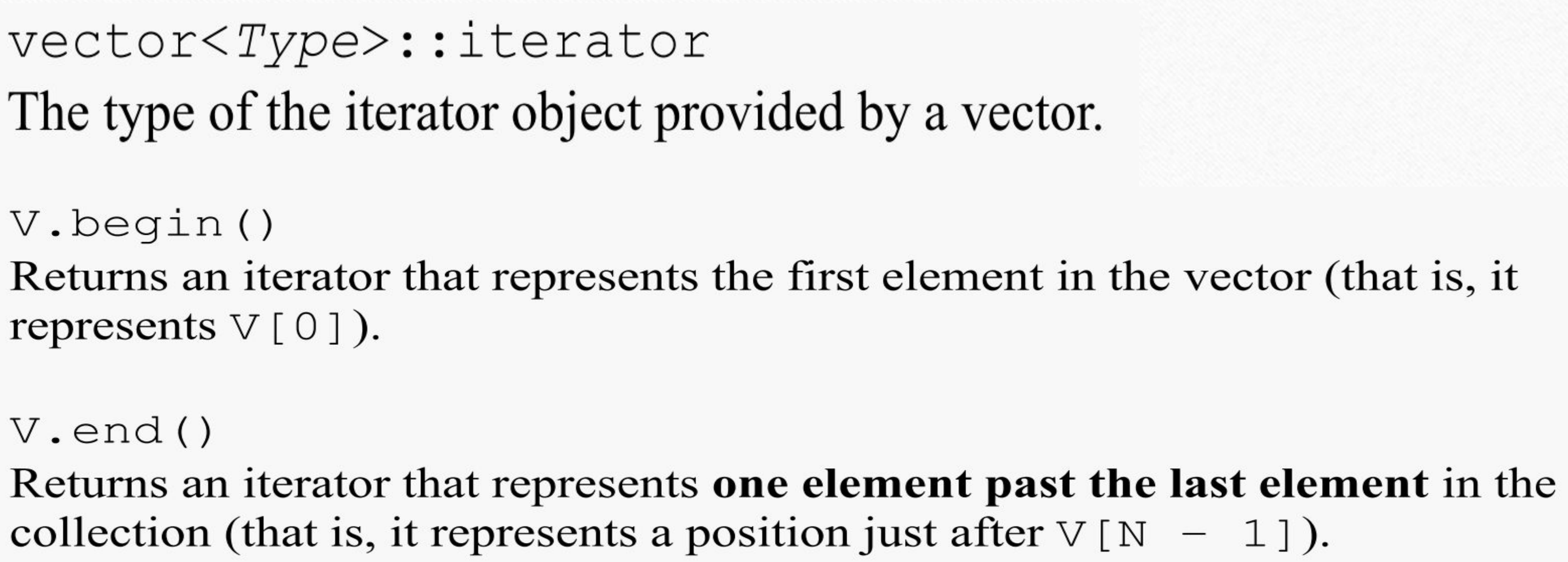
```
vector<int> V;          // V starts out empty
int size = V.size();    // size == 0

V.push_back(10);        // Insert 10 onto the end of V
size = V.size();        // size == 1

V.push_back(20);        // Insert 20 onto the end of V
size = V.size();        // size == 2

int Foo = V[0] + V[1];
```

Containers and Iterators

- Vectors provide the concept of an iterator, which is another form of representing the position of an element in the sequence

- Recall that elements in a vector of size N are identified by their index, an integer between 0 and N – 1. Iterators implement a similar concept

Obtaining an Iterator



vector<*Type*>::iterator

The type of the iterator object provided by a vector.

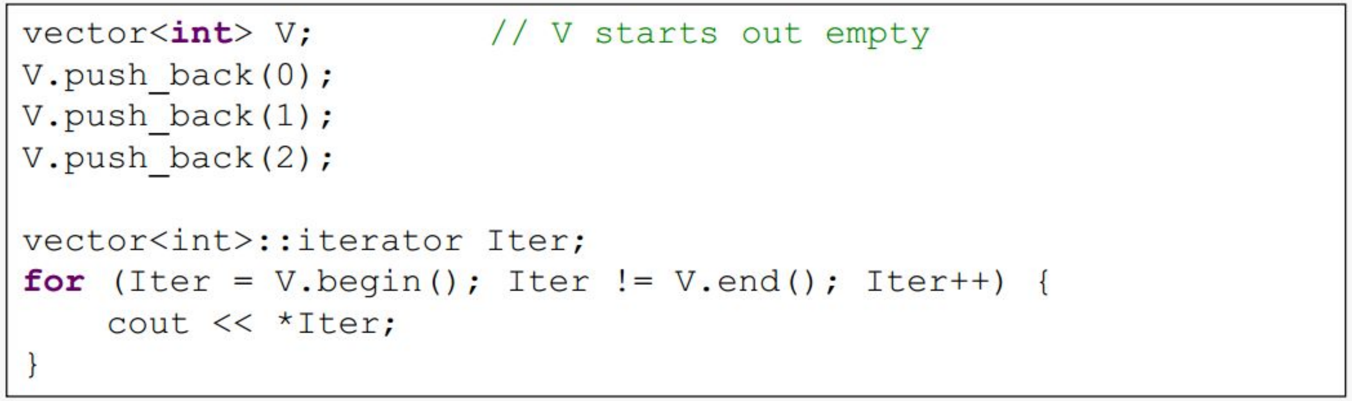
V.begin()

Returns an iterator that represents the first element in the vector (that is, it represents V[0]).

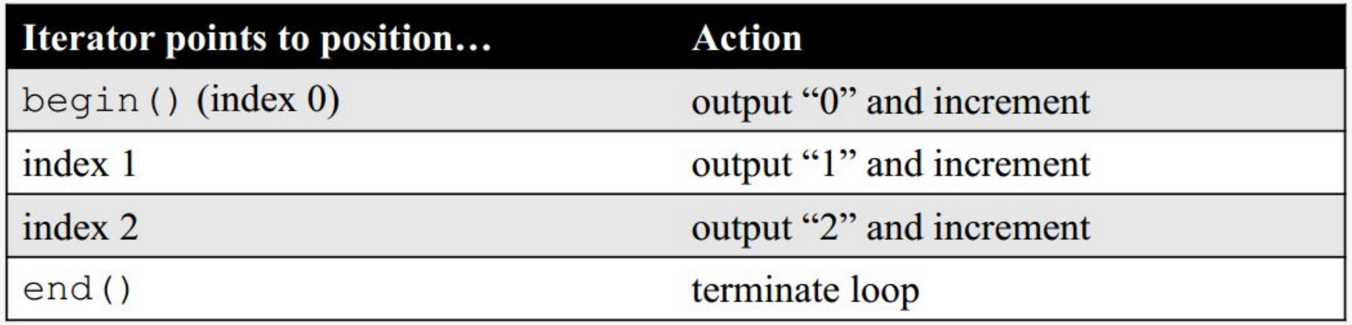
V.end()

Returns an iterator that represents **one element past the last element** in the collection (that is, it represents a position just after V[N – 1]).

Iterator Example

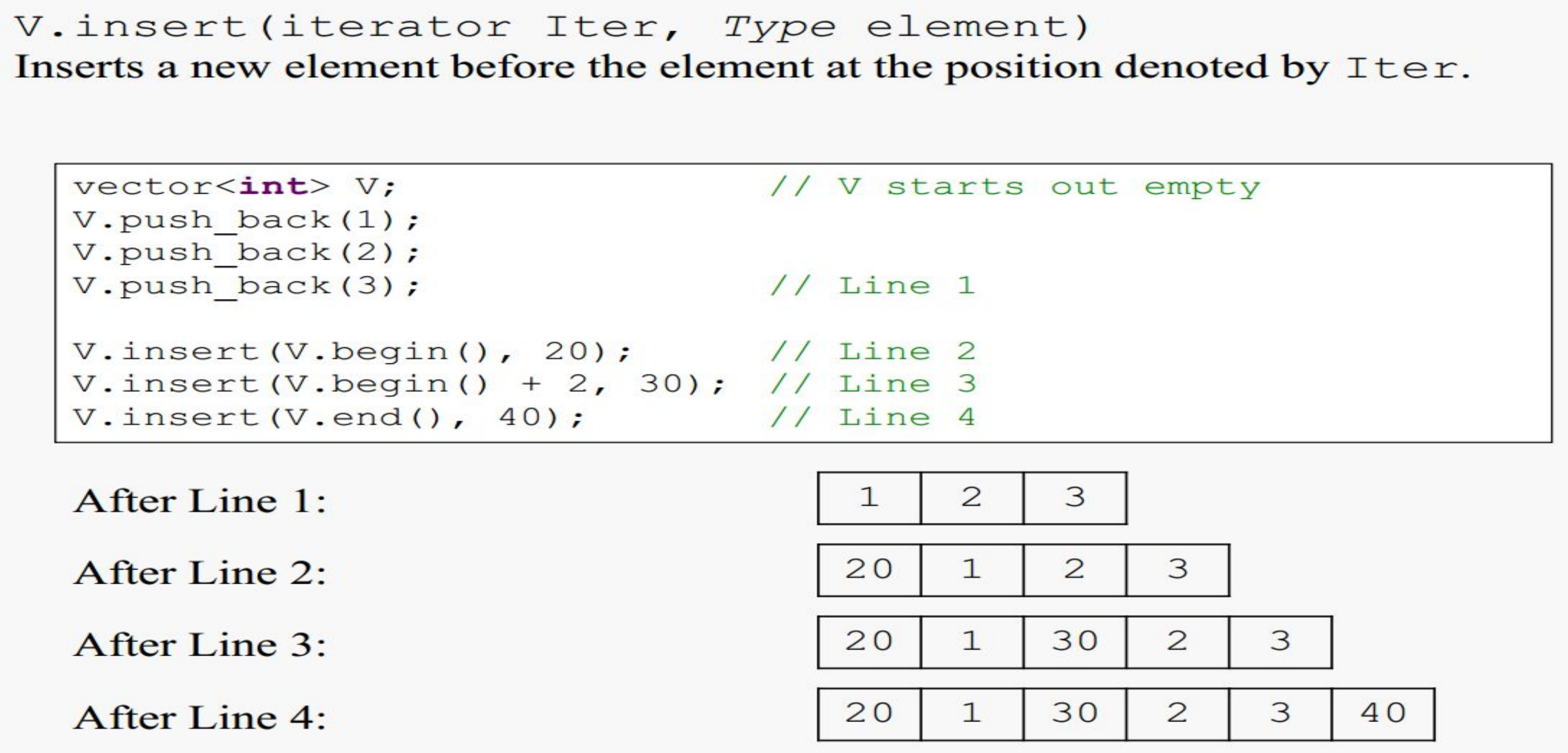


```
vector<int> V;         // V starts out empty
V.push_back(0);
V.push_back(1);
V.push_back(2);

vector<int>::iterator Iter;
for (Iter = V.begin(); Iter != V.end(); Iter++) {
    cout << *Iter;
}
```



Iterator points to position…	Action
`begin()` (index 0)	output “0” and increment
index 1	output “1” and increment
index 2	output “2” and increment
`end()`	terminate loop

Inserting at Arbitrary Positions

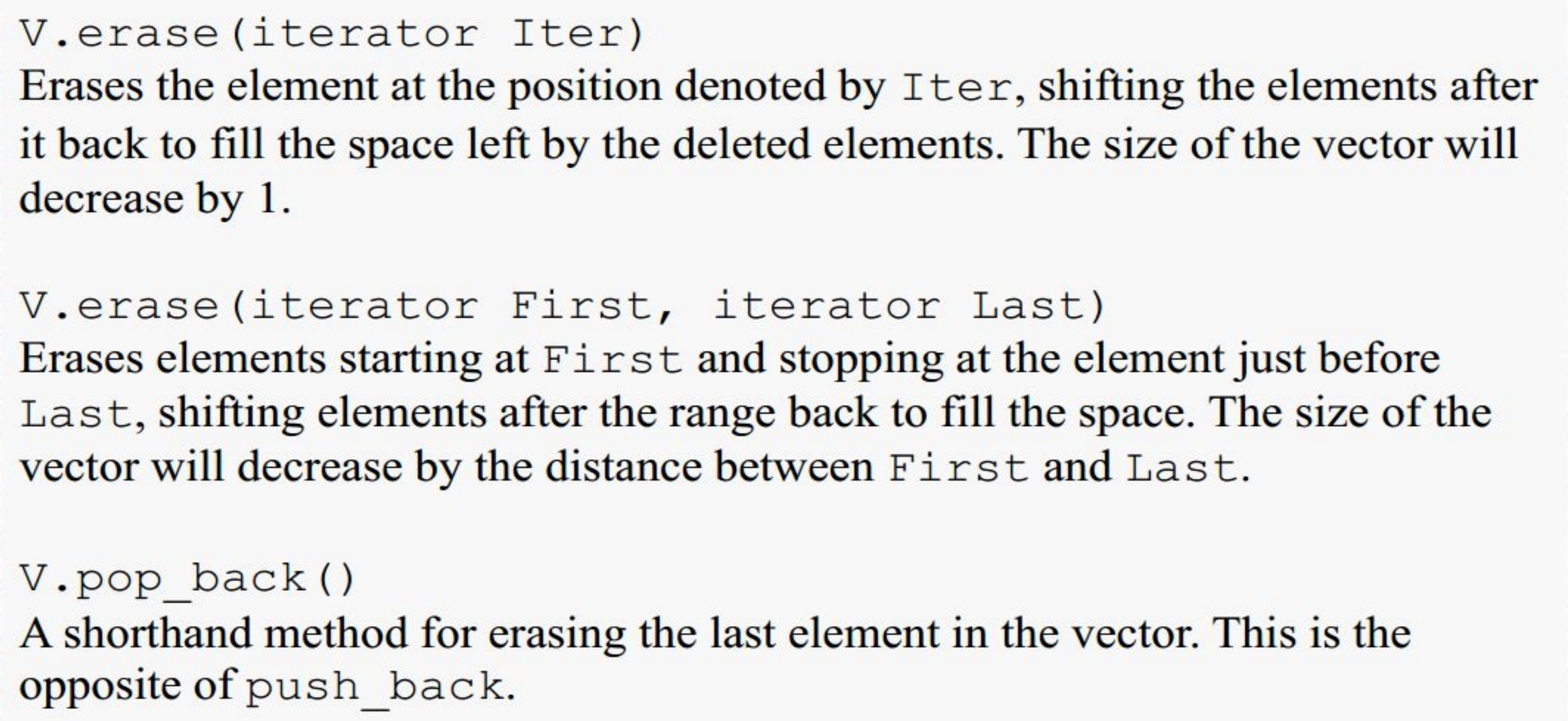


`V.insert(iterator Iter, Type element)`
Inserts a new element before the element at the position denoted by `Iter`.

```
vector<int> V;                      // V starts out empty
V.push_back(1);
V.push_back(2);
V.push_back(3);                     // Line 1

V.insert(V.begin(), 20);            // Line 2
V.insert(V.begin() + 2, 30);        // Line 3
V.insert(V.end(), 40);              // Line 4
```

After Line 1:	1	2	3			
After Line 2:	20	1	2	3		
After Line 3:	20	1	30	2	3	
After Line 4:	20	1	30	2	3	40

Erasing Elements in a Vector

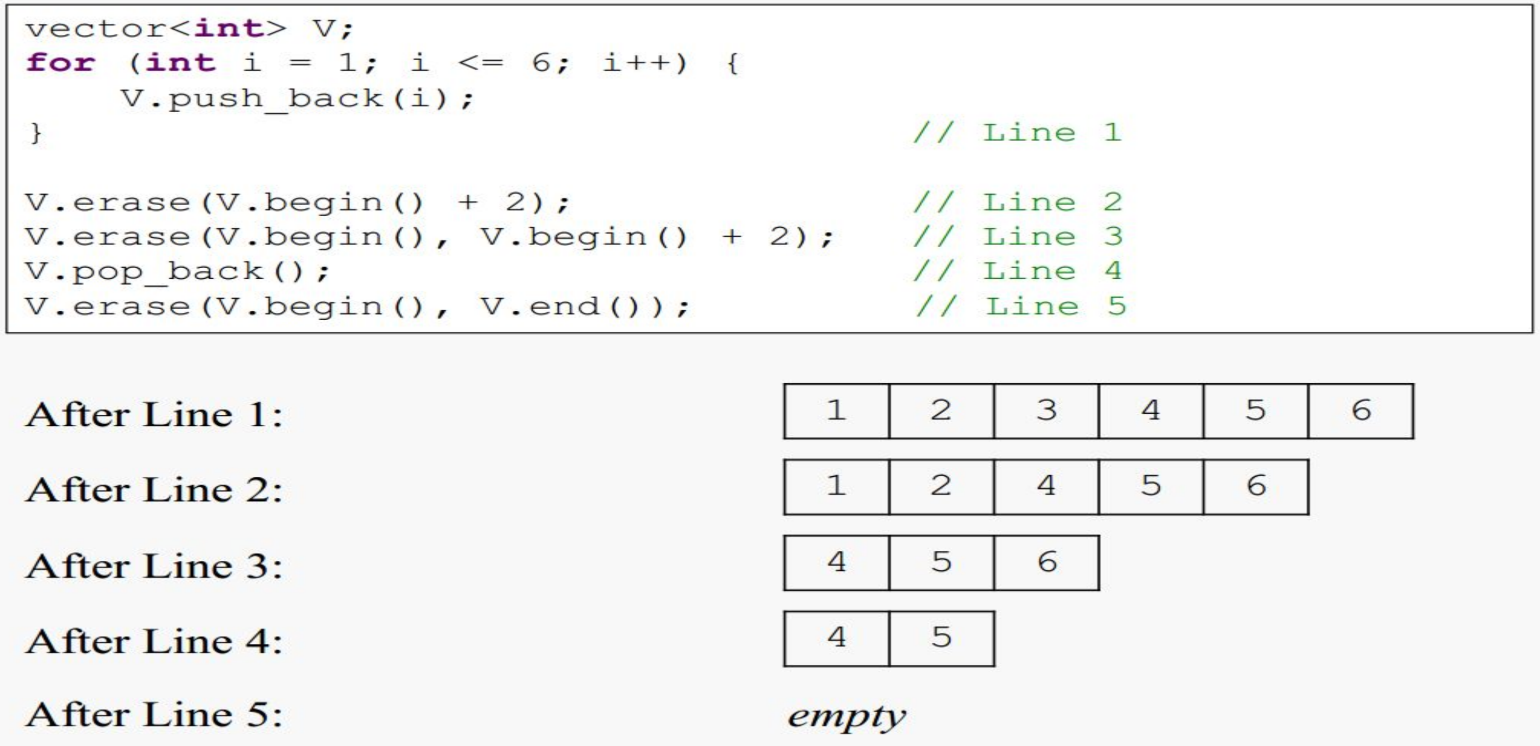


`V.erase(iterator Iter)`
Erases the element at the position denoted by `Iter`, shifting the elements after it back to fill the space left by the deleted elements. The size of the vector will decrease by 1.

`V.erase(iterator First, iterator Last)`
Erases elements starting at `First` and stopping at the element just before `Last`, shifting elements after the range back to fill the space. The size of the vector will decrease by the distance between `First` and `Last`.

`V.pop_back()`
A shorthand method for erasing the last element in the vector. This is the opposite of `push_back`.

Example



```
vector<int> V;
for (int i = 1; i <= 6; i++) {
    V.push_back(i);
}                                       // Line 1

V.erase(V.begin() + 2);                 // Line 2
V.erase(V.begin(), V.begin() + 2);      // Line 3
V.pop_back();                           // Line 4
V.erase(V.begin(), V.end());            // Line 5
```

After Line 1:	1	2	3	4	5	6
After Line 2:	1	2	4	5	6	
After Line 3:	4	5	6			
After Line 4:	4	5				
After Line 5:	*empty*					

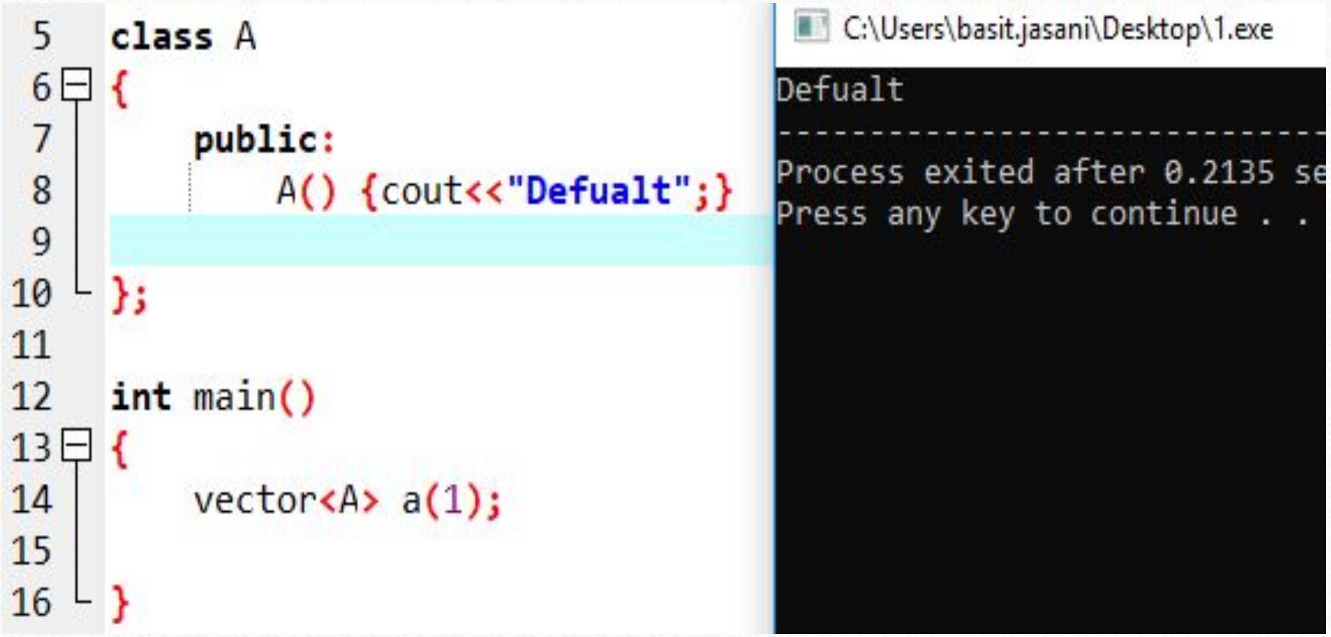
Storing class objects in a Vector

- Vectors can store any type of objects (not just built-in types), including those of classes that we create

- *Example:*

```
vector<MyClass> v;
```

Example



```cpp
class A
{
    public:
        A() {cout<<"Defualt";}

};

int main()
{
    vector<A> a(1);

}
```

```
Defualt
--------------------------------
Process exited after 0.2135 se
Press any key to continue . .
```