CSE 2020 Computer Science II

Module 2.1 – The ADT Class Vector

Instructor: Kerstin Voigt

School of CSE, CSUSB

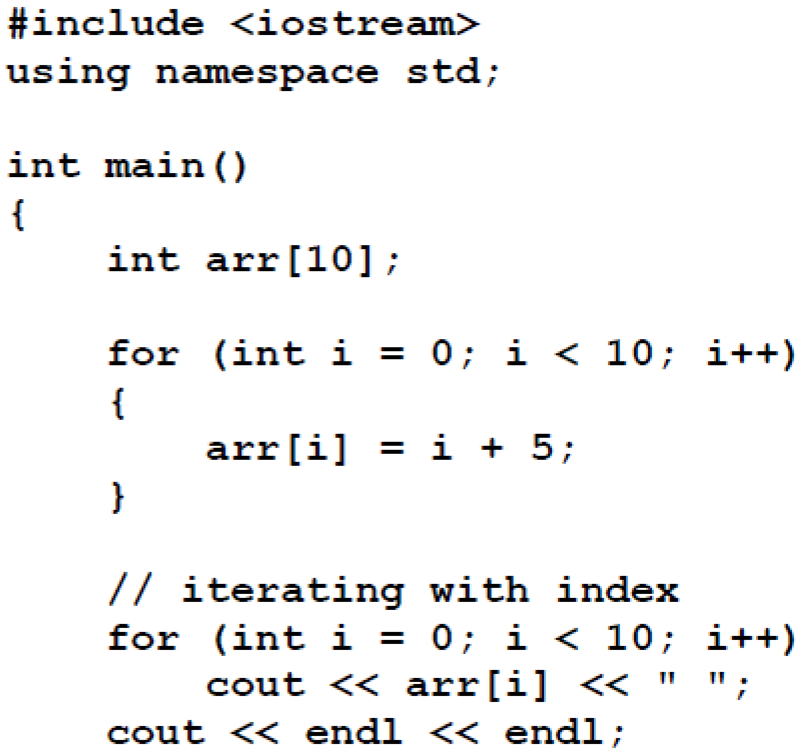
The C++ Vector Abstract Data Type (ADT)

A linear data structure

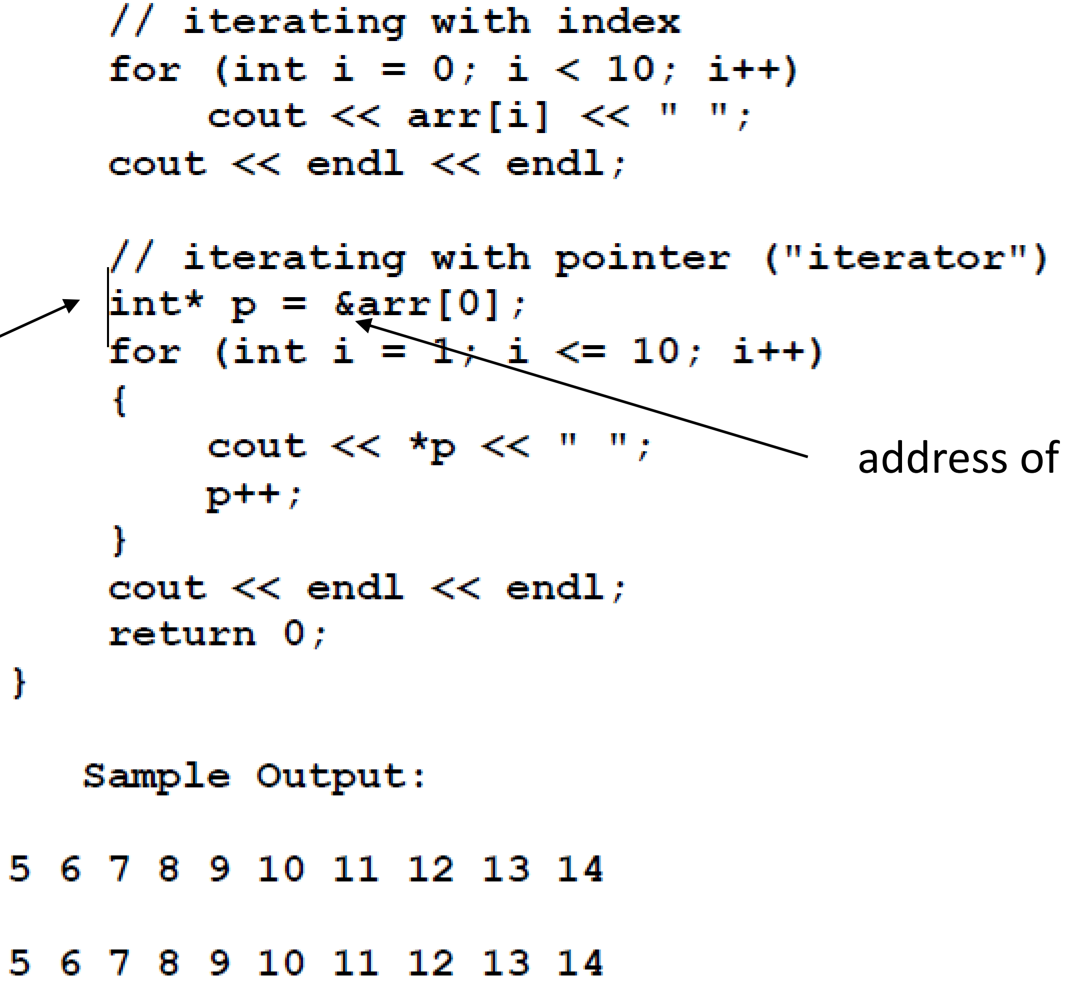
- Indexed from 0 … to N-1, N = number of values stored; cells filled contiguously (= “no gaps”)
- Each stored value must be of the same type
- Supports inserting and erasing of values, iteration over contents, various utilities.
- Efficiency of operations varies between O(1) and O(N)

O(1) means “constant cost”, does not vary with vector size

Why Vectors when there are Arrays?



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

int main()
{
    int arr[10];

    for (int i = 0; i < 10; i++)
    {
        arr[i] = i + 5;
    }

    // iterating with index
    for (int i = 0; i < 10; i++)
        cout << arr[i] << " ";
    cout << endl << endl;
```

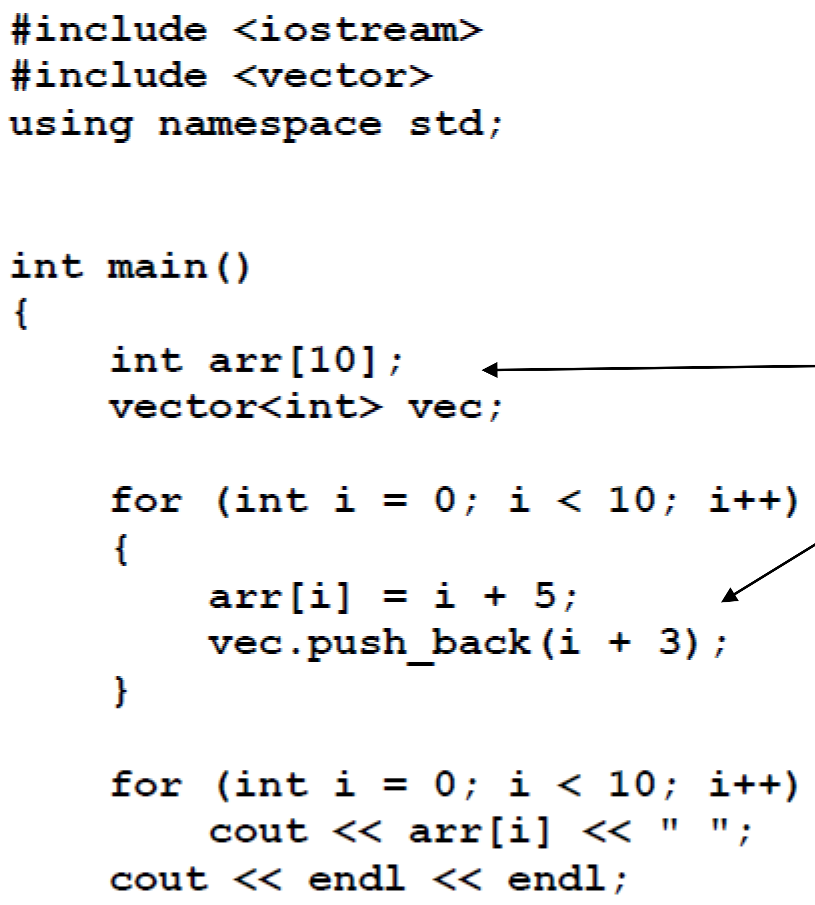


```
    // iterating with index
    for (int i = 0; i < 10; i++)
        cout << arr[i] << " ";
    cout << endl << endl;

    // iterating with pointer ("iterator")
    int* p = &arr[0];
    for (int i = 1; i <= 10; i++)
    {
        cout << *p << " ";
        p++;
    }
    cout << endl << endl;
    return 0;
}
```

```
  Sample Output:

5 6 7 8 9 10 11 12 13 14

5 6 7 8 9 10 11 12 13 14
```

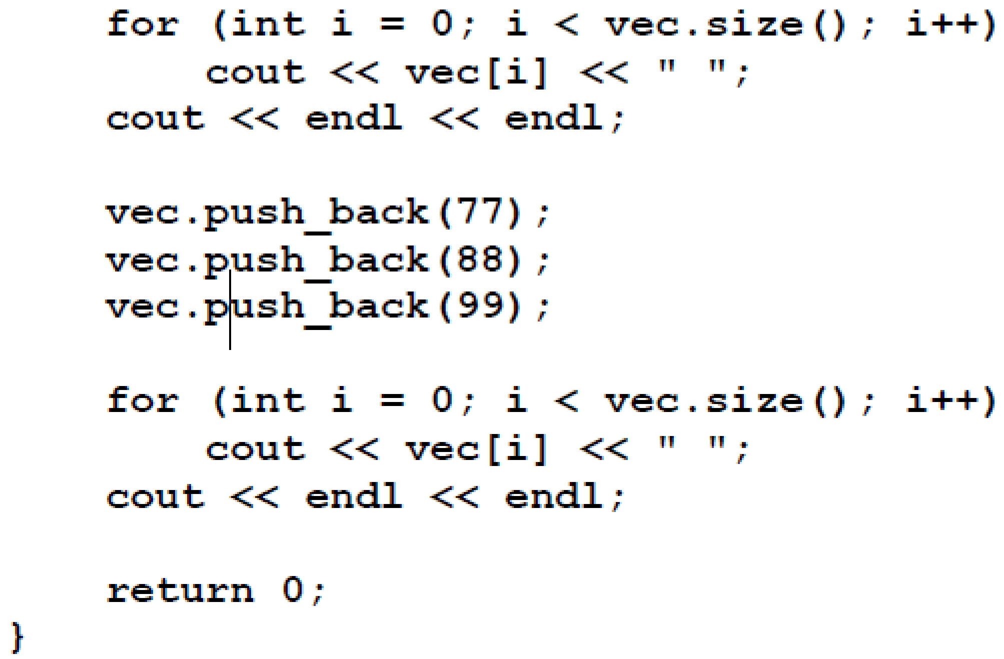
pointer is just a
memory address!

address of first array value



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


int main()
{
    int arr[10];
    vector<int> vec;

    for (int i = 0; i < 10; i++)
    {
        arr[i] = i + 5;
        vec.push_back(i + 3);
    }

    for (int i = 0; i < 10; i++)
        cout << arr[i] << " ";
    cout << endl << endl;
```

array vs. vector



```
    for (int i = 0; i < vec.size(); i++)
        cout << vec[i] << " ";
    cout << endl << endl;

    vec.push_back(77);
    vec.push_back(88);
    vec.push_back(99);

    for (int i = 0; i < vec.size(); i++)
        cout << vec[i] << " ";
    cout << endl << endl;

    return 0;
}
```

why not?

```
arr[10] = 77;
arr[11] = 88;
arr[12] = 99;
```

Answer:

… because arrays have a fixed size that cannot be changed

`int arr[10];` is one chunk of contiguous memory locations;

If more memory locations needed, we need to ask for an entirely new chunk of locations that will be allocated from some suitable parts of memory.

```
int want_more[100];
```

You could now copy all values from array arr into array want_more, and want_more has plenty of space for additional values.

BUT THIS TAKES WORK … frequent copying of values can become EXPENSIVE.

Sample Ouput:

```
5 6 7 8 9 10 11 12 13 14

3 4 5 6 7 8 9 10 11 12

3 4 5 6 7 8 9 10 11 12 77 88 99
```

Vectors are more “user friendly” than arrays in that …

…. They maintain their own size information

… Adjust their size as needed (upward)

Program used <vector> library from the Standard Template Library (STL)

✓***In CSE 2020, we build OUR OWN***

… but we need the C++ built-in arrays to help us

The C++ Vector Class

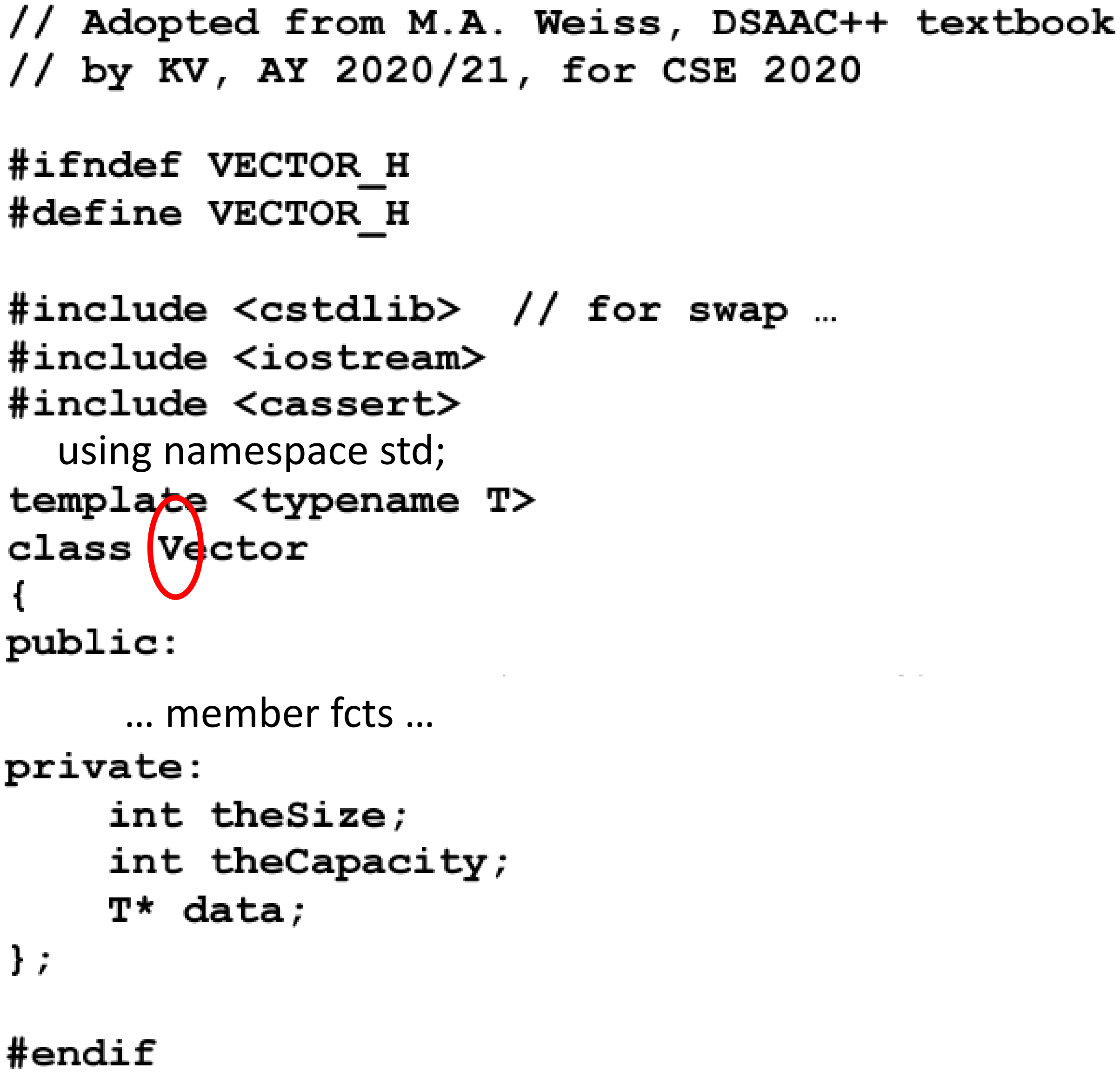
- A ***template class*** so that the stored values can be of any type
- ***All code in a single .h header file***

Be prepared to see:

- A fairly large class; code distributed over many slides
- A fair number of potentially new syntactic structures

- Needing ***attention to detail*** (applies to C++ in general)

C++ Class Vector – the Code



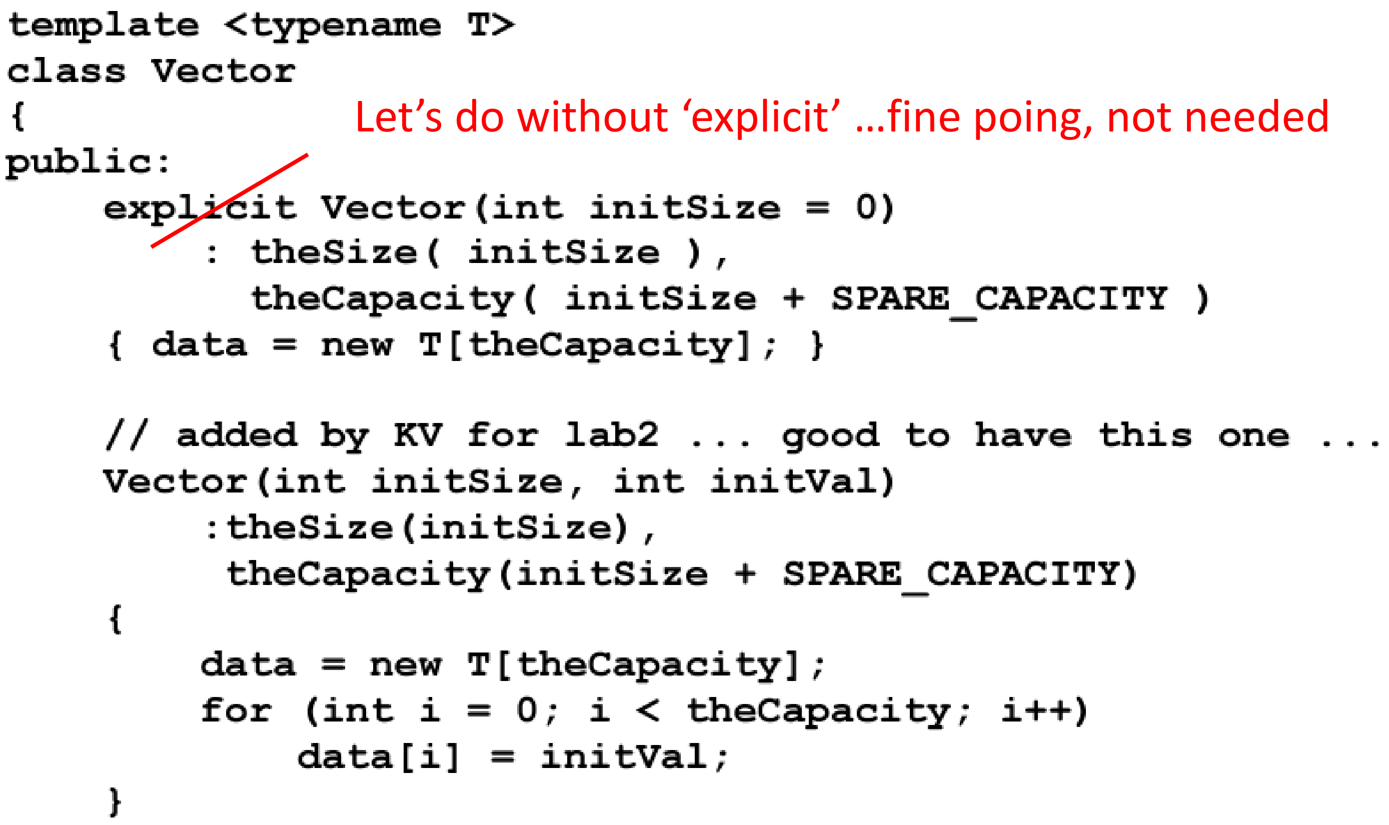
```
// Adopted from M.A. Weiss, DSAAC++ textbook
// by KV, AY 2020/21, for CSE 2020

#ifndef VECTOR_H
#define VECTOR_H

#include <cstdlib>  // for swap …
#include <iostream>
#include <cassert>
  using namespace std;
template <typename T>
class Vector
{
public:

      … member fcts …
private:
    int theSize;
    int theCapacity;
    T* data;
};

#endif
```

Note!

- Compiler directives #...
- Constructors and member functions to follow public:

T* data = …pointer of first type T item in array of size theCapacity, containing theSize many data items

C++ Class Vector – the Code cont.



```
template <typename T>
class Vector
{
public:
    ~~explicit~~ Vector(int initSize = 0)
        : theSize( initSize ),
          theCapacity( initSize + SPARE_CAPACITY )
    { data = new T[theCapacity]; }

    // added by KV for lab2 ... good to have this one ...
    Vector(int initSize, int initVal)
        :theSize(initSize),
         theCapacity(initSize + SPARE_CAPACITY)
    {
        data = new T[theCapacity];
        for (int i = 0; i < theCapacity; i++)
            data[i] = initVal;
    }
```

Let’s do without ‘explicit’ …fine poing, not needed

Simple Example:

```
class A
{
  public:
    A(int x = 0) : value(x){}
  private:
    int x;
};
```

Note that …

```
    A(int x = 0) : value(x){}
```

preferred over

```
    A(int x = 0) { value = x; }
```

Usage:

```
int main()
{ A myA;
  A yourA(55);
  …
}
```

C++ Class Vector – the Code cont.

```
Vector(const Vector& rhs)
    : theSize( rhs.theSize ),
      theCapacity( rhs.theCapacity ),
      data( nullptr )
{
    data = new T[theCapacity];
    for (int k = 0; k < theSize; ++k)
        data[k] = rhs.data[k];
}

Vector& operator= (const Vector& rhs)
{
    Vector copy = rhs;
    std::swap(*this, copy);
    return *this;
}
```

“right hand side” Vector in a declaration

Vector<int> v1; …
Vector<int> v2(v1);

Note!

Copy constructor and assignment operators are very similar

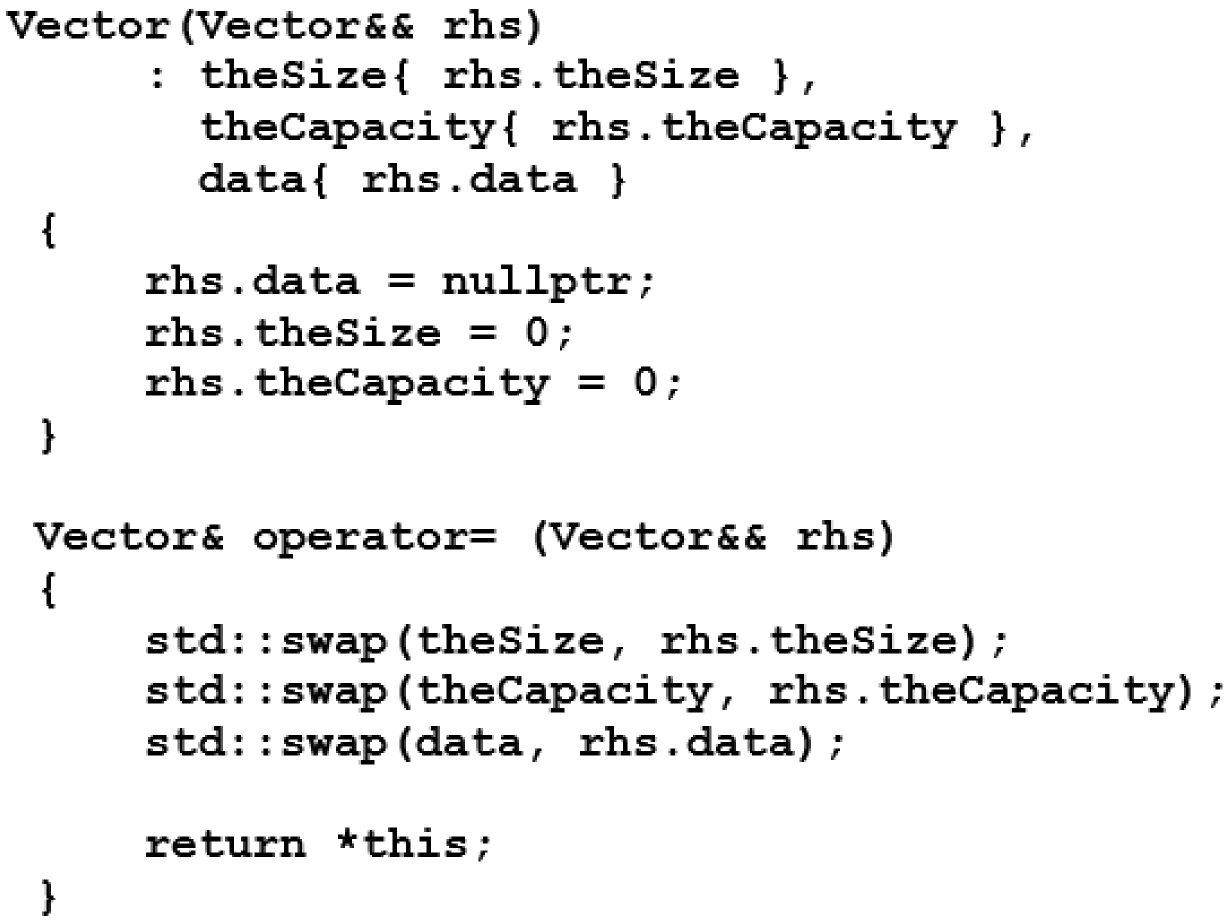
const parameter to preserve integrity of passed object

Reference parameter for efficiency

Exploiting standard swap

‘this’ is pointer of object to self; *this IS the object itself;

C++ Class Vector – the Code cont.



```
Vector(Vector&& rhs)
     : theSize{ rhs.theSize },
       theCapacity{ rhs.theCapacity },
       data{ rhs.data }
 {
     rhs.data = nullptr;
     rhs.theSize = 0;
     rhs.theCapacity = 0;
 }

 Vector& operator= (Vector&& rhs)
 {
     std::swap(theSize, rhs.theSize);
     std::swap(theCapacity, rhs.theCapacity);
     std::swap(data, rhs.data);

     return *this;
 }
```

Syntax && to indicate

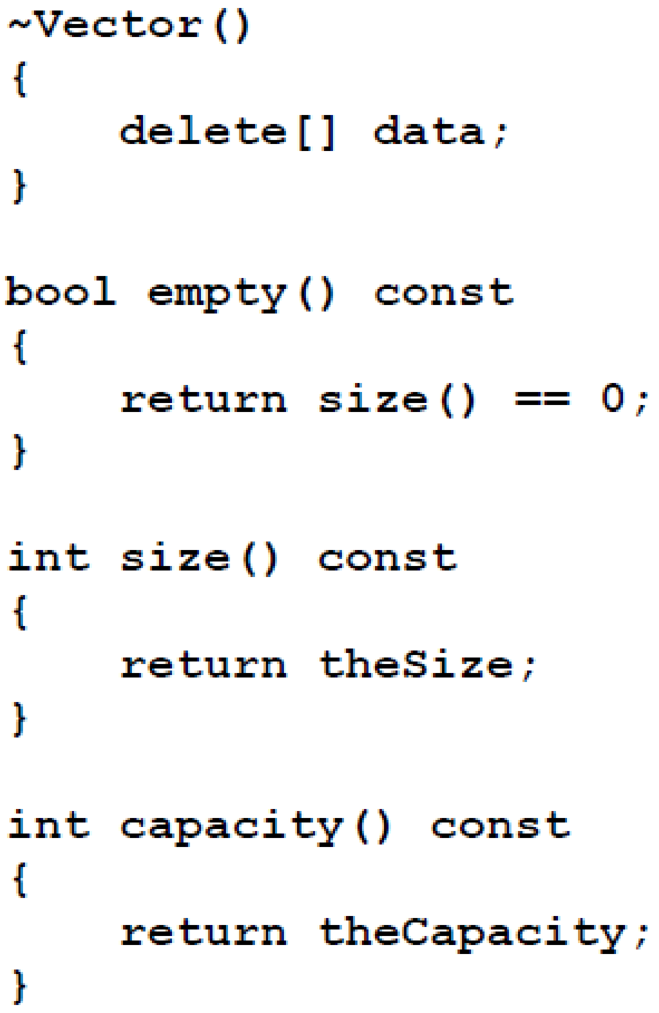
“Move Constructor” and

“Move Assign Operator”

C++11, even more efficient passing of parameter values

Advanced topic … we just copy the code because the compiler expects presence of this type of constructor and assignment operator.

C++ Class Vector – the Code cont.



```
~Vector()
{
    delete[] data;
}

bool empty() const
{
    return size() == 0;
}

int size() const
{
    return theSize;
}

int capacity() const
{
    return theCapacity;
}
```

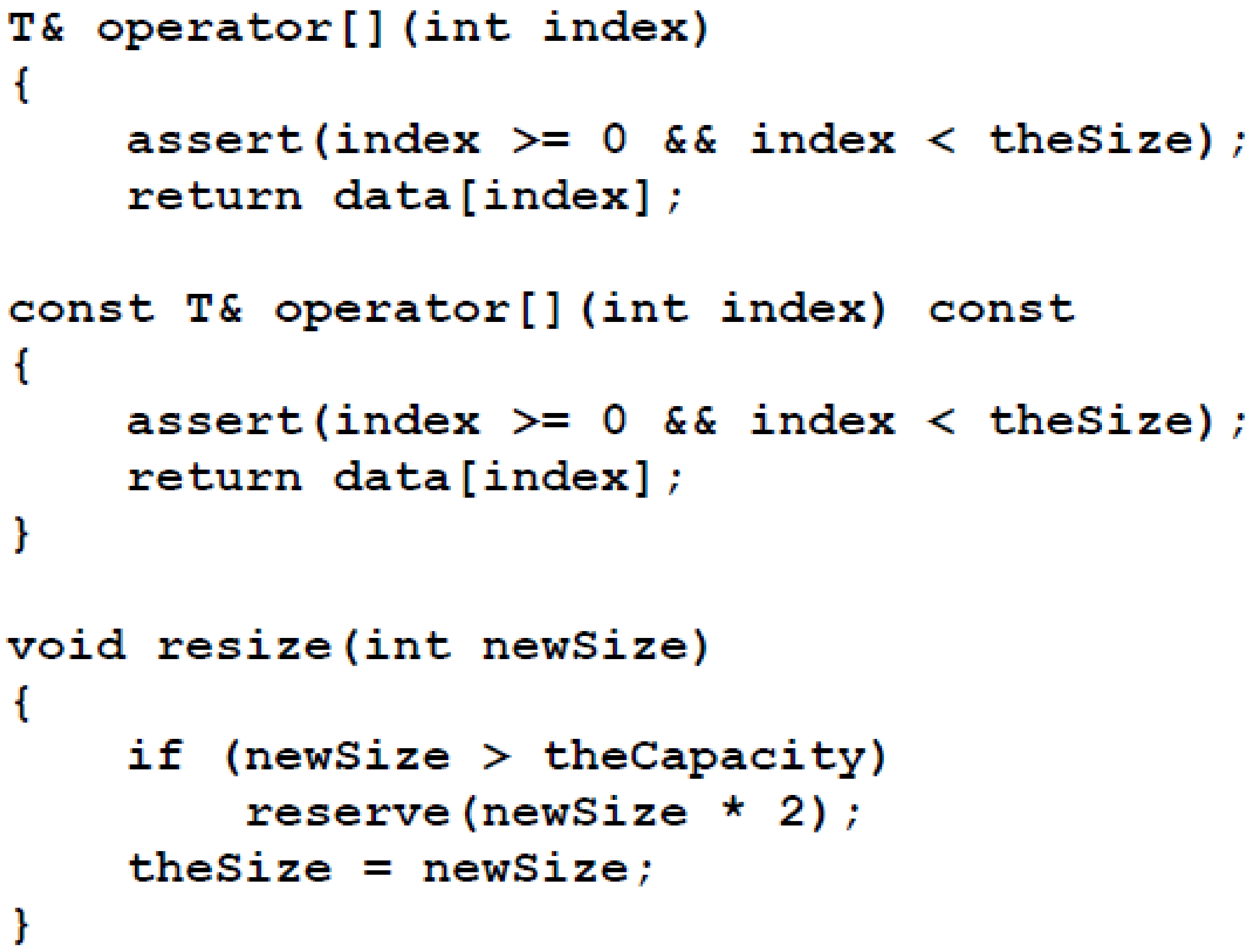
Note!

Classes with pointer data members need DESTRUCTOR;

destructor syntax like constructor with '~' prefix

const member functions are ACCESSOR functions

C++ Class Vector – the Code cont.



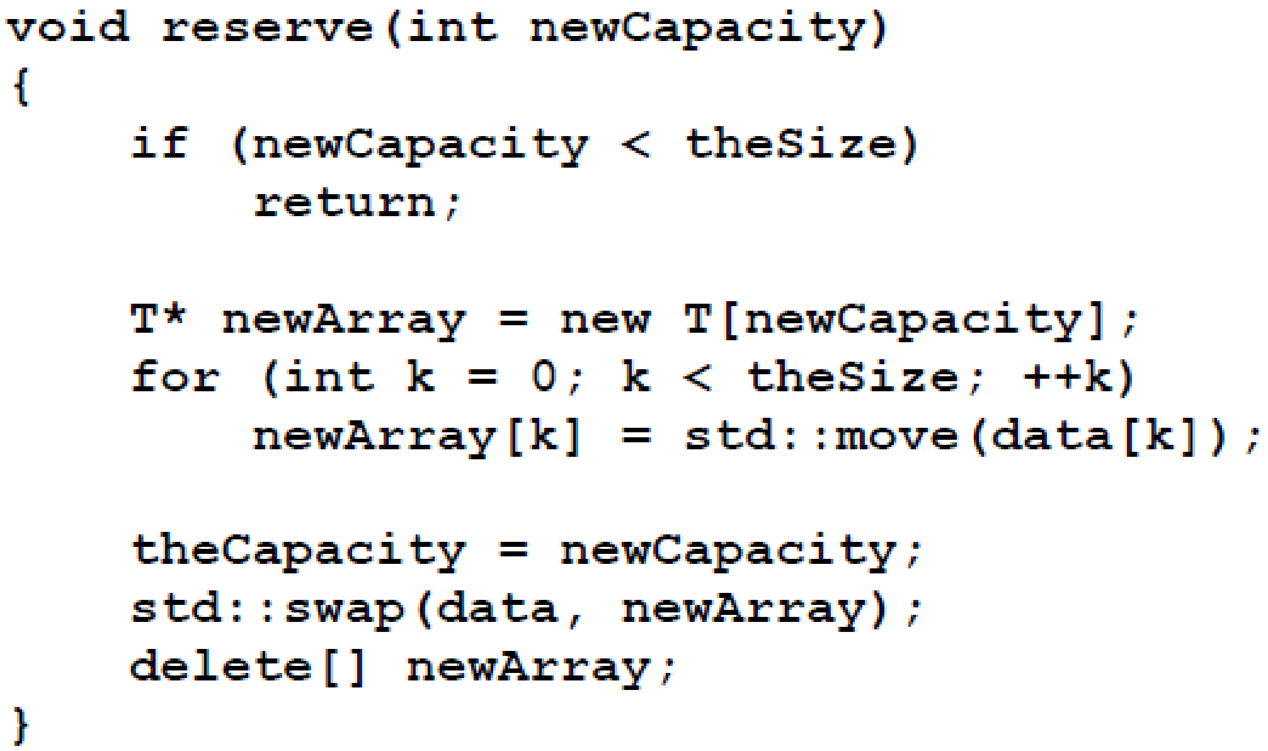
```
T& operator[](int index)
{
    assert(index >= 0 && index < theSize);
    return data[index];

const T& operator[](int index) const
{
    assert(index >= 0 && index < theSize);
    return data[index];
}

void resize(int newSize)
{
    if (newSize > theCapacity)
        reserve(newSize * 2);
    theSize = newSize;
}
```

Note!

`data is an array; thus use of array operator []`

`important vector feature ability to RESIZE as needed`

C++ Class Vector – the Code cont.



```
void reserve(int newCapacity)
{
    if (newCapacity < theSize)
        return;

    T* newArray = new T[newCapacity];
    for (int k = 0; k < theSize; ++k)
        newArray[k] = std::move(data[k]);

    theCapacity = newCapacity;
    std::swap(data, newArray);
    delete[] newArray;
}
```

Note!

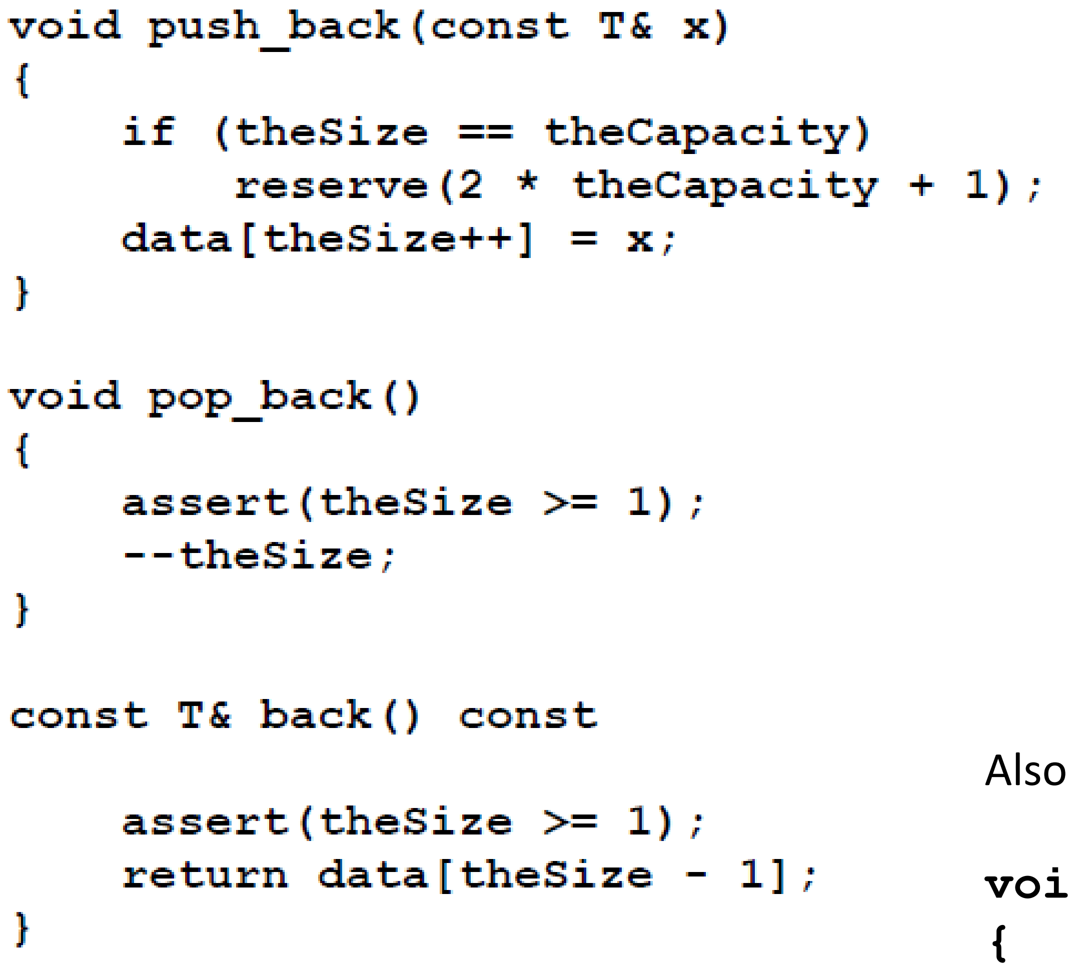
`dynamic allocation of new memory of desired larger size`

`std::move and std::swap for efficiency`

`important to DEALLOCATE helper variable newArray; prevents "memory leak"`

Story: “move data efficiently” from old data array space to new newArray space ; then swap pointers data and newArray; now data points to the newly allocated larger space that contains all the original values; discard the obsolete pointer newArray;

C++ Class Vector – the Code cont.



```
void push_back(const T& x)
{
    if (theSize == theCapacity)
        reserve(2 * theCapacity + 1);
    data[theSize++] = x;
}

void pop_back()
{
    assert(theSize >= 1);
    --theSize;
}

const T& back() const

    assert(theSize >= 1);
    return data[theSize - 1];
}
```

Also add:

```
void clear()
{
    theSize = 0;
}
```

Note!

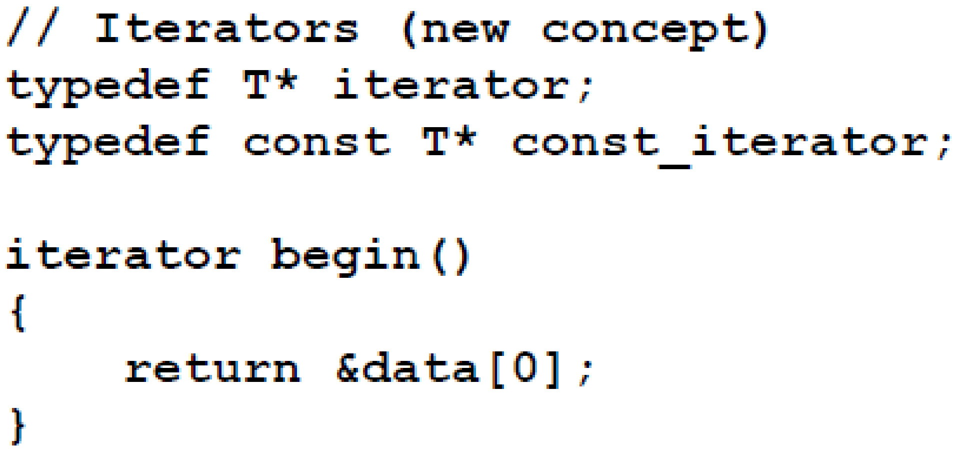
Two cases for push_back:

1. sufficient space for new value
2. insufficient space handled by reservation of new space

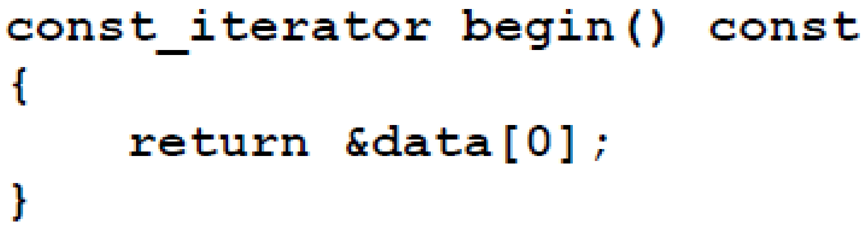
Easy and surprising manner of removal of element at 'right end' of the vector

("just ignore it”)

C++ Class Vector – the Code cont.



```
// Iterators (new concept)
typedef T* iterator;
typedef const T* const_iterator;

iterator begin()
{
     return &data[0];
}
```



```
const_iterator begin() const
{
     return &data[0];
}
```

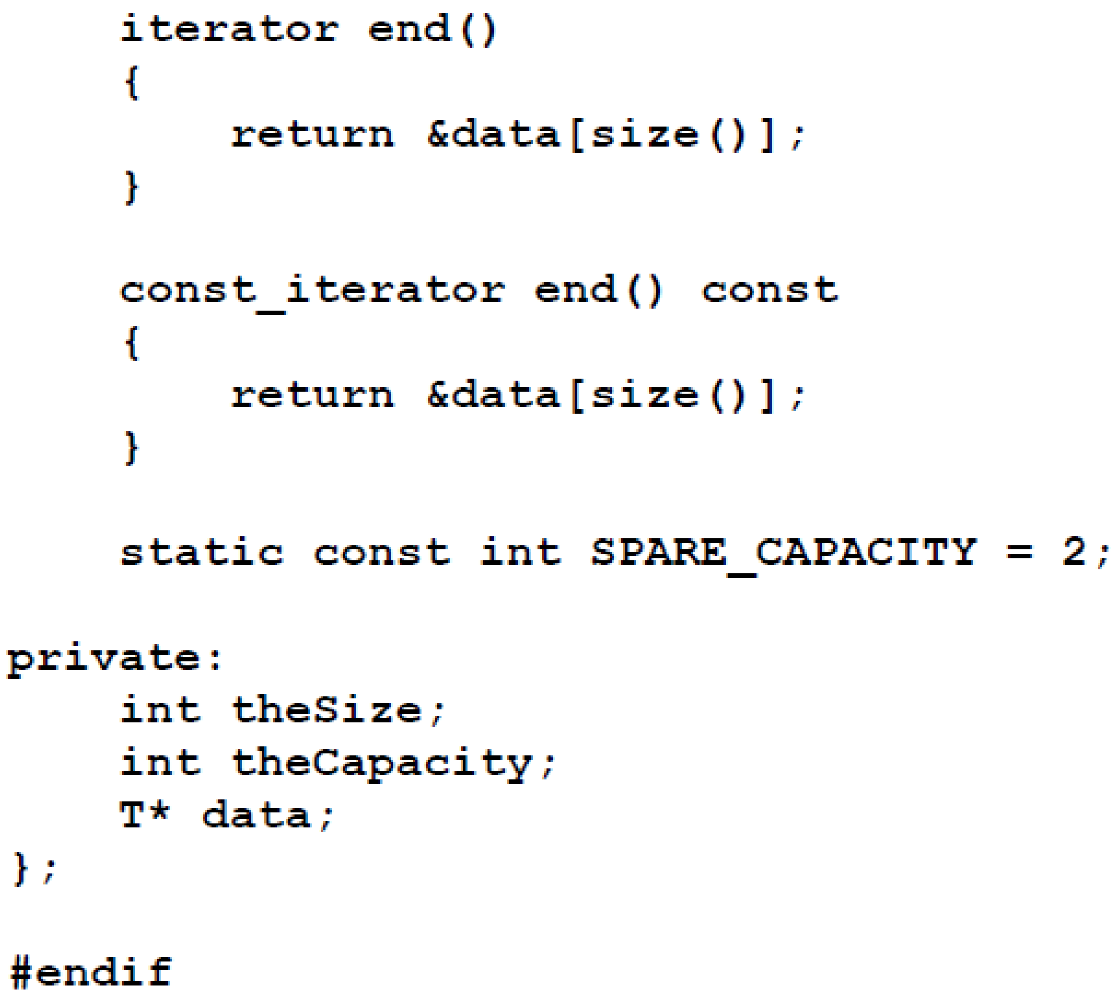
Note!

An iterator is a mechanism to systematically step through a collection of data items

Details of iterator depends on data structure for which it is designed

For vector data structure as implemented, iterators are simply pointers to items in data array

C++ Class Vector – the Code cont.



```
    iterator end()
    {
        return &data[size()];
    }

    const_iterator end() const
    {
        return &data[size()];
    }

    static const int SPARE_CAPACITY = 2;

private:
    int theSize;
    int theCapacity;
    T* data;
};

#endif
```

Note!

SPARE_CAPACITY is a static constant data member

The number of new memory cells to add as 'wiggle room' when sizing data array

Consider value 2 as an arbitrary default

#endif must at bottom of file whenever file lists #ifndef on top

Performance of class Vector

- We do not care about absolute numbers but the TREND by which the computational effort (time, number of significant ops) increases with the size N of the vector (= number of data items stored)

- For vectors, the computational “complexity” of its operations varies between constant O(1) and linear O(N).

  - O(1): the cost of an operation does not change with vector size
  - O(N): the cost of an operation increases in a linear fashion with vector size N.

- For some operations, the run-time context will determine whether performance is O(1) or O(N). (“best case”, “worst case”, “average case”)

Performance of class Vector by Operation

Vector::	O(1)	O(N)	Note
empty()			
size()			
capacity()			
operator []			
resize()			
reserve()			
push_back()			
pop_back()			
back()			

Textbook Reading:

Weiss, DSAC++, Chapters 2 & 3, sections ….

- 2.2, 2.3, 2,4; stress on concepts over mathematical treatment
- 3.3, as it pertains to Vector ADT
- 3.4

Make the effort to read C++ code *line-by-line* and ponder its meaning!

Reading additional sections is not discouraged as preview and understanding material in larger context.

*** End of Module 2.1 ***