Another implementation “Vector”

Vector (Dynamic Arrays)

- Array limitations
  - Must specify size first
  - May not know until program runs!
- Must "estimate" maximum size needed
  - Sometimes OK, sometimes not
  - "Wastes" memory
- Dynamic arrays
  - Can grow and shrink as needed

Creating Vector (Dynamic Arrays)

- Very simple!
- Use new operator
  - Dynamically allocate with pointer variable
  - Treat like standard arrays
- Example:

```
typedef double * DoublePtr;
DoublePtr d;
d = new double[10];    //Size in brackets
```

  - Creates dynamically allocated array variable *d*, with ten elements, base type double

Deleting Vector (Dynamic Arrays)

- Allocated dynamically at run-time
  - So should be destroyed at run-time
- Simple again. Recall Example:

```
d = new double[10];
… //Processing
delete [] d;
```

  - De-allocates all memory for dynamic array
  - Brackets indicate "array" is there
  - Recall: *d* still points there!
    - Should set d = NULL;

Function that Returns an Array

- Array type NOT allowed as return-type of function
- Example:
  int [] someFunction();   // ILLEGAL!
- Instead return pointer to array base type:
  int* someFunction(); // LEGAL!

Destructor Need

- Dynamically-allocated variables
  - Do not go away until "deleted"
- If pointers are only private member data
  - They dynamically allocate "real" data
    - In constructor
  - Must have means to "deallocate" when object is destroyed
- Answer: destructor!

Destructors

- Opposite of constructor
  - Automatically called when object is out-of-scope
  - Default version only removes ordinary variables, not dynamic variables
- Defined like constructor, just add ~

  -
    ```
    MyClass::~MyClass()
    {
       //Perform delete clean-up duties
    }
    ```

STL vector

Introduction to the STL `vector`

- Can hold values of any type:

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

- Automatically adds space as more is needed – no need to determine size at definition
- Can use `[]` to access elements

Declaring Vectors

- You must `#include<vector>`
- Declare a vector to hold `int` element:

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

- Declare a vector with initial size 30:

```
vector<int> scores(30);
```

- Declare a vector and initialize all elements to 0:

```
vector<int> scores(30, 0);
```

- Declare a vector initialized to size and contents of another vector:

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

Adding Elements to a Vector

- Use `push_back` member function to add element to a full array or to an array that had no defined size:

```
scores.push_back(75);
```

- Use `size` member function to determine size of a vector:

```
howbig = scores.size();
```

Removing Vector Elements

- Use `pop_back` member function to remove last element from vector:

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

- To remove all contents of vector, use `clear` member function:

```
scores.clear();
```

- To determine if vector is empty, use `empty` member function:

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

Priority Queue using vector

```
#include <queue>
#include <vector>
int main() {
priority_queue<int, vector<int>> pq;
    pq.push(30);
    pq.push(10);
    pq.push(50);
    pq.push(20);
    cout << "Priority Queue elements (max first): ";
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
}
```

```
/* Output:
Priority Queue elements (max
first): 50 30 20 10
*/
```