Vectors

ITP 165 – Fall 2015

Week 12, Lecture 2

Collections

- ***Collections*** (aka ***containers***) are a type of *data structure* that can store elements

- Examples of collections:
  - `int arr[5];`
  - `double* arr = new double[5];`
  - `std::string name;`

Arrays – The Simplest Collection

- Problem #1: In C++, arrays are created with a fixed size:

```
// Create an array of 5 elements (on the heap)
int* myArray = new int[5];

// Lazy create an array (on the stack)
// It'll be size of 3
double lazyArray[] = { 1.0, 2.0, 3.0 };
```

- Whether on the heap or on the stack, these sizes are fixed, ***we can't change them later***!

Arrays, Cont'd

- Problem #2: There's no way to get the number of elements in an array after you've declared it. This can cause errors:

```
// First five Fibonacci numbers
int fib[5] = { 1, 1, 2, 3, 5 };

// Oops, someone thought it was the first six!
for (int i = 0; i < 6; i++) {
    std::cout << fib[i] << std::endl;
}
```

Arrays, Cont'd

- Problem #3: If we don't initialize the array to values, it will have random garbage data.

```
// Create an array of 5 integers
int myArray[5];

// Since the above wasn't initialized
// who knows what this outputs!
std::cout << myArray[0] << std::endl;
```

Vector to the rescue

- A ***vector*** is a type of collection that:
  - Supports indexing, just like an array
  - Can check to make sure you aren't trying to access an invalid index
  - Automatically grows (and shrinks) in size as needed
  - Allows you to insert and remove elements whenever you want to

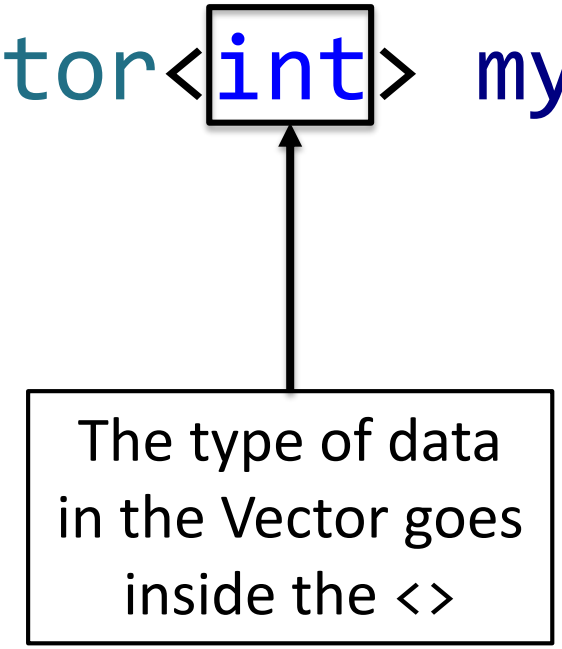
- Vector is part of C++ (just like `std::string`)

Vector in C++

- To use the C++ vector, `#include <vector>`

- To declare a Vector:

```
std::vector<int> myVector;
```



More examples of Vectors

```
// Vector of doubles
std::vector<double> numbers;

// Vector of std::strings
std::vector<std::string> names;

// Vector of Point variables
// (Assume Point was defined elsewhere)
std::vector<Point> coordinates;
```

Vector – size Member Function

- `size` returns the number of elements currently in the vector

- Usage:

```
std::vector<int> myVector;

// Vector starts empty, so this outputs 0
std::cout << myVector.size() << std::endl;
```

Vector – push_back Member Function

- Adds an element to the end of the vector

- Usage:

```
myVector.push_back(element);
```

- Where…
  - Element is what will be added to the end of the Vector. Note that the type of element ***must*** match the type of the Vector

Vector add example, Step by Step

```
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);
```

Vector add example, Step by Step

```
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);
```

Index
Value

size = 0

Vector add example, Step by Step

```
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);
```

Index	0
Value	1

size = 1

Vector add example, Step by Step

```
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);
```

Index	0	1
Value	1	1

size = 2

Vector add example, Step by Step

```
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);
```

Index	0	1	2
Value	1	1	2

size = 3

Vector add example, Step by Step

```
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);
```

Index	0	1	2	3
Value	1	1	2	3

size = 4

Vector add example, Step by Step

```
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);
```

Index	0	1	2	3	4
Value	1	1	2	3	5

size = 5

Vector – [ ] indexing

- You can use [ ] with Vector, just like an array:

```
// Construct Fibonacci number Vector
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);

// Output the Vector
for (int i = 0; i < fibNumbers.size(); i++) {
    std::cout << fibNumbers[i] << std::endl;
}
```

Vector – [ ] indexing, cont'd

- The [ ] can also be used to overwrite elements:

```
// Construct Fibonacci numbers
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);

// Overwrite index 0
fibNumbers[0] = 10;
```

Out-of-bounds Indexing

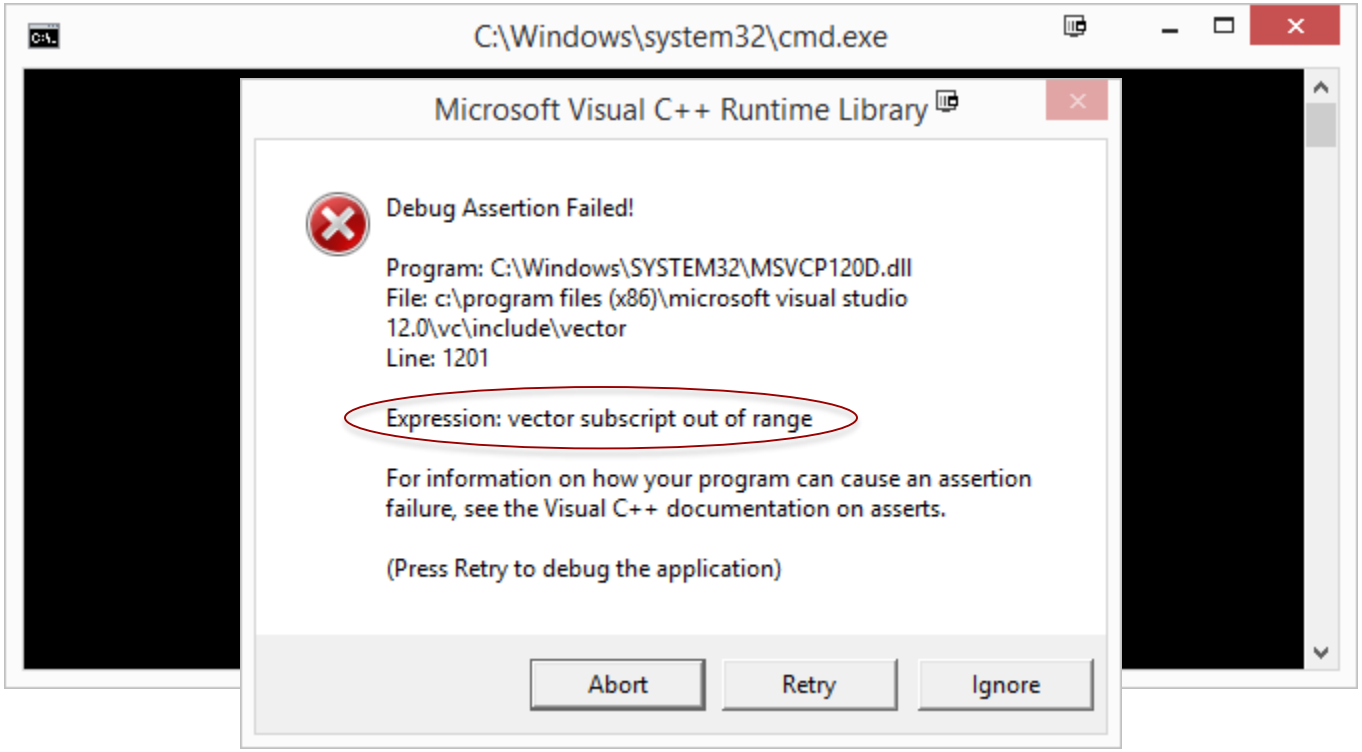
- For safety/debugging purposes, accessing an invalid index will crash the program, but first you will get an error message

- Example:

```
// Construct Fibonacci numbers
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);

// Overwrite index 5
fibNumbers[5] = 10; // OOPS!
```

Out-of-bounds Indexing, Cont'd



- An error message like this allows you to catch these bugs in development – much better than shipping with them

Vector – Another add example

- The advantage of being able to dynamically add elements: you can use loops to construct the Vector!

```
// Construct Fibonacci numbers
Vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);

// Construct through the first 10
for (int i = 1; i < 9; i++) {
   // The next number is the previous two
   int next = fibNumbers[i] + fibNumbers[i - 1];
   fibNumbers.push_back(next);
}
```

Vector – pop_back Member Function

- Removes the last element of the vector

- Usage:

```
myVector.pop_back();
```

Vector – remove, cont'd

- Remove at the last index can just chop off the last element:

```
// Construct Fibonacci numbers
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);
// Remove last element
fibNumbers.pop_back();
```

Index	0	1	2	3	4
Value	1	1	2	3	5

```
size = 5
```

Vector – remove, cont'd

- Remove at the last index can just chop off the last element:

```
// Construct Fibonacci numbers
std::vector<int> fibNumbers;
fibNumbers.push_back(1);
fibNumbers.push_back(1);
fibNumbers.push_back(2);
fibNumbers.push_back(3);
fibNumbers.push_back(5);
// Remove last element
fibNumbers.pop_back();
```

Index	0	1	2	3
Value	1	1	2	3

size = 4

Vector – empty Member Function

- Returns `true` if the Vector is empty

- Usage:

```
if (!myVector.empty()) {
    // If it isn't empty...
}
```

Vector – clear Member Function

- Removes everything from the vector

- Usage:

```
myVector.clear();
```

Lab practical 21

- There's a ZIP file with the lab assignment, make sure to use the included code