An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 20

Classes

From the Standard Library

Based on material developed by Prof. Abhiram G. Ranade

C++ Standard Library

- Many useful classes (and objects)
- We have already seen:
  - ostream (and objects cout, cerr), istream (object cin), string
- Coming up:
  - Containers (and the notion of templated classes)
    - vector, (ordered/unordered) (multi/non-multi) set and map, stack, queue, ...
    - Iterators (like pointers into containers)
  - Algorithms: Example sort
  - File streams

Vector

template argument

```
std::vector<int> A = {1, 2, 3};      // can init. like an array
A[0] = -1;                           // can access like an array
for(int i=0; i < A.size(); i++)
  cout << A[i] << " ";

std::vector<char> B;                 // can init. to be empty
B.push_back('a'); B.push_back('b'); // can grow dynamically
B.resize(4,'z');                     // now {'a','b','z','z'}
```

- std::vector is a flexible alternative to arrays
  - Can dynamically grow/shrink
  - Can be copied, and passed/returned by value as well as reference

Class Templates

template argument

```
std::vector<int> A = {1, 2, 3};      // can init. like an array
A[0] = -1;                           // can access like an array
for(int i=0; i < A.size(); i++)
  cout << A[i] << " ";

std::vector<char> B;                 // can init. to be empty
B.push_back('a'); B.push_back('b'); // can grow dynamically
B.resize(4,'z');                     // now {'a','b','z','z'}
```

vector.h

```
template<typename T>
class vector {
  ...
};
```

- The library doesn't define any vector class! Instead a header file has the template for generating the class definitions
- At the time of compilation, for the specified template arguments, the class definitions are written for you by the compiler using the template

Iterators

```
int A[4] = {1, 2, 3, 4};
for(int* p=A; p!=A+4; ++p)
  cout << *p << " ";
cout << endl;
```

```
typedef std::vector<int>::iterator itr;
std::vector<int> V = {1, 2, 3, 4};
for( itr p=V.begin(),q=V.end(); p!=q; ++p)
  cout << *p << " ";
cout << endl;
```

- Recall that a pointer could be used to iterate through an array
- Containers like vector provide the same ability
  - But instead of a pointer type, a suitable *iterator* class (defined within the container class) is used
  - An iterator class should provide some of the features of "pointer arithmetic" (e.g., `++p`, `p!=q`) and the dereferencing operator

Iterators

```
int A[4] = {1, 2, 3, 4};
for(int* p=A; p!=A+4; ++p)
  cout << *p << " ";
cout << endl;
```

```
std::vector<int> V = {1, 2, 3, 4};
for(auto p=V.begin(),q=V.end(); p!=q; ++p)
  cout << *p << " ";
cout << endl;
```

- Often the compiler can deduce the type of a variable automatically, based on the initialisation
  - But the program should "ask" for it, using the keyword `auto` when declaring the variable
- Especially useful when the type is complicated/non-standard

Range-Based `for` Loop

```
using std::vector;
vector<int> V = {1,2,3,4};
for( int x : V)
  cout << x << " ";
cout << endl;
```

```
std::vector<int> V = {1, 2, 3, 4};
for(auto p=V.begin(),q=V.end(); p!=q; ++p)
  cout << *p << " ";
cout << endl;
```

- There is a simpler syntax specifically for looping over a container

```
for( auto p=V.begin(),q=V.end(); p!=q; ++p) {
  int x = *p;
  cout << x << " ";
}
```

Vector

- Already encountered some member functions:
  - push_back(*value*), `operator[]`, `begin()`, `end()`, `size()`, `resize(`*n*`)`
- Several more available:
  - `operator=` copies all the entries from another vector
  - `at(`*i*`)`, similar to `operator[]`, but also does bound checks
  - `pop_back()`, `front()`, `back()`, `erase(`*first*`,` *last*`)`, `insert(`*pos*`,` *value*`)`, `clear()`, ...
  - Several constructors: `vector()`, `vector(`*another*`)`, `vector(`*init_list*`)`, `vector(`*count*`)`, `vector(`*count*`,` *value*`)`, ...

`vector<int> A = {1,2,3};`
converts {1,2,3} to an std::initializer_list

Example: A Matrix

- An example: Construct a matrix data structure using vectors
  - C++ has multi-dimensional C-style arrays
    - e.g., `double M[5][10];`
  - But we would like to have the same flexibility as offered by vectors over C-style arrays
    - Idea: Use vectors to implement a matrix. Each row of the matrix will be a vector
    - Why not each column?
      - `M[i][j]` should be the element in the i^{th} row and j^{th} column. So we'll let `M[i]` be a vector corresponding to the entire i^{th} row.

Example: A Matrix

- Construct a matrix data structure using vectors
  - Each row of the matrix will be a vector

```
template<typename T> class matrix {
  int nrows, ncols;
  vector<vector<T>> R;
public:
  matrix() = default;  // retain the default constructor. creates a 0 x 0 matrix
  matrix(int m, int n) // a constructor which takes the dimensions
       : nrows(m), ncols(n), R(m,std::vector<T>(n)) {}

  vector<T>& operator[] (int i) { return R[i]; } // reference to i-th row
  int rows() {return nrows;}
  int cols() {return ncols;}
};
```

See example code for another constructor, `matrix(initializer_list<initializer_list<T>>)`

Risky: outside code can resize each row arbitrarily.

See example code for a fix (using std::span in C++20)

Example: Command Line Arguments

- Many programs take command line arguments: e.g.,

```
$  g++ myprog.cpp -o myprog
```

- In C and C++ programs these arguments are accessible as arguments to main
- Each argument is a C-style string: a char array, with "NUL" (0) as the last char

```
int main(int arg_count, char* arg_strings[]) {
  // in the above example, arg_count == 4
  // arg_strings[0] points to first element of {'g','+','+',0}
  // arg_strings[1] : {'m','y','p','r','o','g','.','c','p','p',0}
  // arg_strings[2] : {'-','o',0}
  // arg_strings[3] : {'m','y','p','r','o','g',0}
}
```

Example: Command Line Arguments

- Can easily convert a C-style array of C-style strings to a vector of strings
- Constructors involved: `vector` (*first, last*), where *first* and *last* are iterators; `string`(char* *str*) where *str* is a C-string

```
using std::vector; using std::string;
int main(int arg_count, char* arg_strings[]) {
  vector<string>  args ( arg_strings, arg_strings + arg_count );
  // now can use the vector to access each argument as a string
}
```

More Containers

- Set and Map
  - Multi or non-multi
  - Ordered or unordered
- Set vs map:
  - Set stores a set of "keys." We can check if a given key is in a set or not.
  - Map stores (key, value) pairs. We can check if a given key appears in the map or not, and if it does, what value is associated with it.
- Multi vs. non-multi: A key can occur many times or only once
- Ordered vs. unordered: Can the items be iterated over in a sorted order or not

Example: Counting Words

- Read in some text and print out the frequency of different words
  - Will assume text already given as a vector of strings

```
#include <vector>
#include <map>
using std::map; using std::string; using std::vector;

void print_freqs(const vector<string>& text) {
  map<string,int> counts;
  for(const auto& w: text)
    ++counts[w];
  for(const auto& entry: counts) // can iterate over a map
    cout << entry.first << ": " << entry.second << endl;
}
```

key is string, value is int

operator[] to access value stored for key. If key didn't exist, it will be initialised with value 0.

entry is of type `std::pair`, which has members `first` (of key type) and `second` (of value type)

Algorithms

- The standard library has implementations of several useful algorithms for items in a container
  - Sorting, searching (binary search), finding max/min element, ...
  - Implemented in a flexible manner
    - Templated on a data type that is specified at the time of invoking
    - Can specify the function used for comparison (say, in sorting)

Algorithms: Sorting

- Sort the elements in a container
  - Containers like std::vector (with a "random access" iterator)
  - Elements which have `operator<`

```
#include <iostream>
#include <vector>
#include <algorithm>
using std::string; using std::vector;

void sort_and_print_freqs(vector<string>& text) {
  sort(text.begin(),text.end());  // sort the vector in place
  // now scan the sorted vector, accumulate count and print
  // ..
}
```

to sort in descending order, can add argument `std::greater<string>()`

Files

- Many programs only use their standard input and output (std::cin, std::cout, and std::cerr)
  - Often enough, since the shell can be used to "redirect" them to files

```
$  ./a.out  < inputfile  > outputfile  2> errorfile
```

- However, sometimes will need to handle multiple files (possibly in parallel)
- From a C++ program we can access files on the filesystem (as allowed by the Operating System)
  - Using *file streams* (similar to i/o streams)

Files

Demo

```
#include <iostream>
#include <fstream>
using std::string;

void join(string fname1, string fname2, string outname) {
  std::ifstream f1 (fname1);
  std::ifstream f2 (fname2);
  std::ofstream fout (outname);
  while(f1 || f2) {
    int x, y;
    if (f1)
      (f1 >> x) && fout << x;
    if (f2)
      (f2 >> y) && fout << '\t' << y;
    fout << std::endl;
  }
}
```

The files are opened here by the `ifstream` and of `ofstream` constructors

`operator bool()` is the same as `!fail()`

formatted input from `ifstream` (like from `cin`) using `operator>>`
formatted output to `ofstream` (like from `cout`) using `operator<<`

All the files are closed here by the `ifstream` and `ofstream` destructors

Summary

- Containers (and the notion of templated classes)
  - vector
  - (ordered/unordered) (multi/non-multi) set and map
- Some new concepts
  - Class templates
  - Iterators (like pointers into containers)
  - Range-based for loop
  - auto
  - Command-line arguments
- Algorithms in the standard library
  - Example: sort
- File streams