Ali Malik
CS106L Winter 2018

Handout 12
Jan 30th, 2018

C++ Reference

The purpose of this handout is to provide a summary and reference of the concepts covered in lecture as well as common constructs used in C++. It will also contain quick reference comparisons between the Stanford and standard libraries. At the end of each section is a set of exercises that you could try out to practice the relevant concepts at a less intimidating level than the assignments. I will happy look at your implementations of these and provide style feedback if you email your code to me. Eventually, when time allows, I hope to have some involved examples in each section.

Disclaimer: The stylistic choices and recommendations are, at times, opinions and not necessarily the absolute optimal way to deal with the problem at hand. Over time, you will develop your own style of programming so take these to be a good starting point to branch off from.

Streams

C++ handles I/O by using the abstraction of streams. Streams consist of two subcategories: input streams and output streams. Output streams are used to write data out to an external interface (console, file, socket) using the stream insertion operator <<. Similarly, input streams read data from an interface (console, file, web server). You can extract data from an input stream using the stream extraction operator >> or std::getline(). The extraction and insertion operators are powerful since they do automatic type conversions but, for istreams, the extraction operator only pulls out a whitespace separated token at a time. For purposes where you need to get entire lines at a time, std::getline() is the alternative you need.

Here are some useful stream related idioms in C++.

Opening a file for reading

You can read a file using the ifstream class by creating an ifstream object and telling it which file top open. This can either be done in the constructor (first version) or using the .open(filename) method (second version).

```
// first version
ifstream input("filename.txt");

// second version
ifstream input;
input.open("filename.txt");
```

You can test if the stream opened the file successfully by checking it in a condition i.e.

```
ifstream input("filename.txt");
if(!input) {    // failed to open file
    cout << "File failed to open" << endl;
}
```

Reading from stream line by line

haiku.txt

Space is limited Haiku make it difficult To finish what you	```ifstream input("haiku.txt"); string line; while(std::getline(input, line)) { // do something with line }```

Reading from a list of numbers

numbers.txt

14 125 56 6	```ifstream input("numbers.txt"); int num; while(input >> num) { cout << num + 1 << endl; }```

Using istringstream for conversions

```
string line = "Dubai 24 3.14";
istringstream input(line);
string country;
int num;
double pi;
// First word will be read into the country variable
// and the second number will converted to an int and stored in num
input >> country >> num >> pi;
```

Reading from a list of number pairs

There are two possible ways to do this. The first uses the extraction operator (>>) while the second uses getline. Although the first is shorter, I usually use getline in most of my programs because if you aren't careful, mixing getline and >> causes bugs like the one we saw in lecture. The second version reads a line and then uses a stringstream to do the tokenising and conversions.

coords.txt

```
14 55
21 18
26 25
90 83
```

```
// First way
ifstream input("coords.txt");
int num1, num2;
while(input >> num1 >> num2) {
    cout << num1 + num2 << endl;
}

// Alternative way
ifstream input("coords.txt");
int num1, num2;
string line;
while(std::getline(input, line)) {
    istringstream strStream(line);
    int num1, num2;
    strStream >> num1 >> num2;
    cout << num1 + num2 << endl;
}
```

Practice Exercises:

1. Write a program that reads a list of numbers from the console and prints the max, min, and average of these numbers.

2. Write a program to determine whether a given text file is a Haiku. For the purpose of this question, a Haiku is a three line poem where the first line has five syllables, the second line has seven syllables, and the final line has five syllables. You can

assume you have access to a function that takes a word and returns the total number of syllables in it

```
int syllablesIn(const string& word);
```

Implement the following method which takes any input stream as a parameter and returns whether or not the data in the stream is a haiku. Hint: the getline() and istringstream combination will be useful here.

```
bool isHaiku(istream &input);
```

3. In the study of natural language processing and document forensics, a really common functionality is to get statistics about a particular text. Write a function

```
DocumentInfo statisticsFor(istream& source);
```

that takes as input a stream containing the contents of a file and returns a DocumentInfo object (described later) containing the number of sentences, words, and syllables contained in that file. A word constitutes of any whitespace separated token and any word that ends with a period, exlamation, or question mark signifies the end of a sentence. TheDocumentInfo object is a struct declared as

```
struct DocumentInfo {
    int numSentences;
    int numWords;
    int numVowels;
};
```

Hint: The extraction operator (>>) will be useful. As a first pass, try and just print each token in the file. Then add on the functinality of taking statistics on a word. Also remember if you declare a DocumentInfo object but don't initialise the values to 0, they may contain garbage data. You can initialise a zeroed out DocumentInfo object by declaring

```
DocumentInfo info{0, 0, 0};
```

Containers and Iterators

`std::vector<T>`

What you want to do	Stanford Vector<int>	std::vector<int>
Create an empty vector	`Vector<int> v;`	`vector<int> v;`
Create a vector with n copies of zero	`Vector<int> v(n);`	`vector<int> v(n);`
Create a vector with n copies of a value k	`Vector<int> v(n, k);`	`vector<int> v(n, k);`
Create a vector with initial elements 3,1,4	`Vector<int> v{3,1,4};`	`vector<int> v{3,1,4};`
Add k to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear vector	`v.clear();`	`v.clear();`
Get the element at index i (verify that i is in bounds)	`int k = v.get(i);` `int k = v[i];`	`int k = v.at(i);`
Check if the vector is empty	`if (v.isEmpty()) ...`	`if (v.empty()) ...`
Replace the element at index i (verify that i is in bounds)	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;`
Get the element at index i without bounds checking	`// Impossible!`	`int a = x[i];`
Change the element at index i without bounds checking	`// Impossible!`	`x[i] = v;`

`std::map<T>`

In progress

`std::set<T>`

In progress

`std::multimap<T>`

In progress

Iterators

In progress

Templates

Defining a Tempaltised Function

We can define a function with a single template type as:

```
template <typename T>
T foo(T param1, int param2) {
      // ... implementation
}
```

With multiple tempaltised types, this would look like:

```
template <typename KeyType, typename ValType>
void foo(KeyType param1, const std::map<KeyType, ValType> &m) {
      // ... implementation
}
```

In progress

Algorithms

In progress