CSCI-1200 Computer Science II — Fall 2006
Lecture 10 — String and Character Operations

Review from Lectures 8 & 9

- Maps are associations between keys and values.
- Maps store pairs; map iterators refer to these pairs.
- Maps have fast insert, access and remove operations: O(log n).
- The choice between maps, vectors and lists is based on naturalness, ease of programming, and efficiency of the resulting program.
- Classes can be used as map keys if a well-behaved `operator<` is available.
- Maps can store more complicated values, such as vectors or classes. The *syntax* of using maps in this way can become a bit complicated, you'll get more comfortable with practice.
- We saw how to use maps to solve the-word-to-line-number indexing problem. Using a vector or a list would be significantly more difficult and the solution is less natural.
- Maps should be used when there is an association — natural or easily created — between two classes. As a larger example we looked at the MP3 database.

Homework 4: Using maps to build an efficient library database

- What operations must be sublinear? With respect to which data size?

Today's Class — String and Character Operations

- Motivating problem: input text analysis
- String operations: input a line at a time; substring.
- Character operations: checking character types
- Solving the motivating problem

Koenig & Moo: Sections 5.6-5.9

10.1 Motivation

- Problem: analyzing an input text file to find
  - Number of lines
  - Number of words
  - Number of letters
  - Number of occurrences of letters and words
- Challenges:
  - Distinguishing lines
  - Ignoring whitespace characters
  - Avoiding punctuation
  - Mixture of upper and lower case letters
- Assumptions:
  - A word is a sequence of uninterrupted letters.
  - Whitespace should not be included in the character count, but punctuation should.

10.2 String and Character Manipulation

- Until now, we've been reading strings from the input separated by whitespace. Some of you may have experimented with other operations for homework, but they weren't necessary to solve the problems.
- We can also read a whole line of input (including whitespace) with the function `getline`. This function reads characters until a newline character (or the end-of-file) is encountered. Here's the prototype:

```
istream& getline(istream &, string &);
```

  Returning the `istream` reference may seem a bit strange, but it is common practice. It allows the state of the stream to be tested in a conditional. We've seen this already with loops to read integers and strings, for example:

```
std::string name;
while (std::cin >> name) {
  ...
}
```

- The `string` class has a `substr` member function that extracts a substring starting at a given location. For example:

```
std::string s = "My name is Sally Jones";
std::string t = s.substr(11,5);  // Starting at location 11, extract the next 5 chars.
std::cout << t << std::endl;     // Outputs:  Sally
```

- The header file `<cctype>` provides prototypes for character functions from the C library (hence the 'c' in front of 'ctype'). Here are some examples:
  - `isspace(c)`
  - `isalpha(c)`
  - `isdigit(c)`
  - `ispunct(c)`
  - `isupper(c)`
  - `tolower(c)`

  Each of these functions takes a character and returns true or false.
- The type `char` is a special case of the type `integer`. As such, we can do simple math with values of type character. When we do this, the compiler automatically converts the `char` value to be of `integer` type. We can *cast* the value back to type `char` as illustrated below:

```
'c' - 'a' == 2                            // this is true
char('B' + 4) == 'F'                      // this is true
std::cout << 'a' + 10 << std::endl;       // outputs the integer 107
std::cout << char('a' + 10) << std::endl; // outputs the letter k
```

10.3 Exercise

- For the *last expression* in the fragment of code below, give the *type* and the *value*.

```
char c = 'P' + 2;
tolower(c);
c
```

10.4 Example: Writing a Program Find Palindromes

- A palindrome is a string that reads the same forward and backward.
- We want to write a program to read lines of input and determine if the alphabetic letters on a line form a palindrome.
- To do this, we'll use many of the new functions we learned above.

10.5 Exercise: Finish the Palindrome Code

- Write the details of the `is_palindrome` function. Use the comments in the code as a suggested guide.

```
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>

using namespace std;

bool is_palindrome(const string& line);

int main() {
  cout << "This program will read input, one line at a time, and\n"
       << "determine which input lines are palindromes.  It will also\n"
       << "output a count of palindromes\n";

  unsigned int count=0;
  string line;
  while (getline(cin, line)) {
    if (is_palindrome(line)) {
      count++ ;
      cout << line << endl;
    }
  }
  cout << "There were " << count << " lines containing palindromes.\n";
}


bool is_palindrome(const string& line) {
  string temp;
  string::const_iterator i;

  // Pull out letters and place them in the temp string.  Try to implement
  // this using string iterators and using the += operator  on strings.









  // Determine if the letters are the same in the first half and the
  // second half.  Return false as soon as a difference is found.









  return true;
}
```

10.6 Problem Solving Approach

Now let's address the text analysis posed at the beginning of the lecture. Here's an outline of how you might approach solving problems like this, which do not involve the design of classes:

1. Outline the flow and the major steps of the program.
2. Make note of the information that must be kept by the `main` function.
   This will dictate (most of) the variables.
3. Make a list of the functions that the `main` function needs.
4. Write these functions (and test them).
   If necessary, repeat the above process for these functions.
5. Write the `main` program and test it.

10.7 Returning to the Text Analysis Problem

We want to analyzing an input text file to find:

- Number of lines
- Number of words
- Number of letters
- Number of occurrences of letters and words

10.8 Text Analysis Algorithm Outline

Here's one outline (others are certainly possible). Each of the (*) corresponds to a helper function.

- Main function:
  1. For each line,
     (a) Increment line counter
     (b) Count characters (*) and add to character count
     (c) Add to letter counters (*)
     (d) Break up into words of small letters only (*)
     (e) Save all words
  2. Sort words (including repetitions) and count occurrences (*)
- Variables:
  - Counter: lines, words, letters
  - Vector of 26 individual letter counts
  - Vector of strings to represent words

10.9 Exercise: Write the helper functions!

```
unsigned int count_characters(const string& a_line) {




}
```

```
void add_to_letter_counts(const string & a_line, vector<int>& letter_counters) {




}
```

```
vector<string> break_up_line(const string& a_line) {









}
```

```
void count_word_occurrences(vector<string>& words) {









}
```

10.10 Putting it All Together

```
int main(int argc, char* argv[]) {
  if (argc != 2) {
    cerr << "Usage: " << argv[0] << " text-file";
    return 1;
  }
  ifstream in_str(argv[1]);
  if (!in_str) {
    cerr << "Couldn't open " << argv[1] << " to read.\n";
    return 1;
  }

  unsigned int character_count = 0;
  unsigned int line_count = 0;
  vector<int> letter_counters(26, 0);  // counts for the individual letters
  vector<string> all_words;

  // Handle one line at a time...
  string a_line;
  while (getline(in_str, a_line)) {
    line_count++;
    character_count += count_characters(a_line);
    add_to_letter_counts(a_line, letter_counters);
    vector<string> words_in_line = break_up_line(a_line);
    // Add all words to the back of the all_words vector.
    vector<string>::iterator p;
    for (p = words_in_line.begin(); p != words_in_line.end(); ++ p)
      all_words.push_back(*p);
  }

  // Output char, word and line counters
  cout << "\nHere are the statistics on the input text file:\n"
       << "  char count = " << character_count << "\n"
       << "  word count = " << all_words.size() << "\n"
       << "  line count = " << line_count << "\n";
  // Output the letter counts
  cout << "\nHere are the letter counts:\n";
  for (unsigned int i = 0; i < 26; ++ i) {
    cout << "  " << char('a' + i) << ":  " << letter_counters[ i ] << "\n";
  }
  // Output the word occurences
  count_word_occurrences(all_words);
}
```

10.11 Summary

- When the basic STL string input function is not sufficient we can write our own character manipulation code to implement more complex parsing:
  - Reading one line at a time
  - Substrings
  - Character Operations
- A Strategy for Problem Solving