Eric Roberts
CS 106B

Handout #8
January 11, 2013

Strings and Streams

Strings and Streams

Eric Roberts
CS 106B
January 11, 2013

Administrative Reminders

- All handouts and course information are on the web site:

  http://www.stanford.edu/class/cs106b/

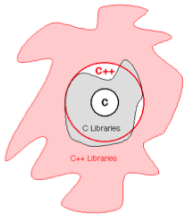
- Extra handouts are placed in the "Handout Hangout" in Gates.
- All CS 106B students must sign up for a section by Sunday at 5:00P.M. The signup form will appear on the web tomorrow:

  http://cs198.stanford.edu/section/

- All undergraduates must take CS 106B for **5 units**. The Axess default is 3 units, and we end up signing lots of petitions each year to correct this error.
- You need to install the appropriate C++ software for the type of computer you intend to use. The instructions for doing so are described in the appropriate version of Handout #7.

Relationship Between C and C++

- Part of the reason behind the success of C++ is that it includes all of C as a subset. This design strategy makes it possible to convert applications from C to C++ incrementally.



- The downside of this strategy is that that the design of C++ is less consistent and integrated than it might otherwise be.

Characters

- Both C and C++ use ASCII (*American Standard Code for Information Interchange*) as their encoding for character representation. The data type `char` therefore fits in a single eight-bit byte.
- With only 256 possible characters, ASCII is inadequate to represent the many alphabets in use throughout the world. In most modern language, ASCII has been superseded by Unicode, which permits a much larger number of characters.
- Even though the weaknesses in the ASCII encoding were clear at the time C++ was designed, changing the definition of `char` was impossible given the decision to keep C as a subset.
- The C++ libraries define the type `wchar_t` to represent "wide characters" that allow for a larger range. Because using this type introduces significant detail complexity, CS 106B will stick with the traditional `char` type.

Functions in the <cctype> Interface

`bool isdigit(char ch)` Determines if the specified character is a digit.
`bool isalpha(char ch)` Determines if the specified character is a letter.
`bool isalnum(char ch)` Determines if the specified character is a letter or a digit.
`bool islower(char ch)` Determines if the specified character is a lowercase letter.
`bool isupper(char ch)` Determines if the specified character is an uppercase letter.
`bool isspace(char ch)` Determines if the specified character is ***whitespace*** (spaces and tabs).
`char tolower(char ch)` Converts `ch` to its lowercase equivalent, if any. If not, `ch` is returned unchanged.
`char toupper(char ch)` Converts `ch` to its uppercase equivalent, if any. If not, `ch` is returned unchanged.

C Strings

- Almost any program that you write in any modern language is likely to use string data at some point, even if it is only to display instructions to the user or to label results.
- Conceptually, a string is just an array of characters, which is precisely how strings are implemented in the C subset of C++. If you put double quotation marks around a sequence of characters, you get what is called a ***C string***, in which the characters are stored in an array of bytes, terminated by a ***null byte*** whose ASCII value is 0. For example, the characters in the C string `"hello, world"` are arranged like this:

h	e	l	l	o	,		w	o	r	l	d	\0
0	1	2	3	4	5	6	7	8	9	10	11	12

- As in Java, character positions in a string are identified by an ***index*** that begins at 0 and extends up to one less than the length of the string.

Strings as an Abstract Data Type

- Because C++ includes everything in its predecessor language, C strings are a part of C++, and you will occasionally have to recognize that this style of string exists.
- For almost every program you write, it will be far easier to use the C++'s **`string`** class, which implements strings as an ***abstract data type***, which is defined by its behavior rather than its representation. All programs that use the string class must include the **`<string>`** library interface.
- The methods C++ provides for working with strings are often subtly different from those in Java's **`String`** class. Most of these differences fall into the *accidental* category. The only *essential* difference in thes models is that C++ allows clients to change the individual characters contained in a string. By contrast, Java strings are ***immutable***, which means that they never change once they are allocated.

Calling String Methods

- Because **`string`** is a class, it is best to think of its methods in terms of sending a message to a particular object. The object to which a message is sent is called the ***receiver***, and the general syntax for sending a message looks like this:

  receiver`.`*name*`(`*arguments*`);`

- For example, if you want to determine the length of a string **`s`**, you might use the assignment statement

  ```
  int len = s.length();
  ```

  just as you would in Java.
- Unlike Java, C++ allows classes to redefine the meanings of the standard operators. As a result, several string operations, such as **`+`** for concatenation, are implemented as operators.

The Concatenation Operator

- As many of you already know from Java, the **`+`** operator is a wonderfully convenient shorthand for ***concatenation***, which consists of combining two strings end to end with no intervening characters. In Java, this extension to **`+`** is part of the language. In C++, it is an extension to the **`string`** class.
- In Java, the **`+`** operator is often used to combine items as part of a **`println`** call, as in

  ```
  println("The total is " + total + ".");
  ```

  In C++, you achieve the same result using the **`<<`** operator:

  ```
  cout << "The total is " << total << "." << endl;
  ```

- Although you might imagine otherwise, you can't use the **`+`** operator in this statement, because the quoted strings are C strings and not **`string`** objects.

Operators on the `string` Class

`str[i]` Returns the i^{th} character of **`str`**. Assigning to **`str[i]`** changes that character.
`s1 + s2` Returns a new string consisting of **`s1`** concatenated with **`s2`**.
`s1 = s2;` Copies the character string **`s2`** into **`s1`**.
`s1 += s2;` Appends **`s2`** to the end of **`s1`**.
`s1 == s2` (and similarly for **`<`**, **`<=`**, **`>`**, **`>=`**, and **`!=`**) Compares to strings lexicographically.

Common Methods in the `string` Class

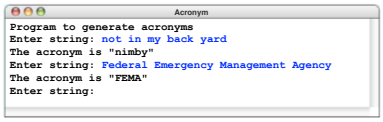
`str.length()` Returns the number of characters in the string **`str`**.
`str.at(index)` Returns the character at position **`index`**; most clients use **`str[index]`** instead.
`str.substr(pos, len)` Returns the substring of **`str`** starting at **`pos`** and continuing for **`len`** characters.
`str.find(ch, pos)` Returns the first index ≥ **`pos`** containing **`ch`**, or **`string::npos`** if not found.
`str.find(text, pos)` Similar to the previous method, but with a string instead of a character.
`str.insert(pos, text)` ←*Destructively changes* **`str`** Inserts the characters from **`text`** before index position **`pos`**.
`str.replace(pos, count, text)` ←*Destructively changes* **`str`** Replaces **`count`** characters from **`text`** starting at position **`pos`**.

Exercise: String Processing

- An ***acronym*** is a word formed by taking the first letter of each word in a sequence, as in

  `"self-contained underwater breathing apparatus"` → **`"scuba"`**

- Write a C++ program that generates acronyms, as illustrated by the following sample run:



Download: Acronym.cpp

Using Data Files

- A ***file*** is the generic name for any named collection of data maintained on the various types of permanent storage media attached to a computer. In most cases, a file is stored on a hard disk, but it can also be stored on removable medium, such as a CD or flash memory drive.
- Files can contain information of many different types. When you compile a C++ program, for example, the compiler stores its output in an ***object file*** containing the binary representation of the program. The most common type of file, however, is a ***text file***, which contains character data of the sort you find in a string.

Text Files *vs.* Strings

Although text files and strings both contain character data, it is important to keep in mind the following important differences between text files and strings:

1. *The information stored in a file is permanent.* The value of a string variable persists only as long as the variable does. Local variables disappear when the method returns, and instance variables disappear when the object goes away, which typically does not occur until the program exits. Information stored in a file exists until the file is deleted.
2. *Files are usually read sequentially.* When you read data from a file, you usually start at the beginning and read the characters in order, either individually or in groups that are most commonly individual lines. Once you have read one set of characters, you then move on to the next set of characters until you reach the end of the file.

Using Text Files

- When you want to read data from a text file as part of a C++ program, you need to take the following steps:
  1. Construct a new **`ifstream`** object that is tied to the data in the file by calling the **`open`** method for the stream. This phase of the process is called ***opening the file***. Note that the argument to **`open`** is a C string rather than a C++ string.
  2. Call the methods provided by the **`ifstream`** class to read data from the file in sequential order. The text describes several strategies for doing so; for today, we'll concentrate on reading a file line by line using the **`getline`** function.
  3. Break the association between the reader and the file by calling the stream's **`close`** method, which is called ***closing the file***.

Opening an Input File

```
/*
 * File: filelib.h
 * ---------------
 * This file exports a standardized set of tools for working with
 * files . . .
 */

#ifndef _filelib_h
#define _filelib_h

/*
 * Function: promptUserForFile
 * Usage: string filename = promptUserForFile(stream, prompt);
 * ------------------------------------------------------------
 * Asks the user for the name of a file.  The file is opened
 * using the reference parameter stream, and the function
 * returns the name of the file.  If the requested file cannot
 * be opened, the user is given additional chances to enter a
 * valid file.  The optional prompt argument provides an input
 * prompt for the user.
 */
```

Opening an Input File

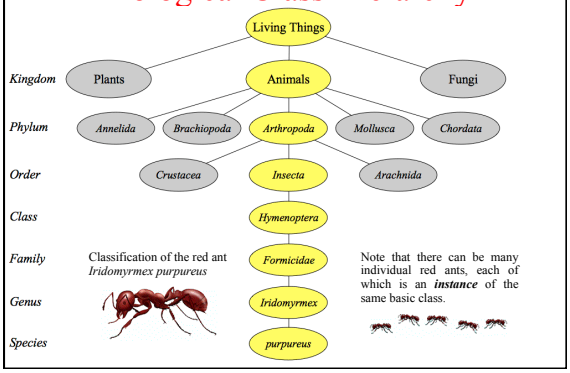
```
string promptUserForFile(ifstream & stream, string prompt) {
   while (true) {
      cout << prompt;
      string filename;
      getline(cin, filename);
      openFile(stream, filename);
      if (!stream.fail()) return filename;
      stream.clear();
      cout << "Unable to open that file.  Try again." << endl;
      if (prompt == "") prompt = "Input file: ";
   }
}
```

Reading Lines from a File

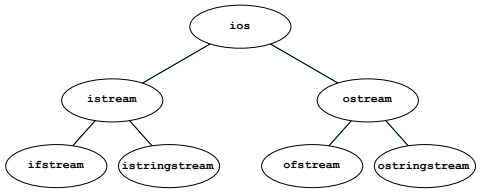
- You can read lines from a text file by calling the free function **`getline`**, which takes an **`ifstream`** and a **`string`**, both as reference parameters. The effect of **`getline`** is to store the next line of data from the file into the string variable after discarding the end-of-line character.
- If you try to read past the end of the data, **`getline`** sets the "fail" indicator for the stream.
- The following code fragment uses the **`getline`** method to determine the length of the longest line in the stream **`infile`**:

```
int max = 0;
while (true) {
   string line;
   getline(infile, line);
   if (infile.fail()) break;
   if (line.length() > max) max = line.length();
}
```

Biological Class Hierarchy



The Stream Hierarchy



UML Diagram for the Stream Hierarchy