C++ Lab 06 - Serialization and Deserialization of C++ Classes

2.680 Unmanned Marine Vehicle Autonomy, Sensing and Communications

Ten Short CPP Labs

IAP 2021

Michael Benjamin, mikerb@mit.edu
Department of Mechanical Engineering
Computer Science and Artificial Intelligence Laboratory (CSAIL)
MIT, Cambridge MA 02139

1 Lab Six Overview and Objectives

This lab builds on the previous lab where classes were introduced and random instances were created and printed to the command line by *serializing* the instance into a short string. In this lab the opposite direction is considered - *deserializing* from a string representation to create an instance of a class. The tools developed in this lab will be very simple and powerful and central to everything we do in MOOS - passing serialized messages.

In this lab the following topics are covered.

- String Parsing I - Finding a Character Within a String and Generating a Substring from Another String
- String Parsing II - Biting a String from the Front of Another String
- String Parsing III - Parsing a String into a Vector of Strings
- A General Deserializing Algorithm
- Re-constructing Class Instances from Deserialized Strings from a File

2 Serialization and Deserialization

In the previous lab we created the `rand_seglist` program that generated an instance of our `SegList` class and wrote a set of vertices in the x-y plane to a file with a line such as:

```
x=-93,y=-51,x=-27,y=-42,x=30,y=-28
```

This is a case of *serializing* an object of a particular class - the creation of a string of characters that uniquely describe the object instance. Presumably the object could be recreated given the information in the string. This step is called *deserialization* with the idea conveyed in the below figure.

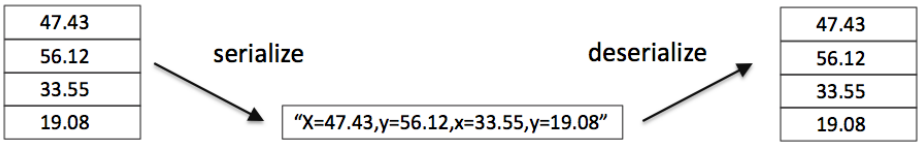

Figure 1: An object serialized into a string and then deserialized back into an object.

We suggest you skim the below Wikipedia discussion about serialization:

- http://en.wikipedia.org/wiki/Serialization
- http://www.parashift.com/c++-faq/serialize-overview.html

There are many ways to handle serialization/deserialization. Some methods use a "human readable" format like the one shown in the figure above, while some use a "binary" or other more compact

format emphasizing the need to have a shorter serialized message over one that is human readable. In certain cases, like underwater low-bandwidth comms, the emphasis is rightly on having shorter cryptic messages. In most other cases, the human-readable string is preferred since it has its advantages in simplicity and debugging while the bandwidth is a non-issue in most applications on modern computers.

The focus of this lab is *deserialization* and the almost synonymous topic of *string parsing*. A strong case could be made that serialization is the easy part, and that deserialization is a bit messier due to the string parsing. That may be true, but we develop here a few simple tools for string parsing that handle a wide variety of cases. They also do not require any external code (like the boost library). Our deserialization tools are comprised of standard C++ tools and a few small functions we build in exercises here.

3 String Parsing I - Finding a Character and Constructing a Sub-String

We start by re-visiting two key C++ utilities discussed in a previous lab, the `find()` and `substr()` functions. Comprehensive descriptions can be found at the two links below. Both functions are powerful and there is a lot of options and info for both functions. We suggest taking a quick skim now, and we will address a couple features relevant to us in the discussion below.

- http://www.cplusplus.com/reference/string/string/find/
- http://www.cplusplus.com/reference/string/string/substr/

In this lab we will build two key utility functions used in deserializing messages. These utility functions will be used frequently in this and future labs (and in general in MOOS-IvP programming). They can be built with the `find()` and `substr()` functions. So we first focus on a couple of the key capabilities of these functions.

3.1 Revisiting the STL String find() Function

Recall the basic usage of the `find()` function can be pretty simple: it just returns the index of first instance of a character in string. For example the below snippet would return: `"index=7"`.

```
string str = "apple_pie";
cout << "index=" << str.find('i') << endl;
```

On the above website, there are four "signatures" to the `find()` function listed. The one signature we are interested in is:

```
size_t find (char c, size_t pos = 0) const;
```

For a function so simple, there are a few things that may look a bit foreign to new C++ users. The first thing that may throw you is that the function actually takes *two* arguments. The second one is optional so that's why we can get away with `find('h')`. If the second argument is provided, it tells

`find()` to start searching at an index other than the default of 0. The other thing that may look new is the `const` at the end of the signature. This simply indicates that the function guarantees that the value of the string will not be changed as a result of invoking this function. Feel free to read up more on const functions in C++, but you can safely leave it at that for now.

The part that may really throw you is the use of the `size_t` as the return type. To understand this, it's worth discussing how this function handles the case when the given character is not found in the string.

When the string *does* contain the search character, the answer will be in the range of [0, N-1] for a string of length N. What should the answer be when the character is *not* found? There are only so many choices in handling this, for example, returning -1 or returning N. The function is implemented instead to return "the largest possible number" to indicate that the character is not found. What is the largest possible number? Well, that depends on what data type you're talking about. Here are some options:

- `unsigned int`: max size 65535
- `unsigned long int`: max size 4294967295
- `unsigned long long int`: max size 18446744073709551615

So one correct but not-so-great way of using this function would be as follows:

```
string str = "apple_pie";

unsigned long long int pos = str.find('i');
if(pos == 18446744073709551615)
  cout << "character NOT found." << endl;
```

This is awkward for a couple reasons. First, anytime a constant appears in source code, this is a red flag, especially with a really ugly and easy-to-mistype number like 18446744073709551615. To ease things a bit, C++ provides an alias for this number: `string::npos`. The second awkward aspect of the above code is the use of the type `"unsigned long long int"`. On most machines this is a 64 bit number with the largest number indeed being 18446744073709551615. It turns out that mapping from a given variable type to a specific bit size is not universal, historically. For this reason the keyword `size_t` is preferred. This tells the compiler to figure out the right version of the `unsigned int` type.

So the above code is better written as:

```
string str = "apple_pie";

size_t pos = str.find('i');
if(pos == string::npos)
  cout << "character NOT found." << endl;
```

You're now in a good position to use this function for the exercises in this lab. The next topic: the `substr()` function.

3.2 Revisiting the STL String substr() Function

We have used the `substr()` function in a previous lab to grab a substring from a command line argument. For example the below snippet would return `"file.dat"`.

```
string argument = "--filename=file.dat";
string filename = argument.substr(10);

// The variable filename would now be equal to "file.dat"
// The variable argument would remain unchanged, equal to "--filename=file.dat"
```

On the cplusplus.com website there is further valuable information, starting with the official signature for this function:

```
string substr (size_t pos = 0, size_t len = npos) const;
```

Like the `find()` function, the `substr()` function is a `const` function, meaning it will not alter the string upon which it is called. The `substr()` function also contains an optional second argument indicating the length of the desired substring. By default, if no second argument is given, the substring runs until the end of the string. Again the keyword `string::npos` is an alias for the position at the end of the original string. The `size_t` type is again shorthand for what is likely the type `unsigned long int`.

Important Note: One handy feature of this function is that it conveniently returns an empty string when the first argument (`pos`, the starting index) is N where N is the length of the string. One may expect that the allowable values for `pos` would be restricted to [0, N-1]. In fact a run-time "out-of-bounds" error would be thrown for `pos` greater than N, but N is indeed allowed and results in a empty substring being returned. This is especially handy in our exercises as we will see.

3.3 Exercise 1: Splitting a String Around a Character

Write a program that uses the `find()` and `substr()` functions to accept a string on the command line and split the string into two new strings. The first string should be everything up to the first `'='` character, and the second string should be everything after the `'='` character. If the given string does not contain the `'='` character the first string should just be the original given string, and the second string should be empty.

Call your file `string_split.cpp` and build it to the executable `string_split`. When your program runs, it should be invocable from the command line and output the answer in the below format. We put brackets around the output to be clearer about when white space is present, as in the final example with the argument `" "`.

```
$ ./string_split one=two
front: [one]
back:  [two]

$ ./string_split something_special
front: [something_special]
back:  []

$ ./string_split special_case=
front: [special_case]
back:  []

$ ./string_split =special_case
front: []
back:  [special_case]

$ ./string_split " "
front: [ ]
back:  []
```

The solution to this exercise is in Section 7.1.

4 String Parsing II - Biting a String from the Front of Another String

In this section the goal is to build a utility function, `biteString()` which be an important piece of our general approach to string parsing and deserialization. It's signature is:

```
string biteString(string& string, char);
```

Note that it is a function *not* defined as a native method on the `string` class, but rather it takes a string as an argument passed by reference. The argument is typically modified by the time the function is returned. The operation is very similar to the operation in the first exercise. Below are some examples:

```
string orig = "apples, pears, bananas";
string one = biteString(orig, ',');
string two = biteString(orig, ',');

cout << "orig: [" << orig << "]" << endl;
cout << "one:  [" << one << "]" << endl;
cout << "two:  [" << two << "]" << endl;
```

The above would produce:

```
orig: [ bananas]
one: [apples]
two: [ pears]
```

4.1 Exercise 2: Write the biteString() Utility Function

Write a program that includes an implementation of the `biteString()` utility function. Use the utilty function to essentially replicate the functionality of the program in Exercise 1. Put your utility function in a file named `BiteString.cpp` and `BiteString.h` and build it to the executable `string_bite`. When your program runs, it should be invocable from the command line with:

```
$ ./string_bite one=two
front: [one]
back:  [two]

$ ./string_bite somthing_special
front: [somthing_special]
back:  []

$ ./string_bite special_case=
front: [special_case]
back:  []

$ ./string_bite =special_case
front: []
back:  [special_case]

$ ./string_bite " "
front: [ ]
back:  []
```

The solution to this exercise is in Section 7.3.

5 String Parsing III - Parsing a String into a Vector of Strings

In this section the goal is to build a utility function, `parseString()` which will be the second important piece of our general approach to string parsing and deserialization. It's signature is:

```
vector<string> parseString(string, char);
```

Note that it is also a function *not* defined as a native method on the `string` class (unlike `find()` and `substr()`). It takes a string as an argument passed by value. The string argument is *not* modified by the function since only a copy of the string is passed to the function (pass by value). The second argument is the "separation character" from which the string is split into substrings. Below is an example:

```
string orig = "x=100,y-23,label=alpha";

vector<string> my_vector = parseString(orig, ',');
for(unsigned int i=0; i<my_vector.size(); i++)
  cout << "[" << i << "]: [" << my_vector[i] << "]" << endl;
```

The above would produce:

```
[0]: [x=100]
[1]: [y=-23]
[2]: [label=alpha]
```

5.1 Exercise 3: Write the parseString() Utility Function

Write a program that includes an implementation of the `parseString()` utility function. Your program will accept a single string argument from the command line and parse the string into a vector of strings, splitting the string based on each comma encountered in the string. *Hint: Use your biteString() utility to greatly simplify the implementation of this function.* Put your utility function in a file named `ParseString.cpp` and `ParseString.h`. Put your `main()` function in a file called `string_parse.cpp` and build it to the executable `string_parse`. Don't forget to include all source files in your build. For example:

```
$ g++ -o string_parse BiteString.cpp ParseString.cpp string_parse.cpp
```

When your program runs, it should be invocable from the command line with:

```
$ ./parse_string x=100,y=-23,label=alpha
[0]: [x=100]
[1]: [y=-23]
[2]: [label=alpha]
```

The solution to this exercise is in Section 7.4.

6 A Simple and General Algorithm for Serialization and Deserialization

You now have built all the tools you need for a handy and simple algorithm for serializing and deserializing objects to and from strings. Let's review the relevant pieces:

- The `biteString()` utility function (from this lab)
- The `parseString()` utility function (from this lab)
- The `fileBuffer()` utility function (from the 4th lab, the File I/O lab)

We will use the above utilities to deserialize our strings back into objects.

6.1 Tips for Choosing a Serialization Format

First let's revisit the serialization step. Remember that the deserialization step depends on some knowledge of how the object was initially serialized. There are options for choosing the serialization format and some may be better than others depending on the situation.

Although generally shorter strings are better than longer, in most situations the length of the string does not really matter unless that string is a message that is being sent over a low-bandwidth

communications channel, or is being stored on media with precious little space. Typically the more important feature is the *robustness* of the format, where robustness is measured in the ease of accommodating changes to the format at later dates without breaking code that was written at an earlier date.

As an example, consider a class with three member variables and three methods for setting the member variables:

```
class MyObject {
  void setLabel(string s) {label=s;};
  void setTotal(string s) {total=s;};
  void setPrice(string s) {price=s;};
 protected:
  string label;
  string total;
  string price;
};
```

Suppose an instance of this class had values `"day_22"`, `"117"`, and `"8.50"`. Two ways of serializing this are:

- "day_22,117,8.50"
- "label=day_22,total=117,price=8.50"

The first one has the upside of being shorter. The second one has the upside of being more human-readable. In the first message, there is no way of knowing for example that the first string represents the label, other than by declaring a convention that this is how it will be interpreted. The assumption of an ordering is prone to problems later on if the convention changes. For example consider what happens if the object later has a fourth field "owner". This can be accommodated with a four-element string, e.g., `"day_22,117,8.580,dave"`. On the deserializing side we can just check for the number of fields (3 for old style, 4 for new style) and handle accordingly. But what happens if, at an even later date, the "label" field is dropped, and all newer style strings now again have three fields. Now this would break any deserializing that depended on out-dated conventions.

This is not as big a problem when we control the serialization and deserialization. By "control", we mean that we are sure that the only code implementing serialization and deserialization is code that we have written. In this case, if the convention changes, we can just make sure the convention is reflected on both sides. There are a couple cases where "control" is not a good assumption. First, in the case of MOOS, the strings are being passed around as messages in a publish-subscribe manner. This means that there may be multiple apps subscribing for and deserializing the same message. Second, the strings could be lines in a data file, one line representing each object. In this case a change in the serialization due to a new convention won't be reflected in a data file created long ago. But suddenly the older data file becomes unreadable because it was written in an older convention.

For these reasons, it is generally much better to serialize in a manner along the lines of the second example above, repeated here:

- "label=day_22,total=117,price=8.50"

In this case, the unwritten convention between the serializer and deserializer has three aspects: (a) all fields are labeled and consist of parameter=value pairs, (b) the order is not guaranteed, and (c) a message in the future may have new types of pairs.

6.2 A General Deserialization Algorithm

A general deserialization algorithm is given here that (a) uses the simple utility functions developed so far in these labs, and (b) assumes a serialized string in the general "comma-separated parameter=value" format such as:

- "label=day_22,total=117,price=8.50"

The algorithm can be treated as boiler-plate and is given by way of example below.

```
string message = "label=day_22,total=117,accuracy=0.18"

vector<string> svector = parseString(message, ',');
for(unsigned int i=0; i<svector.size(); i++) {
   string pair = svector[i];
   string param = biteString(pair, '=');
   string value = pair;
   if(param == "label")
     my_object.setLabel(value);
   else if(param == "total")
     my_object.setTotal(value);
   else if(param == "accuracy")
     my_object.setAccuracy(value);
}
```

Note that there is no assumption of the order of fields, e.g., if `"price=8.50"` came first in the string, this would not be a problem. Also note that unexpected fields would simply be ignored. Of course if the programmer wants to write extra code to raise a warning if an unexpected field were encountered, this is possible and at the discretion of the programmer. Overall though, the above approach is pretty simple and robust.

6.3 Exercise 4: Deserialize a Data File of Serialized Objects

Write a program that reads in the contents of a file and interprets each line as a serialized instance of the `Vertex` class from the previous lab. Your program should deserialize each line back into a valid object. Each line in the file may look like:

```
x=-93,y=-51
x=-60,y=65,
x=3,y=69
...
x=-3,y=-74
```

A test file for this exercise can be found at the below link:

- http://oceanai.mit.edu/cpplabs/vertices.txt

You can also generate a test file yourself from the program you built in the previous lab:

```
$ ./rand_vertices_file --filename=vertices.txt --amt=10
```

Put your `main()` function in a file called `string_deserial.cpp` and build it to the executable `string_deserial`. Don't forget to include all source files in your build. For example:

```
$ g++ -o string_deserial Vertex.cpp FileBuffer.cpp BiteString.cpp ParseString.cpp string_deserial.cpp
```

Your program will simply read in the vertices, one per line, and deserialize each string into a `Vertex` instance, collecting all vertices in to a vector of vertices. Finally after the file is read, all vertices are just printed to the terminal using the `getSpec()` function defined on the `Vertex` class. This function just returns a string (serialized) version of the vertex instance and was part of the prior lab. Your program should handle the cases where the filename was not provided, or the file was not found. When your program runs, it should be invocable from the command line with:

```
$ ./string_deserial
Usage: string_deserial --filename=test.txt

$ ./string_deserial --filename=garbage.txt
Unable to open or empty file: garbage.txt

$ ./string_deserial --filename=vertices.txt
Total vertices found: 10
  Vertex 0: x=5,y=22
  Vertex 1: x=88,y=-59
  Vertex 2: x=24,y=10
  Vertex 3: x=-54,y=13
  Vertex 4: x=20,y=-11
  Vertex 5: x=22,y=-48
  Vertex 6: x=-40,y=-26
  Vertex 7: x=-97,y=88
  Vertex 8: x=69,y=-18
  Vertex 9: x=-72,y=-30
```

The solution to this exercise is in Section 7.5.

7 Solutions to Exercises

7.1 Solution to Exercise 1

```
/*-------------------------------------------------------------------------*/
/*  FILE:  string_split.cpp (Sixth C++ Lab Exercise 1 Version 1)           */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/string_split.cpp            */
/*  BUILD: g++ -o string_split string_split.cpp                            */
/*  RUN:   string_split  one=two                                           */
/*-------------------------------------------------------------------------*/

#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
  // Check for proper usage and get the one string argument
  if(argc != 2) {
    cout << "Usage: string_split STRING" << endl;
    return(1);
  }
  string str = argv[1];

  // First check if the split-character is not at all in the string
  if(str.find('=') == string::npos) {
    cout << "front: ["  << str << "]" << endl;
    cout << "back:  []" << endl;
    return(0);
  }

  // Now that we know the string contains the split-character, split.
  unsigned long int pos = str.find('=');
  cout << "front: [" << str.substr(0, pos) << "]" << endl;
  cout << "back:  [" << str.substr(pos+1) << "]" << endl;

  return(0);
}
```

Comments on the Solution to Exercise 1:

There are many possible variations on the above solution. In this case there are three subtle improvements we would like to make. These fall under the category of "good programming practices" and is admittedly subjective. But nevertheless, we point out three things that could be improved and offer a second version below. Here are the issues:

- The `find()` function is invoked twice for any string that actually contains the split-character (the `'='` character). For many applications this may be a negligible inefficiency, but we can do better.
- There are two similar sets of lines handling the output of the results. This can be improved to just one block. In general this type of simplification helps improve the maintainability of code even though in our case it's pretty small.

- Generally the less return points in a function the better. In this solution there are two points for early return. We can do better.

Compare the above solution and the above three points of concern and compare to the second version of this solution below.

7.2 Solution to Exercise 1 (Version Two)

```
/*-------------------------------------------------------------------------*/
/*  FILE:  string_split_v2.cpp (Sixth C++ Lab Exercise 1 Version 2)     */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/string_split_v2.cpp      */
/*  BUILD: g++ -o string_split_v2 string_split_v2.cpp                   */
/*  RUN:   string_split_v2  one=two                                     */
/*-------------------------------------------------------------------------*/

#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
  if(argc != 2) {
    cout << "Usage: string_split STRING" << endl;
    return(1);
  }
  string str = argv[1];

  // Prepare the answer comprised of the variables "front" and "back".
  // Initialize the values it would have in the special case where
  // the split-character is not found in the string
  string front = str;
  string back  = "";

  // Look for the split-character
  unsigned long int pos = str.find('=');

  // If the split-character is found, modify the answer.
  // Otherwise nothing needs to be done to the answer.
  if(pos != string::npos) {
    front = str.substr(0, pos);
    back  = str.substr(pos+1);
  }

  // Handle the output of the answer.
  cout << "front: [" << front << "]" << endl;
  cout << "back:  [" << back  << "]" << endl;

  return(0);
}
```

7.3 Solution to Exercise 2

```
/*-------------------------------------------------------------------------*/
/*  FILE:  BiteString.h                                                    */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/lib_strings/BiteString.h */
/*-------------------------------------------------------------------------*/

#ifndef BITE_STRING
#define BITE_STRING

#include <string>

std::string biteString(std::string& str, char);

#endif
```

```
/*-------------------------------------------------------------------------*/
/*  FILE:  BiteString.cpp                                                  */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/lib_strings/BiteString.cpp */
/*-------------------------------------------------------------------------*/

#include "BiteString.h"

using namespace std;

//-------------------------------------------------------------------------
// Procedure: biteString()

string biteString(string& str, char c)
{
  size_t pos = str.find(c);

  string return_str;

  if(pos == string::npos) {
    return_str = str;
    str = "";
  }
  else {
    return_str = str.substr(0, pos);
    str = str.substr(pos+1);
  }

  return(return_str);
}
```

```
/*-------------------------------------------------------------------------*/
/*  FILE:  string_bite.cpp (Sixth C++ Lab Exercise 2)                      */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/string_bite.cpp             */
/*  BUILD: g++ -o string_bite BiteString.cpp string_bite.cpp               */
/*  RUN:   string_bite  one=two                                            */
/*-------------------------------------------------------------------------*/

#include <iostream>
#include "BiteString.h"

using namespace std;

int main(int argc, char **argv)
{
  // Check for proper usage and get the one string argument
  if(argc != 2) {
    cout << "Usage: string_bite STRING" << endl;
    return(1);
  }
  string str = argv[1];

  string front = biteString(str, '=');
  string back  = str;

  cout << "front: [" << front << "]" << endl;
  cout << "back:  [" << back  << "]" << endl;

  return(0);
}
```

7.4 Solution to Exercise 3

```
/*-------------------------------------------------------------------------*/
/*  FILE:  ParseString.h                                                   */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/lib_strings/ParseString.h */
/*-------------------------------------------------------------------------*/

#ifndef PARSE_STRING
#define PARSE_STRING

#include <string>
#include <vector>

std::vector<std::string> parseString(std::string str, char);

#endif
```

```
/*-------------------------------------------------------------------------*/
/*  FILE:  ParseString.cpp                                                 */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/lib_strings/ParseString.cpp */
/*-------------------------------------------------------------------------*/

#include "ParseString.h"
#include "BiteString.h"

using namespace std;

//-------------------------------------------------------------------------
// Procedure: parseString()

vector<string> parseString(string str, char c)
{
  vector<string> return_vector;
  while(str != "")
    return_vector.push_back(biteString(str, c));

  return(return_vector);
}
```

```
/*-------------------------------------------------------------------------*/
/*  FILE:  string_parse.cpp (Sixth C++ Lab Exercise 3)                     */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/string_parse.cpp            */
/*  BUILD: g++ -o string_parse BiteString.cpp ParseString.cpp \            */
/*                string_parse.cpp                                         */
/*  RUN:   string_parse  one=two                                           */
/*-------------------------------------------------------------------------*/

#include <iostream>
#include "ParseString.h"

using namespace std;

int main(int argc, char **argv)
{
  // Check for proper usage and get the one string argument
  if(argc != 2) {
    cout << "Usage: string_parse STRING" << endl;
    return(1);
  }
  string str = argv[1];

  vector<string> svector = parseString(str, ',');
  for(unsigned int i=0; i<svector.size(); i++)
    cout << "[" << i << "]: [" << svector[i] << "]" << endl;

  return(0);
}
```

7.5 Solution to Exercise 4

The solution for this exercise includes the previously developed source code in:

- FileBuffer.h/cpp
- BiteString.h/cpp
- ParseString.h/cpp
- Vertex.h/cpp

Plus the following program:

```
/*-------------------------------------------------------------------------*/
/*  FILE:  string_deserial.cpp (Sixth C++ Lab Exercise 4)                  */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/string_deserial.cpp         */
/*  WGET:  wget http://oceanai.mit.edu/cpplabs/vertices.txt                */
/*  BUILD: g++ -o string_deserial BiteString.cpp ParseString.cpp \         */
/*                FileBuffer.cpp Vertex.cpp string_deserial.cpp            */
/*  RUN:   string_deserial --filename=vertices.txt                         */
/*-------------------------------------------------------------------------*/

#include <iostream>
#include "SegList.h"
#include "ParseString.h"
#include "BiteString.h"
#include "FileBuffer.h"

using namespace std;

int main(int argc, char **argv)
{
  // Find the name of the file to be created
  string filename;
  for(int i=1; i<argc; i++) {
    string argi = argv[i];
    if(argi.find("--filename=") == 0)
      filename = argi.substr(11);
  }

  // If no file specified, produce usage information and exit now.
  if(filename == "") {
    cout << "Usage: string_deserial --filename=test.txt" << endl;
    return(1);
  }

  // Open the specified file and read in each line to the buffer
  vector<string> fvector = fileBuffer(filename);
  if(fvector.size() == 0) {
    cout << "Unable to open or empty file: " << filename << endl;
    return(1);
  }

  vector<Vertex> vertices;

  for(unsigned int i=0; i<fvector.size(); i++) {
    string raw_line = fvector[i];
    vector<string> svector = parseString(raw_line, ',');
    string xval, yval;

    for(unsigned int j=0; j<svector.size(); j++) {
      string pair = svector[j];
      string param = biteString(pair, '=');
      string value = pair;
      if(param == "x")
        xval = value;
      else if(param == "y")
        yval = value;
    }
    if((xval != "") && (yval != "")) {
      int int_xval = atoi(xval.c_str());
      int int_yval = atoi(yval.c_str());
      Vertex new_vertex(int_xval, int_yval);
      vertices.push_back(new_vertex);
    }
  }
  cout << endl;
  cout << "Total vertices found: " << vertices.size() << endl;
  for(unsigned int i=0; i<vertices.size(); i++)
    cout << "  Vertex " << i << ": " << vertices[i].getSpec() << endl;

  return(0);
}
```