EECS402 Lecture 10

Andrew M. Morgan

Savitch Ch 12.1-12.2
Streams
Stream States
Input/Output

Intro To Streams

- In C++ a *stream* is simply "a sequence of bytes"
- We use a sequence of bytes as input
- Output is also a sequence of bytes
- Therefore, there are two main types of streams
  - Input streams - streams that flow from a device in to memory
    - cin is an input stream - from the keyboard to memory
  - Output streams - streams that flow from memory out to a device
    - cout is an output stream - from memory to the screen
- All I/O is a sequence of bytes, even if multi-byte data types are being used
  - For example, the int value 4,751 is stored in 4 bytes
  - The data is read in byte-by-byte and can be formed into an int with the value 4,751 by C++ (when you use the >> operator, for example)

A Word About ">>"

- The ">>" operator (call the "extraction operator") is, by default, a whitespace-delimited operation
  - First, it skips any leading whitespace it finds
  - Then, it extracts out characters until:
    - 1) The NEXT character is a whitespace character
      - Or
    - 2) Something "bad" happens (more on that in a few slides)
- Example, say you have this code:

```
cin >> aVal >> bVal >> cVal;
```

- And the user types " 41 8 16"

" 41 8 16"

The ">> aVal" part will skip the whitespace before the 4, then extract the 4 and the 1 to get integer value 41, and will stop.

" 8 16"

The ">> bVal" part will pick up from there, skip the whitespace before the 8, then extract the 8 to get integer value 8, and will stop.

" 16"

The ">> cVal" part will pick up from there, skip the whitespace before the 1, then extract the 1 and the 6 to get integer value 16, and will stop.

Buffered I/O

- Since stream data is read/written byte-by-byte, the data is stored in a "buffer" until the language needs it
- Data that is input is stored in a buffer until a C++ statement requires information from the buffer to be read by the program

```
int main()
{
  int aVal;
  int bVal;

  cout << "Enter a: ";
  cin >> aVal;
  cout << "Enter b: ";
  cin >> bVal;

  cout << "A+B=" << aVal + bVal << endl;

  return 0;
}
```

Enter a: 5
Enter b: 4
A+B=9

Note: Enter was pressed after each int was input

Enter a: 5 4
Enter b: A+B=9

Note: Both integers were entered before enter was pressed. 5 was "consumed" for "a". 4 remained in input buffer and was immediately "consumed" for "b" later in the program.

Bad Input Data Type

- Consider the same program – what if user entered the character 'x' when prompted for "a"?
  - 'x' is not an int, which is what the language expected
  - Stream enters the "fail state" indicating something has gone wrong
  - Further input requests are ignored when a stream is in the "fail state"

```
int main()
{
  int aVal;
  int bVal;

  cout << "Enter a: ";
  cin >> aVal;
  cout << "Enter b: ";
  cin >> bVal;

  cout << "A+B=" << aVal + bVal << endl;

  return 0;
}
```

Enter a: x
Enter b: A+B=1627429282

Expected an int, received a char. cin stream goes into the "fail state"

Since cin stream is still in the "fail state", this input request (and *all future* input requests) are literally ignored and skipped

Therefore, no valid value for either a OR b was provided, and result is garbage.

Functionality Associated With cin

- Several functions are available to help you deal with bad input
  - cin.fail()
    - If this function returns true, then the stream is in the fail state, and the next request for input will be ignored
  - cin.eof()
    - A request for input was made after the end-of-file had been reached (indicating there is no more input)
  - cin.good()
    - If this function returns true, then the stream is in a good state, and the next request for input has the possibility of succeeding
  - cin.clear()
    - Returns the stream to the good state
  - cin.ignore(num, endChar)
    - This function "consumes" up to num characters from the stream, up to, and including, the character indicated by endChar (oftentimes the '\n' character)

(Bad) Example of Overcoming Fail State

```
int aVal;
bool validInputFound;

cout << "Enter a: ";

validInputFound = false;
while (!validInputFound)
{
  cin >> aVal;
  if (cin.fail())
  {
    cout << endl << "Fail state!" << endl;
    cout << "Try again - Enter a: ";
  }
  else
  {
    validInputFound = true;
  }
}
cout << "Entered: " << aVal << endl;
```

```
Enter a: x
Fail state!
Try again - Enter a:
Fail state!
Try again - Enter a:
Fail state!
Try again - Enter a:
Fail state!
Try again - Enter a:
Fail state!
Try again - Enter a:
Fail state!
… (infinite loop)
```

Note: While it was correctly determined that the input caused cin to go in the fail state, it remains in the fail state, and every input request is ignored, resulting in an infinite loop!

(Bad) Example of Overcoming Fail State #2

```
int aVal;
bool validInputFound;

cout << "Enter a: ";

validInputFound = false;
while (!validInputFound)
{
  cin >> aVal;
  if (cin.fail())
  {
    cin.clear();
    cout << endl << "Fail state!" << endl;
    cout << "Try again - Enter a: ";
  }
  else
  {
    validInputFound = true;
  }
}
cout << "Entered: " << aVal << endl;
```

```
Enter a: x
Fail state!
Try again - Enter a:
Fail state!
Try again - Enter a:
Fail state!
Try again - Enter a:
Fail state!
Try again - Enter a:
Fail state!
Try again - Enter a:
Fail state!
… (infinite loop)
```

Note: cin stream was cleared of the fail state, but the 'x' that was input remains in the buffer (it is not consumed by the failed input request). As loop iterates it continues trying to consume the 'x' for the value of a, which results in the fail state over and over.

(Good) Example of Overcoming Fail State

```
int aVal;
bool validInputFound;

cout << "Enter a: ";

validInputFound = false;
while (!validInputFound)
{
  cin >> aVal;
  if (cin.fail())
  {
    cin.clear();
    cin.ignore(200, '\n');
    cout << endl << "Fail state!" << endl;
    cout << "Try again - Enter a: ";
  }
  else
  {
    validInputFound = true;
  }
}
cout << "Entered: " << aVal << endl;
```

cin.clear() takes the stream out of the fail state to allow future requests of input from the stream

cin.ignore() consumes all input from the cin input stream, to allow new data to be consumed for future requests of input from the stream

```
Enter a: x

Fail state!
Try again - Enter a: a

Fail state!
Try again - Enter a: ?

Fail state!
Try again - Enter a: 4
Entered: 4
```


Extending Input To Text Files

- All input discussed has been from keyboard, using the cin stream
- This is not practical if the amount of data to be input is large
  - For example: What if you had to input an entire dictionary with the keyboard every time you wanted to spell-check a document?
- Often, program input will come from a file that is stored with the operating system's file system
  - Dictionary words could be stored in a file called "dictionary.txt" and a program could read the words from the file, rather than from the keyboard
- What is contained in a text file?
  - A sequence of bytes! In other words - a stream
- You already know how to input from the cin stream
  - Input from file streams work in much the same way

Files As Streams In C++

- There are 3 main types of file streams available when you #include <fstream>
  - ifstream: Input file stream (sequence of bytes used as input to program)
  - ofstream: Output file stream (sequence of bytes used as output from program)
  - fstream: Input *and* output file stream (sequence of bytes used as input or output)
- File streams must be associated with a specific file in the file system
  - This is accomplished by "opening a file" for a file stream
  - Any input/output operations performed on that file stream interface with the text file – *not the keyboard or screen*
  - A file stream that has been opened should then be "closed" when the stream will longer be used in the program
    - i.e. All necessary input has been used, or all data to be output has been output

Using An Input File Stream

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  int      aVal;
  int      bVal;
  ifstream inFile;

  inFile.open("inputVals.txt");

  inFile >> aVal; //Similar to "cin >> aVal;"
  cout << "Read a: " << aVal << endl;

  inFile >> bVal;
  cout << "Read b: " << bVal << endl;
  cout << "Sum: " << aVal + bVal << endl;

  inFile.close();

  return 0;
}
```

Required for using "ifstream" type

Indicates that input requests from the stream called "inFile" are to be taken from the sequence of bytes stored in the file named "inputVals.txt".

Inputs an integer from the text stored in the file. "File pointer" is updated to indicate where next input from stream will occur from.

Contents of inputVals.txt on file system

```
5
4
```

Output Of Program

```
Read a: 5
Read b: 4
Sum: 9
```

Problems With Input File Example

- What is missing from previous example?
  - Error checking!
  - Just like the cin stream, file streams may not be in the good state
  - These result in the file stream being in the fail state
    - The file you tried to open does not exist on the file system
    - Prior input from the stream failed due to an incorrect data type
  - A stream can also be in the end-of-file state
    - While attempting to read in a value, the stream encountered the end of the file
- The same functions described for cin (good, fail, eof, clear, ignore) are available for use on input file streams
  - Error checking should be done to ensure valid input is received!

Bad Data In Input File Streams

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  int      aVal;
  int      bVal;
  ifstream inFile;

  inFile.open("inputVals.txt");

  inFile >> aVal;
  cout << "Read a: " << aVal << endl;

  inFile >> bVal;
  cout << "Read b: " << bVal << endl;
  cout << "Sum: " << aVal + bVal << endl;

  inFile.close();

  return 0;
}
```

Contents of inputVals.txt on file system

x
4

Output Of Program

```
Read a: 1627430819
Read b: 2293488
Sum: 1629724307
```

Note: tried to read in 'x' for int variable a, which caused stream to go into the fail state. All future requests for input are ignored, leaving both a and b uninitialized

Handling Bad Data In Input File Streams, p.1

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main()
{
  int      aVal;
  int      bVal;
  ifstream inFile;
  bool     validInputFound;

  inFile.open("inputVals.txt");

  if (inFile.fail())
  {
    cout << "Unable to open input file!" << endl;
    exit(1);
  }

  //Continued on next slide
```

Always check to see if opening the file resulted in a file stream in the good state.

If not – the file didn't exist in the file system or the user opening the file did not have access to it.

Handling Bad Data In Input File Streams, p.2

```
//Continued from previous slide

validInputFound = false;
while (!validInputFound)
{
  inFile >> aVal;

  if (inFile.eof())
  {
    cout << "EOF before reading a" << endl;
    exit(2);
  }
  else if (inFile.fail())
  {
    inFile.clear();
    inFile.ignore(200, '\n');
  }
  else
  {
    validInputFound = true;
  }
}
cout << "Read a: " << aVal << endl;

//Continued on next slide
```

After input, check to see if file is in the end-of-file state.

If so, the input request tried to input data after the end-of-file indicator (which means that there are no more bytes to read in the stream)

If end-of-file has not been reached, the input request could have failed resulting in the fail state. Handled the same way as described for cin.

Handling Bad Data In Input File Streams, p.3

```
validInputFound = false;
while (!validInputFound)
{
  inFile >> bVal;

  if (inFile.eof())
  {
    cout << "EOF before reading b" << endl;
    exit(2);
  }
  else if (inFile.fail())
  {
    inFile.clear();
    inFile.ignore(200, '\n');
  }
  else
  {
    validInputFound = true;
  }
}
cout << "Read b: " << bVal << endl;

cout << "Sum: " << aVal + bVal << endl;
inFile.close();

return 0;
}
```

Reading in b is done the same way as a.

Outputs Of Example Program

- Following are outputs of the program, with several different values in input file

Contents of inputVals.txt

51
14

Output Of Program

```
Read a: 51
Read b: 14
Sum: 65
```

Contents of inputVals.txt

x
4

Output Of Program

```
Read a: 4
EOF reached before reading b
```

Contents of inputVals.txt

This is for a
51
and now
for b
14

Output Of Program

```
Read a: 51
Read b: 14
Sum: 65
```

Understanding the Stream eof Method

- Remember, the extraction operator (>>) will stop parsing things as soon as it sees the next character is a whitespace character
  - Space, Tab, or Newline

fileWithNewlineAtEnd.txt
```
hello there<newline>
eecs402 class<newline>
<eof>
```

fileWithNoNewlineAtEnd.txt
```
hello there<newline>
eecs402 class<eof>
```

```
READING FILE WITH NEWLINE AT END
After word: hello EOF: 0 FAIL: 0
After word: there EOF: 0 FAIL: 0
After word: eecs402 EOF: 0 FAIL: 0
After word: class EOF: 0 FAIL: 0
After word: NOTHING_HERE EOF: 1 FAIL: 1

READING FILE WITH *NO* NEWLINE AT END
After word: hello EOF: 0 FAIL: 0
After word: there EOF: 0 FAIL: 0
After word: eecs402 EOF: 0 FAIL: 0
After word: class EOF: 1 FAIL: 0
```

```
ifstream inFile;
string aWord;

cout << "READING FILE WITH NEWLINE AT END" << endl;
inFile.open("fileWithNewlineAtEnd.txt");
while (!inFile.eof())
{
  aWord = "NOTHING_HERE";
  inFile >> aWord;
  cout << "After word: " << aWord << " EOF: " << inFile.eof() <<
          " FAIL: " << inFile.fail() << endl;
}
inFile.close();

cout << endl;

cout << "READING FILE WITH *NO* NEWLINE AT END" << endl;
inFile.open("fileWithNoNewlineAtEnd.txt");
while (!inFile.eof())
{
  aWord = "NOTHING_HERE";
  inFile >> aWord;
  cout << "After word: " << aWord << " EOF: " << inFile.eof() <<
          " FAIL: " << inFile.fail() << endl;
}
inFile.close();
```


Line-based Input

- Examples have all shown "item-based input" using >>
- Function "getline" gets a complete line from a stream into a string
  - It DOES extract the newline character from the stream
  - It does NOT include the newline character in the string that is populated!

myFile.txt
```
This is a test file.
Used for testing getline!
Hope you found it interesting.
```

```
Read line: This is a test file.
  Line length: 20
Read line: Used for testing getline!
  Line length: 25
Read line: Hope you found it interesting.
  Line length: 30
```

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

int main()
{
  ifstream inFile;
  string lineStr;

  inFile.open("myFile.txt");

  getline(inFile, lineStr);
  cout << "Read line: " << lineStr << endl;
  cout << "  Line length: " << lineStr.size() << endl;

  getline(inFile, lineStr);
  cout << "Read line: " << lineStr << endl;
  cout << "  Line length: " << lineStr.size() << endl;

  getline(inFile, lineStr);
  cout << "Read line: " << lineStr << endl;
  cout << "  Line length: " << lineStr.size() << endl;

  inFile.close();

  return 0;
}
```


Output File Streams

- Instead of sending output to the screen, output can be sent to a text file
  - Think of "printing to a file" instead of printing to the console

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main()
{
  int aVal;
  int bVal;
  ofstream outFile;
  aVal = 104;
  bVal = -19;

  outFile.open("output.txt");
  if (outFile.fail())
  {
    cout << "Unable to open output file" << endl;
    exit(1);
  }
  outFile << "a: " << aVal << endl;
  outFile << "b: " << bVal << endl;
  outFile.close();

  return 0;
}
```

Contents of output.txt file at end of program

```
a: 104
b: -19
```

Output of program (to screen)

```
(none)
```


A Quick Side Note

- The file stream "open" methods require a filename input as a C-string
  - You can NOT pass a C++ "string" object to the open method in our version of C++
- Luckily, you can temporarily convert a C++-string to a C-string with a method of the string class called "c_str()"

```
int main()
{
  string fileName;
  ifstream inFile;
  int intVal;

  fileName = "thisIsMyFile.txt";

  //inFile.open(fileName);
  inFile.open(fileName.c_str());

  inFile >> intVal;

  cout << "I found the int: " << intVal << endl;
  return 0;
}
```

Commented, because this is NOT allowed!

Instead, just use string's c_str() method to convert to a C-string to have the write data type for the open method

File Streams are Objects

- File streams are objects, and therefore have attributes
- One attribute indicates where, in the file, the next read will occur
- This means you can pass file stream objects around to functions *by reference* and have different functions read different parts

- In the following example, I show different functions from the same class reading different parts of the data file
  - This isn't a requirement though!
  - The file stream can be passed to any other function and reading the file will pick up where it left off
  - Could be a function of a completely different class!
    - Allows class "A" to read in data from the file that is specific to "A" objects, while still allowing class "B" to read in data specific to class "B"

Multiple Functions Reading a File, Example

```
const int MAT_ROWS = 4;
const int MAT_COLS = 5;
class MatrixInfo
{
 private:
   int matrixVals[MAT_ROWS][MAT_COLS];

   bool readHeaderRow(ifstream &inFile) const
   {
     string headerVal;
     for (int colInd = 0;
          colInd < MAT_COLS;
          colInd++)
     {
       inFile >> headerVal;
     }
     return !inFile.fail();
   }

   bool readValueRow(ifstream &inFile,
                     const int rowInd)
   {
     for (int colInd = 0;
          colInd < MAT_COLS;
          colInd++)
     {
       inFile >> matrixVals[rowInd][colInd];
     }
     return !inFile.fail();
   }

 public:
   bool readMatrixFile(const string &inFname)
   {
     bool success;
     ifstream inFile;
     inFile.open(inFname.c_str());
     //error checking here!
     success = readHeaderRow(inFile);
     //probably more error checking here!
     for (int rowInd = 0;
          rowInd < MAT_ROWS && success;
          rowInd++)
     {
       success = readValueRow(inFile, rowInd);
       //maybe some more here!
     }
     inFile.close();
   }
```

Consider a data file like this:

```
first 2ndCol 3rd Fourth 5th
50 100 150 200 250
87 187 287 387 487
-11 -22 -33 -44-55
3 1 4 1 5
```

String Stream Motivation

- Write a program that writes out a different file each iteration of a loop, using the loop counter in the filename

```
for (i = 1; i < 3; i++)
{
  if (i == 1)
  {
    fname = "out1.txt";
  }
  if (i == 2)
  {
    fname = "out2.txt";
  }
  outFile.open(fname.c_str());
  //..write to file
  outFile.close();
}
```

This is ok for a loop of 2 iterations.

How to do this for a loop of 100 or more iterations?

fname = "out" + i + ".txt"; //Will NOT work.

string's append operator (+) is for appending one *string* onto the end of another *string*

Would like to "print" strings, integers, doubles, etc. to a string variable

Recall Stream Definition

- A general definition of a ***stream***
  - A sequence of bytes, interpreted in groups of appropriate size, that can be used as input or output
    - Input stream: Such a sequence of bytes, used as input, to set memory locations to appropriately typed values
    - Output stream: Such a sequence of bytes, used as output, to screen, data file, etc.
- Currently known streams
  - Input: cin (keyboard), ifstream (input data file stream)
  - Output: cout (keyboard), ofstream (output data file stream)

Another Type Of Stream

- A general definition of a ***string***
  - A sequence of characters
    - Characters are stored in one byte of data each, so really, this definition could also read "a sequence of bytes"
    - Same definition as a stream from previous page.
- Strings can be used as input or output streams
- istringstream: Data type that will use a string as an input stream
- ostringstream: Data type that will use a string as an output stream
- Both string stream types are in a header file named <sstream>
- Standard stream operators (<< and >>) are available

Output String Streams

- Output string streams allow a user to "print" items of any data type to a sequence of characters (string)
- Often this can be an easy way to form a printable collection of values all of which are different data types
  - ostringstream oss; //Declares an output string stream object
  - oss.str(): Returns the string currently stored in the output string stream object oss
  - insertion operator (<<): Used to add values to the sequence of characters stored in the string stream

Back to Motivation

- Write a program that writes out a different file each iteration of a loop, using the loop counter in the filename

```
int main()
{
  ostringstream fnameStream;
  string fname;
  int i;
  ofstream outFile;

  for (i = 1; i < 8; i++)
  {
    fnameStream.str("");
    fnameStream << "out" << i << ".txt";
    fname = fnameStream.str();

    outFile.open(fname.c_str());
    //... code to write to file
    outFile.close();
  }

  return 0;
}
```

creates the following files:

out1.txt
out2.txt
out3.txt
out4.txt
out5.txt
out6.txt
out7.txt

Input String Streams

- Input string streams allow a user to "read in" values of different data types easily from a sequence of characters
- Often used by reading in a full line from a text file, then parsing it element-by-element
- Some important istringstream methods:
  - iss.str(string inStr): Sets the sequence of characters to use as an input stream to the contents of the inStr parameter
  - Extraction operator (>>): Consumes characters from the input string stream to be used as values
  - Usual state-checking functions available (fail, etc)

Input String Stream Example


```
#include <iostream>
#include <sstream>
#include <string>
#include <string>
using namespace std;

void printFormat()
{
  cout << "ERROR in format: " <<
          "<var> = <lhs> <oper> <rhs>" << endl;
  cout << "Example: x = 54.75 + 3.1" << endl;
  cout << "  Where <lhs> and <rhs> are doubles " <<
          "and <oper> can be + -" << endl;
}

int main()
{
  istringstream iss;
  string lineStr;
  string varStr;
  string eqStr;
  double lhsVal;
  string operStr;
  double rhsVal;
  double resultVal;
  bool success = false; //assume not initially

  cout << "Enter an equation: ";
  getline(cin, lineStr);

  iss.str(lineStr);
  iss >> varStr >> eqStr >> lhsVal >> operStr >> rhsVal;
  //continued next column
```

```
if (iss.fail() || !iss.eof())
{
  printFormat();
}
else
{
  if (operStr == "+")
  {
    resultVal = lhsVal + rhsVal;
    success = true;
  }
  else if (operStr == "-")
  {
    resultVal = lhsVal - rhsVal;
    success = true;
  }
  else //invalid operator!
  {
    printFormat();
  }
}

if (success)
{
  cout << "Result: " << varStr <<
          " = " << resultVal << endl;
}

return 0;
}
```

Enter an equation: xVal = 103.6 - 4.7
Result: xVal = 98.9

Alternative to atoi and atof


```
int main(int argc, char *argv[])
{
  int i;
  ostringstream oss;
  istringstream iss;
  int intVal;
  double doubleVal;
  char checkChar;

  if (argc != 3)
  {
    cout << "ERROR - wrong number of parameters" << endl;
    cout << "Usage: " << argv[0] << " <doubleVal> <intVal>" << endl;
    exit(2);
  }

  for (i = 1; i < argc; i++)
    oss << " " << argv[i];

  iss.str(oss.str());
  iss >> doubleVal;
  iss >> intVal;

  if (iss.fail() || !iss.eof())
  {
    cout << "ERROR - invalid format for parameters" << endl;
    cout << "Usage: " << argv[0] << " <doubleVal> <intVal>" << endl;
    exit(2);
  }
  cout << doubleVal << " / " << intVal << " = " << (doubleVal / intVal) << endl;
  return 0;
```

```
[ 57 ] temp -: altForAtoiAtof
ERROR - wrong number of parameters
Usage: altForAtoiAtof <doubleVal> <intVal>
[ 58 ] temp -: altForAtoiAtof 92.75 16
92.75 / 16 = 5.79688
[ 59 ] temp -: altForAtoiAtof hello 18
ERROR - invalid format for parameters
Usage: altForAtoiAtof <doubleVal> <intVal>
[ 60 ] temp -: altForAtoiAtof 92.75 16.75
ERROR - invalid format for parameters
Usage: altForAtoiAtof <doubleVal> <intVal>
```

Bad # params

Good run

Non-parseable values can be seen as error

"EOF" on stream can indicate "extra" data