Streams

PIC 10A, UCLA

Streams

In C++, a stream is a means of storing data, either data for reading (to set the values of variables, etc.) or for writing (to print or display information). Both **std::cin** and **std::cout** are streams that work with the console; we will look at more streams now, to read/write from/to files, and to read/write values from/to **std::string**s.

Stream Buffer

It can be helpful to remember that a stream manages a collection of data, often called the buffer. At any given time, it may wish to **put** something into the buffer or to **get** something from the buffer.

Stream Buffer: Put Position

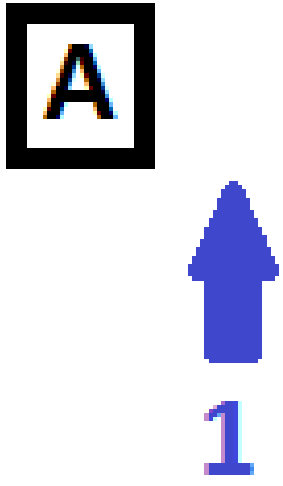
For streams that write outputs, internally they track the position of their next output.

```
out << "A" << "B" << "C";
```

Stream Buffer: Put Position

For streams that write outputs, internally they track the position of their next output.

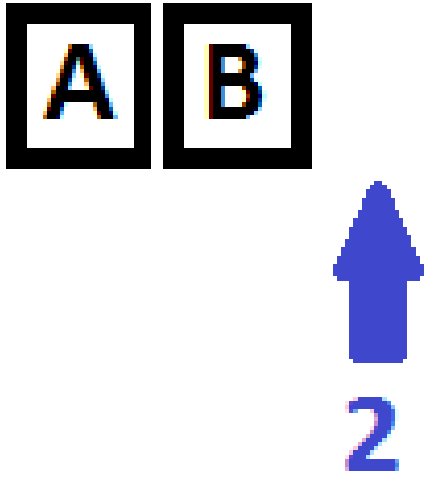
```
out << "A" << "B" << "C";
```



Stream Buffer: Put Position

For streams that write outputs, internally they track the position of their next output.

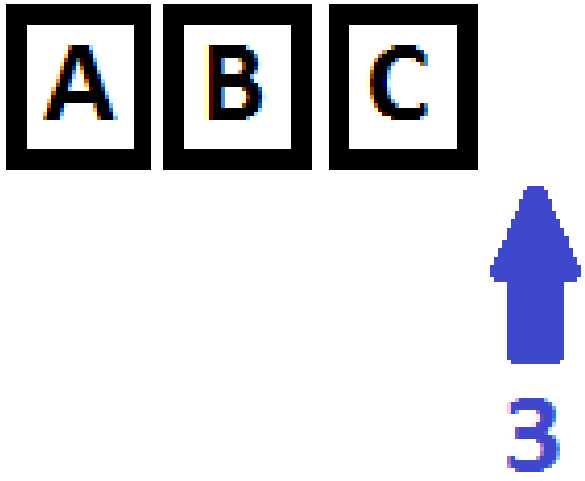
```
out << "A" << "B" << "C";
```



Stream Buffer: Put Position

For streams that write outputs, internally they track the position of their next output.

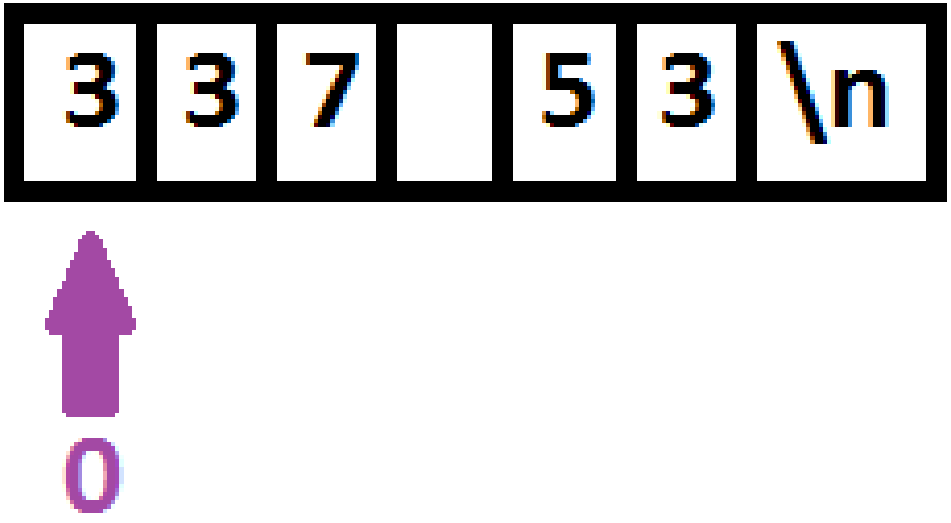
```
out << "A" << "B" << "C";
```



Stream Buffer: Get Position

For streams that read inputs, internally they track the position of their next input.

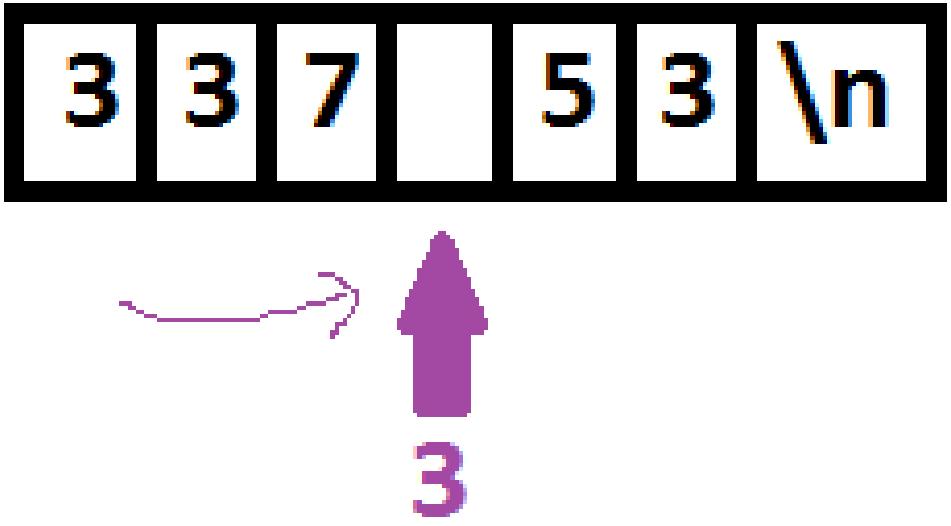
```
int x=0, y=0;
in >> x >> y;
```



Stream Buffer: Get Position

For streams that read inputs, internally they track the position of their next input.

```
int x=0, y=0;
in >> x >> y;
```

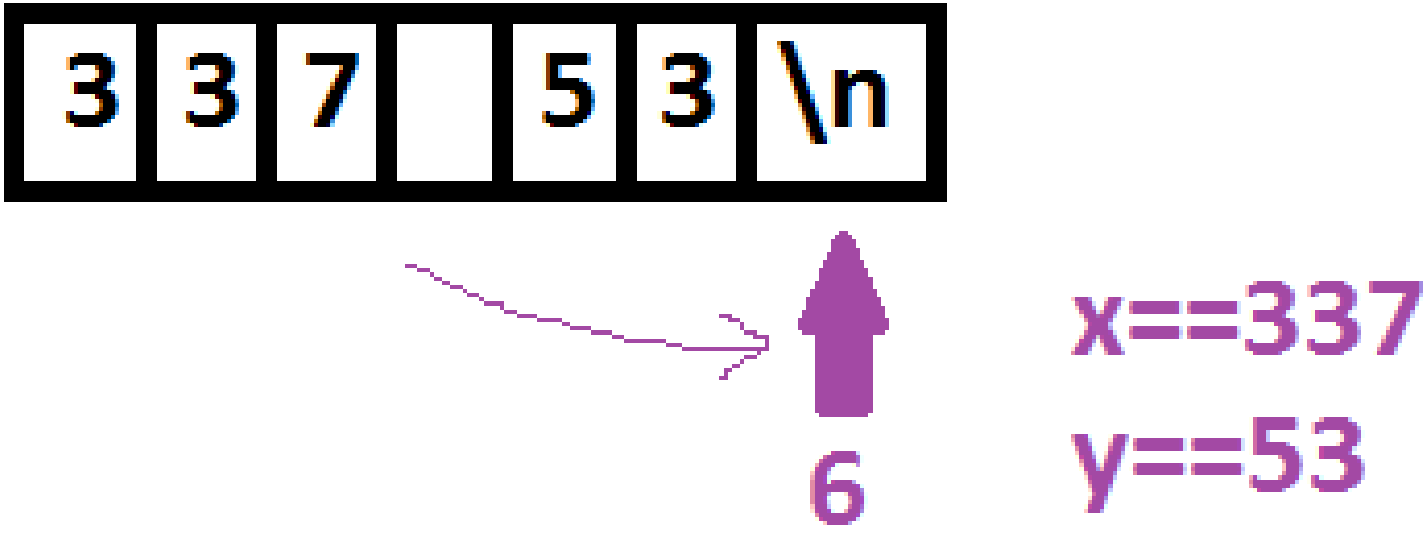


x==337

Stream Buffer: Get Position

For streams that read inputs, internally they track the position of their next input.

```
int x=0, y=0;
in >> x >> y;
```



File Streams

A **file stream** allows us to open up text files and to either read information from the files, like we have done from the console, or to write data to the files, as we have done by displaying information to the console.

If we want to store data long-term, we need to work with files because once our program closes, all the data in the RAM is gone.

String Streams

A **string stream** allows us to treat **std::string**s in a like manner to user input processed with **std::cin** or to concatenate numbers and strings together as an overall **std::string**.

Simple conversions back and forth between **std::string**s such as **"53.18"** and numeric values like **53.18** can be achieved with **string streams**.

Examples

Before even discussing all the functionalities of these streams, we will look at some examples. This is because knowing how to use **std::cin** and **std::cout** provides quite a solid background.

Examples: Output File Stream

```
// create output file stream to write to out.txt
ofstream fout("out.txt");

/* write message on one line and then 99.99 rounded to 1 decimal place
on the next */
fout << "ahoy"<< endl << fixed << setprecision(1) << 99.99;

// close file connection
fout.close();
```

Now there is a file called **"out.txt"** storing:

```
ahoy
100.0
```

Examples: Input File Stream I

Consider a file **bob.txt** storing Bob's employee information:

```
Bob FooBar
60000
```

```
std::string name; // name of employee
double salary = 0; // employee salary

// create input file stream to read from bob.txt
ifstream fin("bob.txt");

// read in the name: get entire line
getline( fin, name );

fin >> salary; // read salary next

fin.close(); // close file connection
```

Examples: Input File Stream II

From the preceding code, we have now set the variables

name == "Bob Foobar"
salary == 60000.

Examples: Output String Stream I

```
// ask for a number and set it
cout <<"Enter an integer: ";
int n = 0;
cin >> n;

// make empty output string stream
ostringstream sout;

sout << n*n; // insert n squared into the stream

string number = sout.str(); // collect number as string

// list how many digits
cout << "Your number squared has: "<< number.size() << " digits.";
```

Examples: Output String Stream II

With this program, a user could enter:

14[ENTER]

and since $14^2 = 196$, the output would be:

```
Your number squared has 3 digits.
```

Examples: Input String Stream

Here, we can add two numbers stored in **std::string**s.

```
// store the numbers in a string
string numbers("14 13");

// make input string stream from the numbers
istringstream sin(numbers);

// read from the stream into numbers
int num1 = 0, num2 = 0;
sin >> num1 >> num2;

// display the sum
cout << "The sum is: "<< num1 + num2;
```

The output is

Streams and Overloading Operators

The operations are very similar to those we have already seen with **std::cin** and **std::cout**.

To gain access to input and output file streams, we need to **#include<fstream>**.

To gain access to input and output string streams, we need to **#include<sstream>**.

Streams and Overloading Operators

There are even deeper similarities between streams...

All istringstreams and ifstreams are also istreams and
all ostringstreams and ofstreams are also ostreams
although the converses are not true (an **istream** is not an **ifstream**, for example).

When we overload operators such as **<<** or **>>**, we typically overload for **ostream** or **istream** and the overload will work for the other streams, too.

Streams and Overloading Operators

```
istream& operator>>(istream&, Foo&);
```

can read from **cin** into a **Foo**, or from an input file stream into a **Foo**, or from an input string stream into a **Foo**, depending on what appears on the left of **>>**.

Output Streams

The following are functions that apply across all output streams:

- operator<<
  -can always be used to write data to the console, to a file, or to a string.

- void flush()
  -ensures that everything within the stream's buffer is written to the console/file/string at the time the function is called. For example, cout.flush();

- **endl**
  - prints a '\n' and flushes the buffer

- The manipulators **setw**, **setfill**, **setpecision**, **left**, and **right**.

Output Streams

Remark: we can now see the difference between

```
cout << "\n";
```

and

```
cout << endl;
```

With **endl**, in addition to the adding a new line, the buffer is flushed.
Using '\n' is more efficient.

Output Streams

The output streams may be optimized during the running of a program to only print periodically. For small programs, we do not see the difference, but when running large simulations and writing to a file, the lack of flushing can mean a delay in being able to process the data. Flushing the buffer ensures the data is written.

Before a std::cin statement, the std::cout buffer is guaranteed to be flushed.

Output File Streams

We can now look at some constructors and member functions for output file streams.

```
/**
Default constructor: makes an output file stream
*/
ofstream();

/**
Makes an output stream tied to the given file
@param fileName the name of the file to open
*/
ofstream( const string& fileName);
```

Output File Streams

```
/**
Links the file stream with the given file
@param fileName the name of the file to open
*/
void open( const string& fileName );

/**
Closes the link to whatever file the stream is linked with
*/
void close();
```

Output File Streams

Remarks:

- When a file is no longer in use, it is important to close the link with that file, both for the input and output file streams.

  Maintaining a link with the file wastes computer resources.

  Footnote: when a file stream goes out of scope, it closes the file it is linked with anyway (but it's better not to rely upon this because we may forget when that is not the case!)

Output File Streams

- For both output and input file streams, file names are relative to the location of the program. For example in Windows, if our **.cpp** file is compiled in **C:\Users\JoeBruin\PIC10A** then when we write,

  ```
  ofstream ofs("A.txt");
  ```

  we mean to open a file **C:\Users\JoeBruin\PIC10A\A.txt**
  If we wanted to open a file **C:\Users\JoeBruin\B.txt**, we would need to write

  ```
  ofstream ofs("..\\B.txt");
  ```

  The **..** indicates one directory up. Note the double backslash: to generate a backslash character, we need two!

Output File Streams

- Linux has a different directory structure, using /s, so this gets confusing. It's best to just place the files with the code using them and avoid relative directories.

- If the file does not yet exist, one is created with the name given. If the file already exists, it is opened **and overwritten** by default (this can be changed, though)!!!

  It is easy to lose data by accidentally overwriting a file!

Output String Streams

We can now look at some constructors and member functions for output string streams.

```
/**
Default constructor: makes an output string stream with nothing
*/
ostringstream();

/**
Makes an output string stream storing the string given to it
@param startString the string to store initially
*/
ostringstream( const string& startString);
```

Output String Streams

```
/**
Returns the string storing the contents of the stream
@return the stream contents
*/
string str() const;

/**
Overwrites what is in the stream with the given input
@param newStreamContent the new content for the stream
*/
void str( const string& newStreamContent );
```

Remark: the **str** member function is overloaded!

Output String Streams

Assuming a variable type can be printed with **operator<<** through an output stream, we can convert it to a **std::string**.

```
/**
This function converts its input argument to a string based on how it
would be rendered through an output stream and operator<<.

@tparam T the type of data being converted to a string
@param input the value being turned into a string
@return the string representation of value
*/
template<typename T>
string toString( const T& input ) {
    ostringstream oss; // create empty output string stream
    oss << input; // insert input, whatever type it may be!
    return oss.str(); // and return the string stored
}
```

Output String Streams

With the preceding template, we can then write:

```
int n = 740000;
string s = toString(n);

// with n as a string, counting digits is counting characters
cout << n << " has "<< s.size() << " digits.";
```

Output String Streams

Remarks:

- This function depends upon there being a valid
  ostream& operator<<(ostream&, const T&);

- Note that within the template, it would be invalid to write

  ostringstream oss(input); // Error: input is not a string!!!

  We had to create an empty string and push the input into it.
  Likewise, we cannot write

  oss.str(input); // ERROR: input is not a string!

Input Streams

The following are functions that apply across all input streams:

- operator>>
  -can always be used to read data from the console, from a file, or from a string.

- istream& getline(istream&, string&)
  -can always read from an input stream into a string. Note the return type is actually the stream itself!

  An optional third argument can be passed to **getline** to determine where to stop reading input: the default is **'\n'**, but we could read up to and not including a tab:

  getline(cin, someString, '\t'); // reads up to tab to set s, removes tab

Input Streams

The following are functions that apply across all input streams:

- int get();
  -member function extracts a **char** and returns its **int** value

- void ignore();
  -member function extracts a **char** from the buffer

Input Streams

- Some extra details on **ignore:**

```
/**
This member function extracts and discards the next ignoreNumber
characters from the input stream, or until it reaches the char
stopChar
@param ignoreNumber the number of chars to discard
@param stopChar the int value of the char to stop at, if it comes up
before ignoreNumber chars have been discarded
*/
void ignore( int ignoreNumber = 1, stopChar = EOF );
```

Subtlety: actually, **ignoreNumber** might be something other than an **int**.

EOF designates "end of file": this is a special trigger for an input stream to stop reading when there is no more to read.

Input Streams

- bool eof() const;
  -member function returns **true** if end of file has been reached, otherwise **false**.

- bool fail() const;
  -member function returns **true** if the stream is in a failed state, otherwise **false**.

  A stream may fail because it could not process user input (expected an int, got a "ABC"instead, for example), or because it could not find/open a desired file, or because it tried to read past the end of file, etc.

Input Streams

- void clear();
  -member function clears the failed state of a stream so it can be used for reading again.

- **conversion to bool:** a stream will be converted to **true** unless it is in a failed state (due to inability to process input, reading past end of file, etc.)

Input Streams

The conversion from **std::cin** to a **bool** can lead to some "interesting" loop conditions:

```
int userNum = 0; // user's number
int sum = 0; // the sum of the numbers

// give user instructions
cout << "Enter numbers to sum. When done, input 'DONE'";

// while user still giving numbers
while( cin >>userNum ) {
    sum += userNum; // add input to sum
}

cout << "The sum is: "<< sum;
```

Input Streams

The user could enter

3 4 5 DONE

to generate the output

```
The sum is:  12
```

Recall: the output of **cin >>userNum** should be **cin** itself. Thus, after each attempt at reading in an **int**, we are able to query whether **cin** has succeeded.

If the user enters something invalid like "DONE" when **cin** expects an **int**, then **cin** fails at that reading and the loop body is not entered.

Input Streams

Assuming the user has entered something invalid into the stream, we can **clear the failed state** of **std::cin** with

```
cin.clear(); // clear failed state so stream an operate
```

and **ignore the rest** of their invalid input with:

```
/* ignore as many character as the stream can store or until the newline from the user */
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

Input Streams

numeric_limits<streamsize>::max() represents the largest number of characters that could be stored by the stream. Gaining access to this functionality requires that we **#include<limits>**.

If we know their invalid characters only span one place, we could just use

```
cin.ignore();
```

Or if we know their invalid characters would only span **INVALID_RANGE** we could simply use

```
cin.ignore( INVALID_RANGE );
```

Input Streams

```
bool defiant = true; // whether user is defiant

cout << "Enter your favourite integer: "; // ask for num
while(defiant) { // while user disobeys
    int favNum = 0; // their fav num
    cin >> favNum; // accept number

    if ( cin.fail() ) { // if int was not given
        cin.clear(); // clear the failed state
        cin.ignore(numeric_limits<streamsize>::max(),'\n'); // ignore bad stuff
        cout << "Enter an integer! ";
    }
    else { // otherwise input is valid
        defiant = false; // and user is not defiant
        cout << "I like "<<favNum << ", too.";
    }
}
```

Input Streams

```
Enter your favourite integer:  toast
Enter an integer!  peanut butter
Enter an integer!  144
I like 144, too.
```

Input File Streams

We can now look at some constructors and member functions for input file streams.

```
/**
Default constructor: makes an input file stream
*/
ifstream();

/**
Makes an input stream tied to the given file
@param fileName the name of the file to open
*/
ifstream( const string& fileName);
```

Input File Streams

```
/**
Links the file stream with the given file
@param fileName the name of the file to open
*/
void open( const string& fileName );

/**
Closes the link to whatever file the stream is linked with
*/
void close();
```

Input File Streams

We can read an entire text file and display it to the console.

```
ifstream in("file.txt"); // open the file

if( in.fail() ) { // if the stream failed to open
    cout << "failed to open"; // give warning
}
else { // then successful
    string line; // a line to read
    while( !in.eof() ) { // while we are not at the end of the file
        getline(in, line); // read the line
        cout << line << endl; // and display the line
    }
}

in.close(); // disassociate file and stream
```

Input File Streams

There is a lot of variety in how we could have implemented the preceding code. For example, we could have replaced the **if** condition with

```
if( ! in )
```

because **!in** is **true** only if **in** is **false**, i.e. has failed somehow.

We could have replaced the **while loop** with

```
while ( getline(in, line) ) {
    cout << line << end;
}
```

because **getline** returns the stream and thus the condition in the while should only be true if there were more to read in the file.

Input String Streams

We can now look at some constructors and member functions for input string streams.

```
/**
Default constructor: makes an input string stream with nothing
*/
istringstream();

/**
Makes an input string stream storing the string given to it
@param startString the string to store initially
*/
istringstream( const string& startString);
```

Input String Streams

```
/**
Returns the string storing the contents of the stream
@return the stream contents
*/
string str() const;

/**
Overwrites what is in the stream with the given input
@param newStreamContent the new content for the stream
*/
void str( const string& newStreamContent );
```

Input String Streams

With this idea, we can write a template function to convert a **std::string** to any data type, assuming its contents can be converted in such as way.

```
/**
This function converts a string to another data type
@tparam T the type we are converting to
@param input the string we are converting
@return the converted value
*/
template<typename T>
T stringTo( const string& input ) {
    istringstream iss( input ); // store the input
    T t; // make a T (assuming it can be default constructed)
    iss >> t; // update t, assuming it can be written to
    return t;
}
```

Input String Streams

Remarks:

- In using this template, the type of **T** cannot be deduced because our input is always a **std::string**. We need to explicitly call the template with the desired output type.

  ```
  string s = "123456";
  int n = stringTo<int>(s); // n == 123456
  ```

- For the template to work, we need to be able to default construct a **T**.

- For the template to work, we need a suitable operator `istream& operator>>(istream&, T&);`.

Summary

- Streams allow us to read/write data.
- Across all input streams, there are many similarities such as **operator>>**, **getline**, **clear**, **fail**, and so on.
- Across all output streams, there are many similarities such as **operator<<**, **endl**, and the manipulators.
- File streams are found in the **<fstream>** header; string streams are found in the **<sstream>** header.
- An input file stream, **ifstream**, reads data from a file to variables.
- An output file stream, **ofstream**, writes data to a file.
- An input string stream, **istringstream**, reads from strings into variables.
- An output string stream, **ostringstream**, writes data into strings.
- Resources should be managed appropriately by **close**ing the links to files.