Streams
Game Plan

- overview
- stringstream
- state bits
- input/output streams
- manipulators

9 January 2020
We often want our programs to interact with external devices.
Here are some common devices we will use.

- console & keyboard
- files
- other programs (pipelines)
- sockets (networking)

Take CS 110!
Take CS 144!
How would you print a Date object to the console?

```java
int month = 9;
int day = 26;
```

A Date object in your program

console
You’d first convert the object to a string.
Then write the string to the console.
How would you read a double from a file?
You would read the characters and convert it to a string first.
Then, you would convert the string to the proper double as a variable.
In general, there are two main challenges.

- `3.14` in your program
- External source
  - Read/write
  - String representation
  - Type conversion
  - `3.14`
First: we need to retrieve/send data from the source in string form.

external source \rightarrow \text{read/write} \rightarrow \text{string representation} \rightarrow \text{type conversion} \rightarrow \text{double in your program}
Second: we need to convert between data in our program and its string representation.
Streams provide a unified interface for interacting with external input.

Streams allow for the read/write of data from external sources, converting string representations to the correct data types for use in a program.
You can imagine a stream to be a character buffer that automatically interacts with the external source.

```
3 . 1 4 p i ...
```

3.14

double in your program

3.14

type conversion
Streams also convert variables to a string form that can be written in the buffer.

3.14

double in your program
stringstream
A stringstream is not connected to any external source.

"3.14" string representation

3.14 double in your program
Example

creating, extracting, and inserting from a stringstream
stringstream constructors

istringstream iss("Initial");

ostringstream oss("Initial");

Constructors with initial text in the buffer. Can optionally provide “modes” such as ate (start at end) or bin (read as binary).

istringstream iss("Initial", stringstream::bin);
ostringstream oss("Initial", stringstream::ate);
stringstream constructors

// “16.9 Ounces”, pos at front
istringstream iss("16.9 Ounces");

// “16.9 Ounces”, pos at front
ostringstream oss("16.9 Ounces");

// “16.9 Ounces”, pos at back
ostringstream oss("16.9 Ounces", stringstream::ate);
stringstream formatted i/o

```
oss << var1 << var2;
```

Inserted into oss's buffer

```
iss >> var1 >> var2;
```

Converted to characters
stringstream formatted i/o

\[ \text{oss} \ll \text{var1} \ll \text{var2}; \]

Inserted into oss’s buffer

Converted to characters

\[ \text{iss} \gg \text{var1} \gg \text{var2}; \]
stringstream formatted i/o

```cpp
oss << var1 << var2;
```

```cpp
iss >> var1 >> var2;
```

- **Extracted from iss’s buffer**
- **Converted to v1’s type**
stringstream formatted i/o

```cpp
oss << var1 << var2;
```

```cpp
iss >> var1 >> var2;
```

- Extracted from iss’s buffer
- Converted to v2’s type
What is a whitespace separated token?

16.9 \n Ounces. . \t \n \n -38271

Token 1  Token 2  Token 3  Token 4

iss >> token1 >> token2 >> token3 >> token4;
Types matter! Stream stops reading at any whitespace or any invalid character for the type.

```cpp
int token1, string token2, char token3, bool token4
iss >> token1 >> token2 >> token3 >> token4;
```
// buffer contains "Ito En Green Tea ", pos at end
ostringstream oss("Ito En Green Tea ", stringstream::ate);

// str function returns characters in buffer as a string
cout << oss.str() << endl;

// Converts 16.9 to string and insert into buffer
oss << 16.9 << " Ounce ";

// prints "Ito En Green Tea 16.9 Ounce "
cout << oss.str() << endl;
stringstream formatted i/o

// buffer contains "Ito En Green Tea ", pos at front
istringstream iss("16.9 Ounces");

double amount, string unit;

// reads next whitespace-separated token "16.9"
// converts to correct type (double) and placed into 'amount'
// same for unit, except no conversion needed
iss >> amount >> unit;

// amount is now 8.45
amount /= 2;
Questions to Ponder

• What exactly does >> do?
• Why can you chain << and >>?
• Is there a stringstream that can you can both insert and extract?
Key Takeaways

• >> extracts the next variable of a certain type, up to the next whitespace.
• The >> and << operators return a reference to the stream itself, so in each instance the stream is the left-hand operand.
• Yes, it’s called a stringstream. Reading and writing simultaneously can often lead to subtle bugs, be careful!
stringstream positioning functions

get position
oss.tellp();
iss.tellg();

set position
oss.seekp(pos);
iss.seekg(pos);

create offset
streamoff(n)

These methods let you manually set the position. Most useful is the offset which can be added to positions.

Note: the types are a little funky. Read the documentation!
Learn more about stringbuf!

For more fine-tuned control, you can access and modify the underlying buffer.

Requires knowledge of pointers and C-arrays. Learn more about them after taking CS 107!
Example

implementing stringToInteger (first attempt)
First attempt: no error-checking.

```c++
int stringToInteger(const string& str) {
    istringstream iss(str);

    int result;
    iss >> result; // problem: what if this fails?

    return result;
}
```
state bits
Four bits indicate the state of the stream.

- **G**: Good bit: ready for read/write.
- **F**: Fail bit: previous operation failed, all future operations frozen.
- **E**: EOF bit: previous operation reached the end of buffer content.
- **B**: Bad bit: external error, likely irrecoverable.
Common reasons why that bit is on.

G. Nothing unusual, on when other bits are off.

F. Type mismatch, file can’t be opened, seekg failed.

E. Reached the end of the buffer.

B. Could not move characters to buffer from external source. (e.g. the file you are reading from suddenly is deleted)
Important things about state bits.

- G and B are not opposites! (e.g. type mismatch)
- G and F are not opposites! (e.g. end of file)
- F and E are normally the ones you will be checking.

Conclusion: You should rarely be using G.
Example

print the stream bits in our function implementing stringToInteger (second attempt)
Second attempt: incomplete error-checking.

```cpp
int stringToInteger(const string& str) {
    istream iss(str);

    int result;
    iss >> result;
    if (iss.fail()) throw domain_error(...);

    return result;
}
```

Check if the operation failed (due to type mismatch).
Third attempt: complete error-checking.

```cpp
int stringToInteger(const string& str) {
    istringstream iss(str);
    int result;
    iss >> result;
    if (iss.fail()) throw domain_error(...);
    char remain;
    iss >> remain;
    if (!iss.fail()) throw domain_error(...);
    return result;
}
```

Why do we need ‘remain’? Ensure no characters after the integer.
These are equivalent!

```java
iss >> remain;
if (iss.fail()) { // report error }

if (!(iss >> remain)) { // report error }
```

Reason: `>>` returns `iss` itself, which can act like a boolean for `iss.fail()`. 
Third attempt: complete error-checking.

```cpp
int stringToInteger(const string& str) {
    istringstream iss(str);

    int result; char remain;
    if (!(iss >> result) || iss >> remain)
        throw domain_error(…);

    return result;
}
```

Notice the short circuiting!
cout and cin
Key difference: there is an external source.
Data is sent between the external source and the buffer.

keyboard → read → “42” → type conversion → 42

int in your program
There are four standard iostreams.

- **cin** Standard input stream
- **cout** Standard output stream (buffered)
- **cerr** Standard error stream (unbuffered)
- **clog** Standard error stream (buffered)
Example

output streams, buffering, and flushing
Buffered stream: characters are stored in an intermediate buffer before being moved to the external source.

```cpp
3.14

3.14
```

double in your program

```
cout << 3.14;
```
To push the characters to the external source, the stream must be “flushed”

cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;

“CS106” only shows up at this point.
To push the characters to the external source, the stream must be “flushed”

cout << “CS”;  
cout << 106;  
cout << flush;  
cout << ‘L’;  
cout << endl;  

Include a flush. Console: “CS106L\n”
To flush or not to flush?

// Option 1: flush every int
for (int i = 0; i < 100000; ++i) {
    cout << i << endl;
}

// Option 2: flush only at the end
for (int i = 0; i < 100000; ++i) {
    cout << i << '\n';
}
cout << flush;

What’s the pros/cons of each option?
Other streams using the console may also flush `cout`.

```cpp
int num;
cout << "CS";
cout << 106;
cout << 'L';
cin >> num;
```

`cout` is flushed when `cin` is waiting for user input.
Streams also convert variables to a string form that can be written in the buffer.

3.14

3.14

double in your program
Example

input streams, buffering, and waiting for user input.
The lecture code included `cout` statements.

```cpp
// cin

name = cin >> name;
age = cin >> age;
cout << name << age;
cin >> response;
```

The lecture code included `cout` statements.
Since there is nothing in the buffer, `cin` waits for the user to type something in.

```cpp
name >> age;
cout << name << age;
cin >> response;
```
After typing in my name and pressing enter, cin transfers what I typed into the buffer.
Then we read from the buffer into the variable name, just like a stringstream.

```cpp
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

```
Avery
```

```
name (string)
```

```
"Avery"
```

```
age (int)
```

```
???
```

```
response (string)
```

```
???
```
cin >> name;  
cin >> age;  
cout << name << age;  
cin >> response;

name (string)  
"Avery"  
response (string)  
??

age (int)  
??

Avery

G F E B

Avery

9 January 2020

9 January 2020
Everything I type is transferred to the buffer.
We read directly into an int, stopping at a whitespace.

```cpp
Avery
20
```

```cpp
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

- **name (string)**: “Avery”
- **age (int)**: 20
- **response (string)**: ???

We read directly into an int, stopping at a whitespace.
We read directly into an int, stopping at a whitespace.

```cpp
Avery

20

cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

We read directly into an int, stopping at a whitespace.
We now print the variables (don’t forget cout is buffered!)

cin >> name;
cin >> age;
cout << name << age;
cin >> response;

Avery
20

name (string)

“Avery”

age (int)

20

response (string)

???
cin >> name;
cin >> age;
cout << name << age;
cin >> response;

Avery
20
Avery20

name (string)
“Avery”
age (int)
20
response (string)
???

But attempting reading again will flush cout.
We prompt the user again.

```cpp
Avery
20
Avery20 |
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

<table>
<thead>
<tr>
<th>name</th>
<th>(string)</th>
<th>“Avery”</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>(int)</td>
<td>20</td>
</tr>
<tr>
<td>response</td>
<td>(string)</td>
<td>???</td>
</tr>
</tbody>
</table>

We prompt the user again.
We type something, it's transferred to the buffer, and read into the variable.

```cpp
Avery

20

YES

name
(string)

age
(int)

response
(string)

"Avery"

20

"YES"

We type something, it's transferred to the buffer, and read into the variable.

cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```
Key Takeways

• When does the program prompt the user for input?
• Why does the `cout` operation not immediately print the output onto the console? When is the output printed?
• Does the position pointer skip whitespace before the token or after the token with each `>>` operation? (this is important!)
• Does the position pointer always read up to a whitespace? If not, come up with a counterexample.
Key Takeways

• The program hangs and waits for user input when the position reaches EOF, past the last token in the buffer.
• All input operations will flush cout.
• The position pointer does the following:
  • consume all whitespaces (spaces, newlines, etc.)
  • reads as many characters until:
    • a whitespace is reached, or...
    • for primitives, the maximum number of bytes necessary to form a valid variable.
    • example: if we extract an int from “86.2”, we’ll get 86, with pos at the decimal point.
Example

when input streams go wrong
Let’s try something innocuous. I type in my full name.

```cpp
let name = cin >> name;
let age = cin >> age;
let response = cin >> response;

cout << name << age << response;
```
After typing in my name and pressing enter, cin transfers what I typed into the buffer.
cin >> name;
cin >> age;
cout << name << age;
cin >> response;

Remember cin reads up to a whitespace.
Avery Wang

Avery Wang

name (string) "Avery"

age (int) ???

response (string) ???

```cpp
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

cin now tries to read an int. It skips past the initial whitespace.
Avery Wang

Avery Wang reads in a string:

```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

And prints:

```
Avery
```

But it tries to read in an int,

but fails.
cin >> name;
cin >> age;
cout << name << age;
cin >> response;

name (string)  "Avery"
age (int) ??
response (string) ??

Avery Wang

The fail bit is turned on.
cin >> name;  
cin >> age;  
cout << name << age;  
cin >> response;

Avery Wang
Avery-2736262

name (string)  “Avery”
response (string)  ???
age (int)  ???

cout now prints the name and age (which is uninitialized!)
Worst, since the fail bit is on, all future cin operations fail.

```cpp
Avery Wang
Avery-2736262

name (string)  "Avery"
age (int) ???
response (string) ???

Avery Wang
Avery-2736262

cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

Worst, since the fail bit is on, all future cin operations fail.
3 reason why >> with cin is a nightmare.

1. cin reads the entire line into the buffer but extracts whitespace-separated tokens.

2. Trash in the buffer will make cin not prompt the user for input at the right time.

3. When cin fails, all future cin operations fail too.
manipulators

BTW these exist, they’re easy to use, just look them up.
There are some keywords that will change the behavior of the stream when inserted.

- `endl`: insert newline and flush stream
- `ws`: skips all whitespace until it finds another char
- `boolalpha`: prints “true” and “false” for bools.
- `hex`: prints numbers in hexadecimal
- `setprecision`: adjusts the precision of printed numbers
We can use manipulators to pad the output.

```cpp
cout << "[" << setw(10) << "Ito" << "]";
```

Output: `[Ito]`

```cpp
cout << "[" << left << setw(10) << "Ito" << "]";
```

Output: `[Ito]`

```cpp
cout << "[" << left << setfill('-') << setw(10) << "Ito" << "]";
```

Output: `[Ito------]`
Your challenge for Tuesday

// Given a string that has whitespace separated tokens
// return a vector containing those tokens

ingput string: “I 3 \n MM 6”
output vector: {“I”, “3”, “MM”, “6”}

vector<string> stringSplit(const string& str) {
    vector<string> tokens;
    // use tokens.push_back(token);
    return tokens;
}
Your challenge for Tuesday

// Given a start time and a duration,
// calculate the end time.
// Assume correct formatting of string.

input string: “1:30 PM \n 1 hour 20 minute”
console string: “2:50 PM”

void printEndTime(const string& input) {
   // you fill this out!
}
How do you avoid this bug?

```cpp
int hour = 3;
int minute = 0;

cout << hour << ":" << minute << "\n";
// prints 3:0 instead of 3:00
```
Your challenge for Tuesday

Play around with streams and try printing out the state bits.

See if you can use the state bits to help you control the streams.

On Friday’s 106B lecture, you’ll learn about file streams. Test the state bits on those too!
Next time

Stream error-handling: implementing simpio and other Stanford i/o libraries