Streams I
Game Plan

- overview + stringstream
- state bits
- input/output streams
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!
We might print a date to the console.

A Date struct in your program

```
int month
int day
```

String representation

"Sep. 26"

Write
We might read a double from a file.

file → read → "3.14" → type conversion → 3.14

string representation

double in your program
There are two main challenges.

- External source
- String representation
- Read/write conversion
- Type conversion
- Double in your program
First: we need to retrieve/send data from the source in string form.
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 read/write operations and type conversion.

- **External Source**: Input data from sources like files, commands, or user input.
- **String Representation**: The string "3.14" is read and written to the stream.
- **Type Conversion**: The string "3.14" is converted to a double in the program.

This diagram illustrates the process of reading and converting data from an external source to a usable format within a program.
You can imagine a stream to be a character buffer that automatically interacts with the external source.

stream

3.14

type conversion

double in your program
Streams also convert variables to a string form that can be written in the buffer.

```cpp
3.14
```

**input**

**output**

```cpp
<<
```

**double in your program**

3.14
Don’t worry about how read/write to the source actually happens!
Although...when read/writes happen will matter later when we discuss buffering.
stringstream
A stringstream is not connected to any external source.

nothing

string representation → 3.14

"3.14"

double in your program

A stringstream is not connected to any external source.

nothing

string representation → 3.14

"3.14"

double in your program
Example
creating, extracting, and inserting from a stringstream
Construct oss with the string parameter as the initial string.

```cpp
ostringstream oss("Ito En Green Tea ");
```
Construct `oss` with the string parameter as the initial string.

```cpp
ostringstream oss("Ito En Green Tea ");
```
The `str` method outputs the string in the entire buffer.

```cpp
ostringstream oss("Ito En Green Tea ");
cout << oss.str() << endl; // Ito En Green Tea
```
We convert 16.9 to the string form “16.9” and insert into oss.

oss << 16.9 << “ Ounce ”;
The position started in the front, so we are overwriting the buffer!

oss << 16.9 << " Ounce ";
The position started in the front, so we are overwriting the buffer!

```cpp
oss << 16.9 << " Ounce ";
cout << oss.str() << endl; // 16.9 Ounce n Tea
```
The buffer is as big as it (reasonably) needs to be. Don’t worry about the details.

```
oss << "(Pack of " << 12 << ")\n";
```
The buffer is as big as it (reasonably) needs to be. Don’t worry about the details.
The buffer is as big as it (reasonably) needs to be. Don’t worry about the details.

```cpp
oss << "(Pack of " << 12 << ")\n"
cout << oss.str() << endl; // 16.9 Ounce (Pack of 12)\n```
We intended to append to the initial string.
Now the position starts at end.

```cpp
ostringstream oss("Ito En Green Tea ", stringstream::ate);
```
Rest of the program works the same.

```cpp
ostringstream oss("Ito En Green Tea ", stringstream::ate);
oss << 16.9 << " Ounce ";
```
Let’s now create an input stream using the same string.
Declare two variables.

```cpp
istringstream iss(oss.str()); // 16.9 Ounce (Pack of 12)
double amount, string unit;
```
```cpp
istringstream iss(oss.str()); // 16.9 Ounce (Pack of 12)
double amount, string unit;
iss >> amount >> unit;
```

Try reading in a double then a string.
istringstream iss(oss.str()); // 16.9 Ounce (Pack of 12)

double amount, string unit;

iss >> amount >> unit;

amount 16.9

unit ???
istringstream iss(oss.str()); // 16.9 Ounce (Pack of 12)
double amount, string unit;
iss >> amount >> unit;

amount: 16.9
unit: "Ounce"

It also skips any leading whitespace.
istringstream iss(oss.str()); // 16.9 Ounce (Pack of 12)

double amount, string unit;
iss >> amount >> unit;
amount /= 2;

amount 8.45
unit “Ounce”

This proves that amount is indeed a double.
Example

manually repositioning the stream position
One more time:
position at index 4

oss <<< 16.9;
Calculate a new index, which is the current index plus an offset of 3.

oss << 16.9;

fpos pos = oss.tellp() + streamoff(3); // index 3+4=7
Calculate a new index, which is the current index plus an offset of 3.

oss <= 16.9;

fpos pos = oss.tellp() + streamoff(3);  // index 3+4=7

oss.seekp(pos);  // move to index 7
Write and advance position
oss << "Black";

oss.seekp(streamoff(1), stringstream::cur);

Move offset of 1 from current position.
oss << "Black";
oss.seekp(streamoff(1), stringstream::cur);
oss << "Boba";

Write and advance.
stringstream key methods

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 oss("Initial", stringstream::bin);
ostringstream oss("Initial", stringstream::ate);
stringstream key methods

oss << var1 << var2;

iss >> var1 >> var2;

Insert or extract into the buffer.
Converts type of var to and from string type.

Read about the get/put and read/write functions which provide unformatted input/output!
stringstream key methods

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!
Example

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

```cpp
int stringToInteger(const string& str) {
    istringstream iss(str);
    int result;
    iss >> result;
    return result;
}
```
First attempt: no error-checking.

```cpp
int stringToInteger(const string& str) {
    istringstream iss(str);
    int result;
    iss >> result;
    return result;
}
```

How do we know if this line succeeded?
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.
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)
First attempt: no error-checking.

```cpp
int stringToInteger(const string& str) {
    istringstream iss(str);
    int result;
    iss >> result;
    return result;
}
```

How do we know if this line succeeded?
Second attempt: incomplete error-checking.

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

We also need to ensure there’s nothing left to read in the stream.
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 >> ch;
    if (!iss.fail()) throw domain_error(…);
    return result;
}
```

Check if the operation failed (due to type mismatch).
Very helpful shortcut.

```cpp
iss >> ch;
if (iss.fail()) { // report error }

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

The >> operator returns the stream which is converted to stream.fail().
Third attempt: complete error-checking.

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

    int result; char remain;
    if (!(iss >> result) || iss >> ch)
        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.

1. **keyboard**
2. Read
3. "42"
4. String representation
5. Type conversion
6. **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)
Let’s first discuss the output streams.

- **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
In the lecture code, I inserted slow function calls between each line.

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

Added to buffer
Notice that nothing shows up on the console yet!

```cpp
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
```
cout | "CS"
cout << 106;
cout << flush;
cout << 'L';
cout << endl;

(buffered)

Same thing here.
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;

(buffered)

Same thing here.
Now that we flush the stream, everything in the buffer is flushed to the console.

cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
Now that we flush the stream, everything in the buffer is flushed to the console.

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

The stream is still buffered.
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;

The stream is still buffered.
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;

This is equivalent to adding '\n' and then flushing.
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;

This is equivalent to adding '\n' and then flushing.
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;

This is equivalent to adding \n' and then flushing.
For unbuffered streams everything inserted shows up immediately.
For unbuffered streams, everything inserted shows up immediately.

```cpp
cerr << "CS";
cerr << 106;
cerr << flush;
cerr << 'L';
cerr << endl;
```
cerr << “CS”;
cerr << 106;
cerr << flush;
cerr << ‘L’;
cerr << endl;

For unbuffered streams everything inserted shows up immediately.
24 September 2019

```
cerr << "CS";
cell << 106;
cerr << flush;
cerr << '\L';
cerr << endl;
```

(unbuffered)

Flushing doesn’t do anything.
Flushing doesn’t do anything.
```
cerr << "CS";
cerr << 106;
cerr << flush;
cerr << 'L';
cout << endl;
```

And `endl` still adds a new line.
Example

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

```
    cin >> name;
    cin >> age;
    cout << name << age;
    cin >> response;
```
The lecture code included `cout` statements.

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

- **name** (string)
- **age** (int)
- **response** (string)

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

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

```cpp
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```
Then we read from the buffer into the variable name, just like a stringstream.

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

Avery

name
(string)

“Avery”

age
(int)

???

response
(string)

???

Then we read from the buffer into the variable name, just like a stringstream.
cin >> name;
cin >> age;
cout << name << age;
cin >> response;

Avery

name (string)

“Avery”

age (int)

???

response (string)

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

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

Avery
20

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
#include <iostream>

int main() {
    std::string name, response;
    int age;

    std::cin >> name;  // Avery
    std::cin >> age;    // 20
    std::cout << name << age;
    std::cin >> response;  // ???

    return 0;
}
```

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

```cpp
Avery
20
```

```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```
cin A v e r y \n 2 0 \n
Avery
20
Avery20

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

But attempting reading again will flush cout.

cin >> name;
cin >> age;
cout << name << age;
cin >> response;
We prompt the user again.

```cpp
Avery
20
Avery20|
```

```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```
We type something, it's transferred to the buffer, and read into the variable.

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

```
Avery
20
Avery20YES
```

```
name (string)  “Avery”
age (int)  20
response (string)  “YES”
```

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

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

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

```cpp
Avery Wang

name (string) cin >> name;

age (int)    cin >> age;

response (string) cout << name << age;

    cin >> response;
```
Remember `cin` reads up to a whitespace.

```cpp
Avery Wang

name (string) = "Avery"

age (int) = ???

response (string) = ???

```
Avery Wang

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

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

- **cin now tries to read an int.**
  - It skips past the initial whitespace.
It tries to read in an int, but fails.

```cpp
int main() {
    cout << "Enter your name: ";
    cin >> name;  // Read name
    cout << "Enter your age: ";
    cin >> age;   // Read age
    cout << "Enter your response: ";
    cin >> response;  // Read response
    cout << "Avery Wang\n";  // Output name
    cout << name << age;  // Output name and age
    cout << response;  // Output response
}
```
It tries to read in an int, but fails.

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

```
Avery Wang
```

```
name
(string)
```

```
“Avery”
```

```
response
(string)
```

```
???
```

```
age
(int)
```

```
???
```

It tries to read in an int, but fails.
The fail bit is turned on.

```cpp
Avery Wang

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

The fail bit is turned on.

Avery Wang

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

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

```cpp
Avery Wang
Avery -2736262

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

```
Avery Wang
Avery

name (string)
“Avery”

age (int)
???

response (string)
???
```
There are 3 reasons why >> with cin is a nightmare.

1. cin reads the entire line into the buffer but gives you 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.
Summary
External devices are complicated. Streams hide make them all seem like an array of character.

3.14

3 1 4 p i ...

<< output

<double in your program>

3.14

>> input
There is a unified interface for all streams.

- `<<` and `>>` for formatted input/output
- `tell/seek` to get/set the position

```cpp
oss.seekp(streamoff(1), stringstream::cur);
oss << "Boba";
```
Some output streams are buffered and require flushing.

cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
The >> operator for input streams find whitespace-separated tokens, which is annoying.

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

Avery Wang
Avery -2736262

```
name (string) "Avery"
age (int) ???
```
Use state bits to help with error-checking.

Avery Wang

Avery -2736262

name (string) “Avery”

age (int) ???

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

Implementing simpio and other Stanford libraries