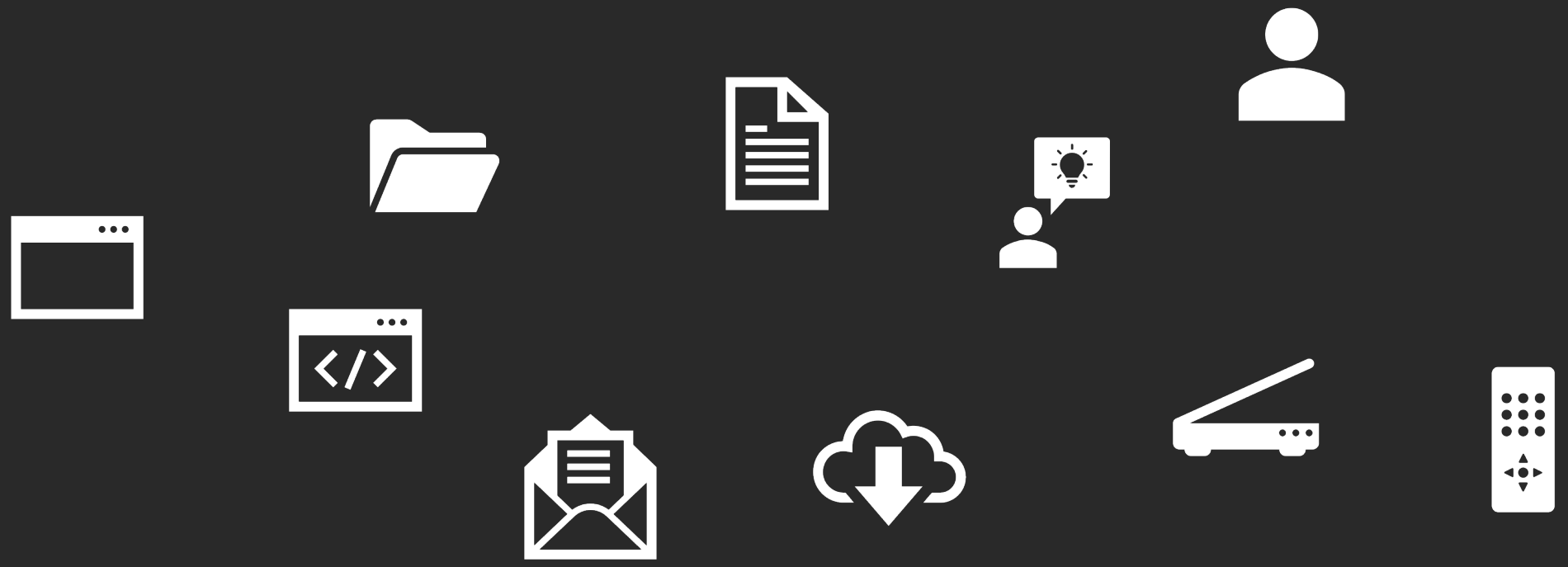
Streams I

CS 106L Fall 2019 – Avery Wang and Anna Zeng

Game Plan

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

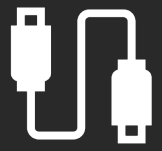
We often want our programs to interact with external devices.



Here are some common devices we will use.

console & keyboard

files



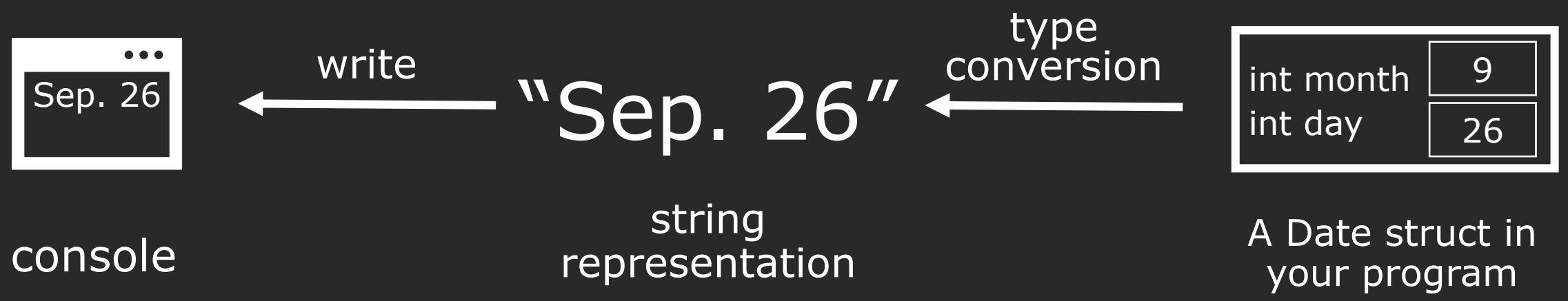
other programs (pipelines)

sockets (networking)

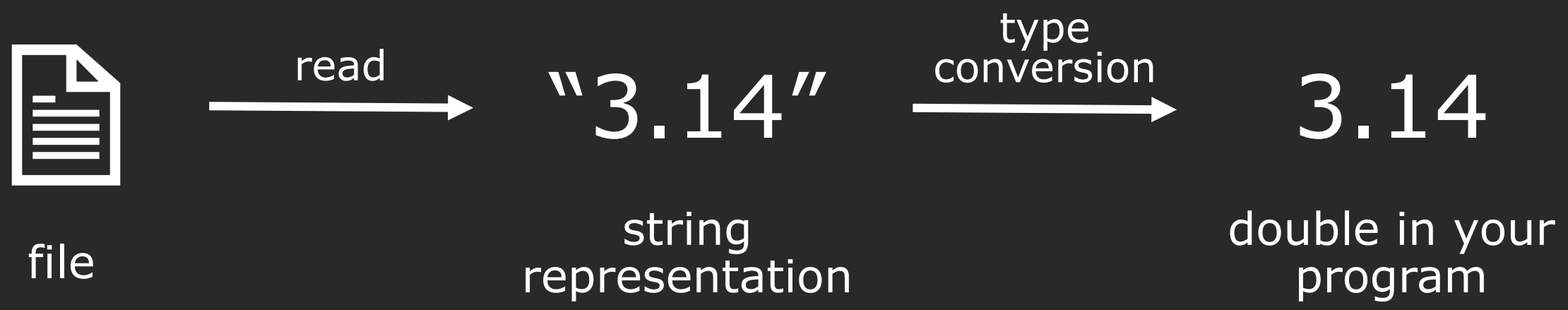
Take CS 110!

Take CS 144!

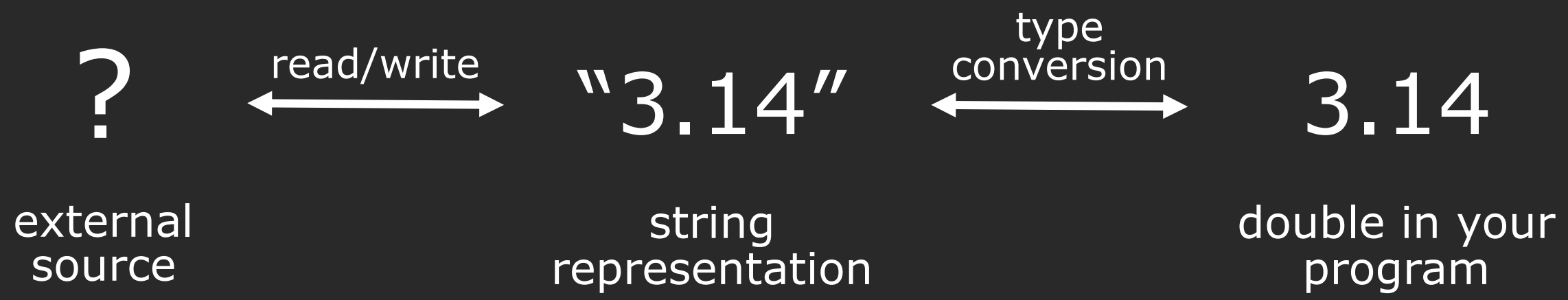
We might print a date to the console.



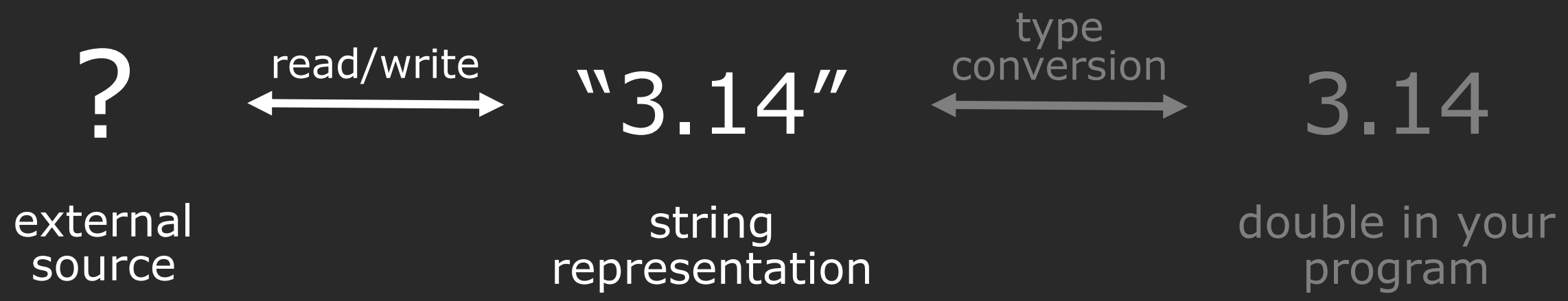
We might read a double from a file.



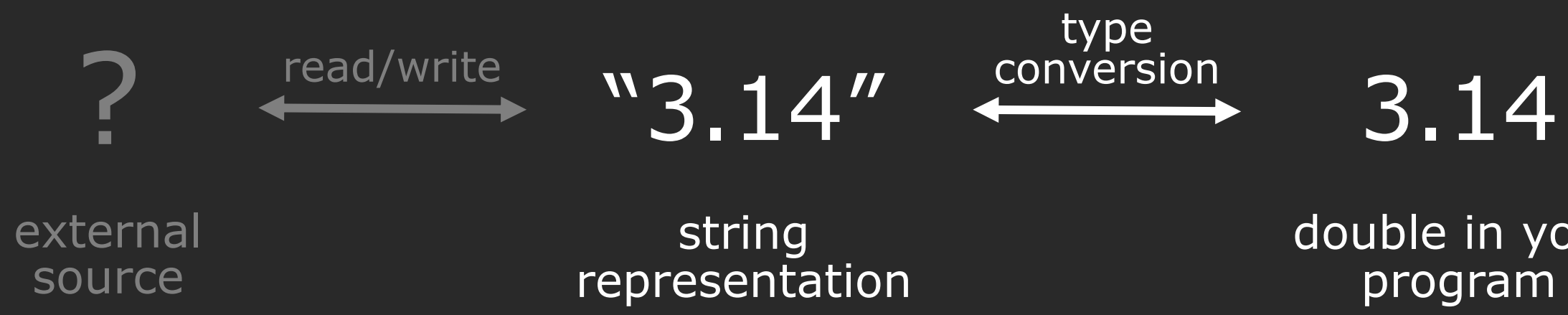
There are two main challenges.



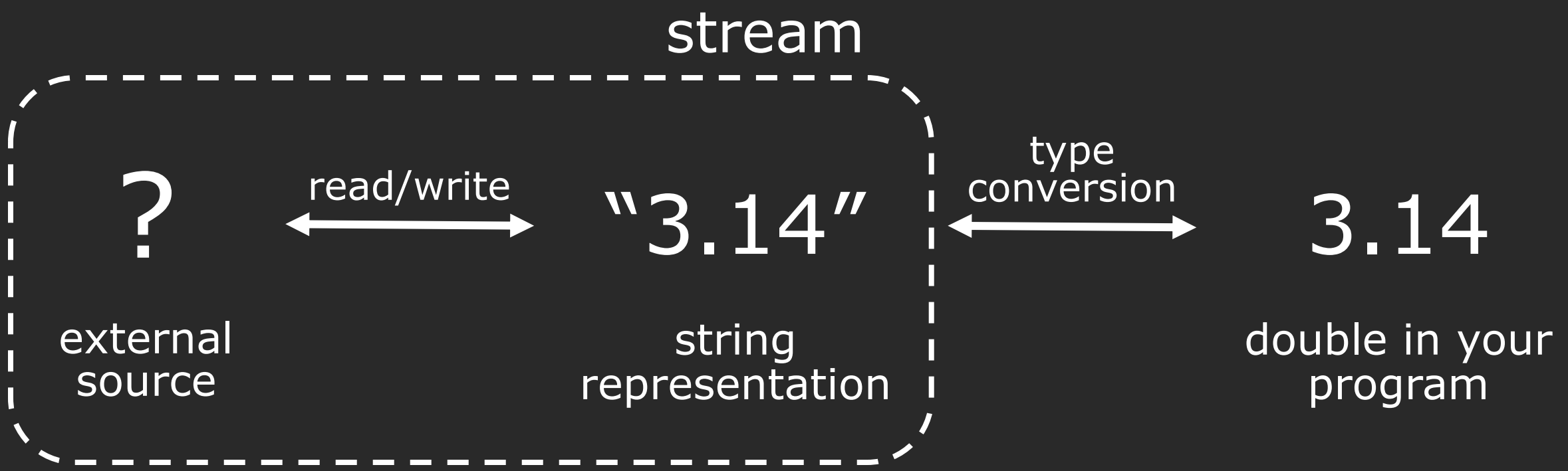
First: we need to retrieve/send data from the source in string form.



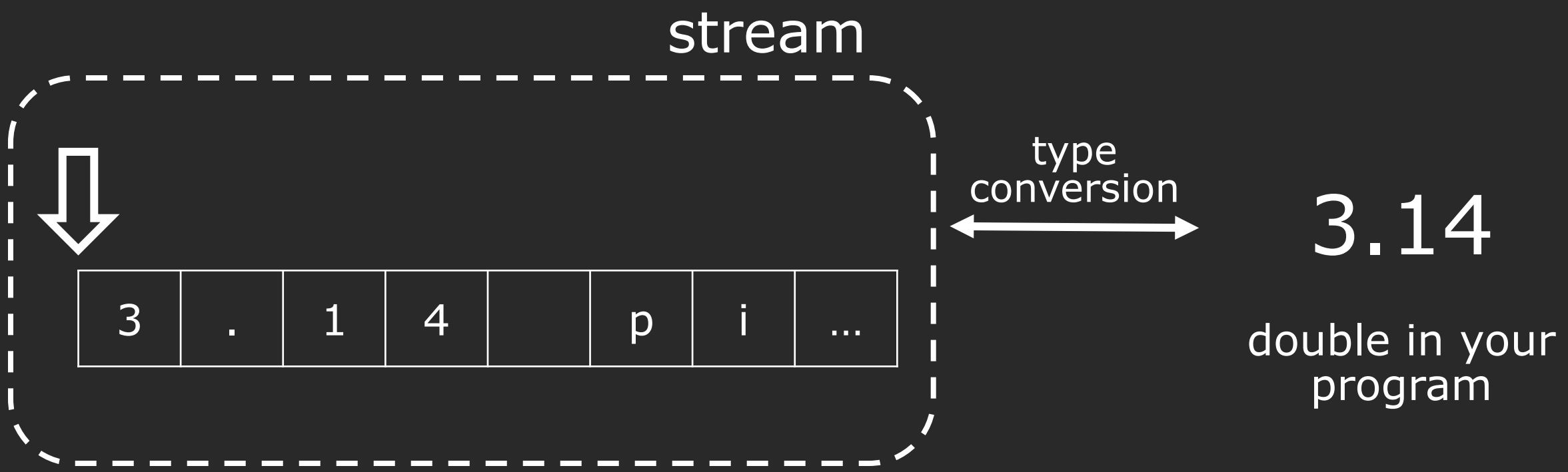
Second: we need to convert between data in our program and its string representation.



Streams provide a unified interface for interacting with external input.



You can imagine a stream to be a character buffer that automatically interacts with the external source.



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

<<

output

>>

input

3.14

double in your program

Don’t worry about how read/write to the source actually happens!

<<

output

>>

input

3.14

double in your program

Although…when read/writes happen will matter later when we discuss buffering.

<<

output

>>

input

3.14

double in your program

stringstream

A stringstream is not connected to any external source.

nothing

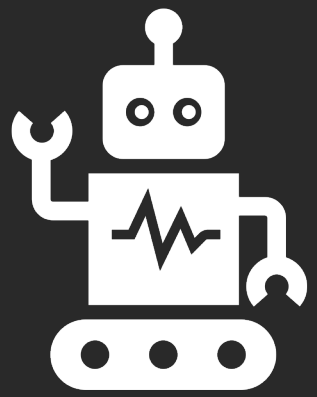
"3.14"

string representation

type conversion

3.14

double in your program



Example

creating, extracting, and inserting
from a stringstream


```
ostringstream oss("Ito En Green Tea “);
```

Construct oss with the string parameter as the initial string.


```
ostringstream oss("Ito En Green Tea “);
```

Construct oss with the string parameter as the initial string.


```
ostringstream oss("Ito En Green Tea “);
cout << oss.str() << endl; // Ite En Green Tea
```

The str method outputs the string in the entire buffer.


```
oss << 16.9 << " Ounce ";
```

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 ";
cout << oss.str() << endl; // 16.9 Ounce n Tea
```

The position started in the front, so we are overwriting the buffer!


```
oss << "(Pack of " << 12 << ")\n";
```

The buffer is as big as it (reasonably) needs to be. Don't worry about the details.


```
oss << "(Pack of " << 12 << ")\n";
```

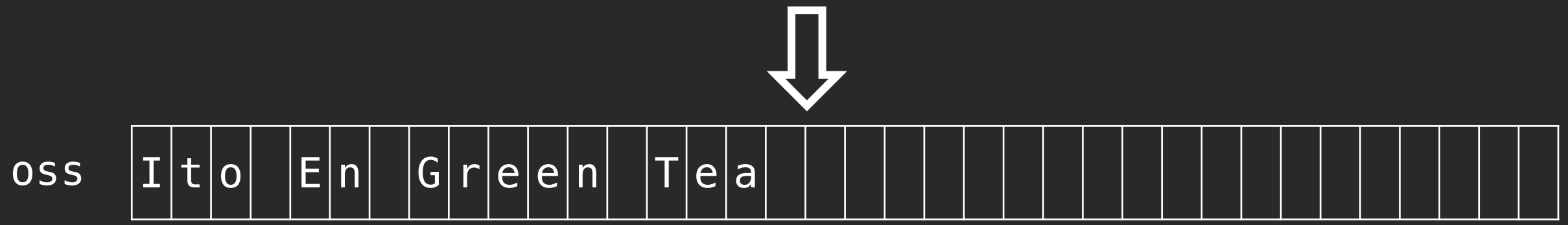
The buffer is as big as it (reasonably) needs to be. Don't worry about the details.


```
oss << "(Pack of " << 12 << ")\n";
cout << oss.str() << endl; // 16.9 Ounce (Pack of 12)\n
```

The buffer is as big as it (reasonably) needs to be. Don't worry about the details.

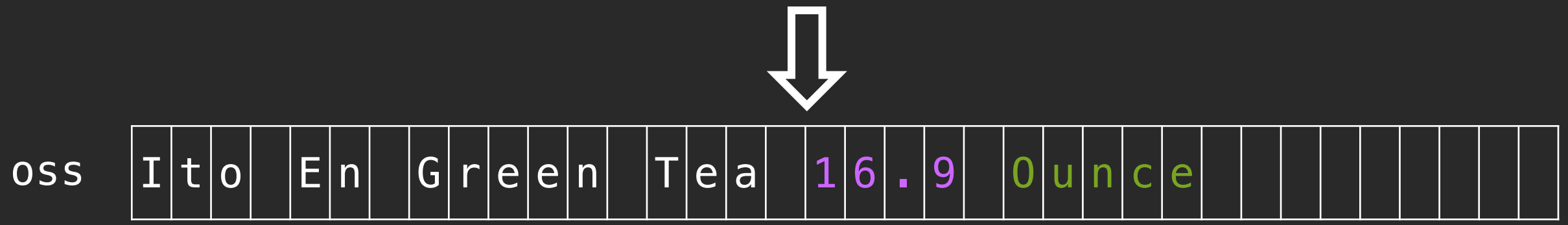

```
ostringstream oss("Ito En Green Tea “);
```

We intended to append to the initial string.




```
ostringstream oss("Ito En Green Tea “, stringstream::ate);
```

Now the position
starts at end.



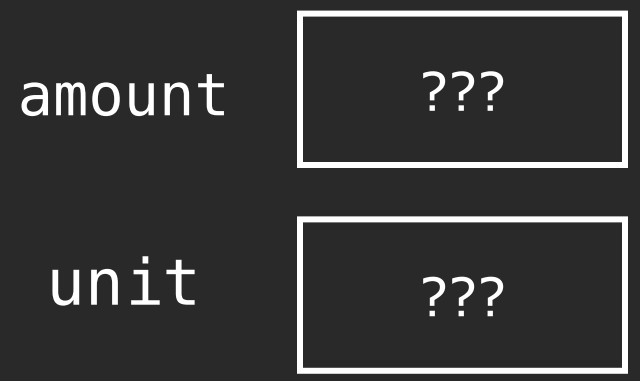

```
ostringstream oss("Ito En Green Tea “, stringstream::ate);
oss << 16.9 << “ Ounce ”;
```

Rest of the program works the same.


```
istringstream iss(oss.str()); // 16.9 Ounce (Pack of 12)
```

Let’s now create an input sstream using the same string.


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



Declare two variables.


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

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	???

input streams read up to the next whitespace.


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

amount	16.9
unit	"Ounce"

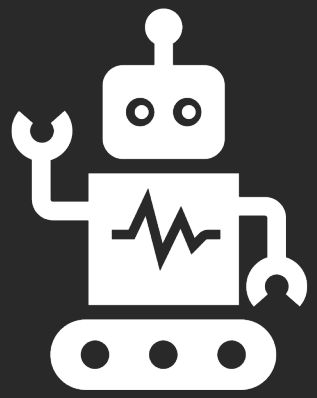
It also skips any leading whitespace.

iss	1	6	.	9		0	u	n	c	e		(	P	a	c	k		o	f		1	2	)	\n											

```
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


```
oss << 16.9;
```

One more time:
position at index 4


```
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
```

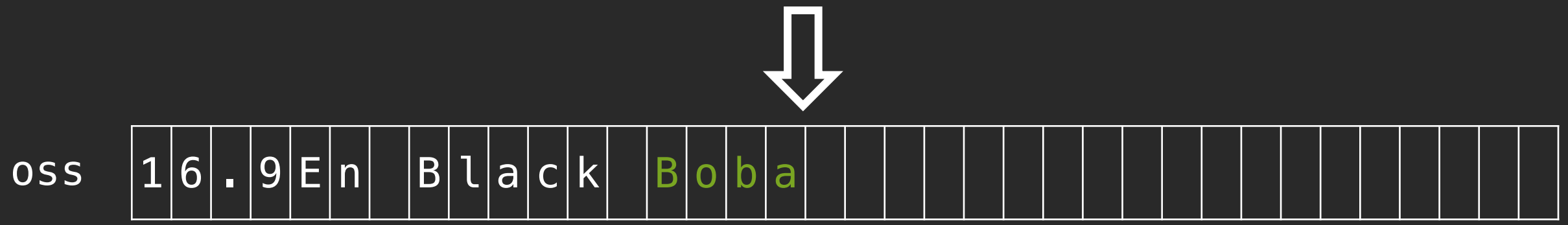
Calculate a new index, which is the current index plus an offset of 3.


```
oss << “Black”;
```

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.

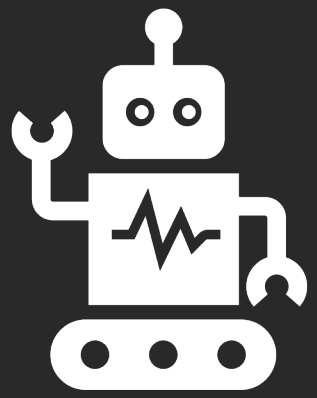
Read about the get/put and read/write functions which provide unformatted input/output!

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.

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

    int result;
    iss >> result;

    return result;
}
```

First attempt: no error-checking.

```
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.

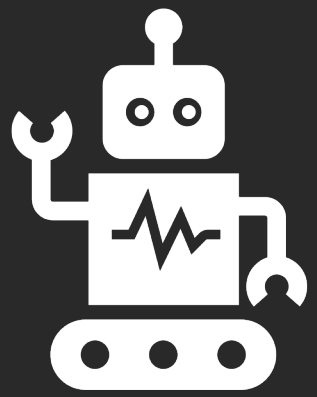
Important things about state bits.

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.

```
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.

```
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.

```
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;
}
```

We also need to ensure there's nothing left to read in the stream.

Third attempt: complete error-checking.

```
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.

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

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

The >> operator returns the stream which is converted to stream.fail().

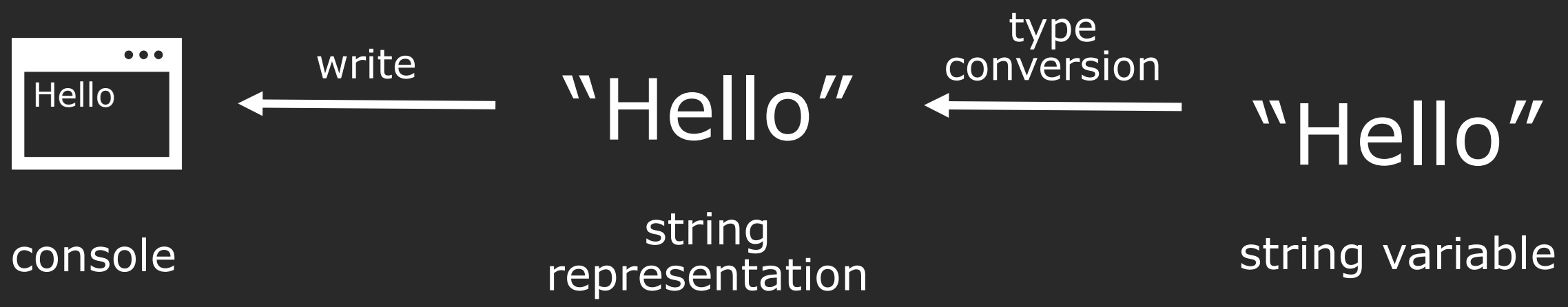
Third attempt: complete error-checking.

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

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

    return result;
}
```

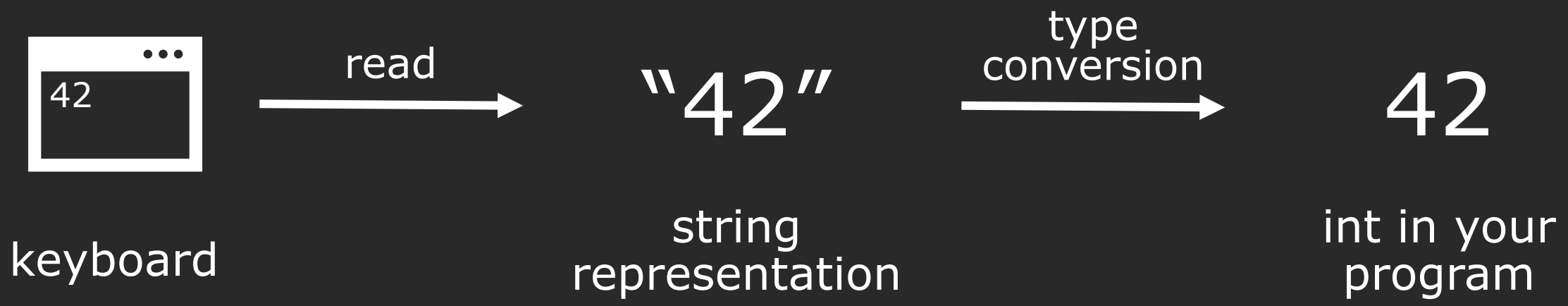
Notice the short circuiting!

cout and cin

Key difference: there is an external source.



Data is sent between the external source and the buffer.

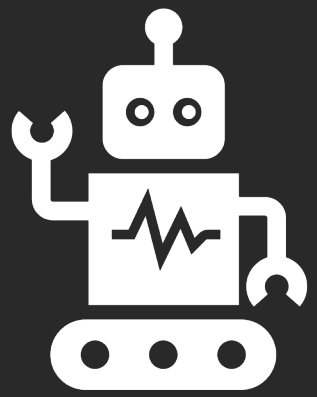


There are four standard iostreams.

`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.

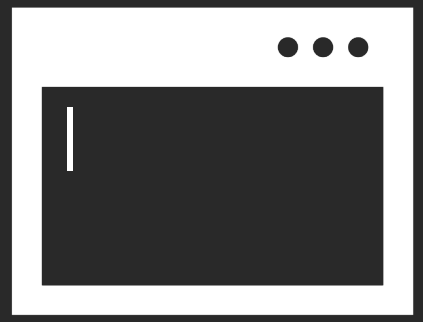
`cin`	Standard input stream
`cout`	Standard output stream (buffered)
`cerr`	Standard error stream (unbuffered)
`clog`	Standard error stream (buffered)



Example

output streams, buffering, and flushing

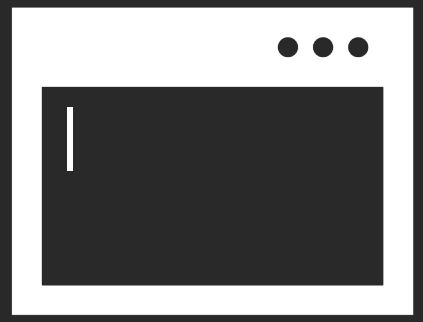
(buffered)



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

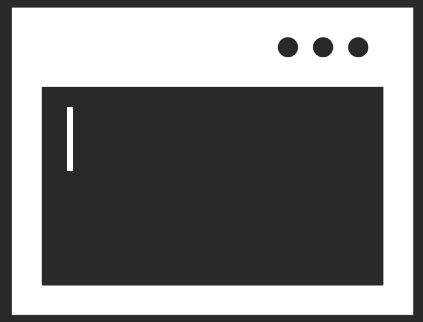
In the lecture code, I inserted slow function calls between each line.

(buffered)



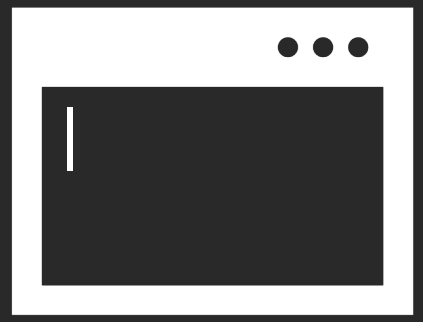
```
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
```

Added to buffer



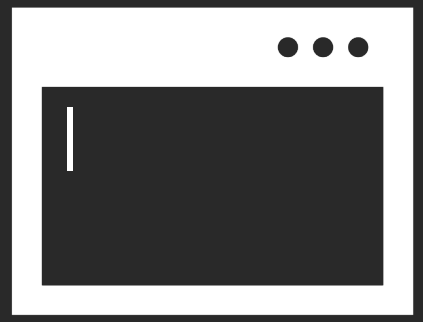
```
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
```

Notice that nothing shows up on the console yet!



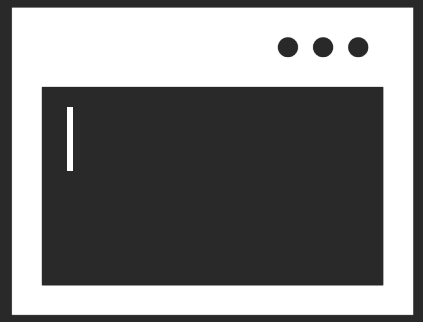
```
cout << “CS”;
cout << 106;
cout << flush;
cout << ‘L’;
cout << endl;
```

Same thing here.



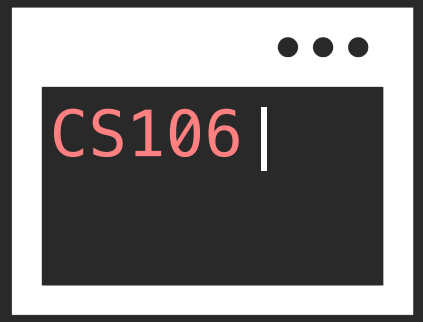
```
cout << “CS”;
cout << 106;
cout << flush;
cout << ‘L’;
cout << endl;
```

Same thing here.



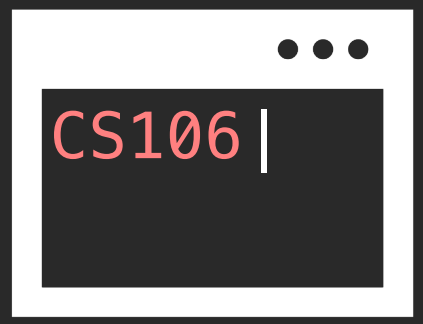
```
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
```

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



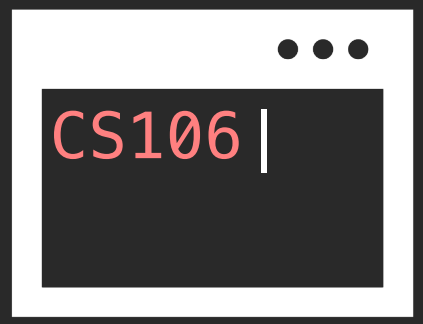

```
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
```

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



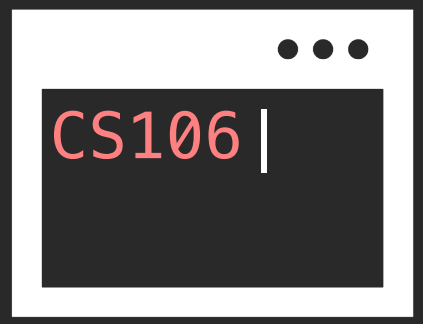

```
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
```

The stream is still buffered.



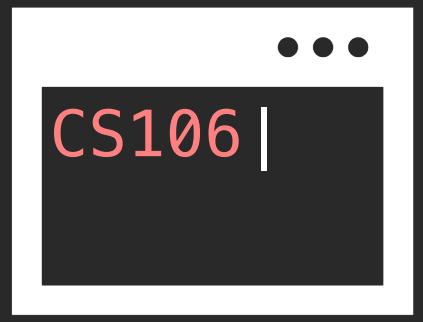

```
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
```

The stream is still buffered.




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

This is equivalent to adding '\n' and then flushing.



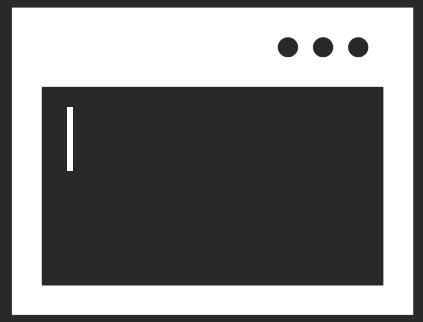

```
cout << "CS";
cout << 106;
cout << flush;
cout << 'L';
cout << endl;
```

This is equivalent to adding '\n' and then flushing.


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

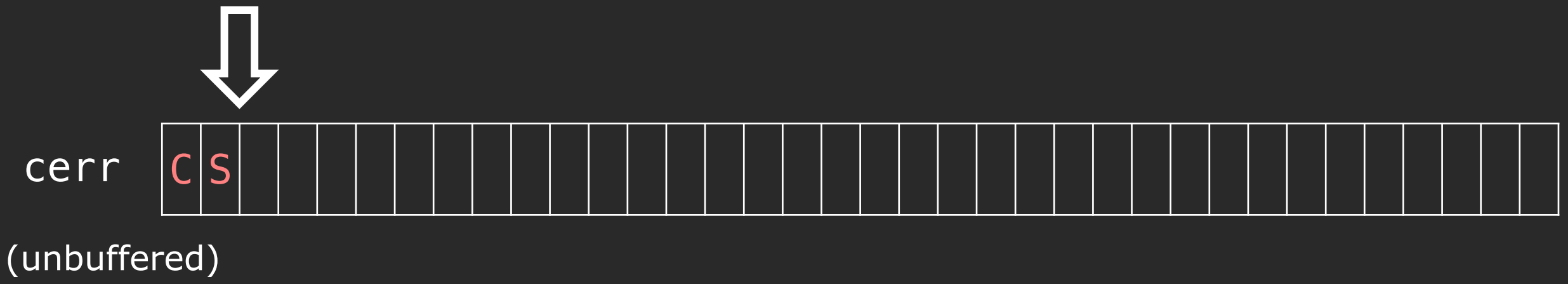
This is equivalent to adding '\n' and then flushing.

(unbuffered)



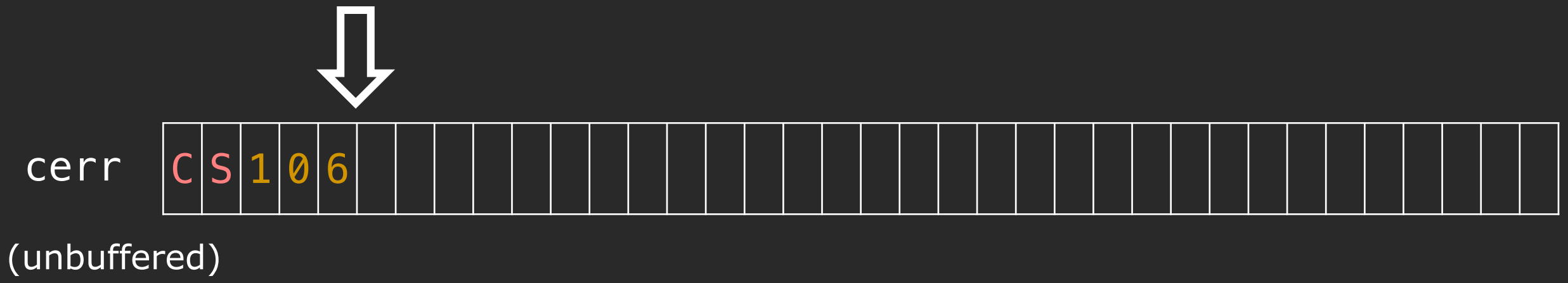
```
cerr << "CS";
cerr << 106;
cerr << flush;
cerr << 'L';
cerr << endl;
```

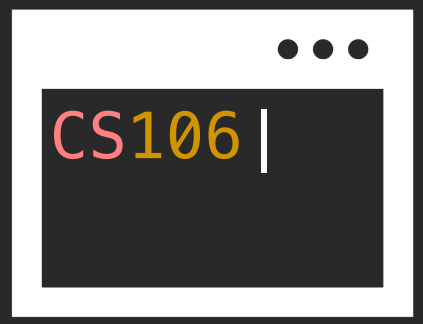
For unbuffered streams everything inserted shows up immediately.




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

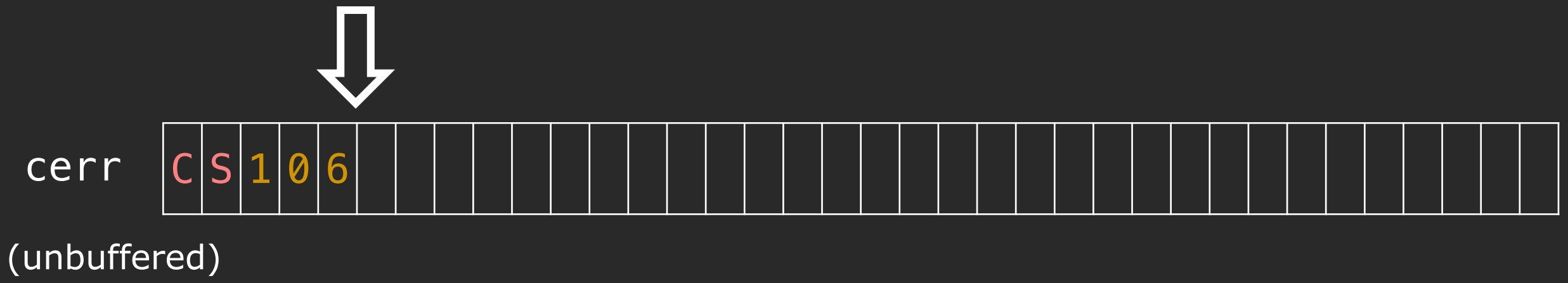
For unbuffered streams everything inserted shows up immediately.

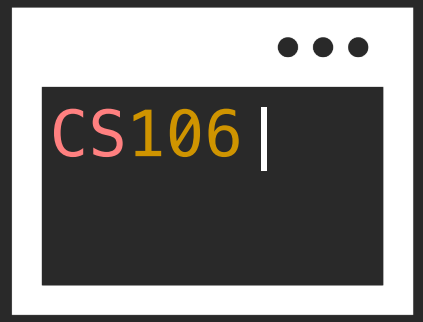





```
cerr << "CS";
cerr << 106;
cerr << flush;
cerr << 'L';
cerr << endl;
```

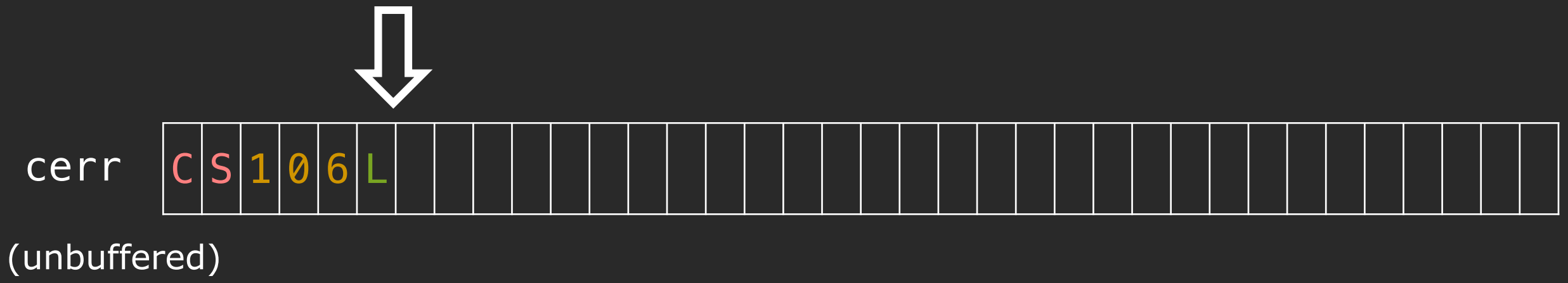
For unbuffered streams everything inserted shows up immediately.





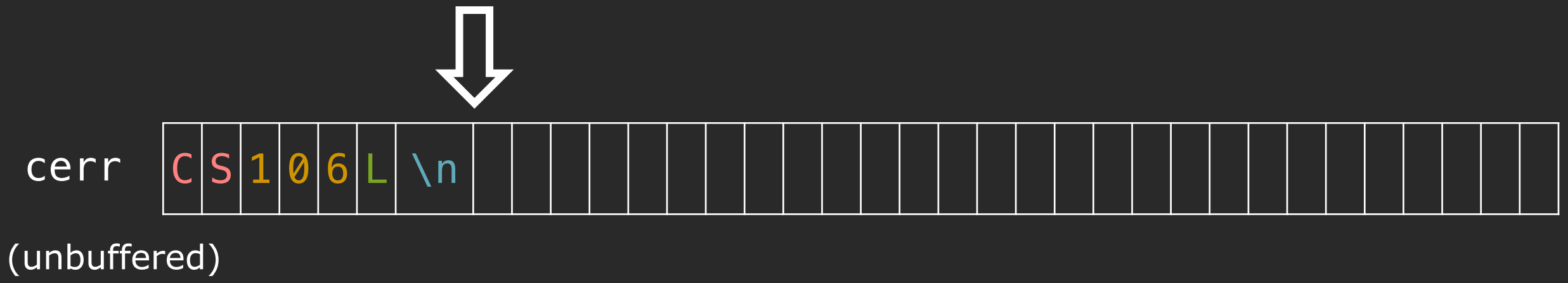

```
cerr << "CS";
cerr << 106;
cerr << flush;
cerr << 'L';
cerr << endl;
```

Flushing doesn't do anything.



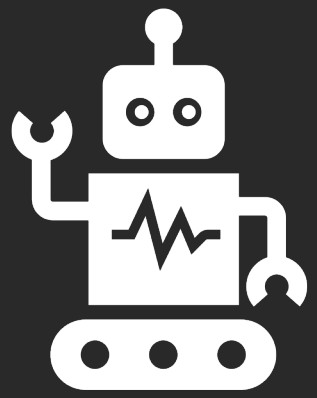

```
cerr << "CS";
cerr << 106;
cerr << flush;
cerr << 'L';
cerr << endl;
```

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.

cin

G F E B

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

name
(string)

???

response
(string)

???

age
(int)

???

The lecture code included cout statements.

cin

G F E B

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

name
(string)

???

response
(string)

???

age
(int)

???

The lecture code included cout statements.


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

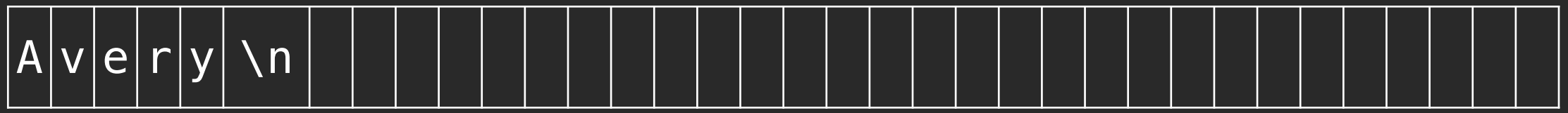
name (string) ???

response (string) ???

age (int) ???

Since there is nothing in the buffer, cin waits for the user to type something in.

cin



Avery

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

name
(string)

???

response
(string)

???

age
(int)

???

After typing in my name and pressing enter, cin transfers what I typed into the buffer.

Avery

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

name
(string)

"Avery"

response
(string)

???

age
(int)

???

Then we read from the buffer into the variable name, just like a stringstream.


```
Avery
|
```

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

name (string): "Avery"

response (string): ???

age (int): ???

cin skips whitespace, sees no more input, and prompts the user again.


```
Avery
20
```

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

name (string): "Avery"

response (string): ???

age (int): ???

Everything I type is transferred to the buffer.


```
Avery
20
```

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

name (string): "Avery"

response (string): ???

age (int): 20

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


```
Avery
20
```

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

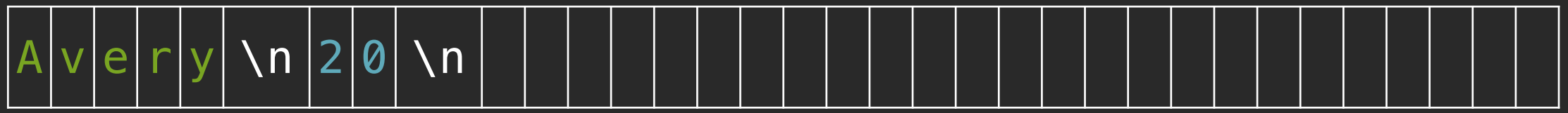
name (string): "Avery"

response (string): ???

age (int): 20

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

cin




```
Avery
20
```

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

name
(string)

"Avery"

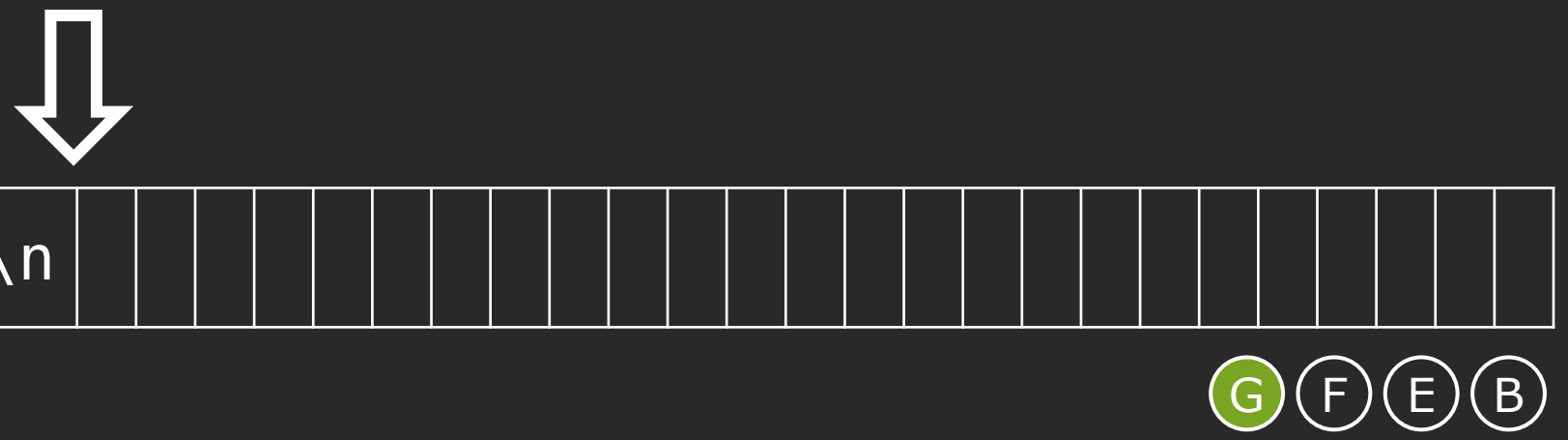
response
(string)

???

age
(int)

20

We now print the variables
(don't forget cout is buffered!)




```
Avery
20
Avery20
```

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

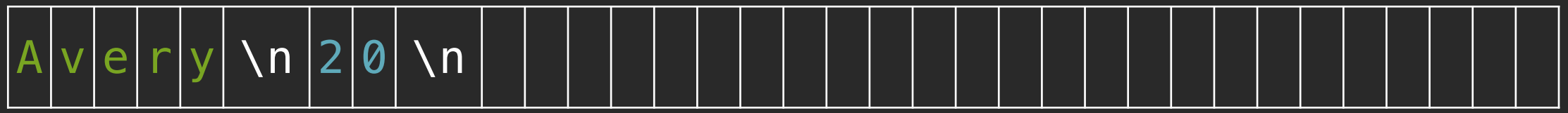
name (string): "Avery"

response (string): ???

age (int): 20

But attempting reading again will flush cout.

cin




```
Avery
20
Avery20|
```

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

name
(string)

"Avery"

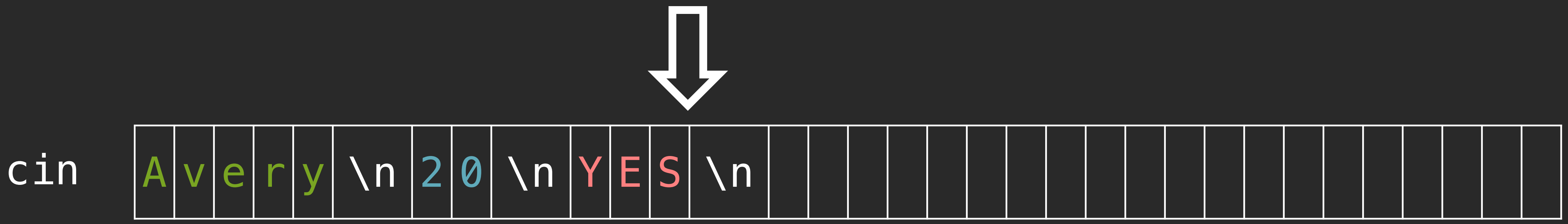
response
(string)

???

age
(int)

20

We prompt the user again.



cin

A	v	e	r	y	\n	2	0	\n	Y	E	S	\n																	

G F E B

```
Avery
20
Avery20YES
```

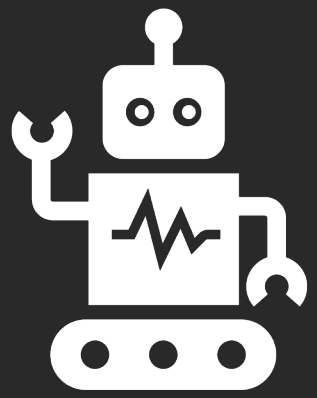
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): “Avery”

response (string): “YES”

age (int): 20

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



Example

when input streams go wrong

cin

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

name
(string)

???

response
(string)

???

age
(int)

???

Let's try something innocuous.
I type in my full name.

G F E B

Avery Wang

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

name (string): ???

response (string): ???

age (int): ???

After typing in my name and pressing enter, cin transfers what I typed into the buffer.

Avery Wang

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

name (string): "Avery"

response (string): ???

age (int): ???

Remember cin reads up to a whitespace.

Avery Wang

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

name (string): "Avery"

response (string): ???

age (int): ???

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

Avery Wang

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

name (string): "Avery"

response (string): ???

age (int): ???

It tries to read in an int, but fails.

Avery Wang

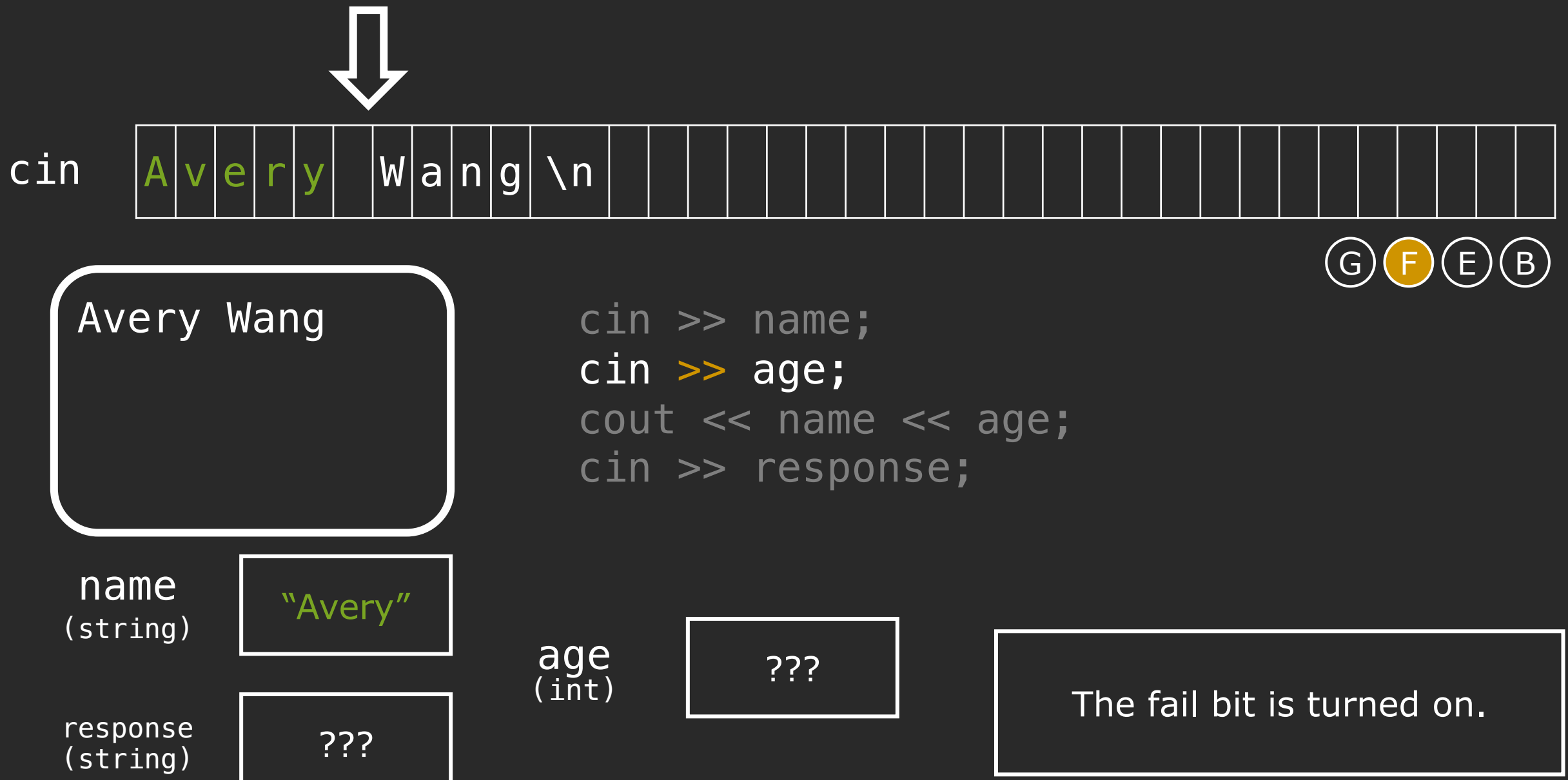
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): "Avery"

response (string): ???

age (int): ???

It tries to read in an int, but fails.


cin
A v e r y W a n g \n
G F E B
Avery Wang
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
name
(string)
"Avery"
response
(string)
???
age
(int)
???
The fail bit is turned on.


```
Avery Wang
Avery -2736262
```

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

name (string): "Avery"

response (string): ???

age (int): ???

cout now prints the name and age (which is uninitialized!)


```
Avery Wang
Avery -2736262
```

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

name (string): "Avery"

response (string): ???

age (int): ???

Worst part, since the fail bit is on, all future cin operations fail.

There are 3 reason 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.

<<

output

>>

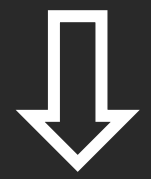
input

3.14

double in your program

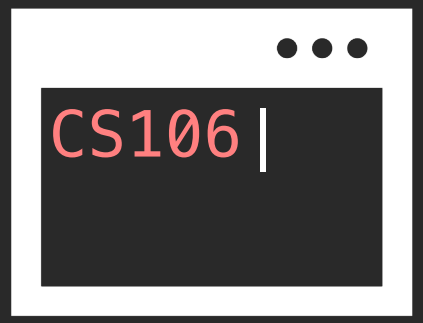
There is a unified interface for all streams.

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



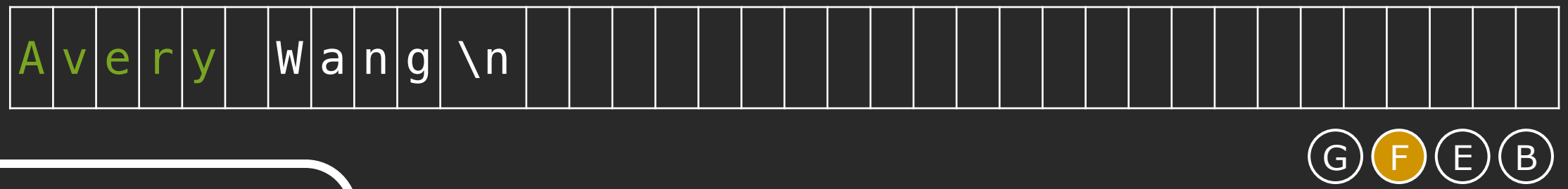

```
oss.seekp(streamoff(1), stringstream::cur);
oss << “Boba”;
```

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.




```
Avery Wang
Avery -2736262
```

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

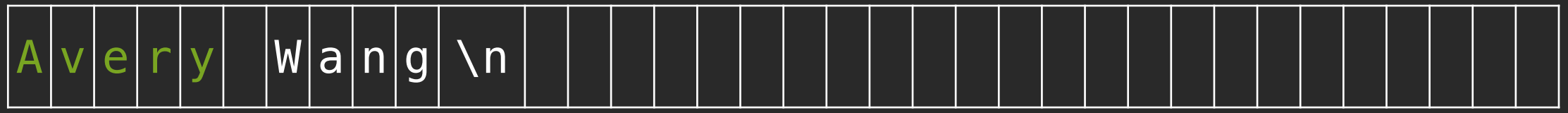
name
(string)

"Avery"

age
(int)

???

Use state bits to help with error-checking.




```
Avery Wang
Avery -2736262
```

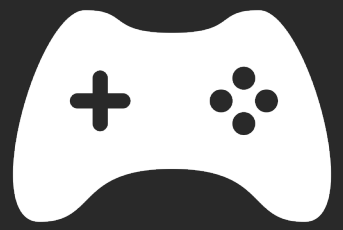
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name
(string)

"Avery"

age
(int)

???



Next time

Implementing simpio and other Stanford libraries