CS106L Lecture 5: Streams 

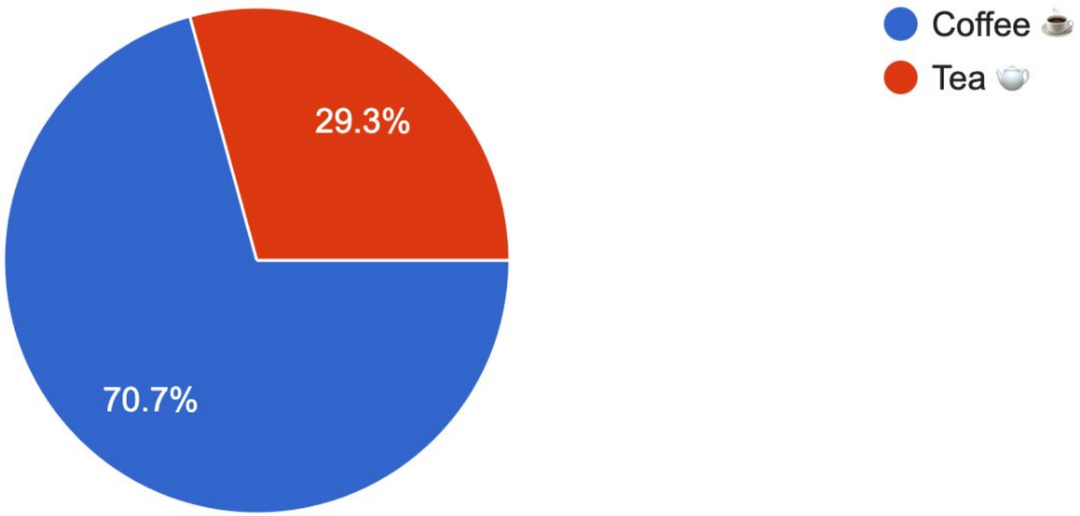
Fabio Ibanez, Jacob Roberts-Baca

Attendance 🪄

Interesting Stats 🪄

Coffee or Tea? (There is one right answer)

41 responses

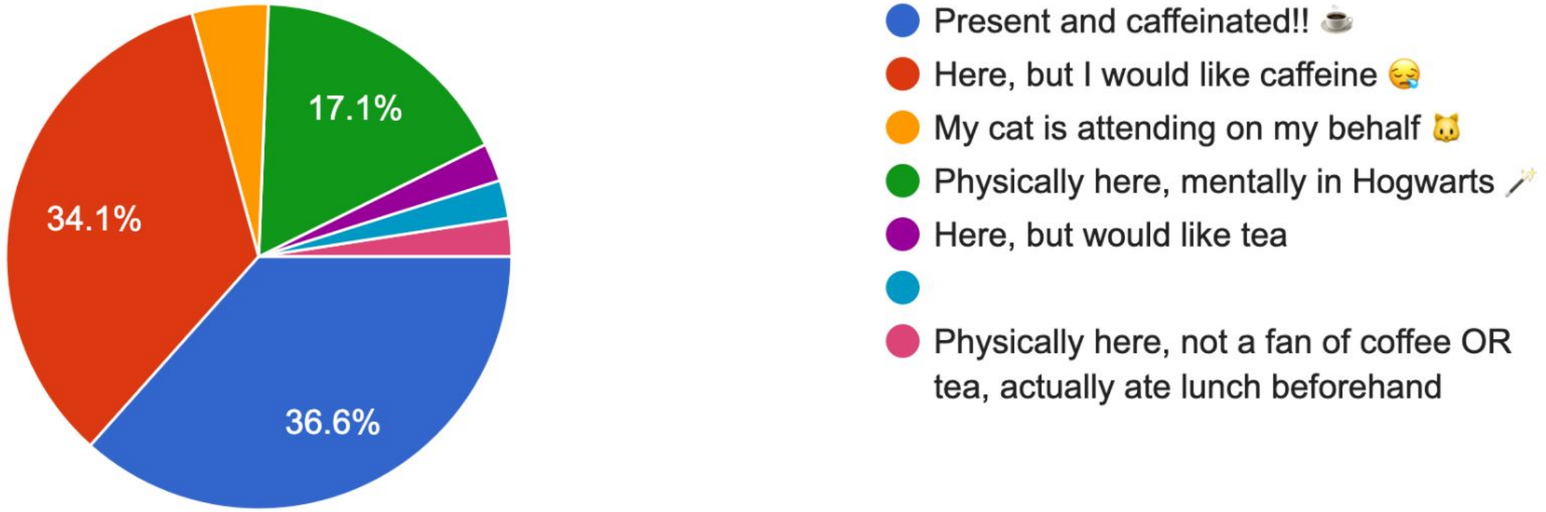


Personally

Interesting Stats 🪄

Which describes your current status?

41 responses



For the people in Hogwarts (or anyone)

This is a friendly reminder to let us know how to make this class better for you by submitting feedback using our anonymous feedback form here. ***We're interested in why your cat is attending on your behalf, or why you're in Hogwarts!***

I've even make a QR code for your convenience 🤠 (the slides are up on the website):

Plan

1. Quick recap
2. What are streams??!!
3. `stringstreams`
4. `cout` and `cin`
5. Output streams
6. Input streams

A quick recap

1. **Uniform Initialization** 🦄
   a. A *ubiquitous and safe* way of initializing things using {}

A quick recap

1. **Uniform Initialization** 🦄
   a. A *ubiquitous and safe* way of initializing things using {}
2. **References** 🦄
   a. A way of giving variables ***aliases*** and having multiple variables all refer the the **same memory.**

Plan

1. Quick recap
2. **What are streams??!!**
3. `stringstreams`
4. `cout` and `cin`
5. Output streams
6. Input streams

Why streams?

"Designing and implementing a general **input/output** facility for a programming language is notoriously difficult"

- *Bjarne Stroustrup*

Streams

"~~Designing and implementing~~ a general **input/output** facility for ~~a programming language is notoriously difficult~~ C++"

- a stream :)

Streams

a general **input/output** facility for C++

Streams

~~a general **input/output** facility for C++~~

a general **input/output(IO)** abstraction for C++

Abstractions

Abstractions provide a consistent ***interface***, and in the case of streams the interface is for reading and writing data!

A familiar stream!

```
std::cout << “Hello, World” << std::endl;
```

A familiar stream!

```
std::cout << “Hello, World” << std::endl;
```

This is a stream

A familiar stream!

```
std::cout << “Hello, World” << std::endl;
```

This is a stream

The **`std::cout`** stream is an instance of **`std::ostream`** which represents the standard output stream!

std::cout

```
std::cout << “Hello, World” << std::endl;
```

```
std::cout
```

“Hello, World”

std::cout

```
std::cout << “Hello, World” << std::endl;
```

```
std::cout
```

“Hello, World”

But how do we go from external source to program?

An Input Stream

How do you read a `double` from your console?

`std::cin` is the console input stream!

The `std::cin` stream is an instance of **`std::istream`** which represents the standard input stream!

```
void verifyPi()
{
  double pi;
  std::cin >> pi;
  /// verify the value of pi!
  std::cout << pi / 2 << '\n';
}
```

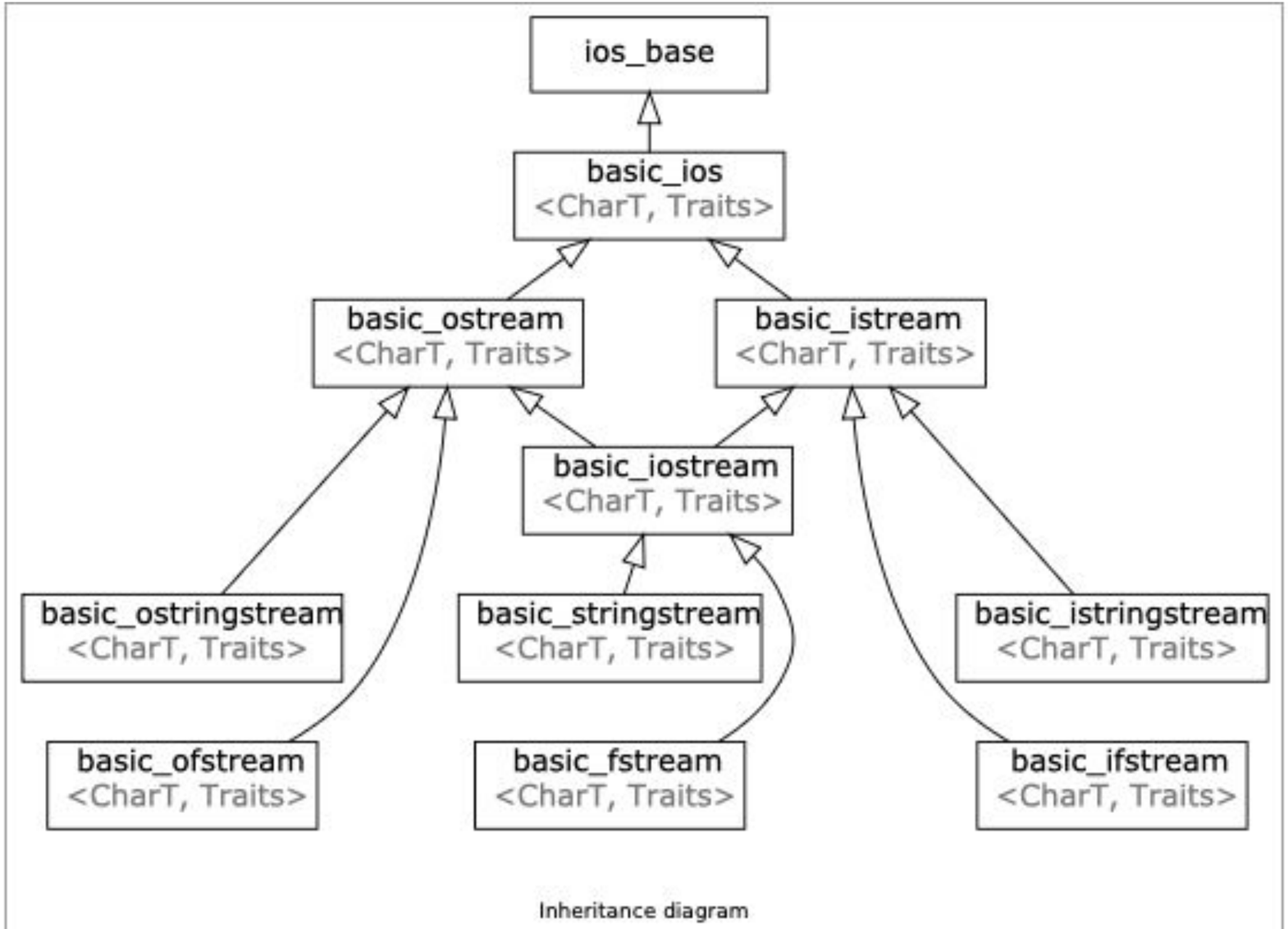
std::cin

```
int main()
{
  double pi;
  std::cin >> pi;
  /// verify the value of pi!
  std::cout << pi / 2 << '\n';

  return 0;
}
```

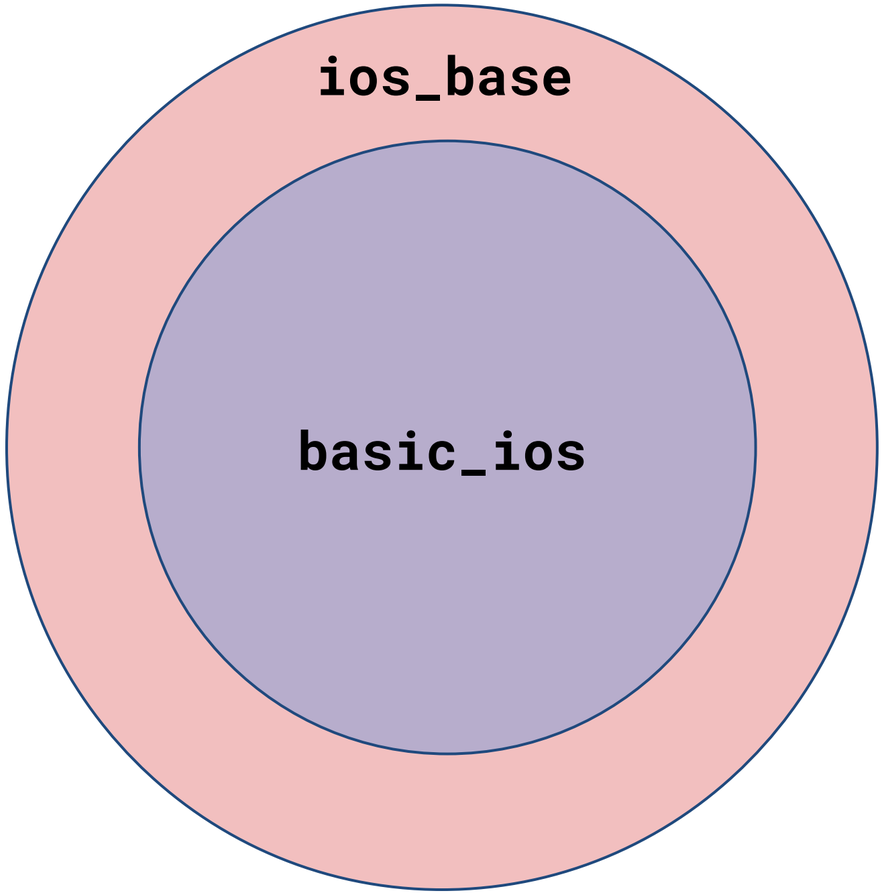
Console

"1.57"

What streams actually are

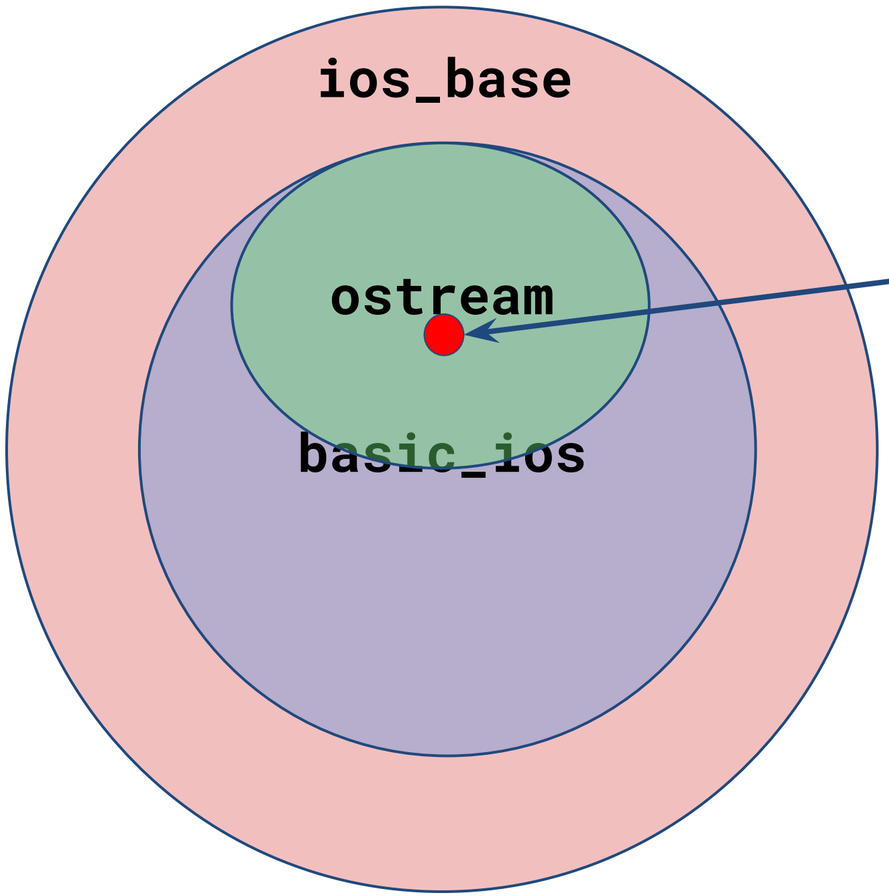


streams and types



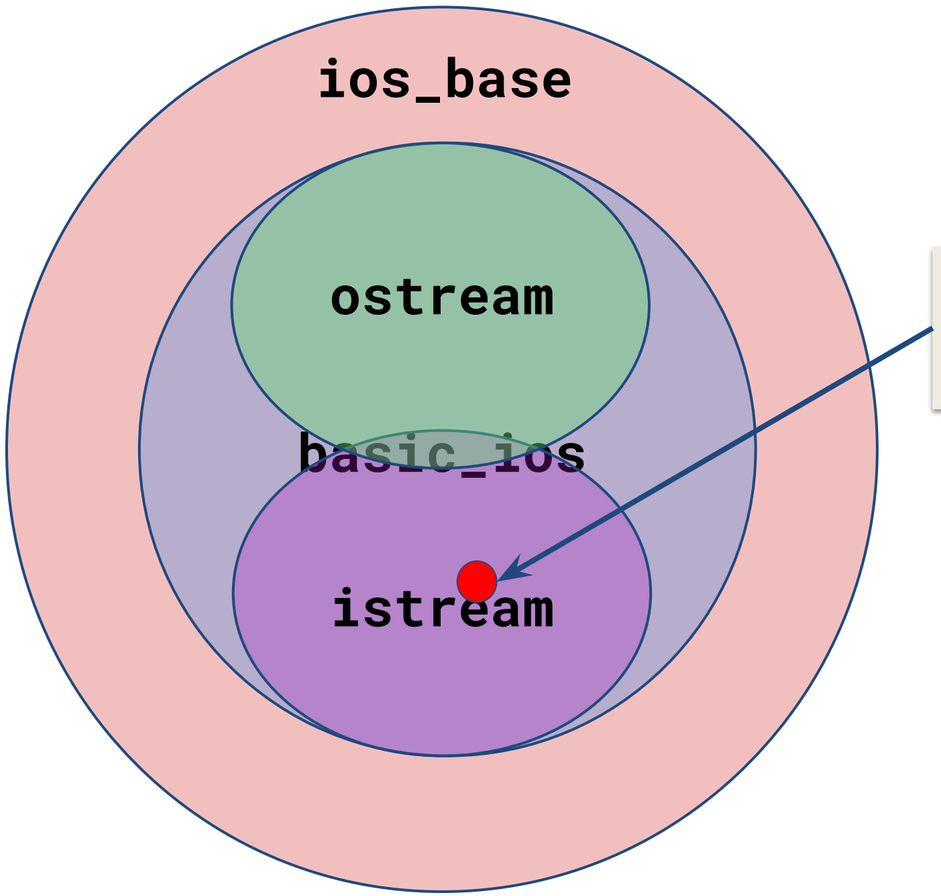
Each of these types are associated with some functionality – more on this later

streams and types



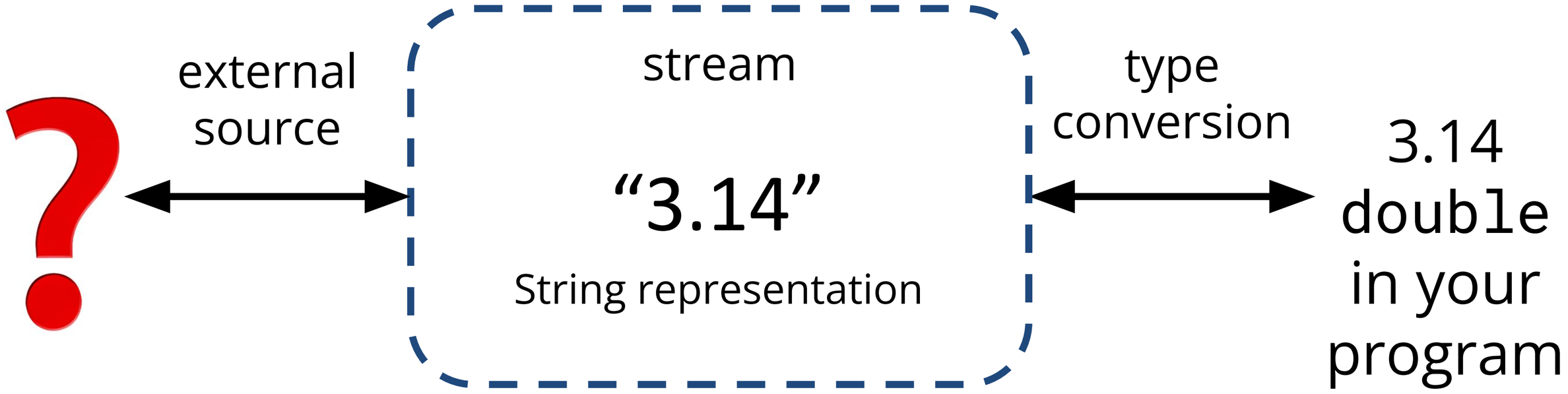
`std::cout` has the property of the **`std::basic_ostream`** type

streams and types

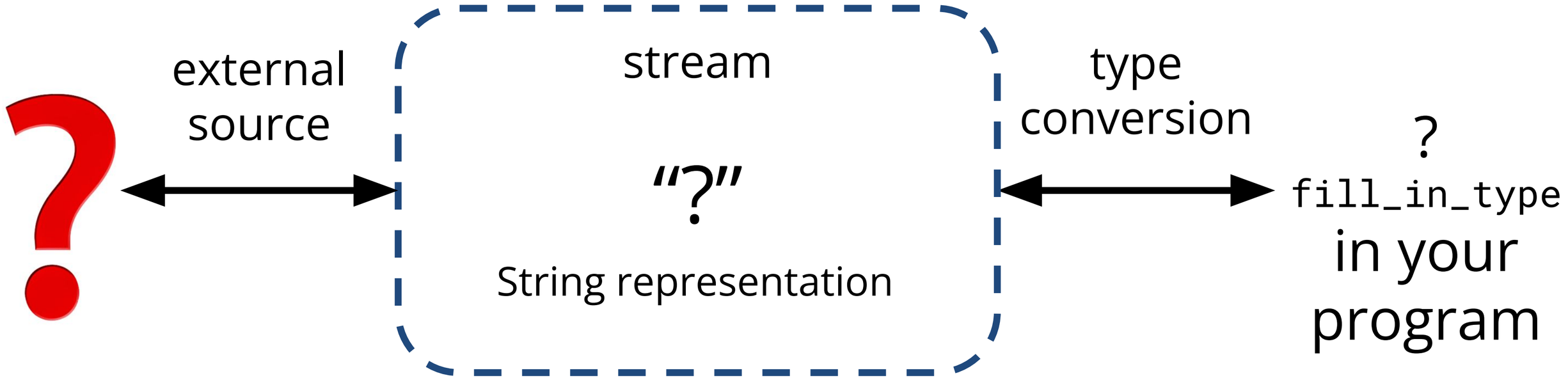


`std::cin` has the property of the **`std::istream`** type

Generalizing the Stream



Implementation vs Abstraction



Why is this even useful?

Streams allow for a **universal** way of **dealing with external data**

What streams actually are

Classifying different types of streams

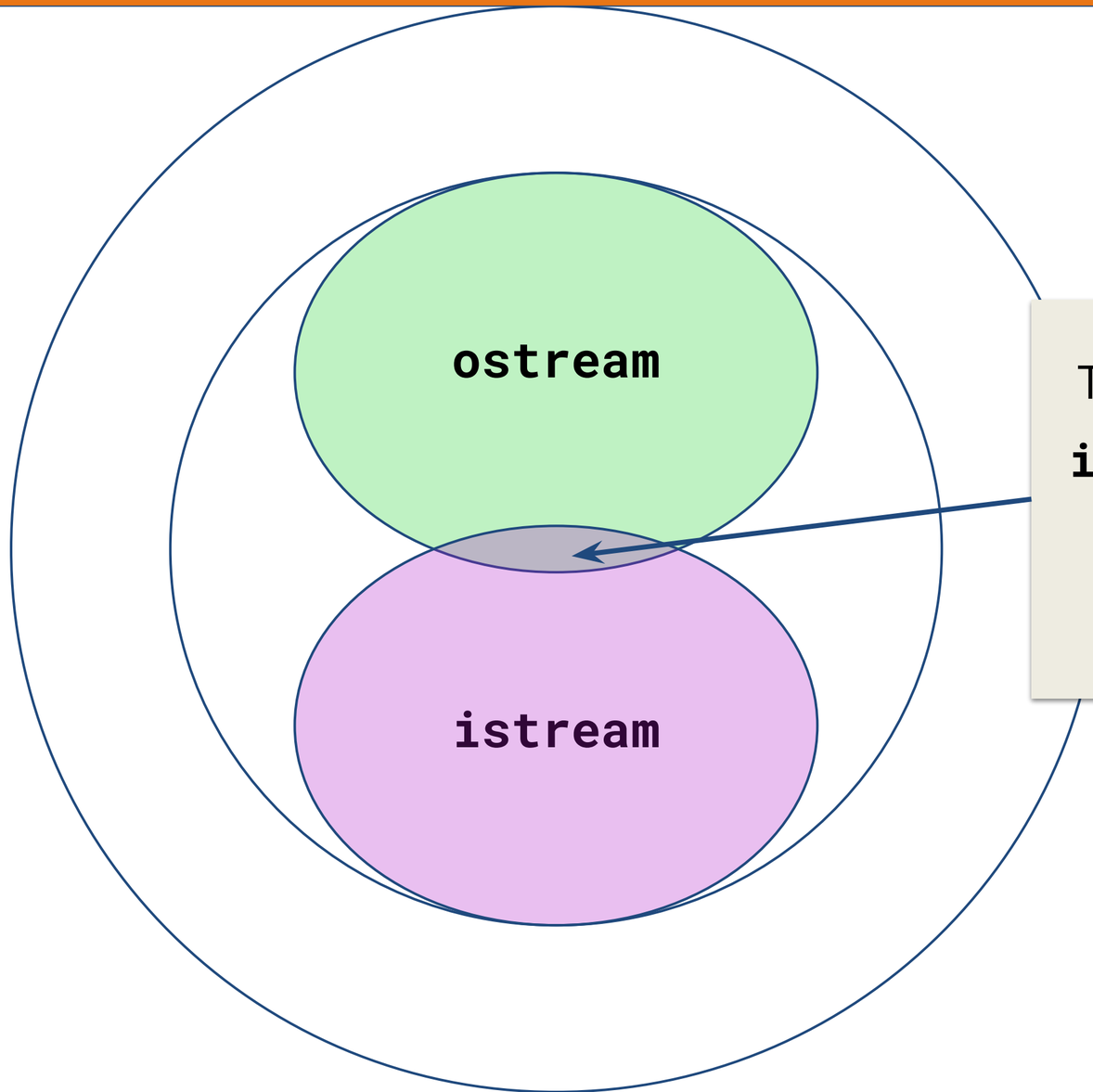
Input streams (I)

- a way to read data from a source
  - Are inherited from **`std::istream`**
  - ex. reading in something from the console (**`std::cin`**)
  - primary operator: **>>** (called the extraction operator)

Output streams (O)

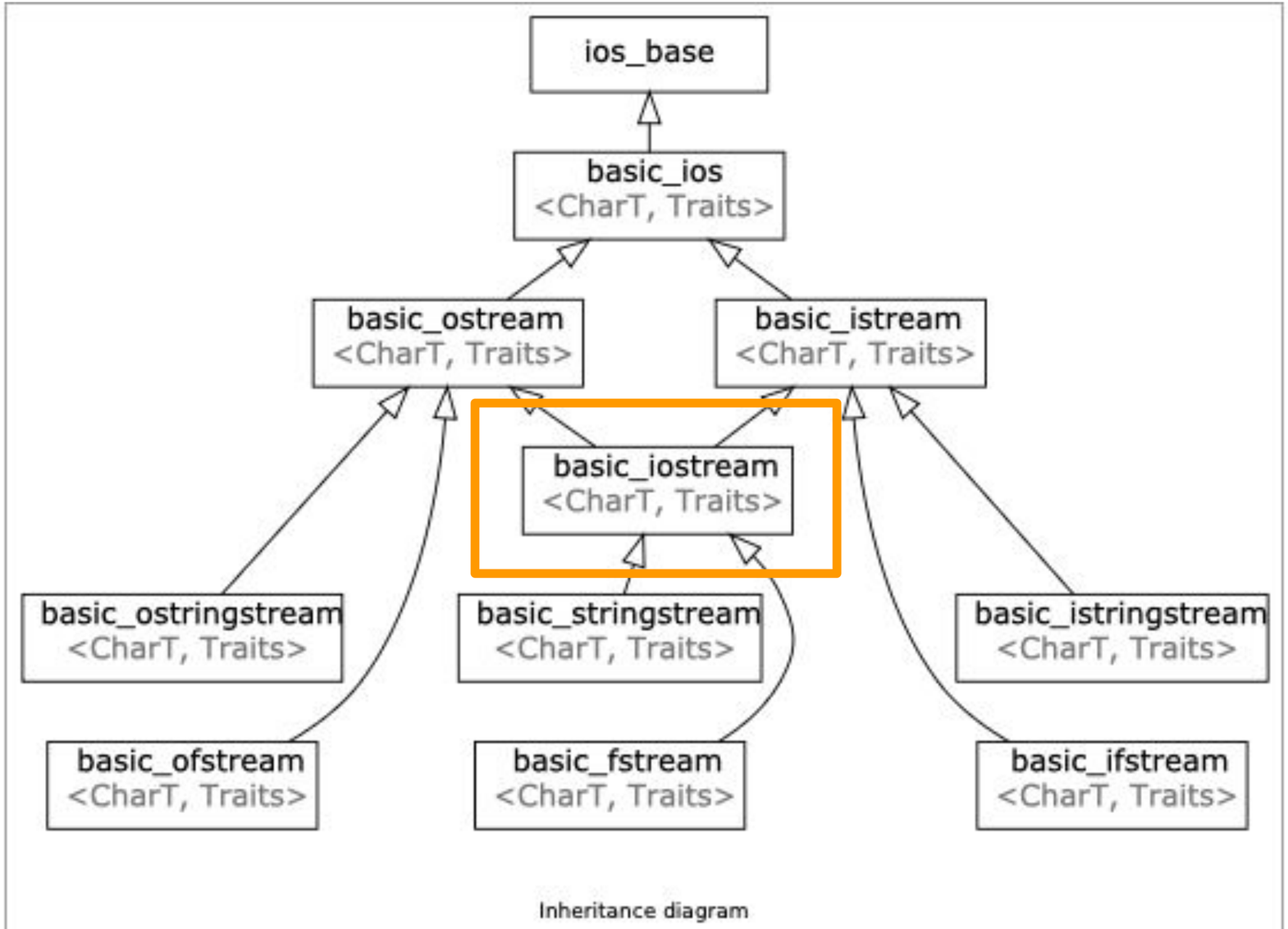
- a way to write data to a destination
  - Are inherited from **`std::ostream`**
  - ex. writing out something to the console (**`std::cout`**)
  - primary operator: **<<** (called the insertion operator)

streams and types

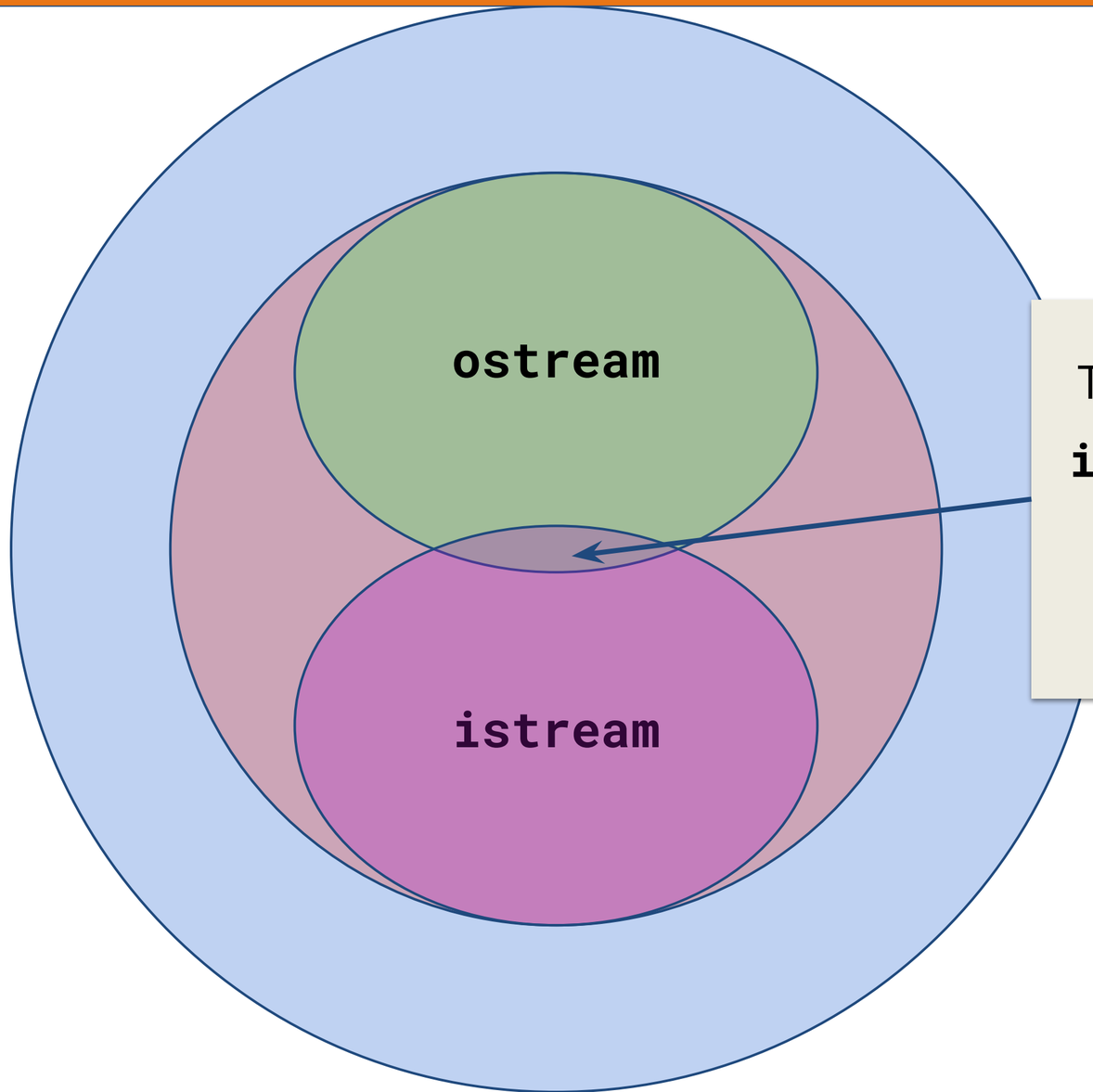


This intersection is known as **`iostream`** which takes has all of the characteristics of **`ostream`** *and* **`istream`**!

What streams actually are

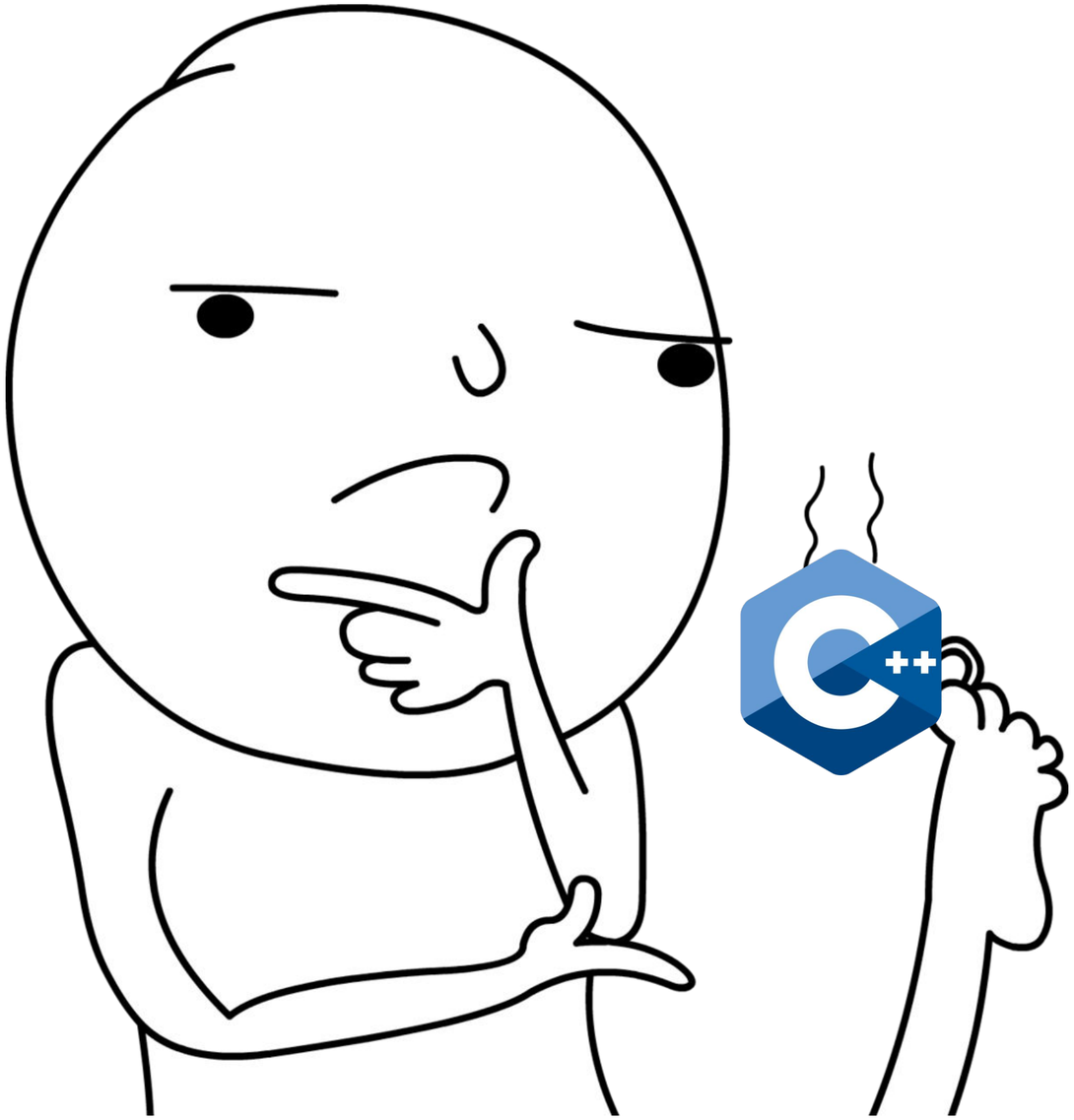


streams and types



This intersection is known as **`iostream`** which takes has all of the characteristics of **`ostream`** *and* **`istream`**!

What questions do we have?



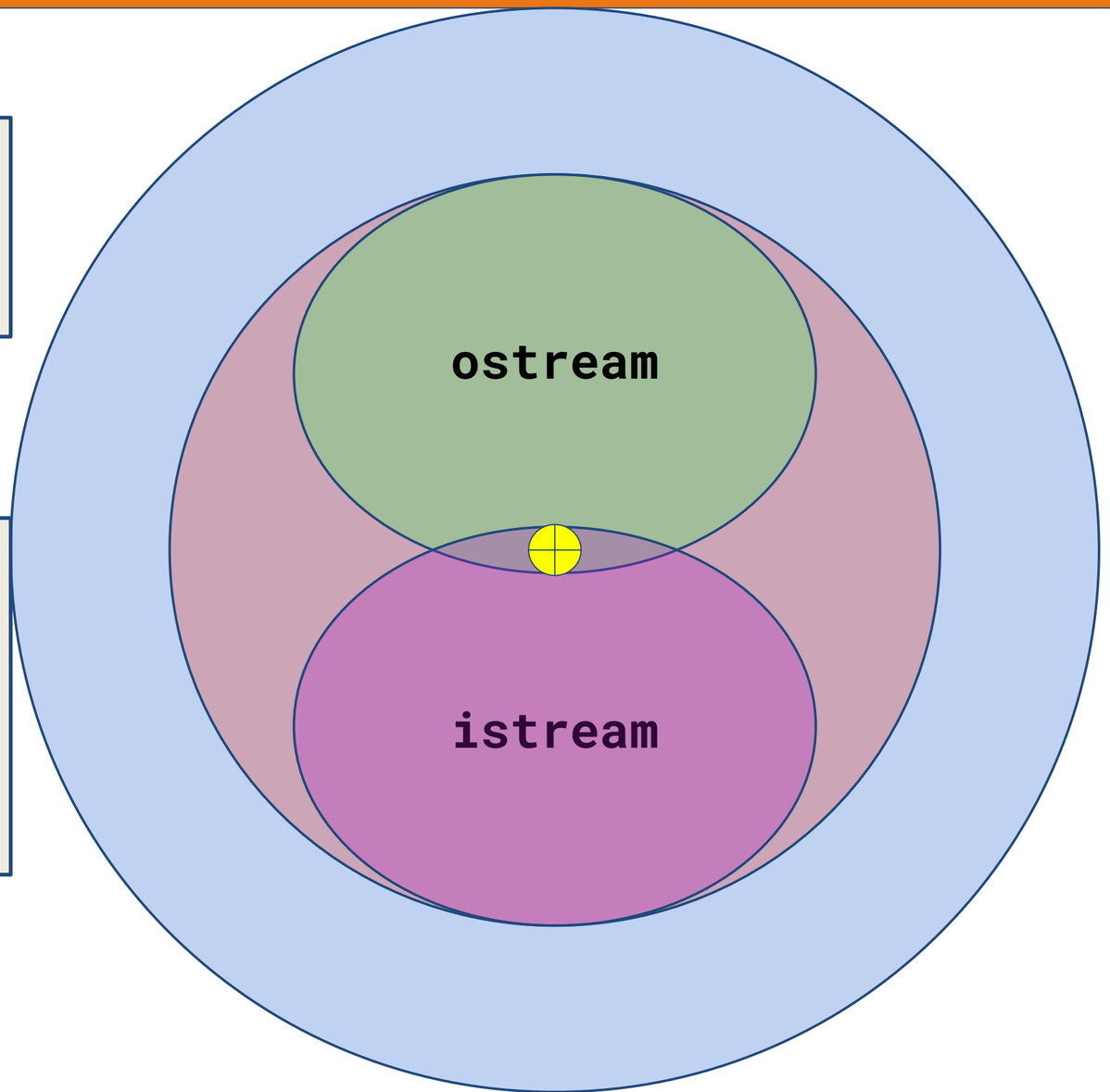
Plan

1. Quick recap
2. What are streams??!!
3. **`stringstreams`**!
4. `cout` and `cin`
5. Output streams
6. Input streams

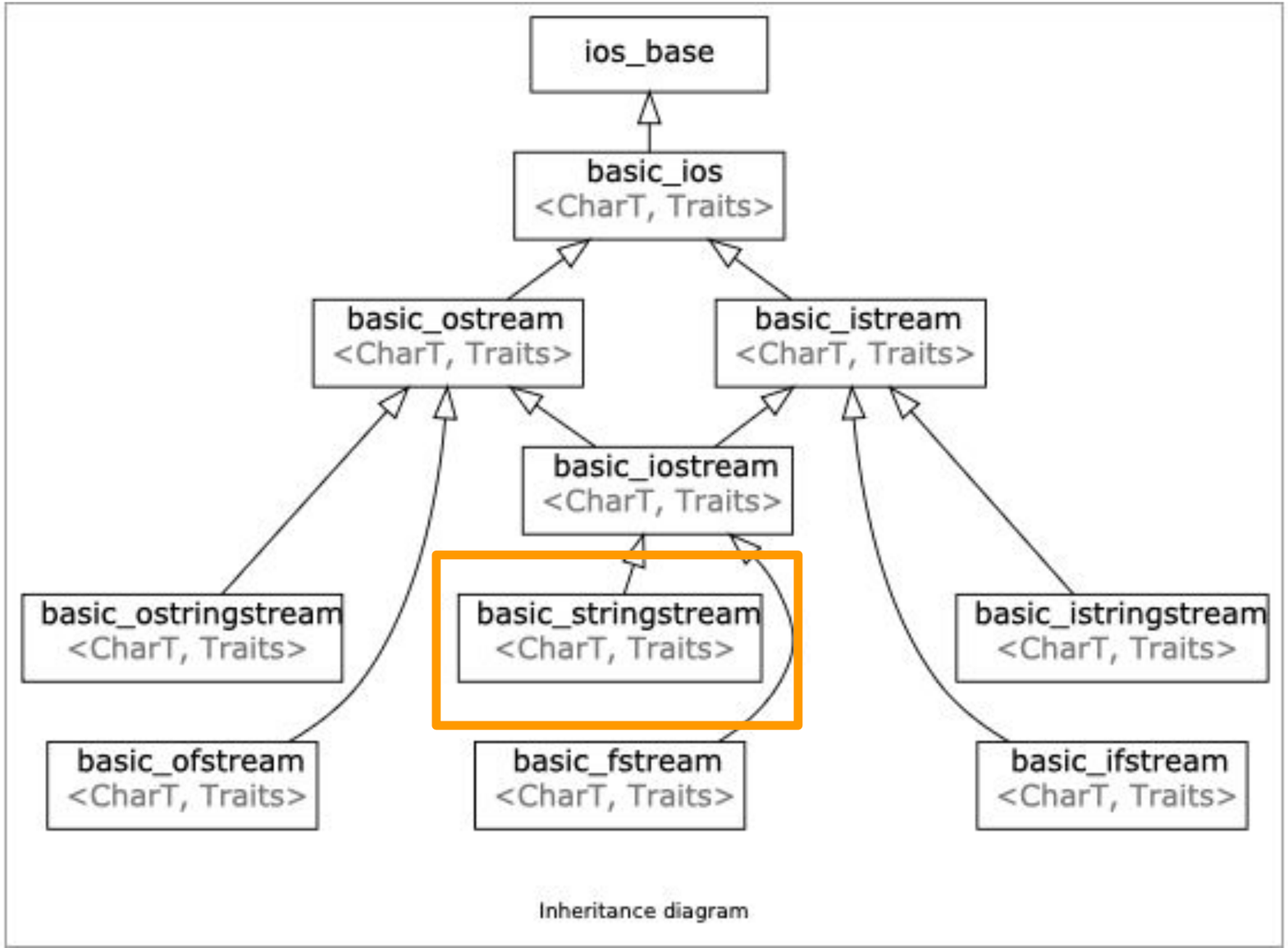
`std::stringstream`

What?
a way to treat strings as streams

Utility?
`stringstreams` are useful for use-cases that deal with mixing data types



What streams actually are



std::stringstream example

```
void foo() {
  /// partial Bjarne Quote
  std::string initial_quote = “Bjarne Stroustrup C makes it easy to shoot
  yourself in the foot”;

  /// create a stringstream
  std::stringstream ss(initial_quote);

  /// data destinations
  std::string first;
  std::string last;
  std::string language, extracted_quote;

  ss >> first >> last >> language >> extracted_quote;
  std::cout << first << “ ” << last << “ said this: ”<< language << “ “ <<
  extracted_quote << std::endl;
}
```

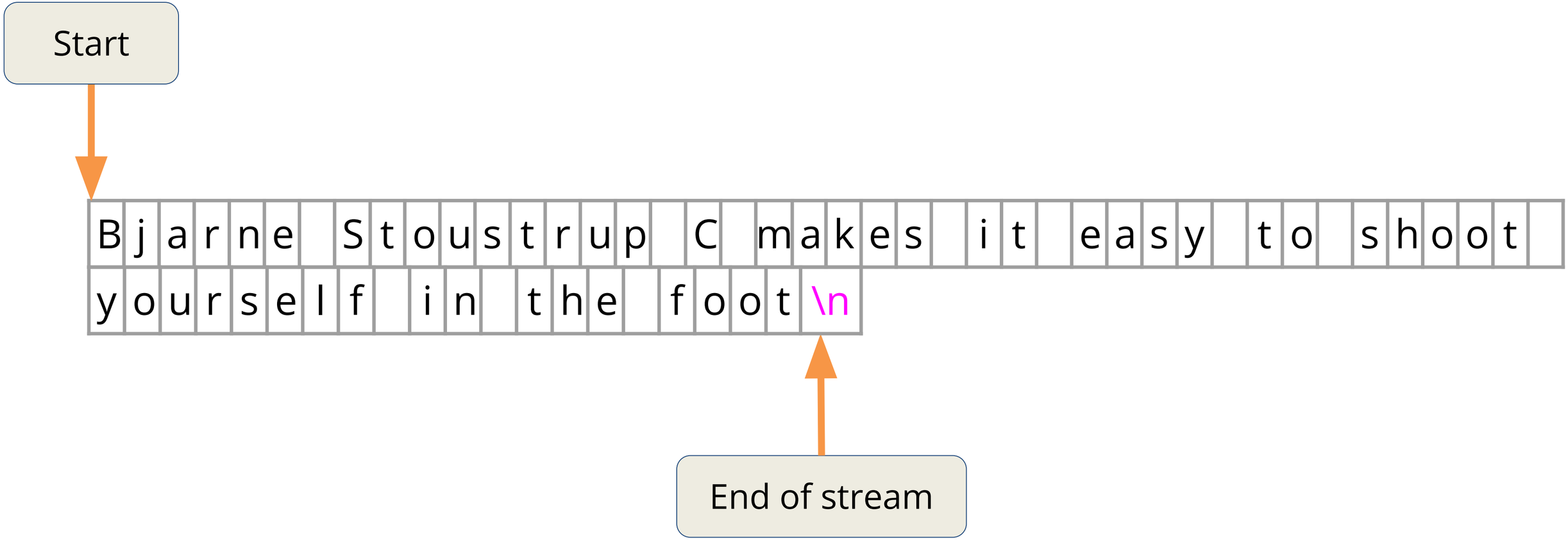
initialize stringstream with string constructor

std::stringstream example

```
void foo() {
  /// partial Bjarne Quote
  std::string initial_quote = “Bjarne Stroustrup C makes it easy to shoot
  yourself in the foot”;

  /// create a stringstream
  std::stringstream ss;
  ss << initial_quote;

  /// data destinations
  std::string first;
  std::string last;
  std::string language, extracted_quote;

  ss >> first >> last >> language >> extracted_quote;
  std::cout << first << “ ” << last << “ said this: ”<< language << “ “ <<
  extracted_quote << std::endl;
}
```

since this is a stream we can also **insert** the `initial_quote` like this!

what the stream looks like!



std::stringstream example

```
void foo() {
  /// partial Bjarne Quote
  std::string initial_quote = “Bjarne Stroustrup C makes it easy to shoot
  yourself in the foot”;

  /// create a stringstream
  std::stringstream ss(initial_quote);

  /// data destinations
  std::string first;
  std::string last;
  std::string language, extracted_quote;

  ss >> first >> last >> language;
  std::cout << first << “ ” << last << “ said this: ”<< language << “ “ <<
  extracted_quote << std::endl;
}
```

Remember! Streams move data from one place to another

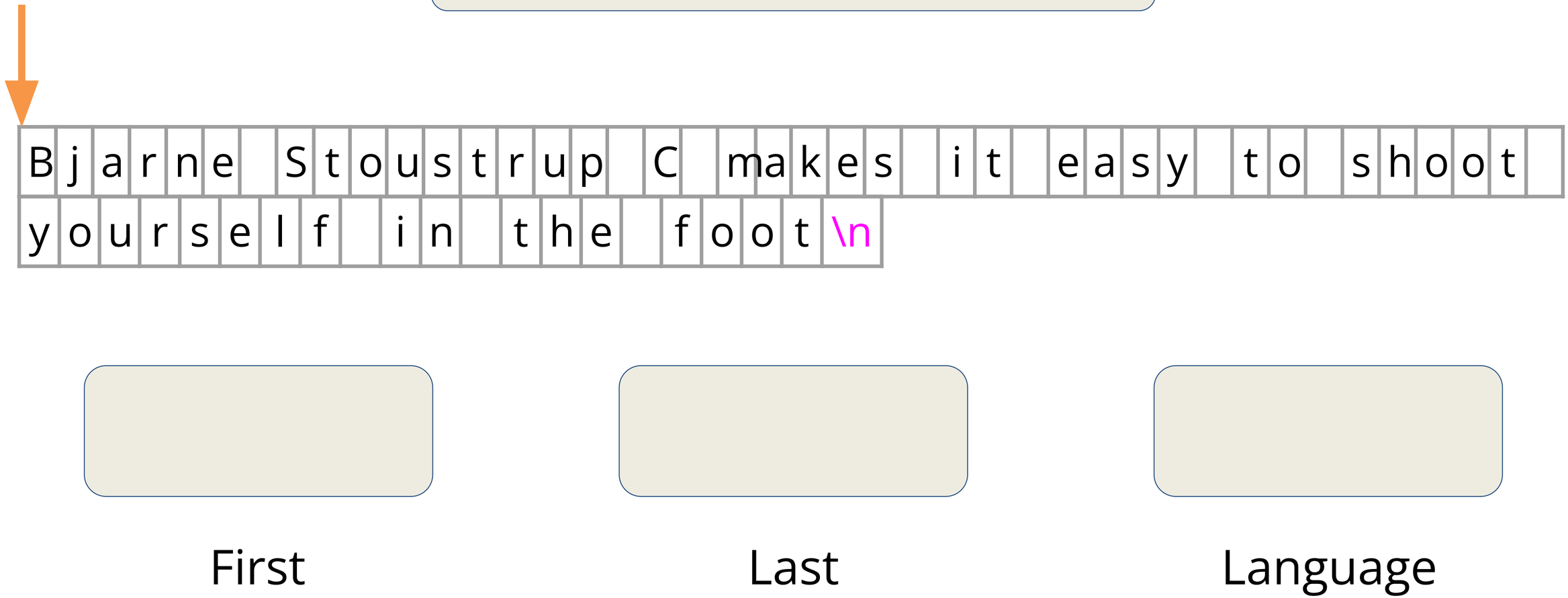
std::stringstream example

```
void foo() {
  /// partial Bjarne Quote
  std::string initial_quote = “Bjarne Stroustrup C makes it easy to shoot
  yourself in the foot”;

  /// create a stringstream
  std::stringstream ss(initial_quote);

  /// data destinations
  std::string first;
  std::string last;
  std::string language, extracted_quote;

  ss >> first >> last >> language;
  std::cout << first << “ ” << last << “ said this: ”<< language << “ “ <<
  extracted_quote << std::endl;
}
```

We’re making use of the insertion operator

what the stream looks like!

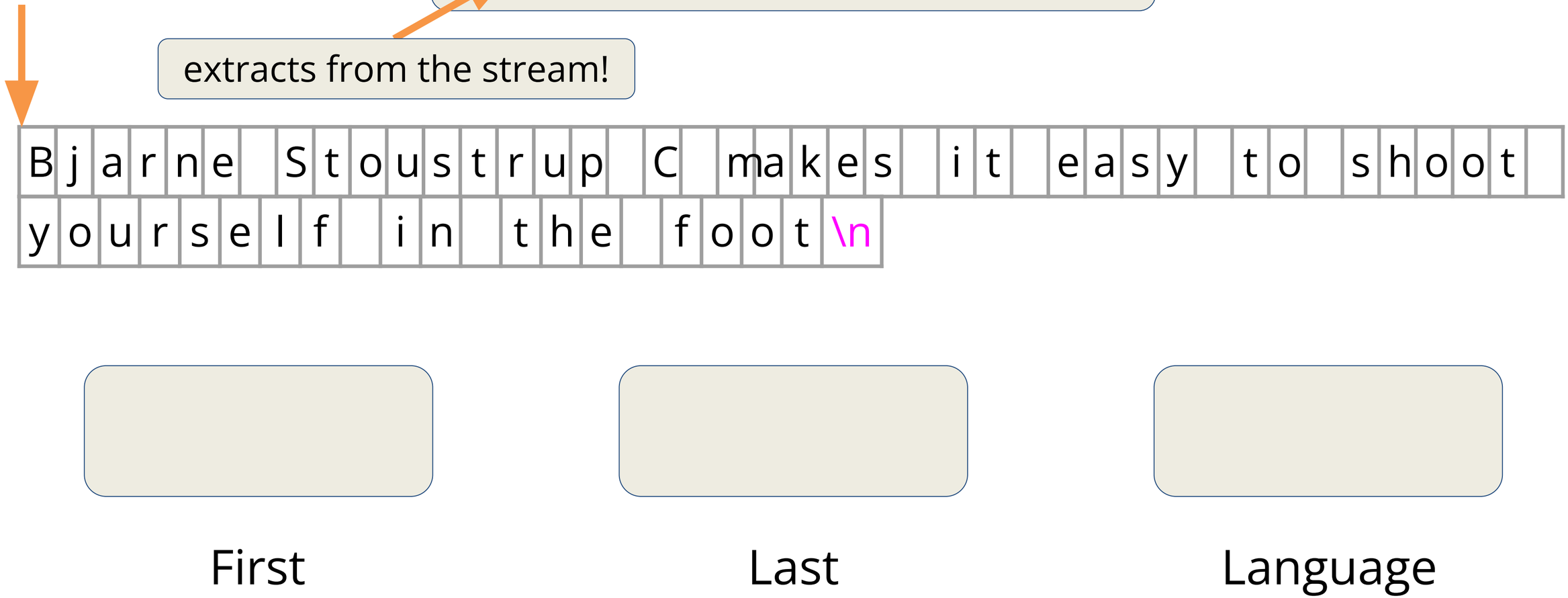
```
ss >> first >> last >> language;
```



what the stream looks like!

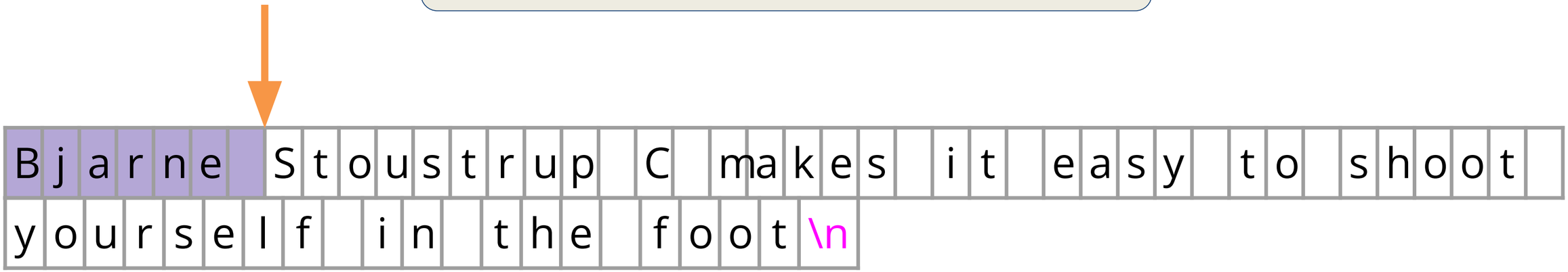
```
ss >> first >> last >> language;
```

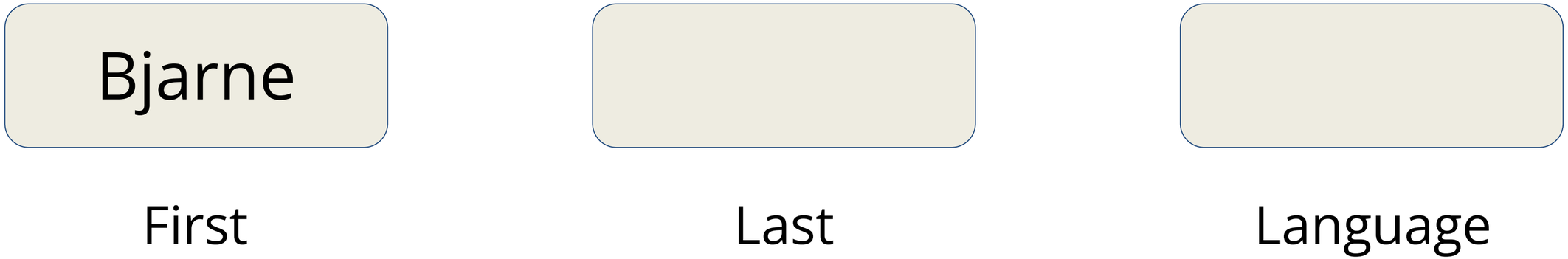
extracts from the stream!



what the stream looks like!

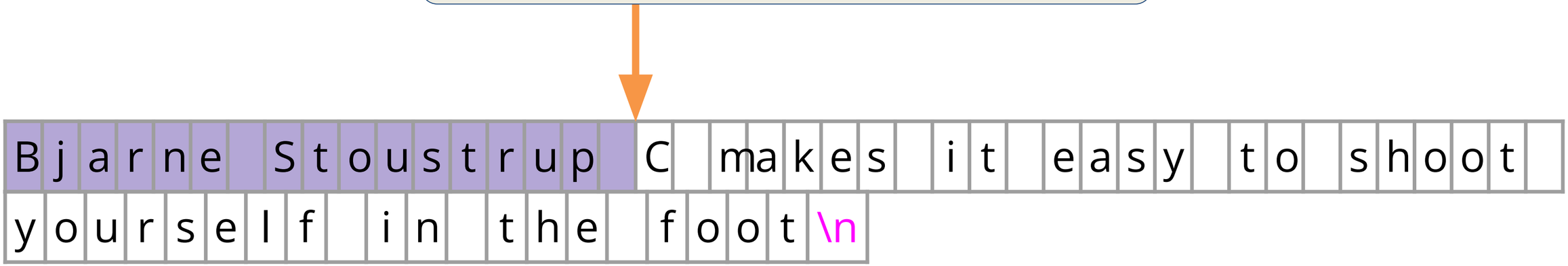
```
ss >> first >> last >> language;
```





what the stream looks like!

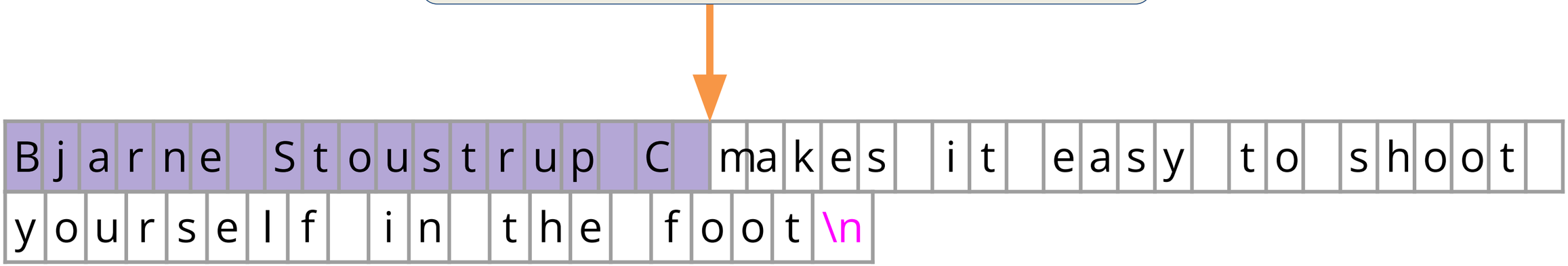
```
ss >> first >> last >> language;
```



Bjarne	Stroustrup	
First	Last	Language

what the stream looks like!

```
ss >> first >> last >> language;
```



Bjarne	Stroustrup	C
First	Last	Language

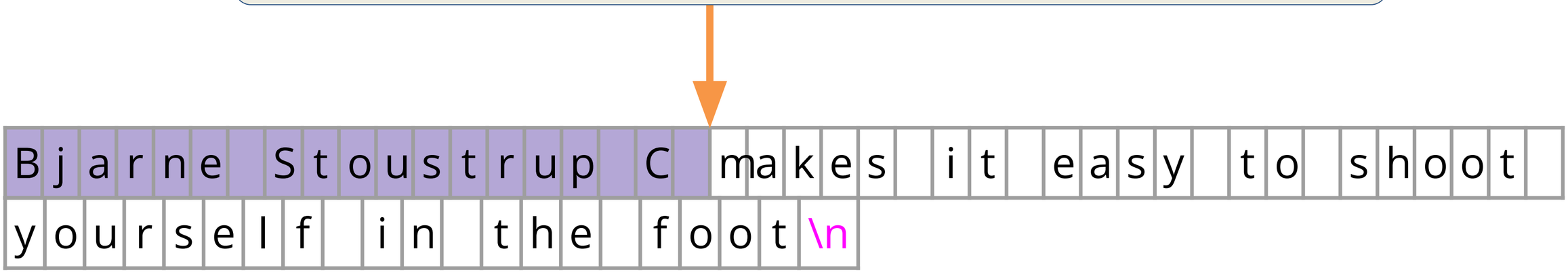
std::stringstream example

```
void foo() {
  /// partial Bjarne Quote
  std::string initial_quote = “Bjarne Stroustrup C makes it easy to shoot
  yourself in the foot”;

  /// create a stringstream
  std::stringstream ss(initial_quote);

  /// data destinations
  std::string first;
  std::string last;
  std::string language, extracted_quote;

  ss >> first >> last >> language;
  std::cout << first << “ ” << last << “ said this: ” << language << “ “ <<
  extracted_quote << std::endl;
}
```

We want to extract the quote!

what the stream looks like!

```
ss >> first >> last >> language >> extracted_quote;
```



Bjarne

First

Stroustrup

Last

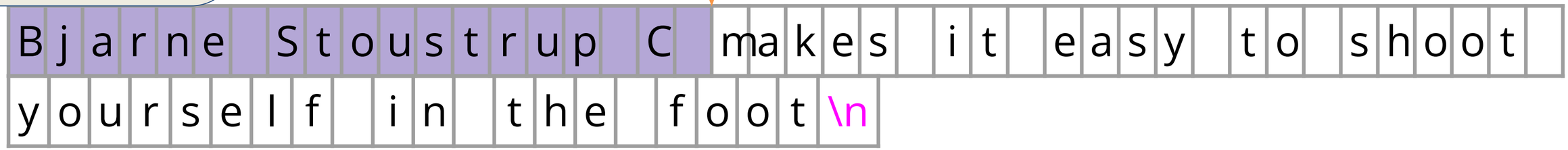
C

Language

what the stream looks like!

Problem:
?

```
ss >> first >> last >> language >> extracted_quote;
```



Bjarne

First

Stroustrup

Last

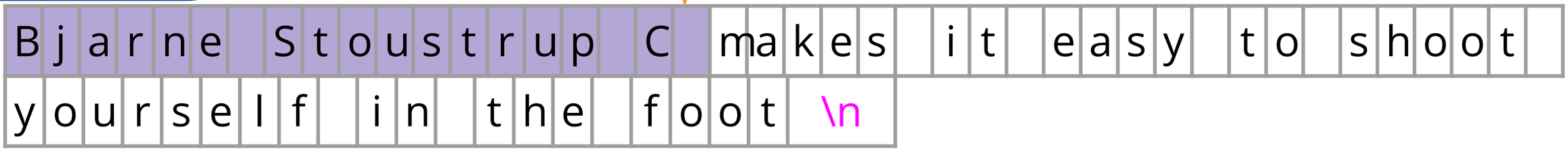
C

Language

what the stream looks like!

Problem:
The **>>** operator only reads until the next whitespace!

```
ss >> first >> last >> language >> extracted_quote;
```



Bjarne

First

Stroustrup

Last

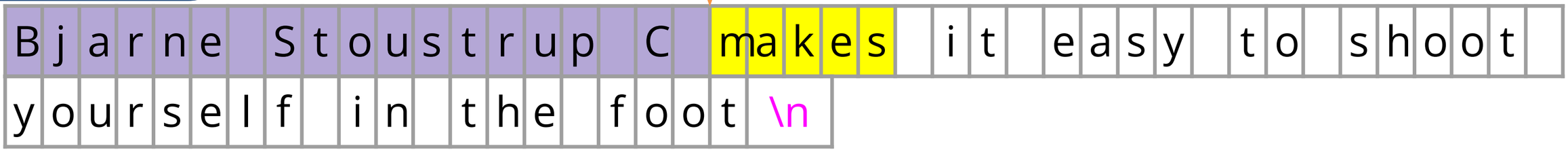
C

Language

what the stream looks like!

Problem:
The **>>** operator only reads until the next whitespace!

```
ss >> first >> last >> language >> extracted_quote;
```



Bjarne	Stroustrup	C
First	Last	Language

Use `getline()`!

```
istream& getline(istream& is, string& str, char delim)
```

- **`getline()`** reads an input stream, **`is`**, up until the **`delim`** char and stores it in some buffer, **`str`**.

Use `getline()`!

```
istream& getline(istream& is, string& str, char delim)
```

- **`getline()`** reads an input stream, **`is`**, up until the **`delim`** char and stores it in some buffer, **`str`**.

- The `delim` char is by default '`\n`'.

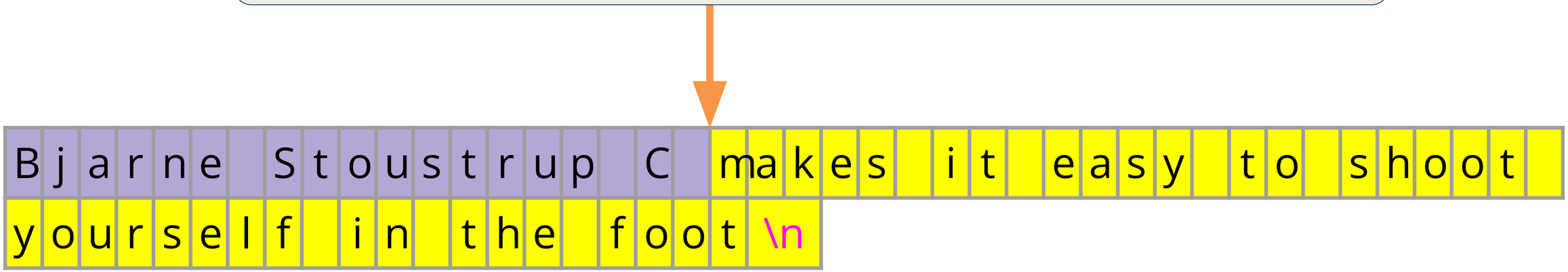
Use `getline()`!

```
istream& getline(istream& is, string& str, char delim)
```

- **`getline()`** reads an input stream, **`is`**, up until the **`delim`** char and stores it in some buffer, **`str`**.
- The `delim` char is by default '`\n`'.
- **`getline()`** ***<u>consumes</u>*** the `delim` character!
  - PAY ATTENTION TO THIS :)

use std::getline()!

```
ss >> first >> last >> language >> extracted_quote;
```

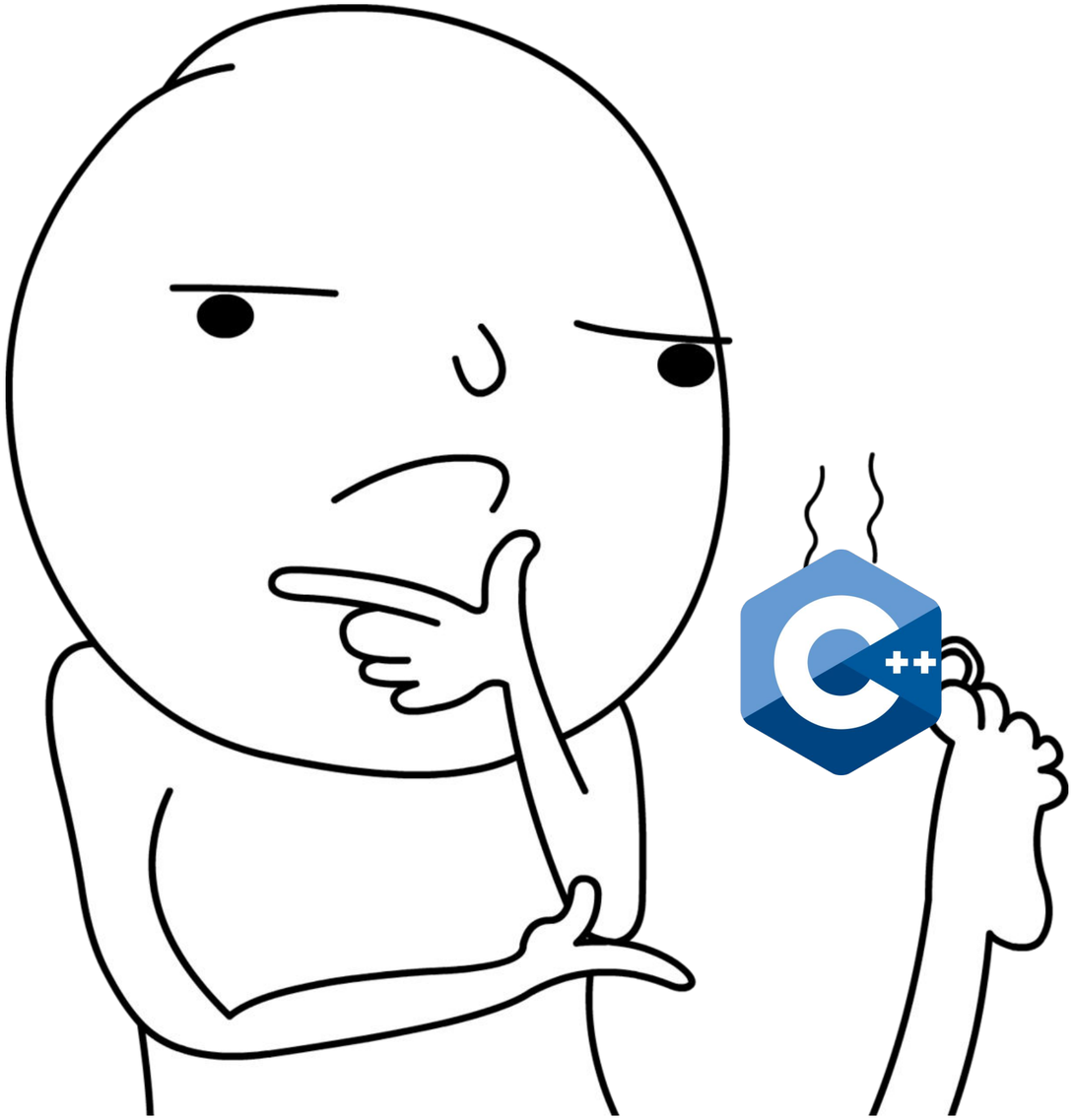


Bjarne	Stroustrup	C
First	Last	Language

std::stringstream example

```
void foo() {
  /// partial Bjarne Quote
  std::string initial_quote = “Bjarne Stroustrup C makes it easy to shoot
  yourself in the foot”;

  /// create a stringstream
  std::stringstream ss(initial_quote);

  /// data destinations
  std::string first;
  std::string last;
  std::string language, extracted_quote;
  ss >> first >> last >> language;
  std::getline(ss, extracted_quote);
  std::cout << first << “ ” << last << “ said this: ‘” << language << “ “ <<
  extracted_quote + “‘“ << std::endl;
}
```

What questions do we have?



Plan

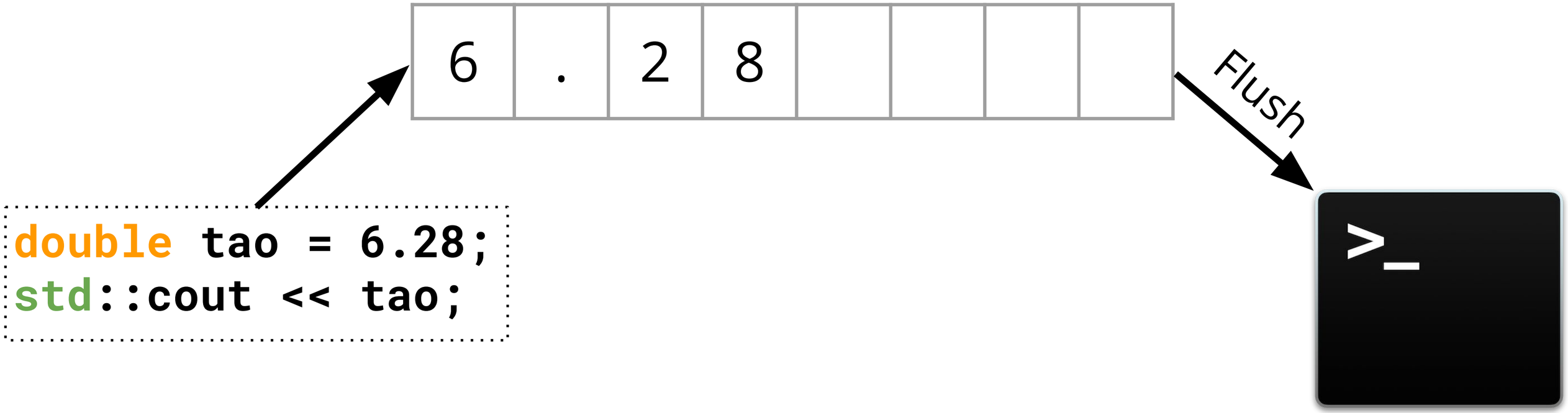
1. Quick recap
2. What are streams??!!
3. `stringstreams`!
4. **`cout` and `cin`**
5. Output streams
6. Input streams

Output Streams

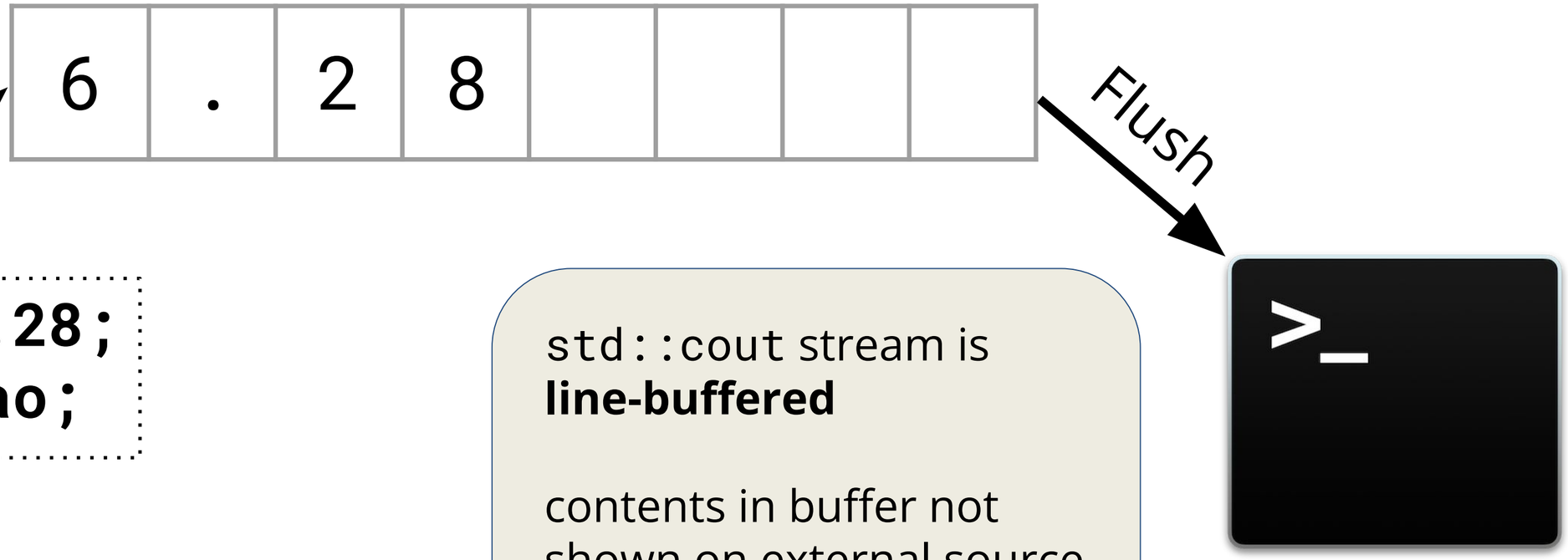
- a way to write data to a destination/external source
  - ex. writing out something to the console (`std::cout`)
  - use the << operator to ***send*** to the output stream

Zooming in on Output Streams!

Character in output streams are stored in an intermediary buffer before being flushed to the destination



Zooming in on Output Streams!

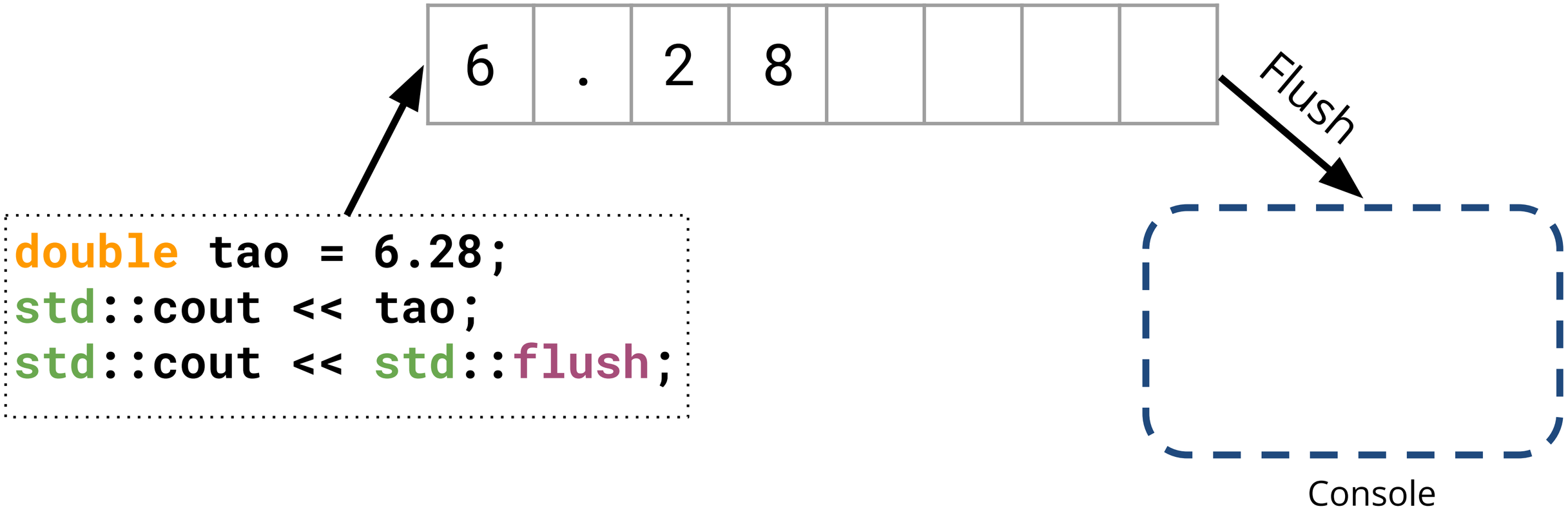



```
double tao = 6.28;
std::cout << tao;
```

`std::cout` stream is **line-buffered**

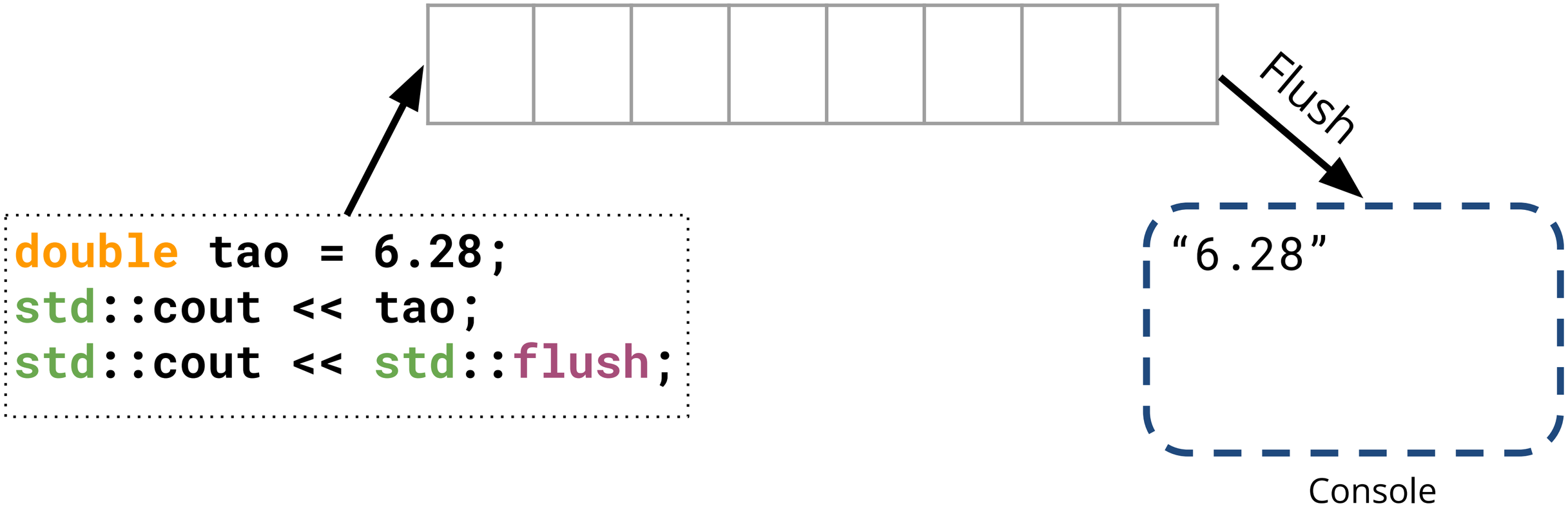
contents in buffer not shown on external source until an explicit flush occurs!

Zooming in on Output Streams!

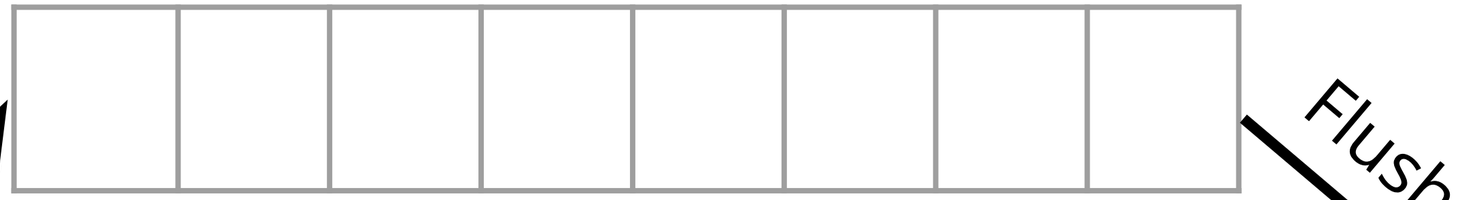


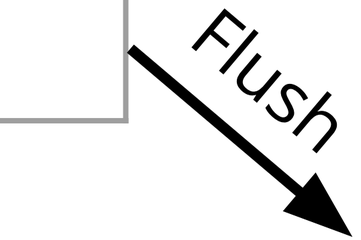

```
double tao = 6.28;
std::cout << tao;
std::cout << std::flush;
```

Zooming in on Output Streams!

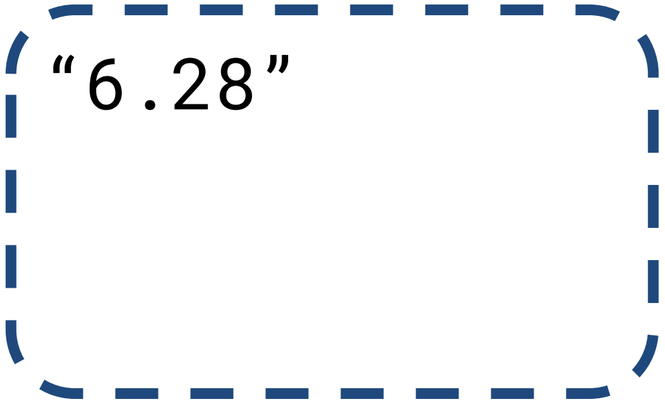


Zooming in on Output Streams!






```
double tao = 6.28;
std::cout << tao;
/// Also flushes!
std::cout << std::endl;
```



std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

Output:

"1"
"2"
"3"
"4"
"5"

`std::endl` tells the `cout` stream to end the line!

Here's without `std::endl`

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i;
  }
  return 0;
}
```

Output:

"12345"

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

Output:

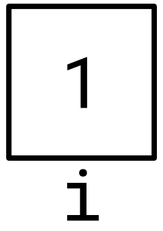
"1"
"2"
"3"
"4"
"5"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

intermediate buffer

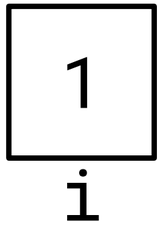


Output:

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```



intermediate buffer

'1'	'\n'						

endl also flushes! So it is immediately sent to destination

Output:

std::endl ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

intermediate buffer

2

i

When a stream is flushed the **intermediate buffer is cleared!**

Output:

"1"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

intermediate buffer

2	'\n'						

2

i

Next integer is put into the stream and immediately flushed!

Output:

"1"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

3

i

intermediate buffer

Next integer is put into the stream and immediately flushed!

Output:

"1"
"2"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

intermediate buffer

3	'\n'						

3

i

Next integer is put into the stream and immediately flushed!

Output:

"1"
"2"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

intermediate buffer

4

i

Next integer is put into the stream and immediately flushed!

Output:

"1"
"2"
"3"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

intermediate buffer

4	'\n'						

4

i

Next integer is put into the stream and immediately flushed!

Output:

"1"
"2"
"3"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

intermediate buffer

5

i

Next integer is put into the stream and immediately flushed!

Output:

"1"
"2"
"3"
"4"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

intermediate buffer

5	'\n'						

5

i

Next integer is put into the stream and immediately flushed!

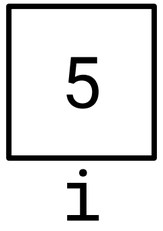
Output:

"1"
"2"
"3"
"4"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```



intermediate buffer

Next integer is put into the stream and immediately flushed!

Output:

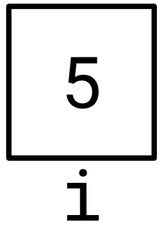
"1"
"2"
"3"
"4"
"5"

`std::endl` ***also*** tells the stream to **flush**

std::endl

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << std::endl;
  }
  return 0;
}
```

intermediate buffer



flushing is an expensive operation!

Output:

"1"
"2"
"3"
"4"
"5"

`std::endl` ***also*** tells the stream to **flush**

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

1

i

intermediate buffer

C++ is (kinda) smart! It knows when to auto flush

Output:

Let’s try just adding the **‘\n’** character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

intermediate buffer

1	‘\n’						

1

i

C++ is (kinda) smart! It knows when to auto flush

Output:

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

intermediate buffer

1	‘\n’						

2
i

C++ is (kinda) smart! It knows when to auto flush

Output:

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

2

i

intermediate buffer

1	‘\n’	2	‘\n’				

C++ is (kinda) smart! It knows when to auto flush

Output:

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

3

i

intermediate buffer

1	‘\n’	2	‘\n’				

C++ is (kinda) smart! It knows when to auto flush

Output:

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

3

i

intermediate buffer

1	‘\n’	2	‘\n’	3	‘\n’		

C++ is (kinda) smart! It knows when to auto flush

Output:

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

4

i

intermediate buffer

1	‘\n’	2	‘\n’	3	‘\n’		

C++ is (kinda) smart! It knows when to auto flush

Output:

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

4

i

intermediate buffer

1	‘\n’	2	‘\n’	3	‘\n’	4	‘\n’

C++ is (kinda) smart! It knows when to auto flush

Output:

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

5

i

intermediate buffer

1	‘\n’	2	‘\n’	3	‘\n’	4	‘\n’

C++ is (kinda) smart! It knows when to auto flush

Output:

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

5

i

intermediate buffer

1	‘\n’	2	‘\n’	3	‘\n’	4	‘\n’

😱🤯 Our intermediate buffer is full!

Output:

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

5

i

intermediate buffer

C++: FLUSH

Output:

“1”
“2”
“3”
“4”

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

intermediate buffer

5	‘\n’						

5

i

Yay!

Output:
“1”
“2”
“3”
“4”

Let’s try just adding the ‘\n’ character

‘\n’ 🤠

```
int main()
{
  for (int i=1; i <= 5; ++i) {
    std::cout << i << ‘\n’;
  }
  return 0;
}
```

intermediate buffer

5

i

Yay!

Output:

“1”
“2”
“3”
“4”
“5”

Let’s try just adding the ‘\n’ character

Recall

- **`cerr` and `clog`**

  `cerr`: used to output errors (unbuffered)

  `clog`: used for non-critical event logging (buffered)

read more here: [GeeksForGeeks]

A shoutout and clarification

So there’s a small caveat to this

A shoutout and clarification

However, upon testing these examples, I observed that '\n' seems to flush the buffer in a manner similar to std::cout. Further research led me to the CPP Reference std::endl, which states, "In many implementations, standard output is line-buffered, and writing '\n' causes a flush anyway, unless std::ios::sync_with_stdio(false) was executed." This suggests that in many standard outputs, '\n' behaves the same as std::cout. Additionally, when I appended | cat to my program, I noticed that in file output, '\n' does not immediately flush the buffer.

In case you're looking at these slides Aolin, thank you for pointing this out!

A shoutout and clarification

However, upon testing these examples, I observed that '\n' seems to flush the buffer in a manner similar to std::cout. Further research led me to the CPP Reference std::endl, which states, "In many implementations, standard output is line-buffered, and writing '\n' causes a flush anyway, unless `std::ios::sync_with_stdio(false)` was executed." This suggests that in many standard outputs, '\n' behaves the same as std::cout. Additionally, when I appended | cat to my program, I noticed that in file output, '\n' does not immediately flush the buffer.

In case you're looking at these slides Aolin, thank you for pointing this out!

A shoutout and clarification

```
int main()
{
  std::ios::sync_with_stdio(false)
  for (int i=1; i <= 5; ++i) {
    std::cout << i << '\n';
  }
  return 0;
}
```

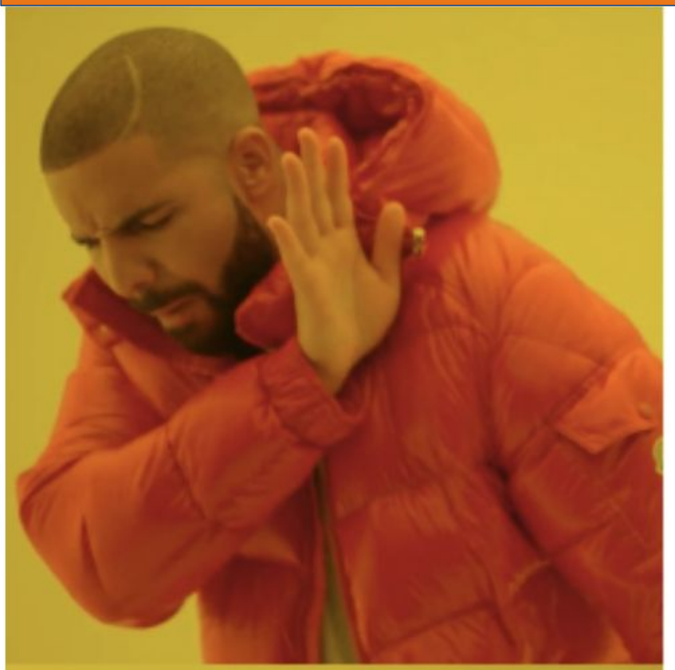
You ***may*** get a massive performance boost from this. Read more about this here

In case you're looking at these slides Aolin, thank you for pointing this out!


YOU WASTED ENOUGH OF MY TIME
CAN I PLEASE HAVE MY LIFE BACK?
memeshappen.com

ASIDE: If you're interested in how computers are able to do multiple things at the same time take CS149!

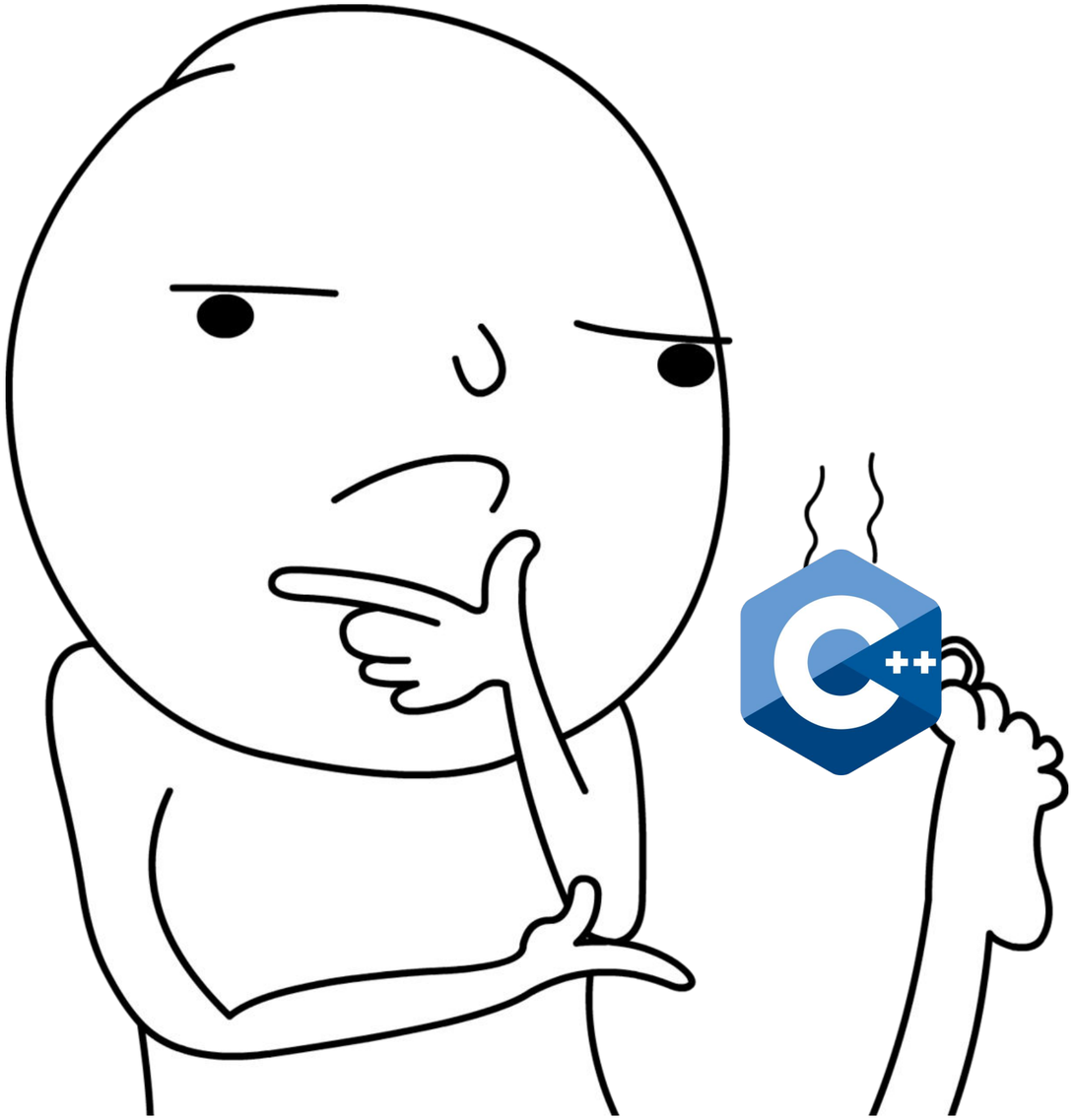
Use ‘\n’!



```
std::cout << “Draaaakkkkeeeeeeeeeee” << std::endl;
```

```
std::cout << “Draaaakkkkeeeeeeeeeee” << ‘\n’;
```

What questions do we have?



Output File Streams

- Output file streams have a type: `std::ofstream`
- a way to write data to a file!
  - use the `<<` insertion operator to ***send*** to the file
  - There are some methods for `std::ofstream` check them out
  - Here are some you should know:
    - `is_open()`
    - `open()`
    - `close()`
    - `fail()`

Output File Streams

```
int main() {
  /// associating file on construction
  std::ofstream ofs("hello.txt");
  if (ofs.is_open()) {
    ofs << "Hello CS106L!" << '\n';
  }
  ofs.close();
  ofs << "this will not get written";

  ofs.open("hello.txt");
  ofs << "this will though! It's open
again";
  return 0;
}
```

Output File Streams

```
int main() {
  /// associating file on construction
  std::ofstream ofs("hello.txt");
  if (ofs.is_open()) {
    ofs << "Hello CS106L!" << '\n';
  }
  ofs.close();
  ofs << "this will not get written";

  ofs.open("hello.txt");
  ofs << "this will though! It's open
again";
  return 0;
}
```

Creates an output file stream to the file "hello.txt"

Output File Streams

```
int main() {
  /// associating file on construction
  std::ofstream ofs("hello.txt");
  if (ofs.is_open()) {
    ofs << "Hello CS106L!" << '\n';
  }
  ofs.close();
  ofs << "this will not get written";

  ofs.open("hello.txt");
  ofs << "this will though! It's open
again";
  return 0;
}
```

Checks if the file is open and if it is, then tries to write to it!

Output File Streams

```
int main() {
  /// associating file on construction
  std::ofstream ofs("hello.txt");
  if (ofs.is_open()) {
    ofs << "Hello CS106L!" << '\n';
  }
  ofs.close();
  ofs << "this will not get written";

  ofs.open("hello.txt");
  ofs << "this will though! It's open
again";
  return 0;
}
```

This closes the output file stream to "hello.txt"

Output File Streams

```
int main() {
  /// associating file on construction
  std::ofstream ofs("hello.txt");
  if (ofs.is_open()) {
    ofs << "Hello CS106L!" << '\n';
  }
  ofs.close();
  ofs << "this will not get written";

  ofs.open("hello.txt");
  ofs << "this will though! It's open
again";
  return 0;
}
```

Will silently fail

Output File Streams

```
int main() {
  /// associating file on construction
  std::ofstream ofs("hello.txt");
  if (ofs.is_open()) {
    ofs << "Hello CS106L!" << '\n';
  }
  ofs.close();
  ofs << "this will not get written";

  ofs.open("hello.txt");
  ofs << "this will though! It's open
again";
  return 0;
}
```

Reopens the stream

Output File Streams

```
int main() {
  /// associating file on construction
  std::ofstream ofs("hello.txt");
  if (ofs.is_open()) {
    ofs << "Hello CS106L!" << '\n';
  }
  ofs.close();
  ofs << "this will not get written";

  ofs.open("hello.txt");
  ofs << "this will though! It's open
again";
  return 0;
}
```

Successfully writes to stream

Let's checkout some code!

My cue to see code :)

Output File Streams

```
int main() {
  /// associating file on construction
  std::ofstream ofs("hello.txt")
  if (ofs.is_open()) {
    ofs << "Hello CS106L!" << '\n';
  }
  ofs.close();
  ofs << "this will not get written";

  ofs.open("hello.txt", std::ios::app);
  ofs << "this will though! It's open
again";
  return 0;
}
```

Flag specifies you want to append, not truncate!

Input File Streams

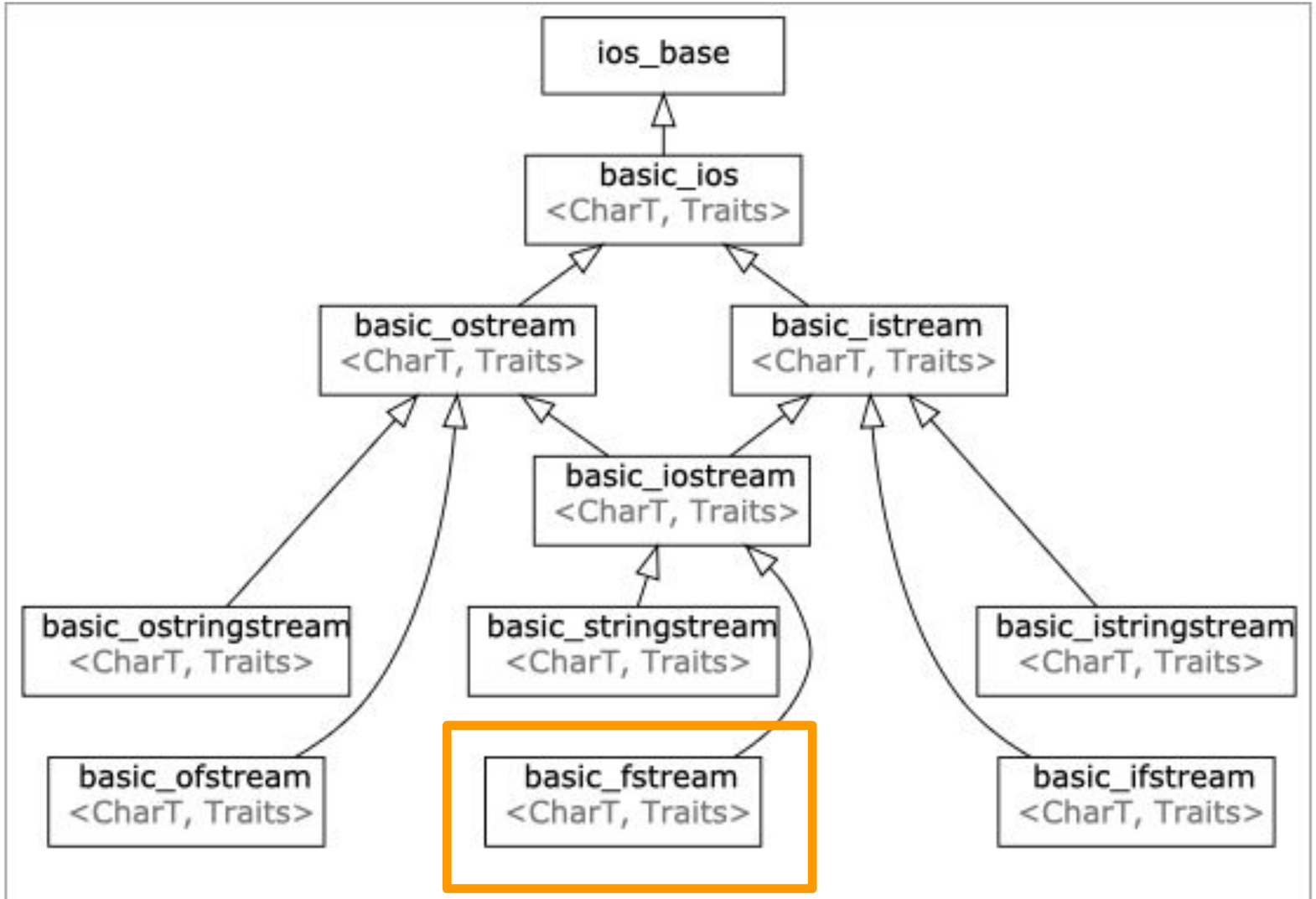
```
int inputFileStreamExample() {
  std::ifstream ifs("input.txt");
  if (ifs.is_open()) {
      std::string line;
      std::getline(ifs, line);
      std::cout << "Read from the file: " << line << '\n';
  }
  if (ifs.is_open()) {
      std::string lineTwo;
      std::getline(ifs, lineTwo);
      std::cout << "Read from the file: " << lineTwo << '\n';
  }
  return 0;
}
```

Input File Streams

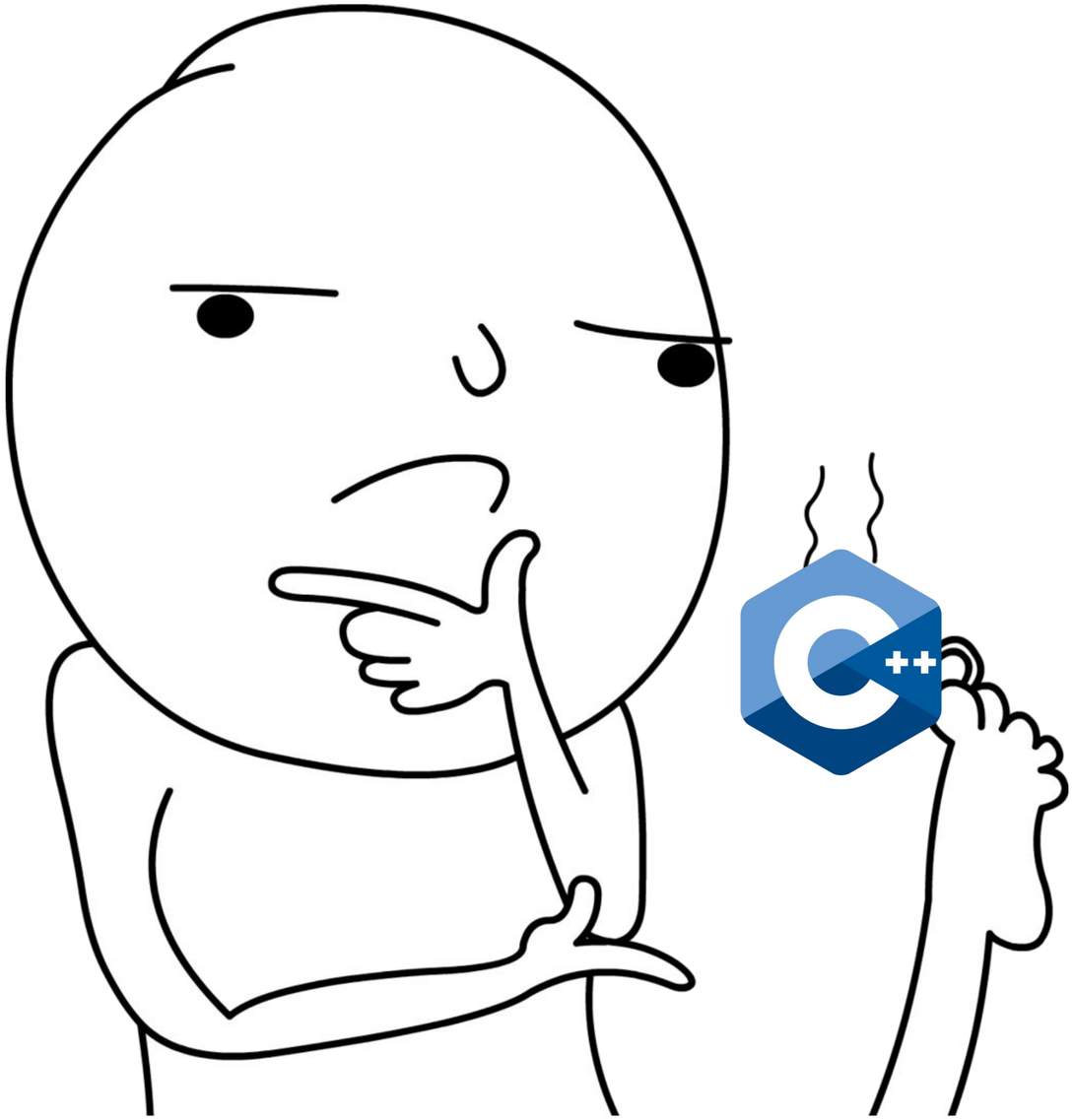
```
int inputFileStreamExample() {
  std::ifstream ifs("input.txt");
  if (ifs.is_open()) {
      std::string line;
      std::getline(ifs, line);
      std::cout << "Read from the file: " << line << '\n';
  }
  if (ifs.is_open()) {
      std::string lineTwo;
      std::getline(ifs, lineTwo);
      std::cout << "Read from the file: " << lineTwo << '\n';
  }
  return 0;
}
```

Input and output streams on the same source/destination type are complimentary!

IO File Streams



What questions do we have?



Plan

1. Quick recap
2. What are streams??!!
3. `stringstreams`!
4. `cout` and `cin`
5. Output streams
6. **Input streams**

Input Streams

- Input streams have the type `std::istream`
- a way to read data from an destination/external source
  - use the `>>` extractor operator to ***read*** from the input stream
  - Remember the `std::cin` is the console input stream

`std::cin`

- `std::cin` is buffered
- Think of it as a place where a user can store some data and then read from it
- `std::cin` buffer stops at a whitespace

`std::cin`

- `std::cin` is buffered
- Think of it as a place where a user can store some data and then read from it
- `std::cin` buffer stops at a whitespace
- Whitespace in C++ includes:
  - “ ” – a literal space
  - `\n` character
  - `\t` character

std::cin


```
int main()
{
  double pi;
  std::cin; /// what does this do?
  std::cin >> pi;
  std::cout << "pi is: " << pi << '\n';
  return 0;
}
```

cin buffer is empty so prompts for input!

std::cin


```
int main()
{
  double pi;
  std::cin; /// what does this do?
  std::cin >> pi;
  std::cout << "pi is: " << pi << '\n';
  return 0;
}
```

3.14

std::cin

cin	3	.	1	4	'\n'														

```
int main()
{
  double pi;
  std::cin; /// what does this do?
  std::cin >> pi;
  std::cout << "pi is: " << pi << '\n';
  return 0;
}
```

cin not empty so it reads up to white space and saves it to **double pi**

3.14

std::cin


```
int main()
{
  double pi;
  std::cin; /// what does this do?
  std::cin >> pi;
  std::cout << "pi is: " << pi << '\n';
  return 0;
}
```

cout

"3.14"
"pi is: 3.14"

Alternatively

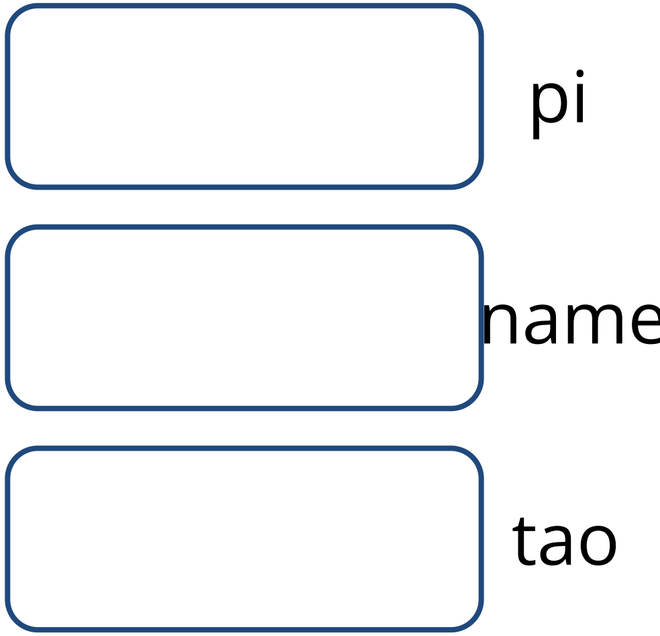
cin	3	.	1	4	'\n'													

```
int main()
{
  double pi;
  std::cin >> pi; /// input directly!
  std::cout << "pi is: " << pi << '\n';
  return 0;
}
```

"3.14"
"pi is: 3.14"

When std::cin fails!

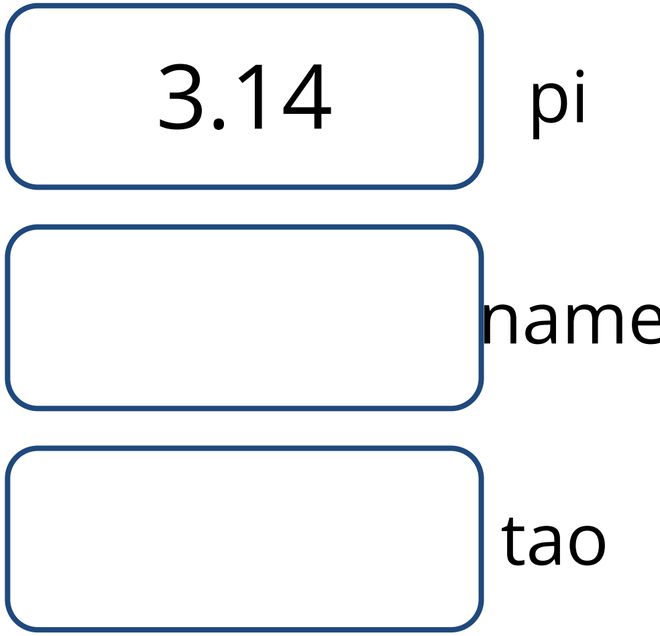

```
int main()
{
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::cin >> name;
  std::cin >> tao;
  std::cout << “my name is: “ << name <<
  “ tao is: “ << tao << “ pi is: “ << pi << ‘\n’;
  return 0;
}
```



When std::cin fails!


```
int main()
{
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::cin >> name;
  std::cin >> tao;
  std::cout << “my name is: “ << name <<
  “ tao is: “ << tao << “ pi is: “ << pi << ‘\n’;
  return 0;
}
```

cin prompts user to enter a value saved in **pi**



When std::cin fails!

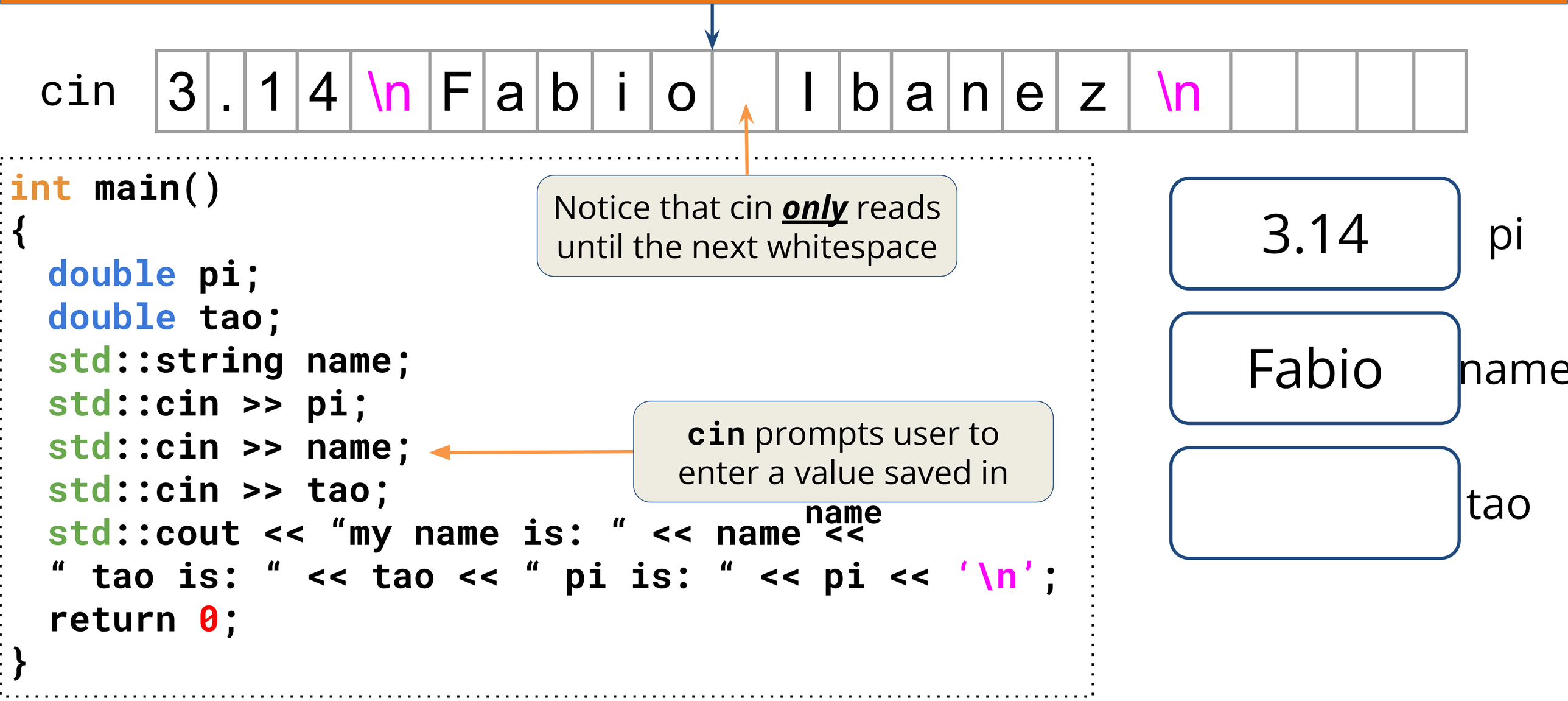
cin	3	.	1	4	\n	F	a	b	i	o		I	b	a	n	e	z	\n				

```
int main()
{
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::cin >> name;
  std::cin >> tao;
  std::cout << “my name is: “ << name <<
  “ tao is: “ << tao << “ pi is: “ << pi << ‘\n’;
  return 0;
}
```

cin prompts user to enter a value saved in **name**

3.14	pi
Fabio	name
	tao

When std::cin fails!



When std::cin fails!

cin	3	.	1	4	\n	F	a	b	i	o		I	b	a	n	e	z	\n				

```
int main()
{
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::cin >> name;
  std::cin >> tao;
  std::cout << “my name is: “
  “ tao is: “ << tao << “ pi is: “ << pi << ‘\n’;
  return 0;
}
```

cin buffer is not empty, so it reads until the next whitespace

3.14	pi
Fabio	name
?	tao

When std::cin fails!

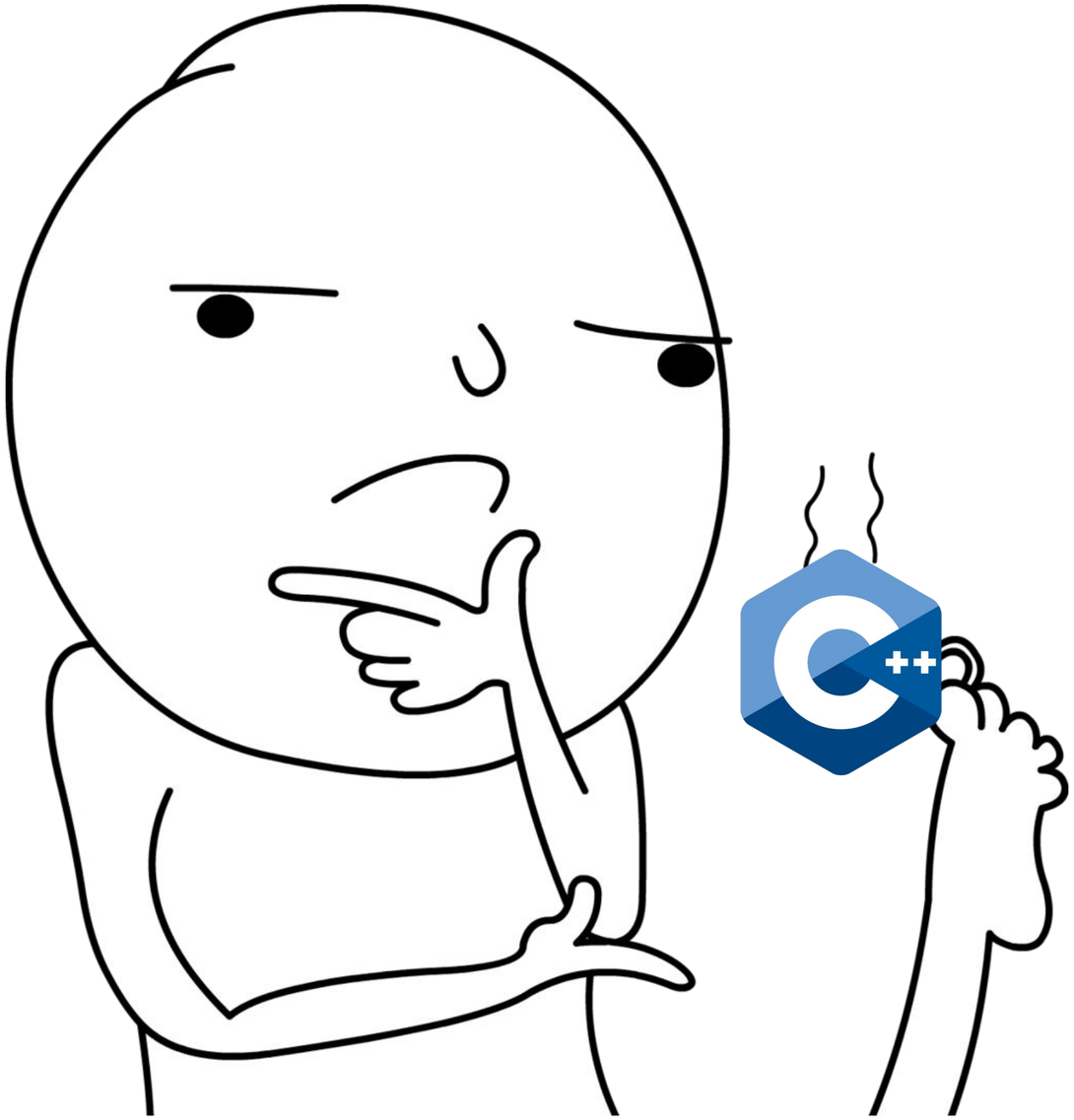
cin	3	.	1	4	\n	F	a	b	i	o		I	b	a	n	e	z	\n				

```
void cinFailure() // replit name
{
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::cin >> name;
  std::cin >> tao;
  std::cout << “my name is: “
  “ tao is: “ << tao << “ pi is: “ << pi << ‘\n’;
}
```

cin buffer is not empty, so it reads until the next whitespace

3.14	pi
Fabio	name
0	tao

What questions do we have?

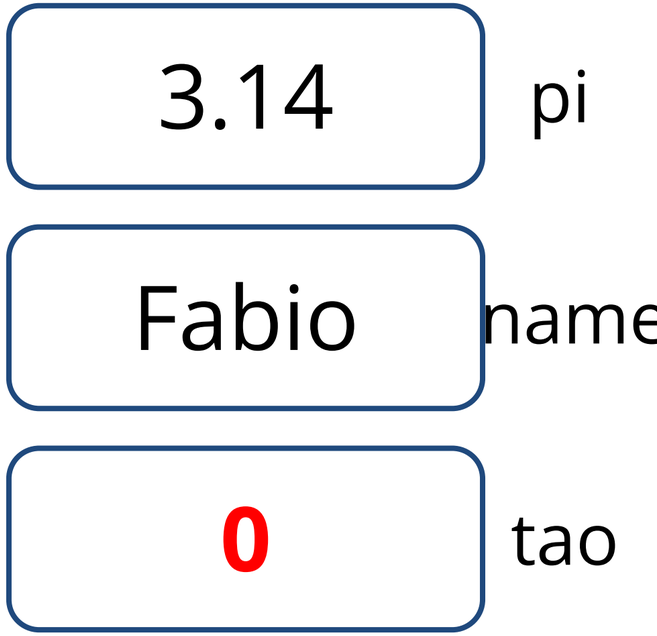


How do we fix this?

Anyone want to take a guess?

Fix?


```
void cinGetlineBug() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << name << " tao is :
" << tao
            << " pi is : " << pi << '\n';
}
```



Fix?


```
void cinGetlineBug() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << name << " tao is : " << tao
            << " pi is : " << pi << '\n';
}
```

3.14	pi
Fabio	name
0	tao

Fix?


```
void cinGetlineBug() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << name << " tao is : " << tao
            << " pi is : " << pi << '\n';
}
```

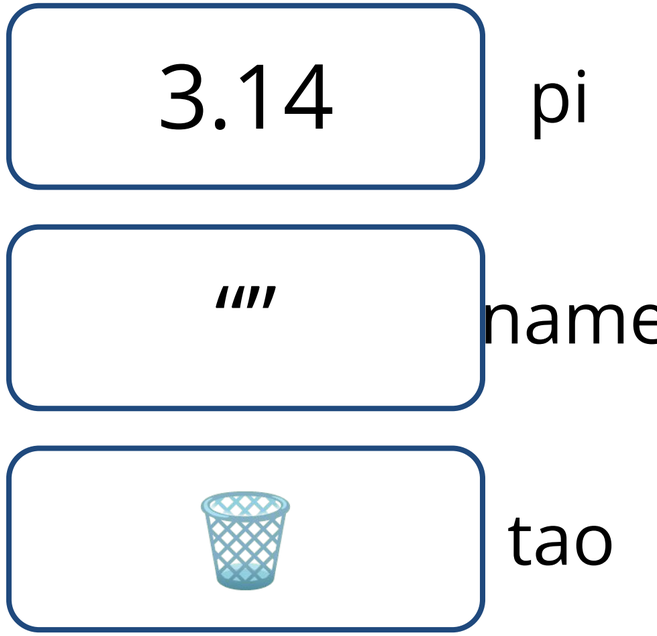
Any guesses for what happens here?

3.14	pi
Fabio	name
0	tao

Fix?


```
void cinGetlineBug() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << name << " tao is : " << tao
            << " pi is : " << pi << '\n';
}
```

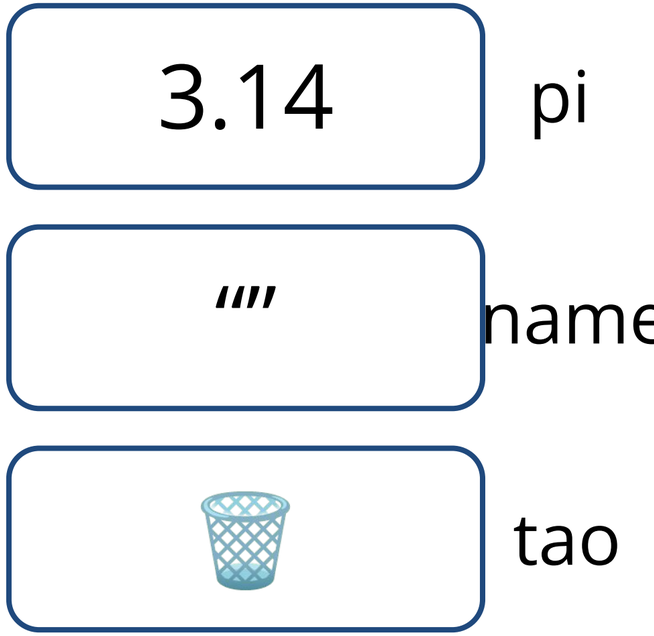
getline consumes the newline character



Fix?


```
void cinGetlineBug() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << na
" << tao
            << " pi is : " << pi << '\n';
}
```

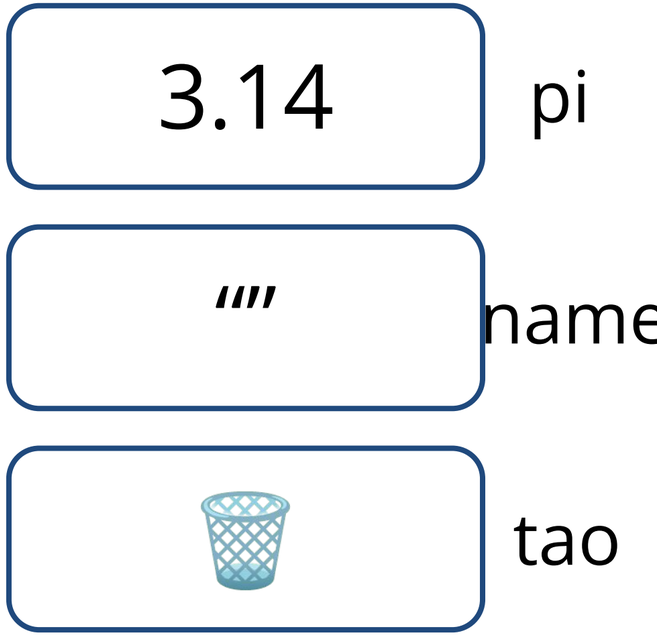
tao is going to be garbage because the buffer is not empty



Fix?


```
void cinGetlineBug() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << na
" << tao
            << " pi is : " << pi << '\n';
}
```

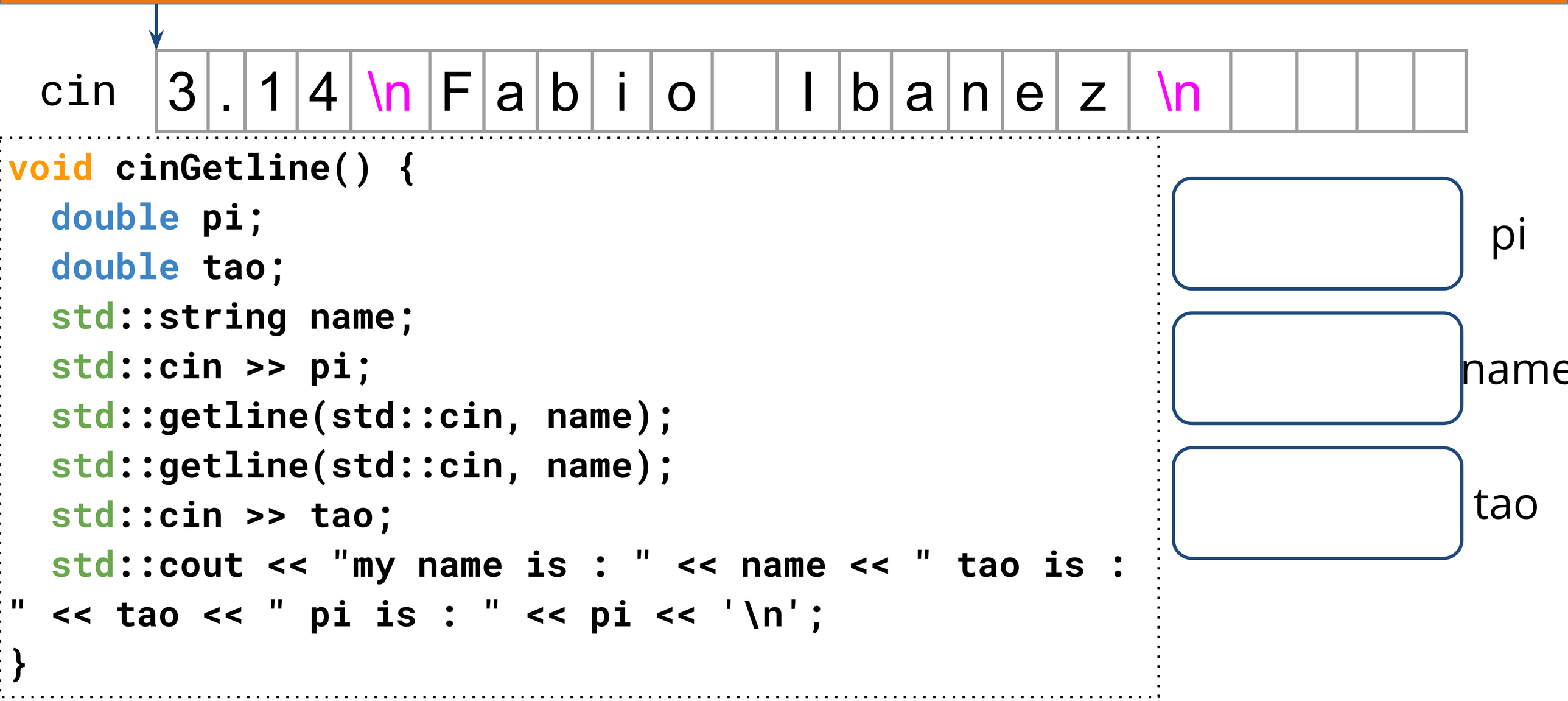
It's going to try to read the green stuff (name). But tao is a **double**!



How do we fix this?

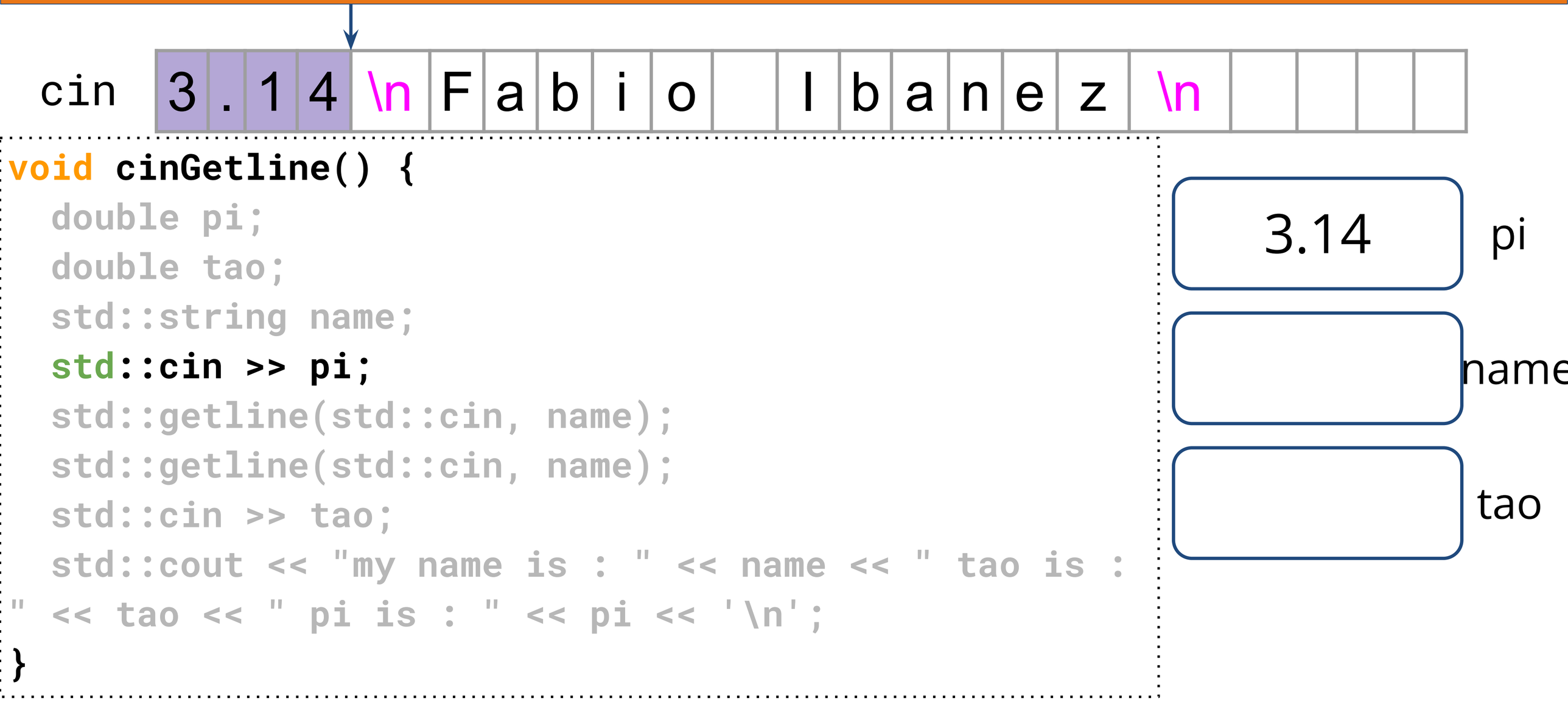
Anyone want to take another guess?

Fix?

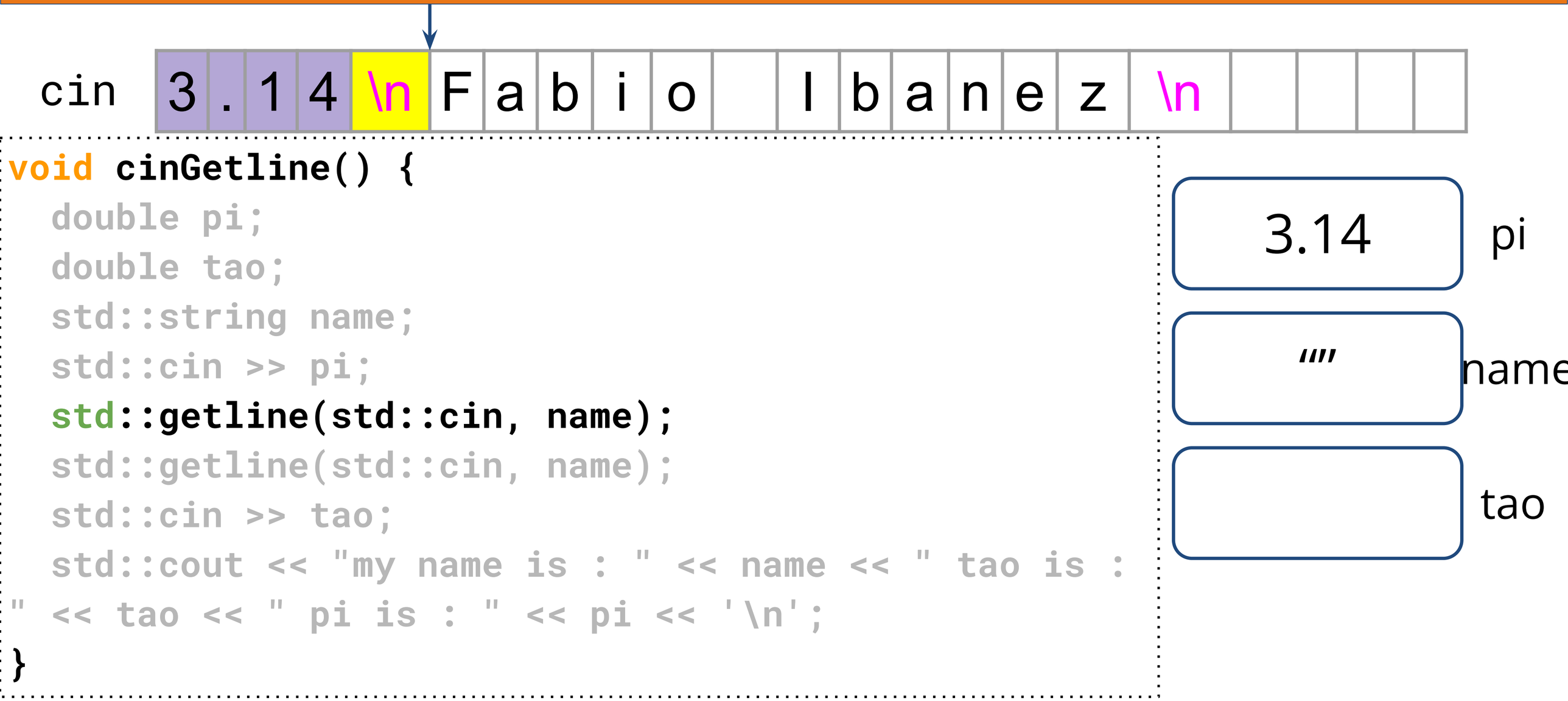



```
void cinGetline() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << name << " tao is :
" << tao << " pi is : " << pi << '\n';
}
```

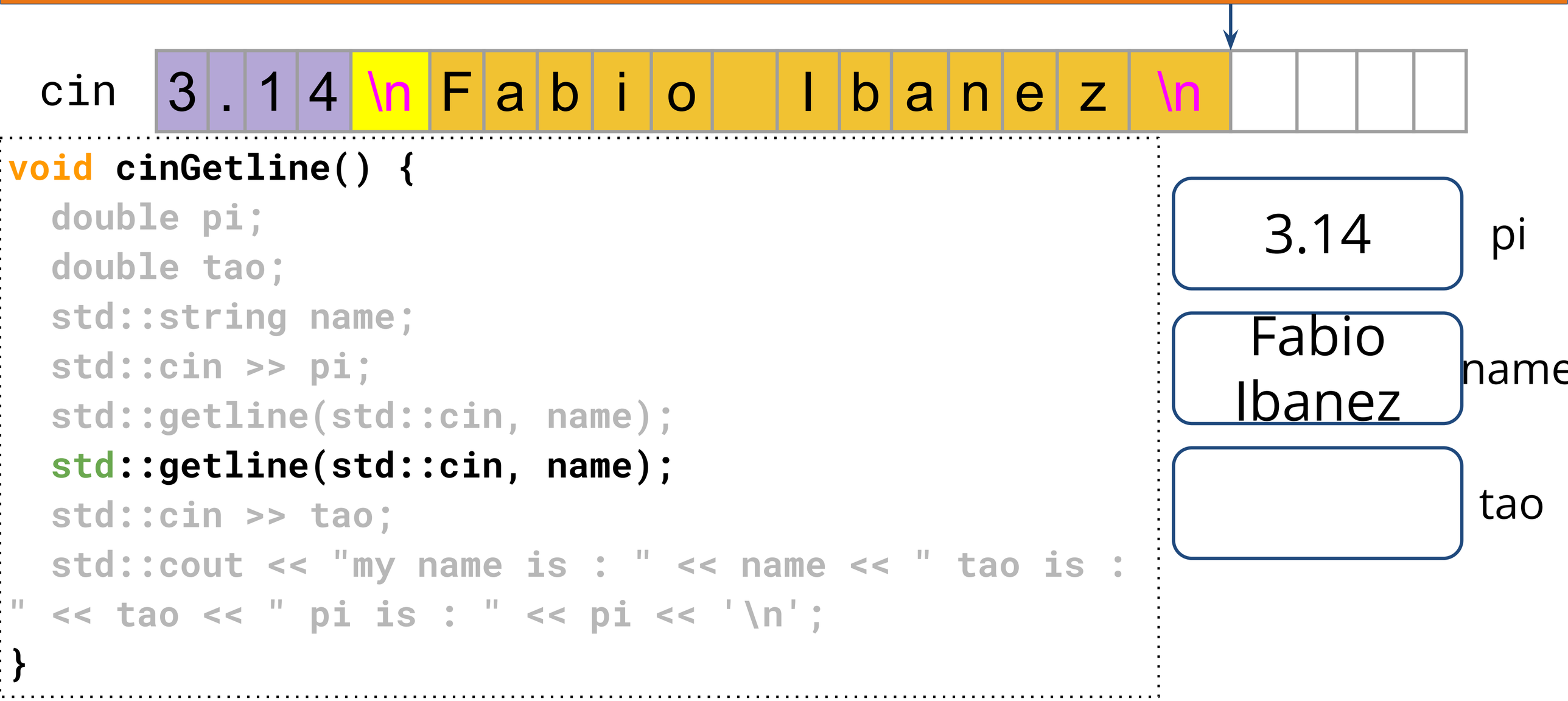
Fix?


cin
3 . 1 4 \n F a b i o l b a n e z \n
void cinGetline() {
double pi;
double tao;
std::string name;
std::cin >> pi;
std::getline(std::cin, name);
std::getline(std::cin, name);
std::cin >> tao;
std::cout << "my name is : " << name << " tao is : " << tao << " pi is : " << pi << '\n';
}
3.14
pi
name
tao

Fix ✅

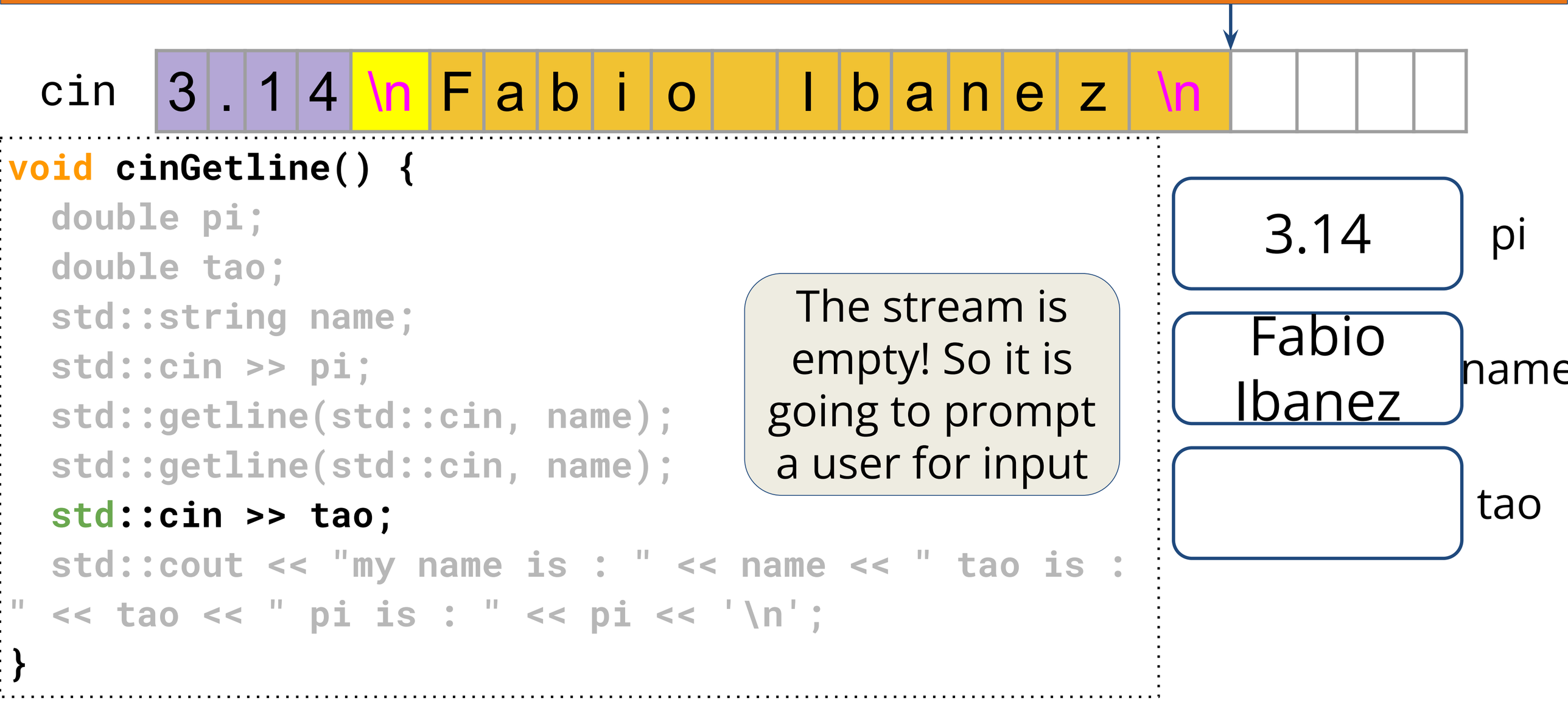

cin
3 . 1 4 \n F a b i o l b a n e z \n
void cinGetline() {
double pi;
double tao;
std::string name;
std::cin >> pi;
std::getline(std::cin, name);
std::getline(std::cin, name);
std::cin >> tao;
std::cout << "my name is : " << name << " tao is : " << tao << " pi is : " << pi << '\n';
}
3.14
pi
""
name
tao

Fix ✅




```
void cinGetline() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << name << " tao is : " << tao << " pi is : " << pi << '\n';
}
```

Fix ✅


cin
3 . 1 4 \n F a b i o l b a n e z \n
void cinGetline() {
double pi;
double tao;
std::string name;
std::cin >> pi;
std::getline(std::cin, name);
std::getline(std::cin, name);
std::cin >> tao;
std::cout << "my name is : " << name << " tao is : " << tao << " pi is : " << pi << '\n';
}
The stream is empty! So it is going to prompt a user for input
3.14
pi
Fabio Ibanez
name
tao

Fix ✅

cin	3	.	1	4	\n	F	a	b	i	o		I	b	a	n	e	z	\n	6	.	2	8	\n

```
void cinGetline() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << name << " tao is : " << tao << " pi is : " << pi << '\n';
}
```

3.14	pi
Fabio Ibanez	name
6.28	tao

That being said

You shouldn’t use **`getline()`** and **`std::cin()`** together because of the difference in how they parse data.

`std::cin()` - leaves the newline in the buffer
`getline()` - gets rid of the newline

Whew that was a lot!

To conclude (Main takeaways):

1. Streams are a general interface to read and write data in programs
2. Input and output streams on the same source/destination type compliment each other!
3. Don't use **`getline()`** and **`std::cin()`** together, unless you *really really* have to!

Acknowledgements

Credit to **Avery Wang's** streams lecture which I took a lot of inspiration from, particularly for formatting and flow.