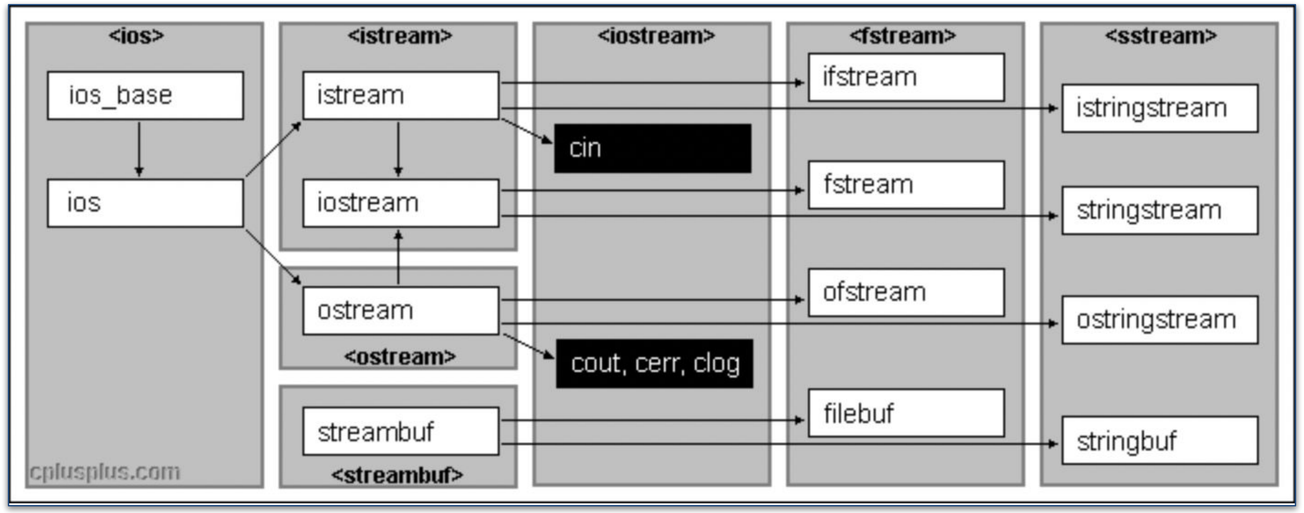
Lecture 4: Streams 

Stanford CS106L, Fall 2025

Rachel Fernandez, Thomas Poimenidis



Plan

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

Attendance 🪄

Slides are available at...

cs106l.stanford.edu

Last Lecture

CS106L Lecture 3:
Initialization & References 

Thomas Poimenidis, Rachel Fernandez

A quick recap

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

A quick recap

1. **Uniform Initialization** 🦄

   A *ubiquitous and safe* way of initializing things using {}

2. **References** 🦄

   A way of giving variables ***aliases*** and having multiple variables all refer the the **same memory.**

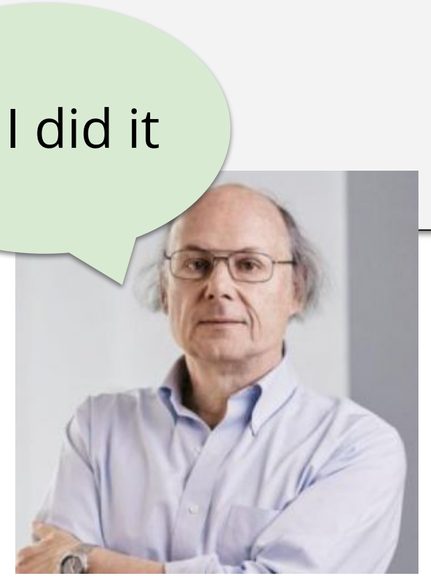
Plan

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++

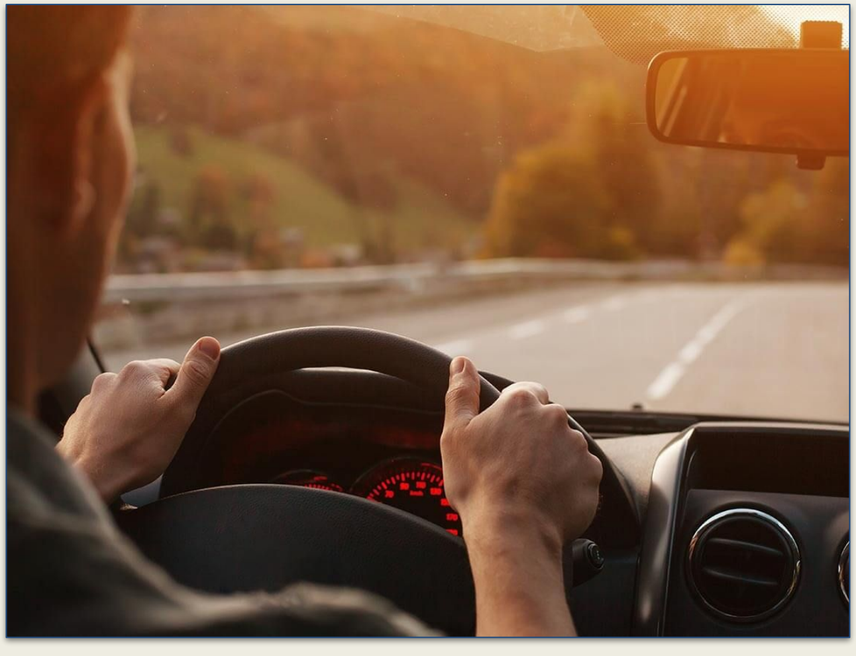
Streams

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

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

Abstractions

Abstraction = hide unnecessary details and expose what is only relevant



Abstractions

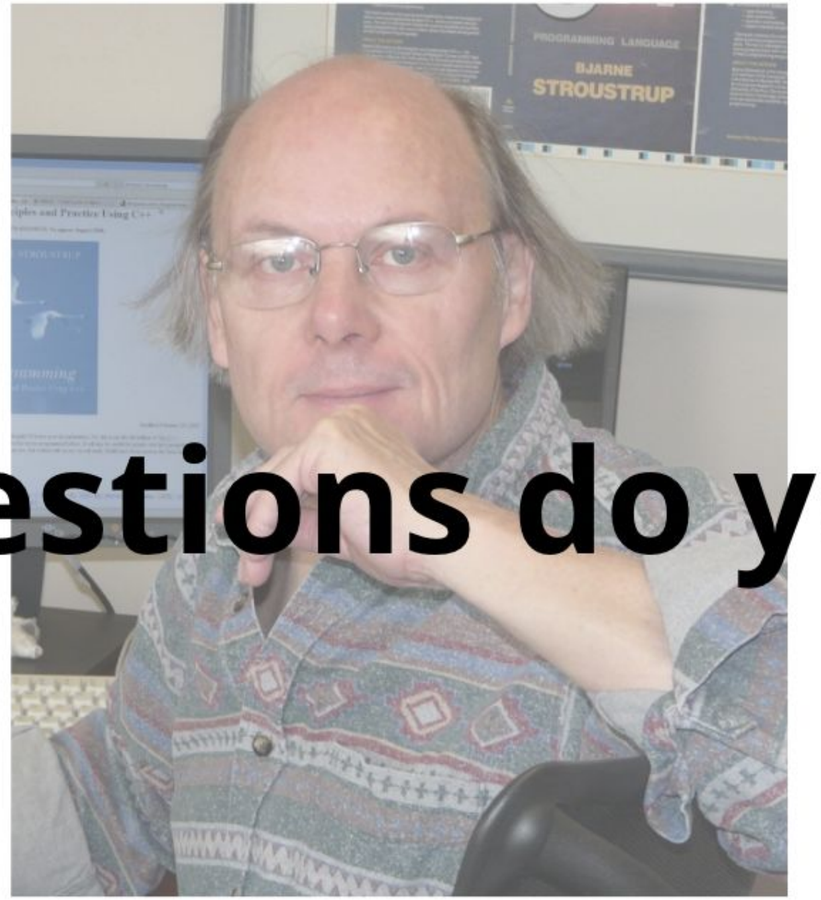
Abstractions provide a consistent ***interface***, and in the case of streams the interface is for reading and writing data!

Abstractions

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

if you are still confused! that is ok :D We will see an example in a few slides!

What questions do you have?



bjarne_about_to_raise_hand

🧠 THE BIG IDEA 🧠

Streams help us read and write data

But what is a stream?

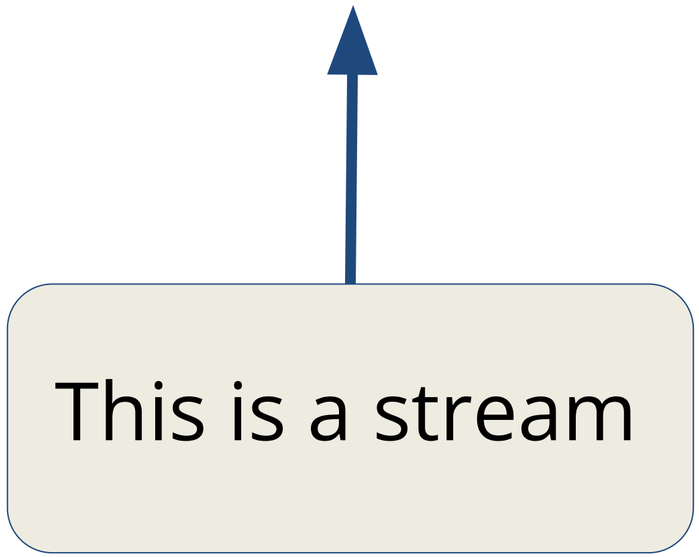
You may not know what a stream is, but chances are **you probably use them all the time!**

A familiar stream!

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

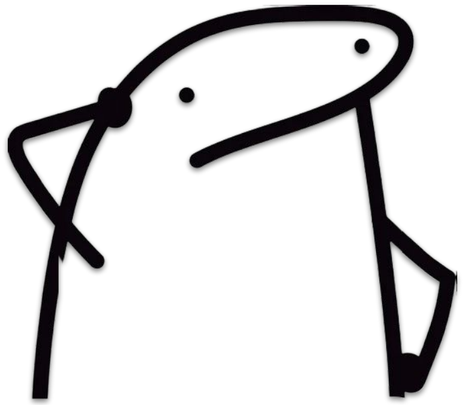
A familiar stream!

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



CS106B examples

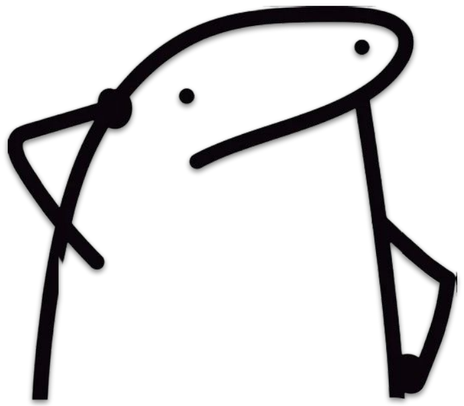
Have you ever taken CS106B and you had to read in files for your assignment(s)? kinda like this..?



```
ifstream in;
openFile(in, my_file);

Vector<std::string> lines = readLines(in);
```

CS106B examples

Have you ever taken CS106B and you had to read in files for your assignment(s)? kinda like this..?



```
ifstream in;
openFile(in, my_file);

Vector<std::string> lines = readLines(in);
```

***Note**: This is using the Stanford library and <u>not the STD</u> but it's still an example of streams*

you were using stream all along!!

More familiar examples

```
std::cout << “hello CS106L!”;

// Allows user to write something into
// student_input
std::string student_input;
std::cin >> student_input;
```

More examples

```
//create a file called “data.txt”
std::ofstream fout(“data.txt”);
fout << “I’m writing to this file”
```

More examples

```
//create a file called “data.txt”
std::ofstream fout(“data.txt”);
fout << “I’m writing to this file”

std::ifstream fin(“data.txt”);
std::string first_word;
//store the first word from the file into
//student_input
fin >> student_input;
```

More examples

```
//create a file called “data.txt”
std::ofstream fout(“data.txt”);
fout << “I’m writing to this file”

std::ifstream fin(“data.txt”);
std::string first_word;
//store the first word from the file into fin
fin >> student_input;
```

```
std::cout << “hello CS106L!”

std::string student_input;
std::cin >> student_input;
```

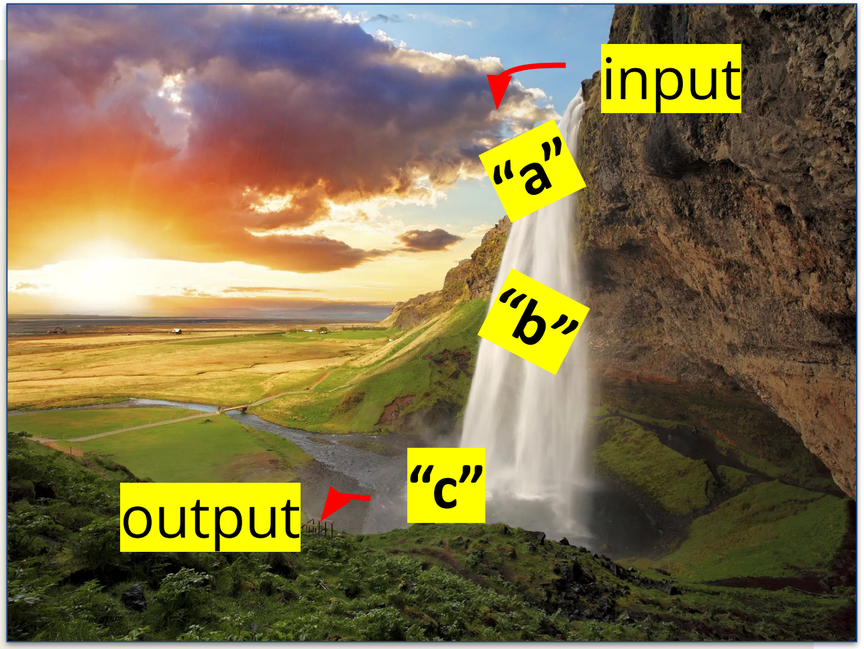
notice the << and >>? That’s abstraction at work! We can use a **consistent interface** to work with **input** and **output**

What are all these types of streams?

ios_base

- `ios_base` is the foundation for everything streams related

- What data does `ios_base` maintain?
  - State Information
  - Control Information
  - These things have to do with making sure our stream is a-ok!



Stream carrying your data

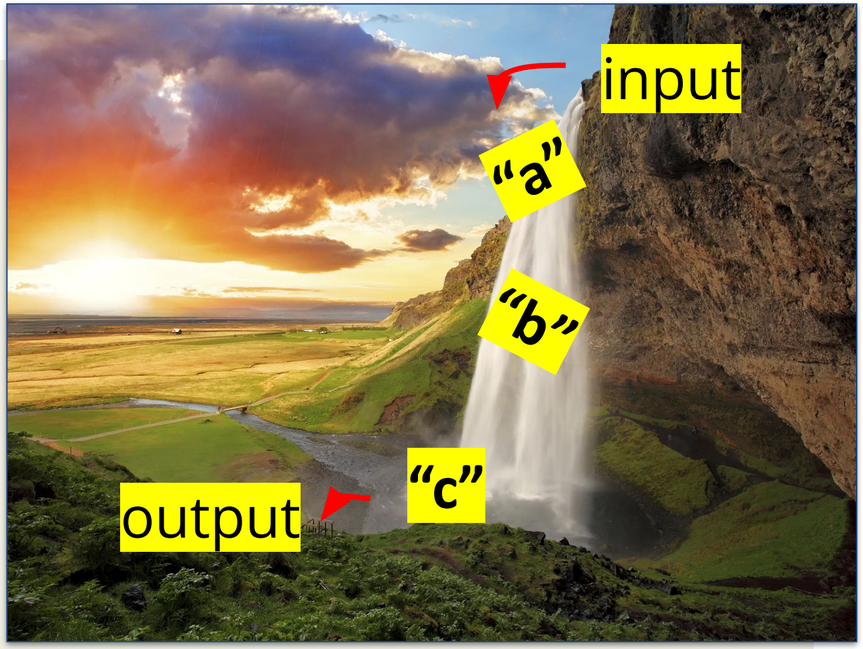
ios_base explained

State Information = flags that tell you the status/health of your stream

1. ex. `failbit` → logical error (ex. type error)
2. ex. `eofbit` → reached end of string

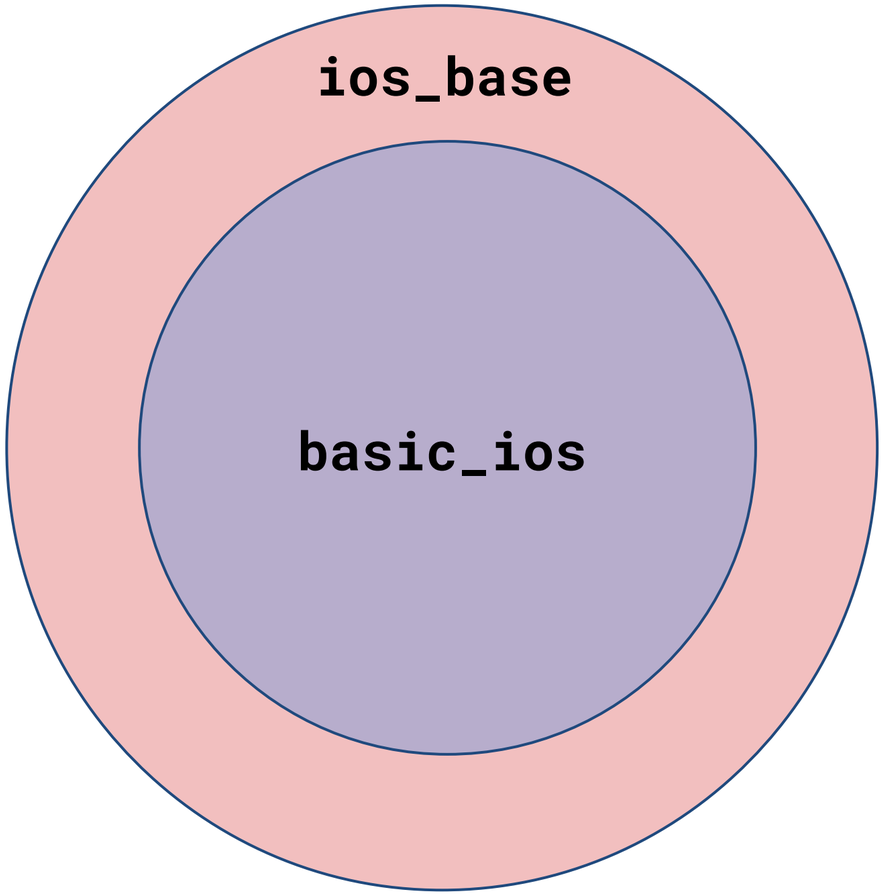
Control Information = how does the stream present the data?

1. ex. should 255 be printed as “255”, “FF” or “377”?



Stream carrying your data

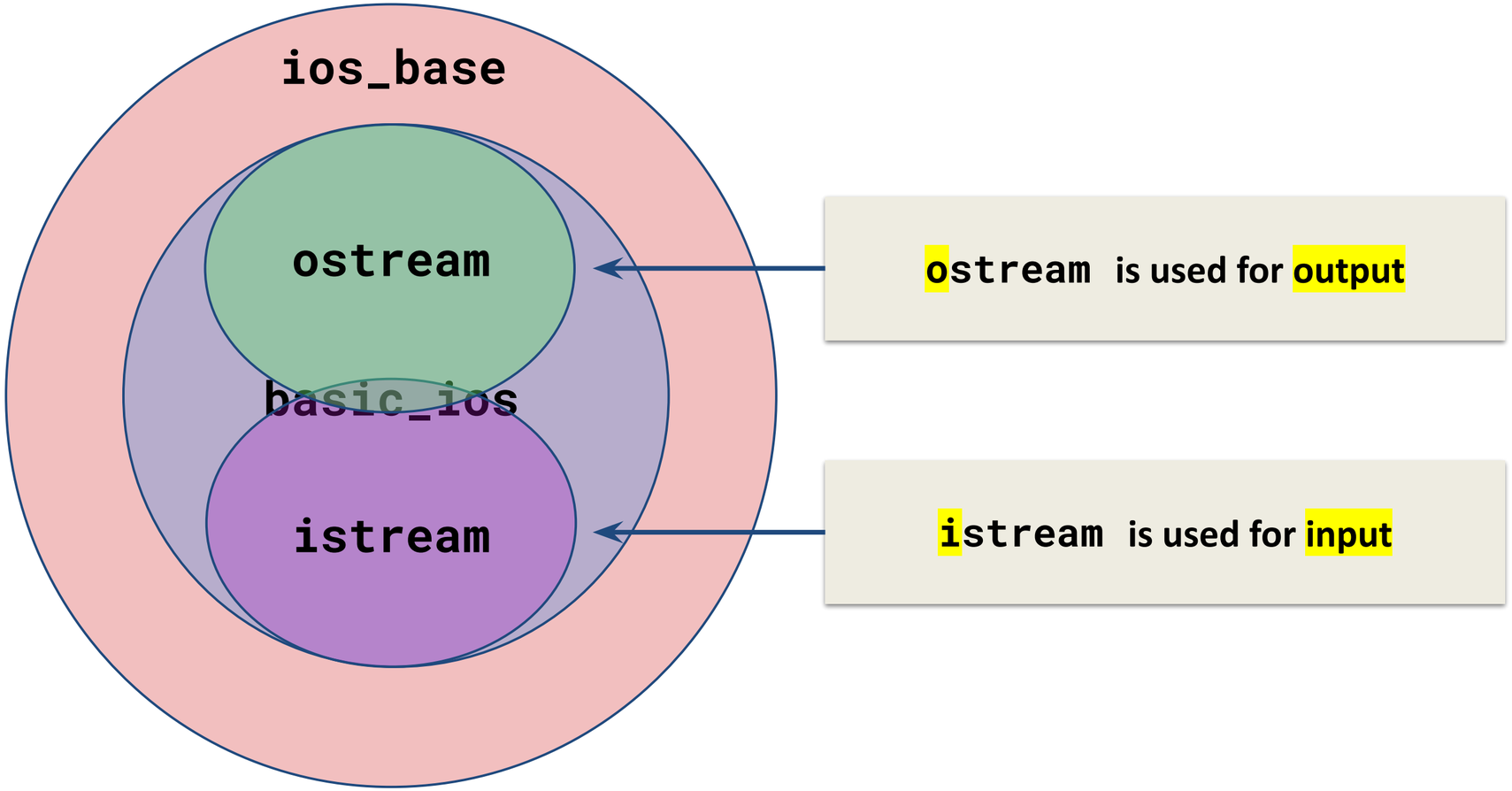
What builds off ios_base?



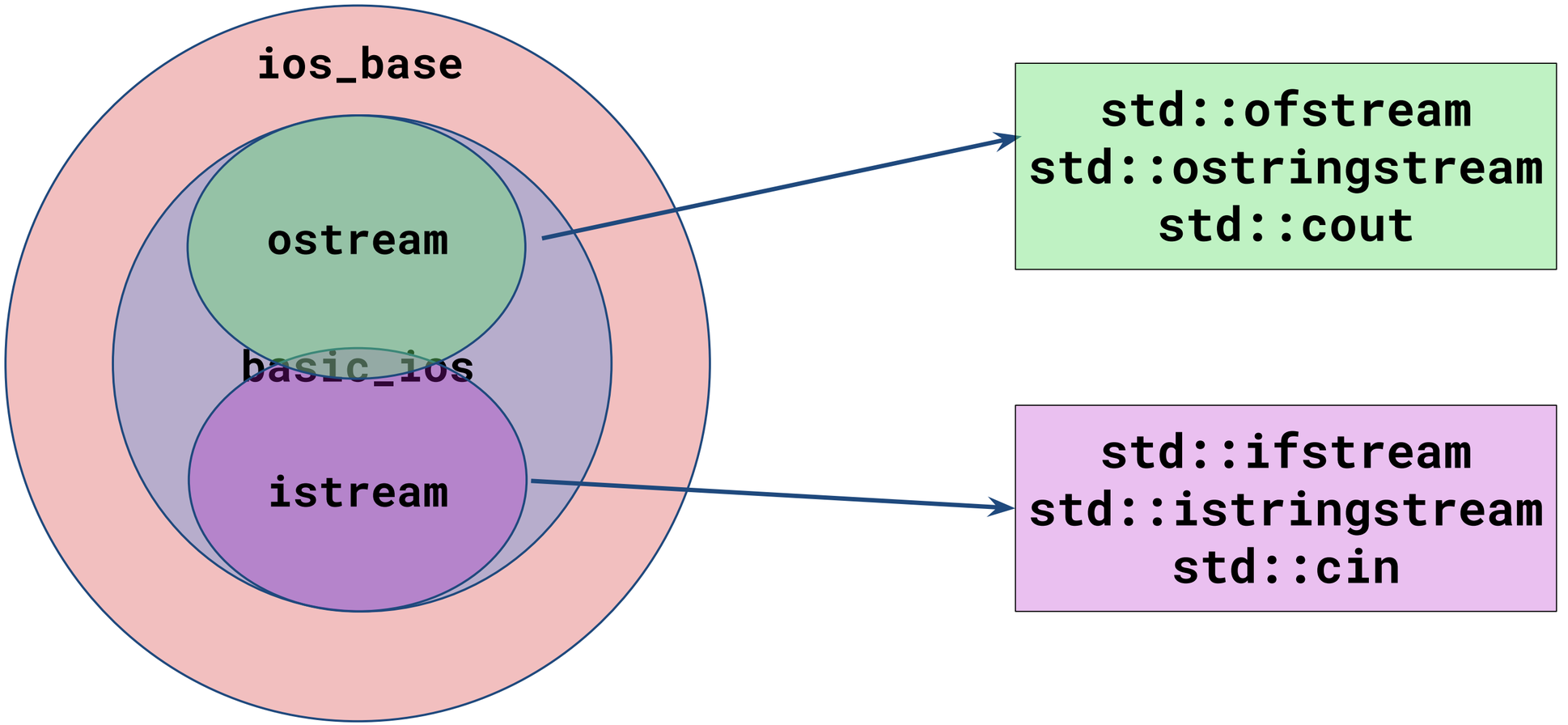
basic_ios ensures the stream is working correctly and where the stream comes from! maybe its the console, keyboard, or a file

more on this later >:D

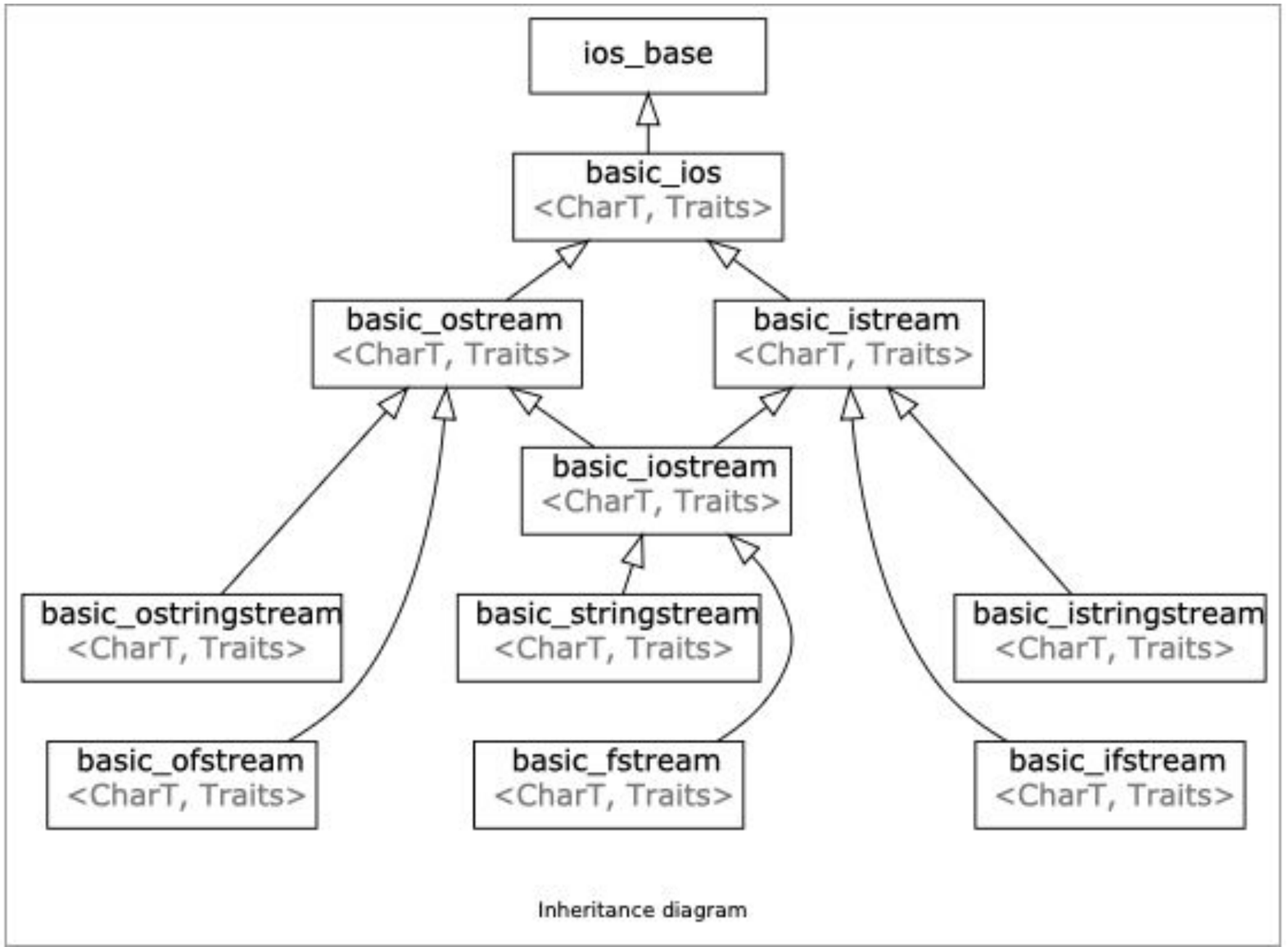
ostream and istream



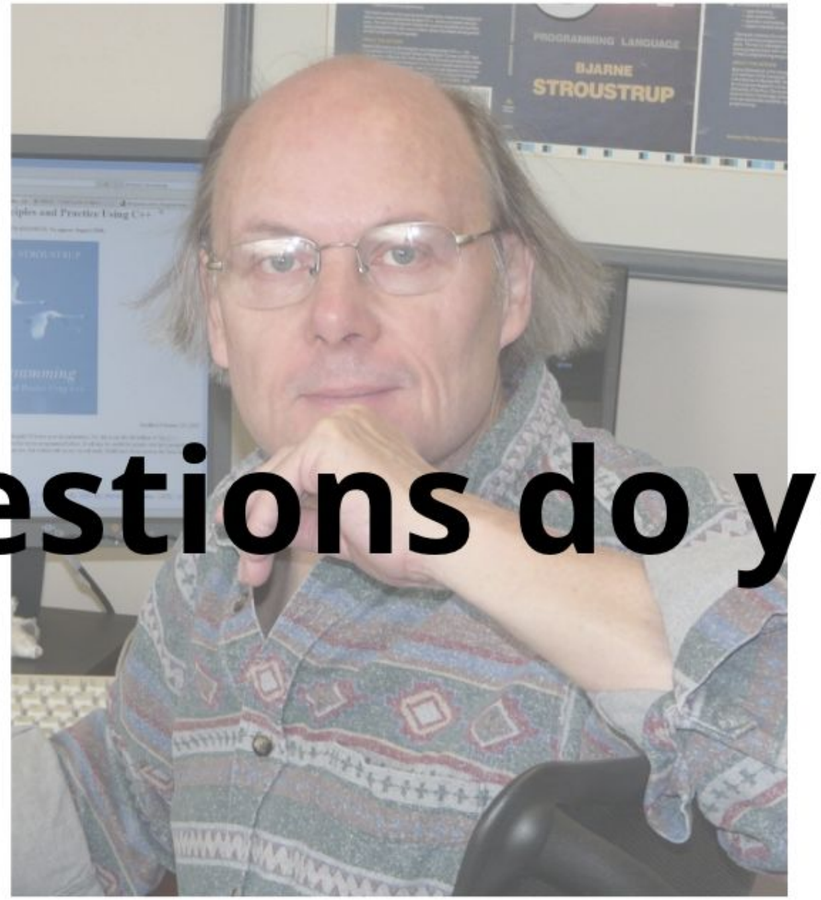
ostream and istream



What streams actually are



What questions do you have?



bjarne_about_to_raise_hand

How does data move from an external source (keyboard or file) into your C++ 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"
```

std::cin

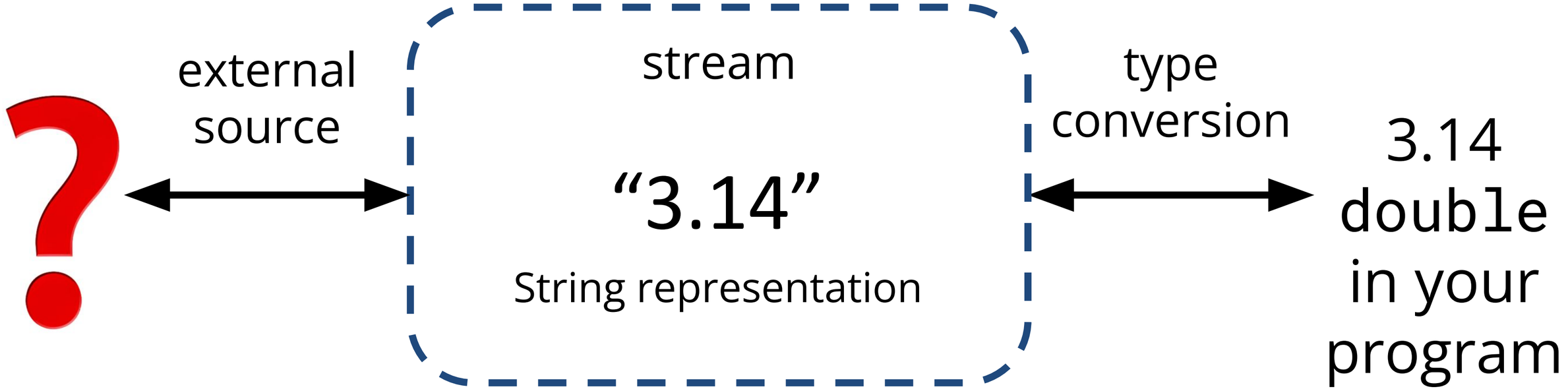
```
int main()
{
  double pi;
  std::cin >> pi;
  /// verify the value of pi!
  std::cout << pi / 2 << '\n';

  return 0;
}
```

Console

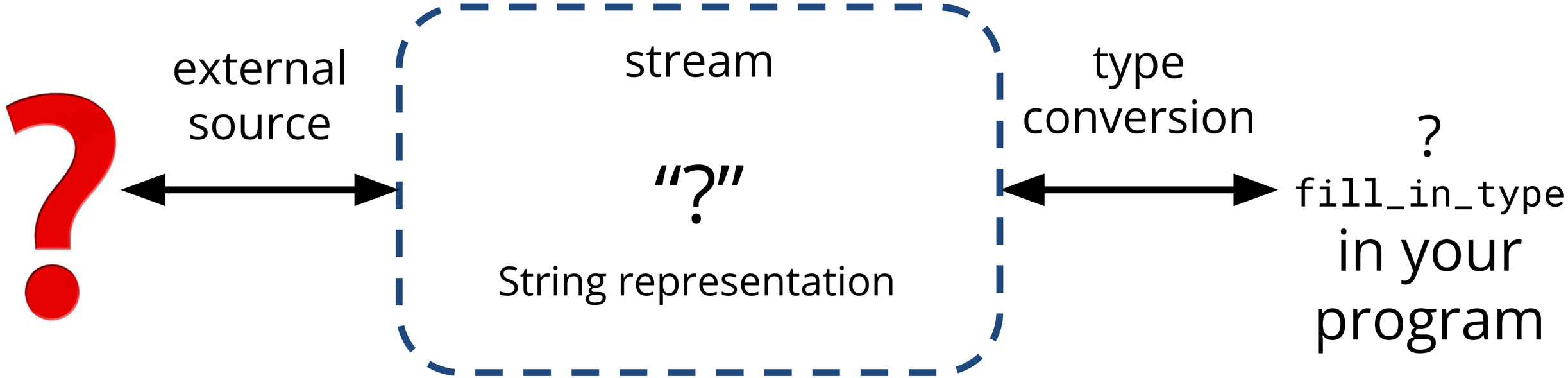
"1.57"

Woah! So we stored a string into a double? is that allowed?? 🤨

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

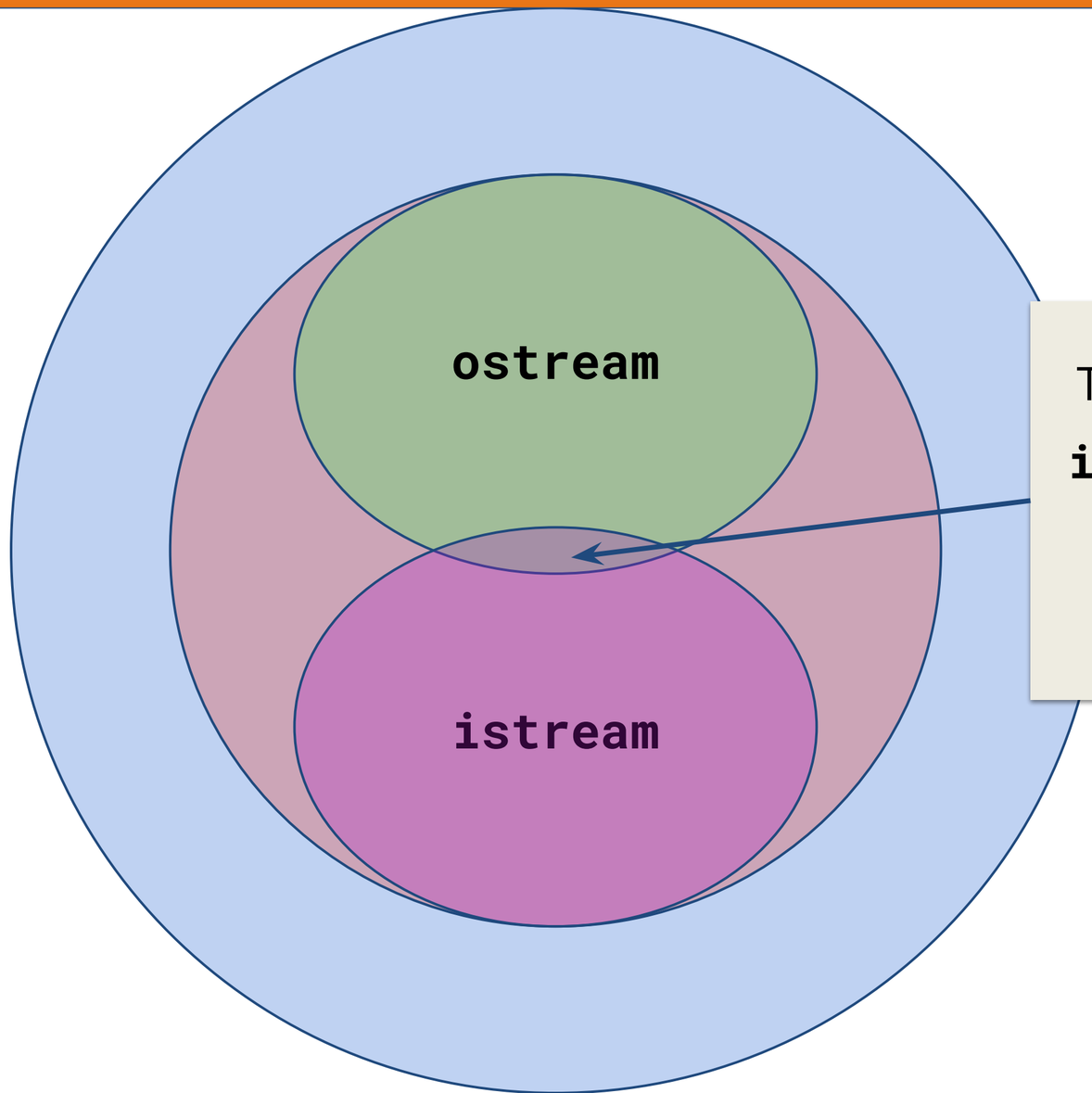
Input streams (I)

- 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)

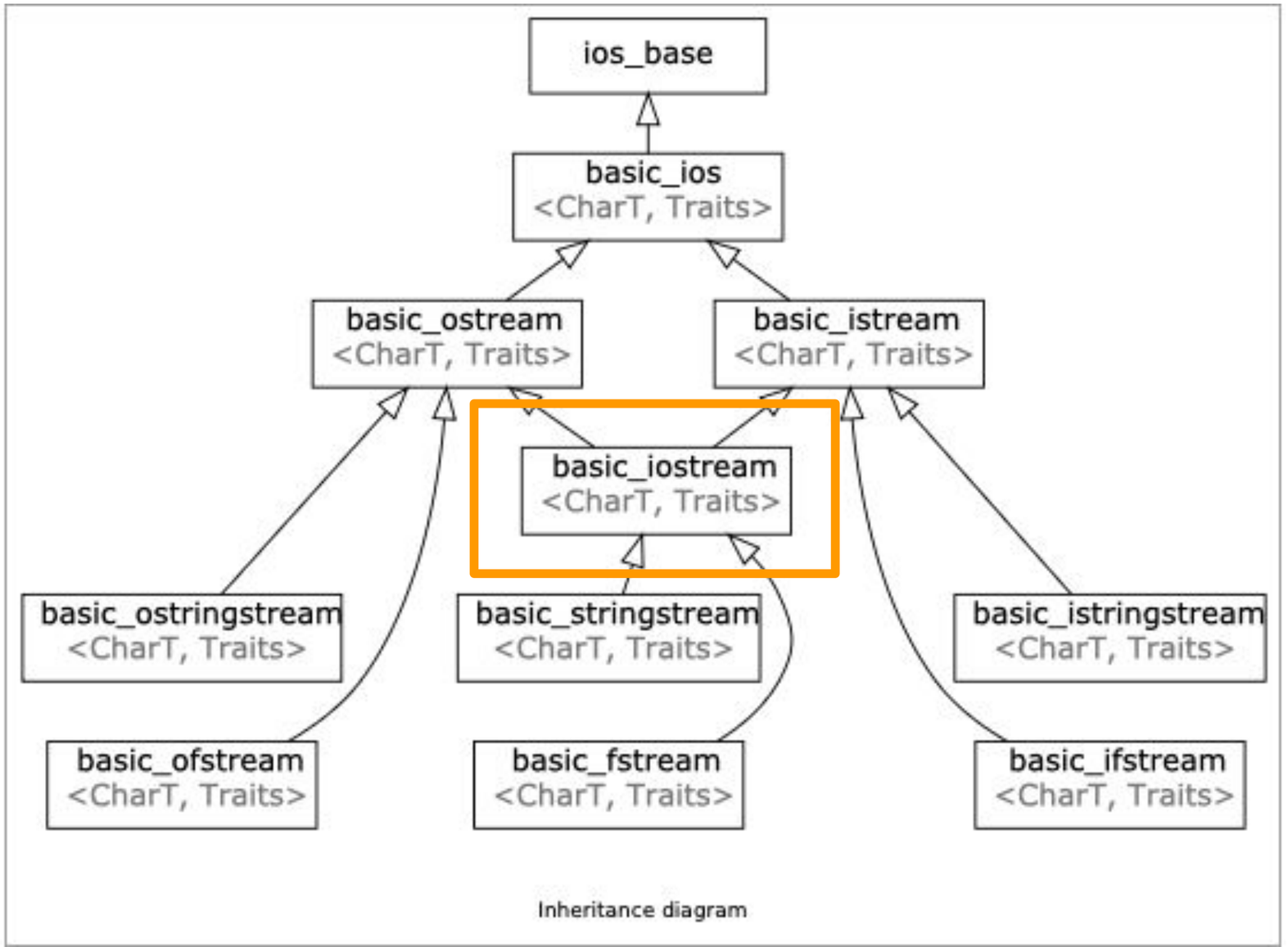
- a way to write data to a destination
  - Are inherited from **`std::ostream`**
  - ex. writing out something to the console (**`std::cout`**)
  - primary operator: **`<<`** (called the insertion operator)

streams and types

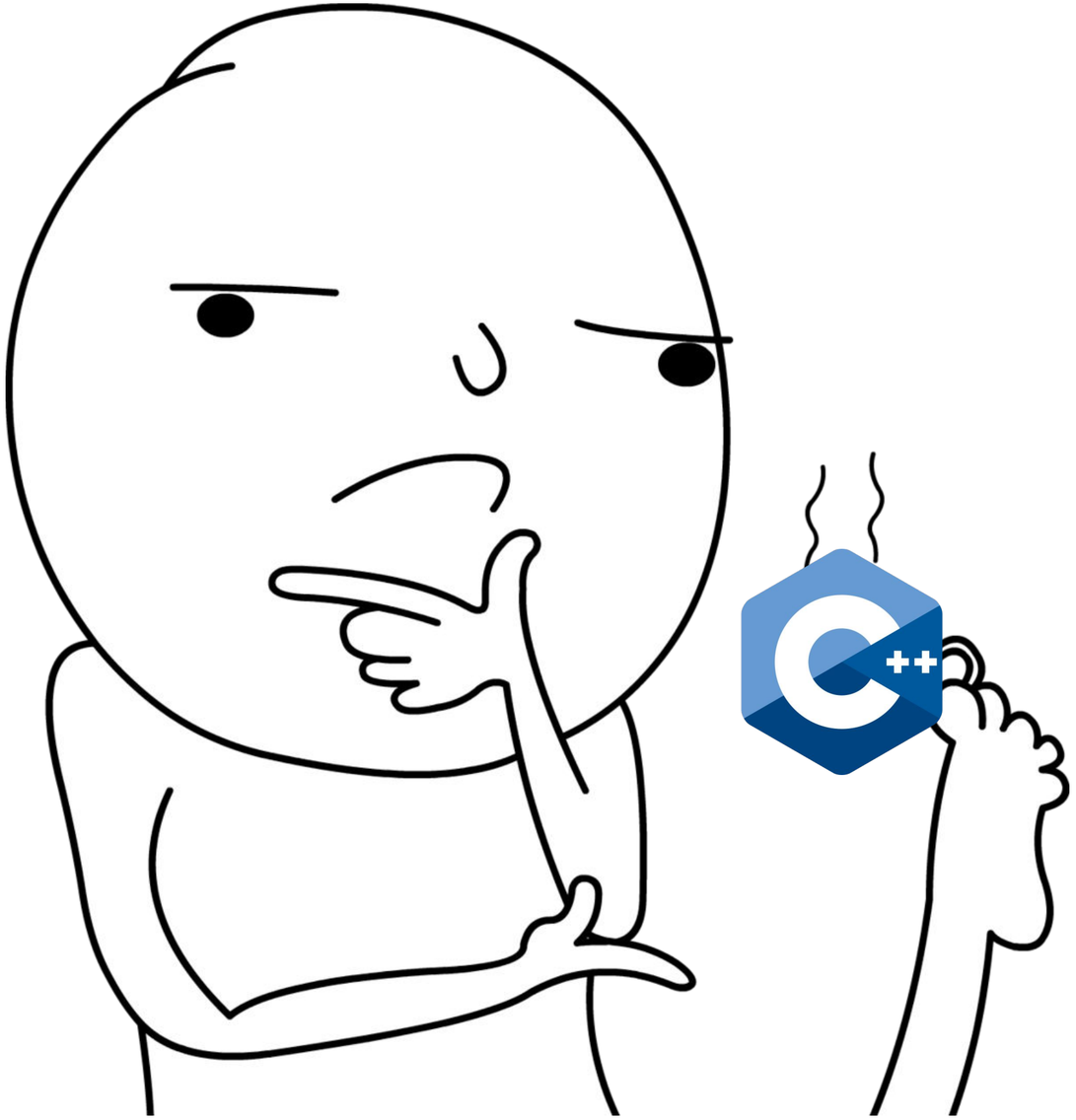


This intersection is known as **`iostream`** which takes has all of the characteristics of **`ostream`** *and* **`istream`**!

What streams actually are



What questions do we have?



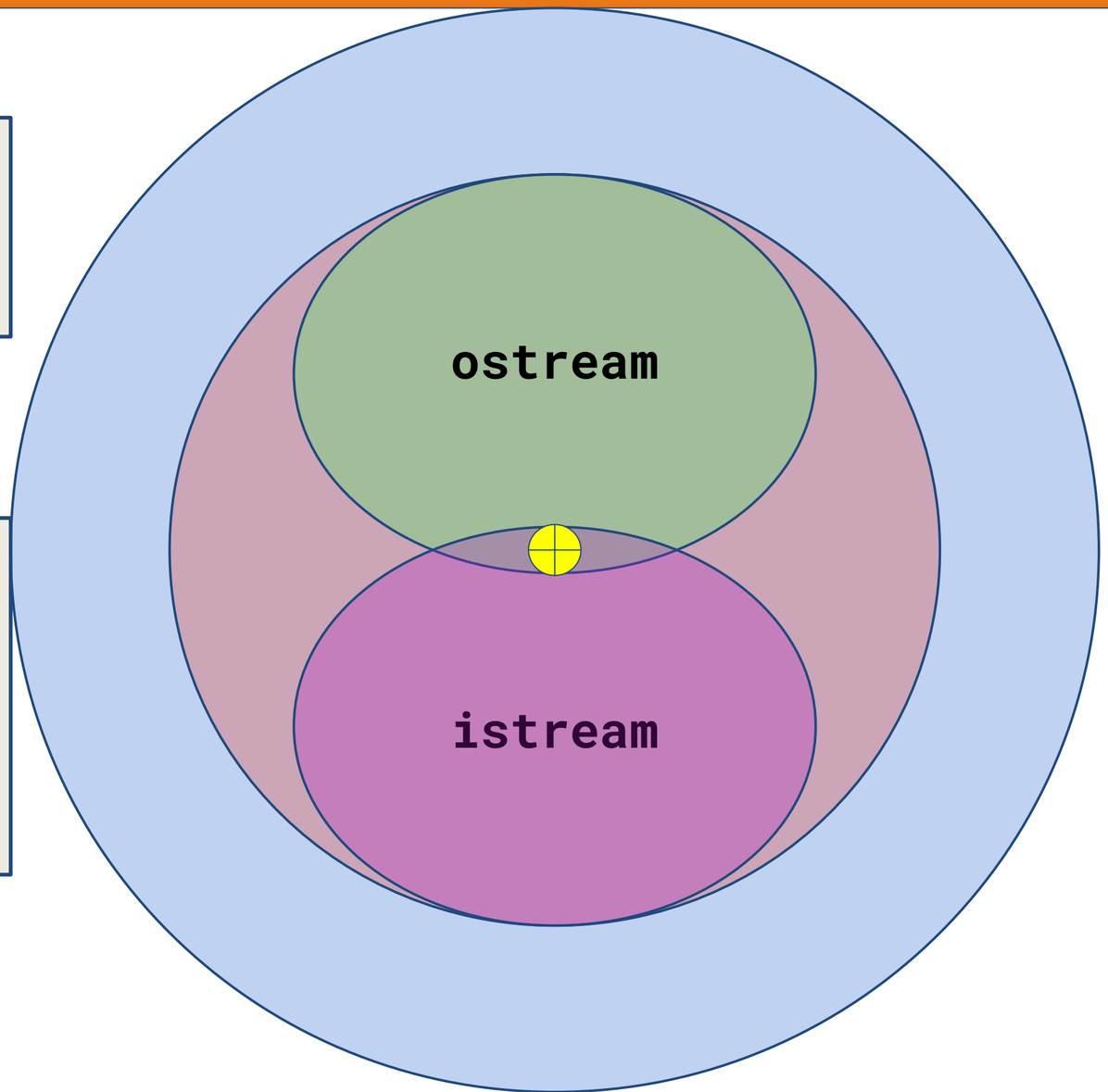
Plan

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

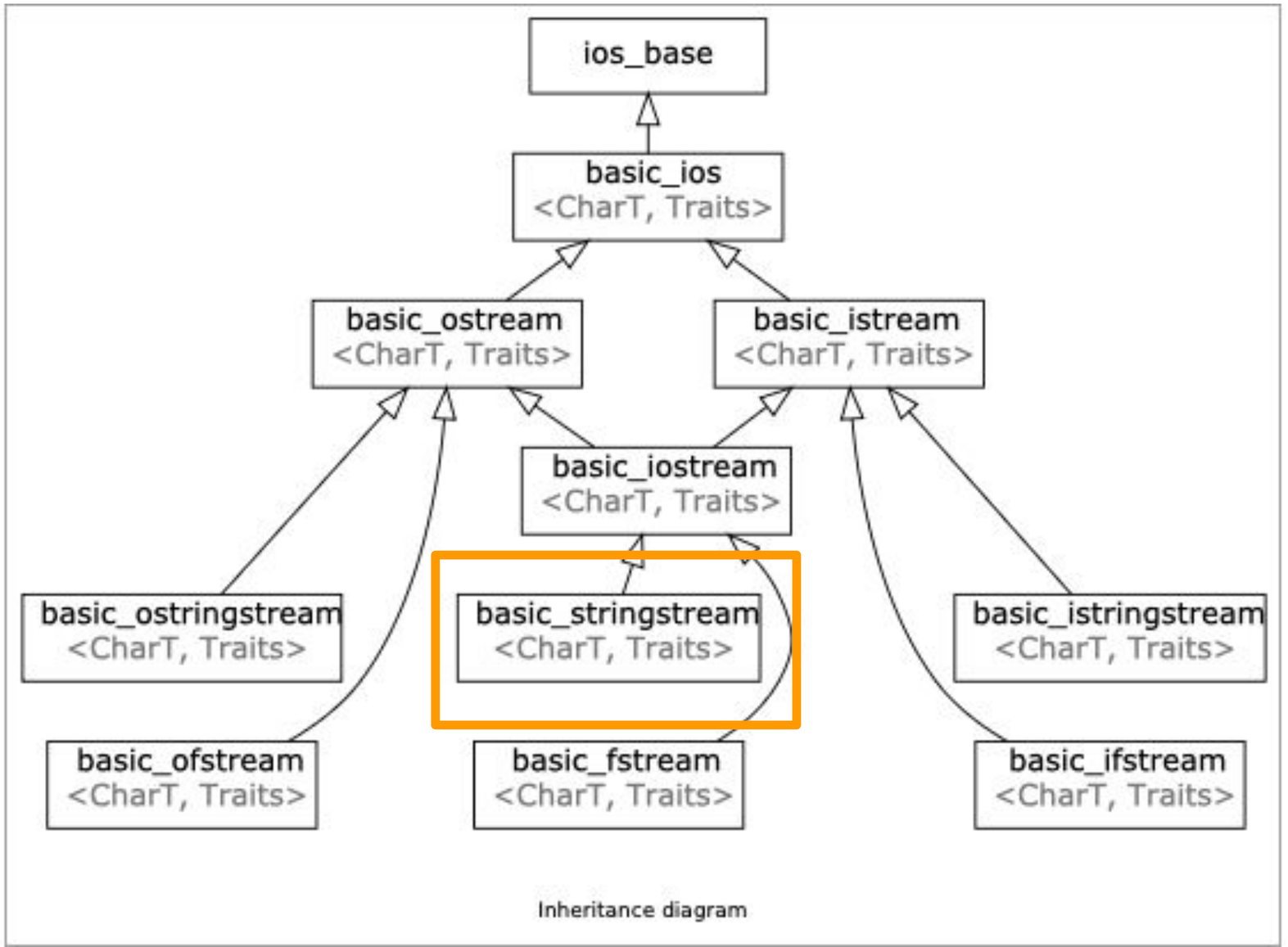
`std::stringstream`

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\n”;

}
```

std::stringstream example

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

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

}
```

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\n”;

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

}
```

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

std::stringstream example

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

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

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

}
```

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\n”;

  /// 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;

}
```

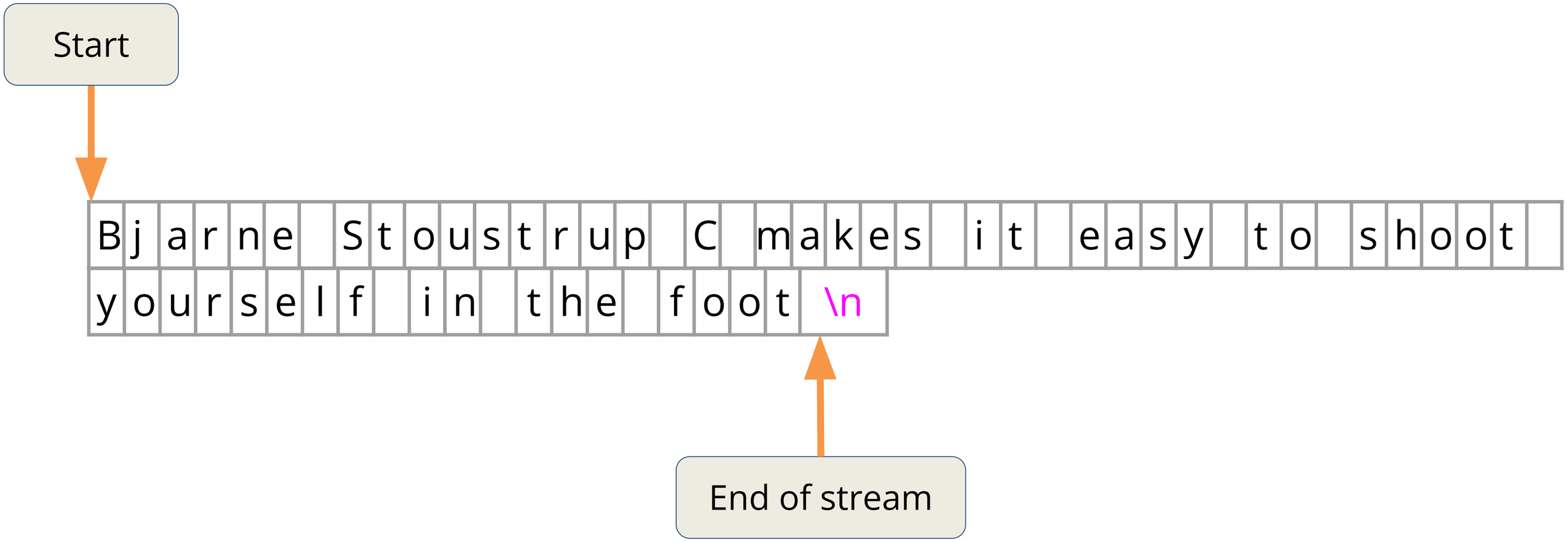
initialize `stringstream` with string constructor

std::stringstream example

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

  /// 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

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\n”;

  /// 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

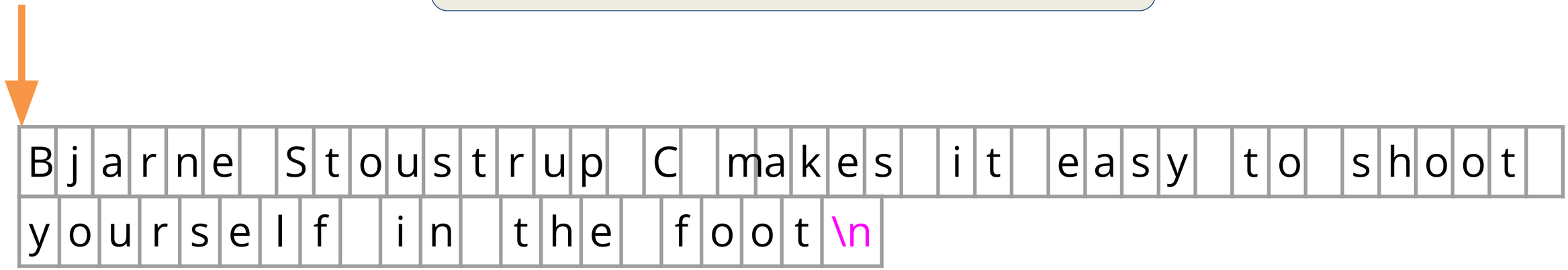
std::stringstream example

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

  /// 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;
```



First

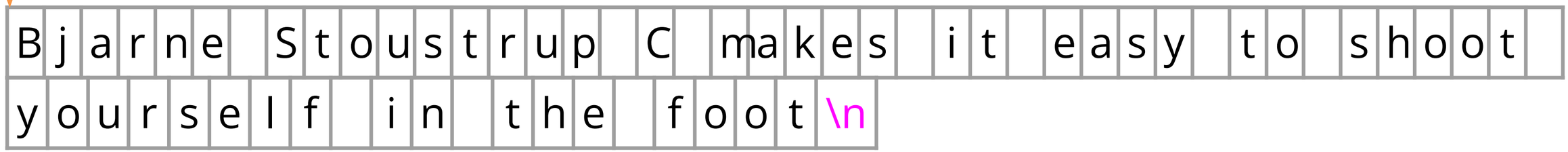
Last

Language

what the stream looks like!

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

extracts from the stream!



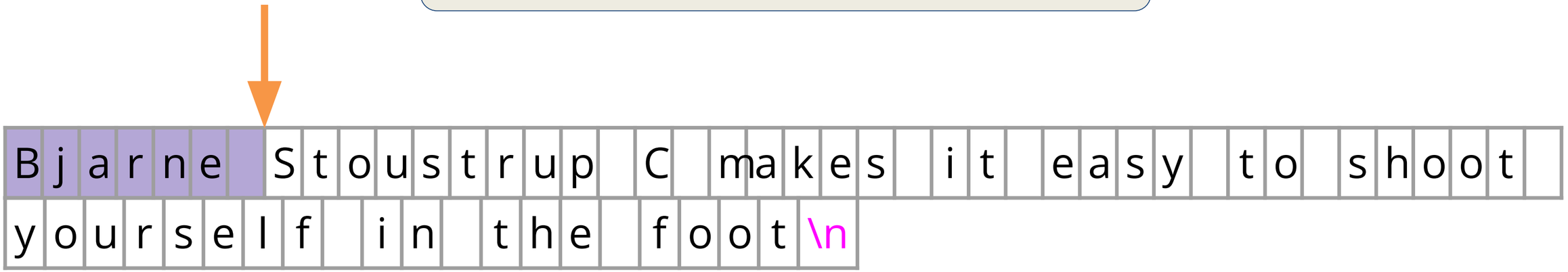
First

Last

Language

what the stream looks like!

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



Bjarne

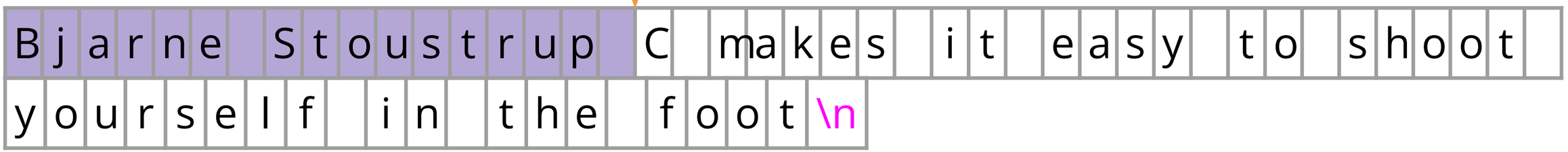
First

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

First

Stroustrup

Last

C

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!

Problem:
?

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

Bjarne

Stroustrup

C

First

Last

Language

what the stream looks like!

```
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`’.

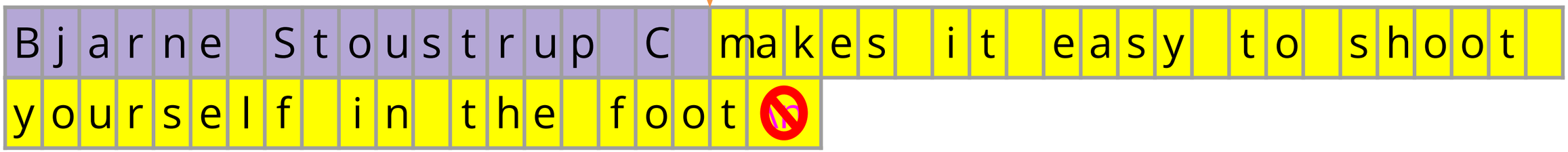
Use `getline()`!

```
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()`** ***consumes*** the `delim` character!
  - PAY ATTENTION TO THIS :)

use std::getline()!

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



Bjarne

First

Stroustrup

Last

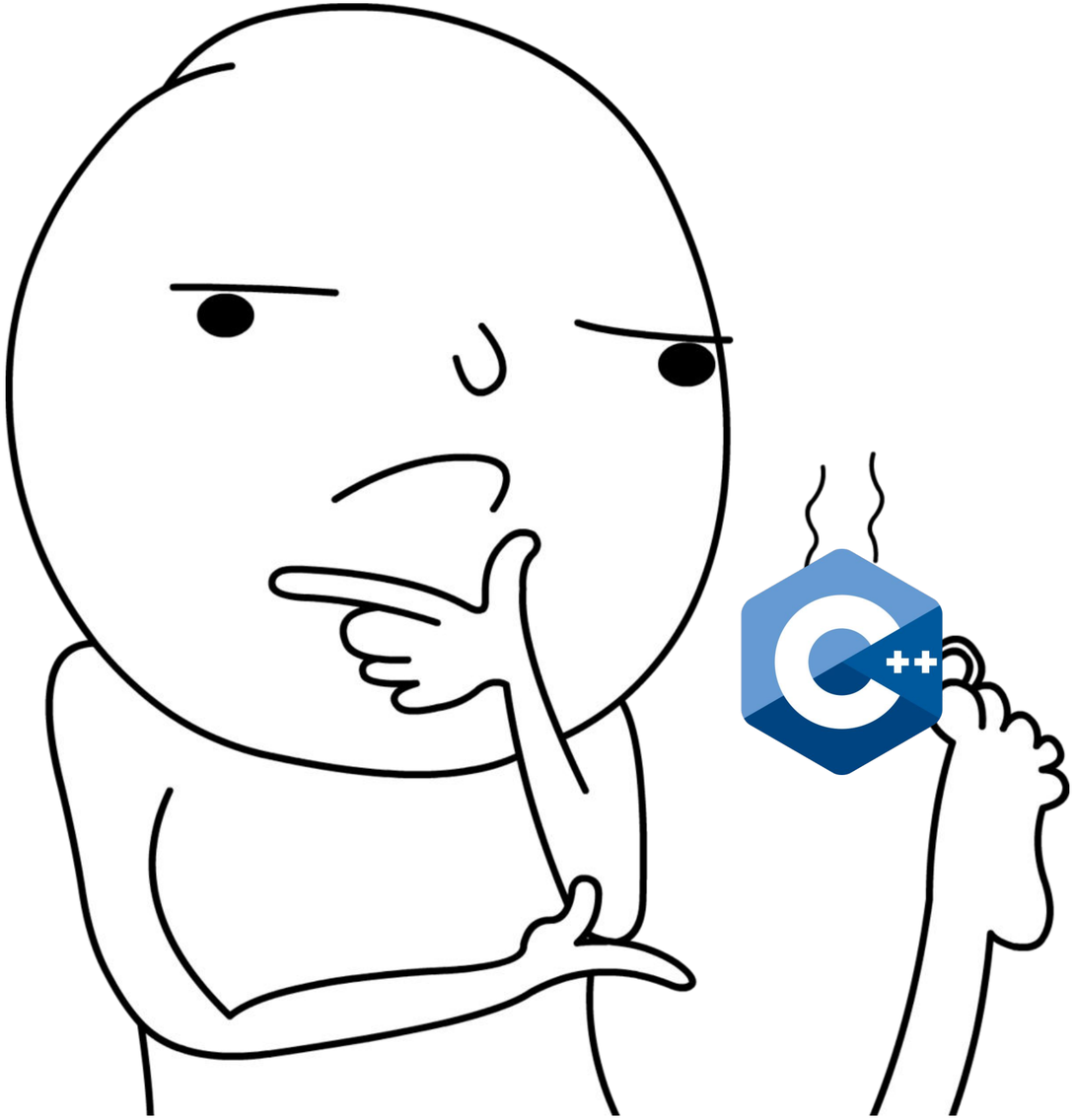
C

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

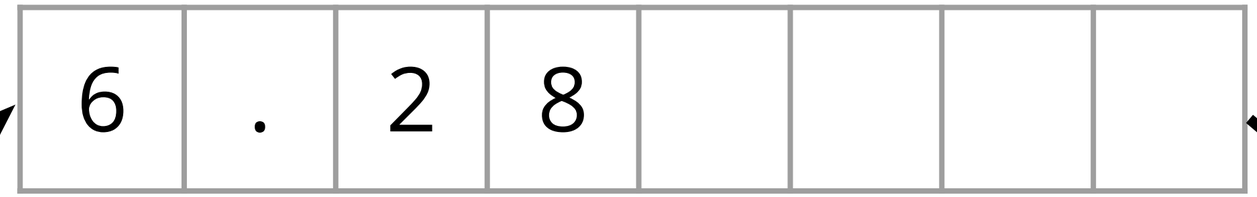
1. Quick recap
2. What are streams??!!
3. `stringstreams!`
4. **`cout` and `cin`**
5. Output streams
6. Input streams

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

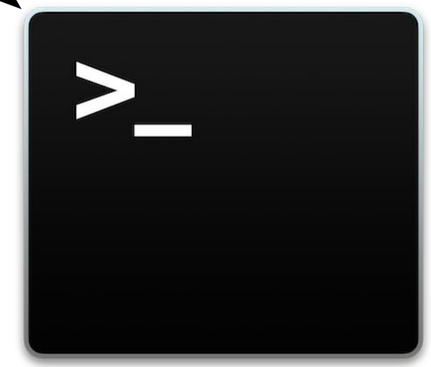
Zooming in on Output Streams!

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

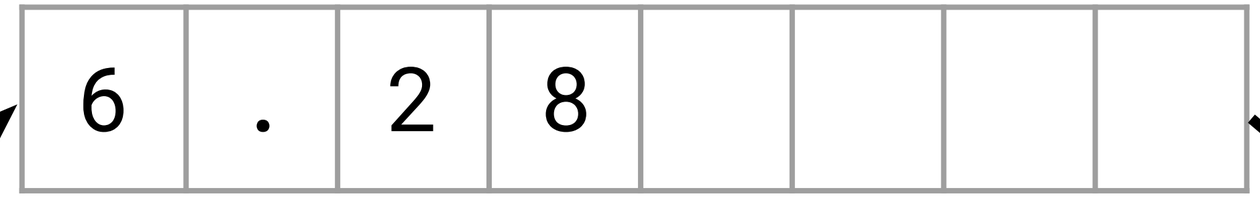


Flush

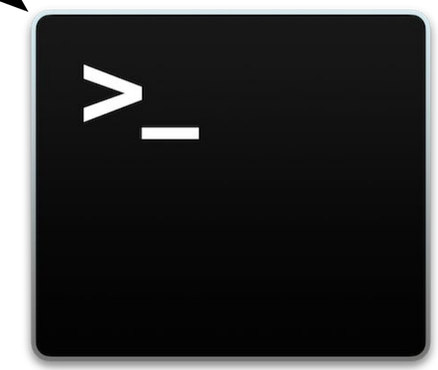
```
double tao = 6.28;
std::cout << tao;
```



Zooming in on Output Streams!



Flush



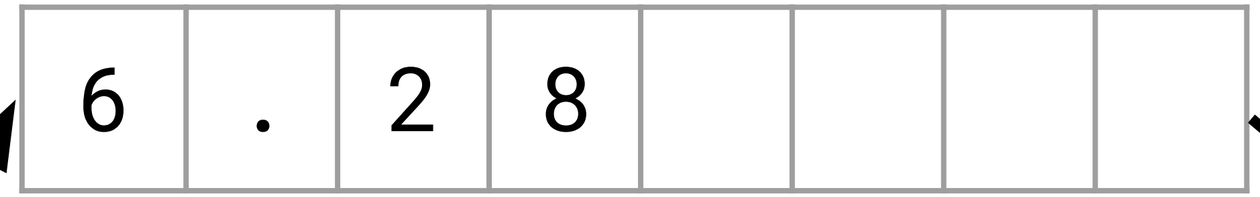

```
double tao = 6.28;
std::cout << tao;
```

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

When do we flush?

- `std::cout << std::flush`
- `std::cout << std::endl`
- When you reach the **end of your program**
- When the buffer is **full**
- When **tied streams interact** (ie. cout has to flush before you take input via cin)

Zooming in on Output Streams!

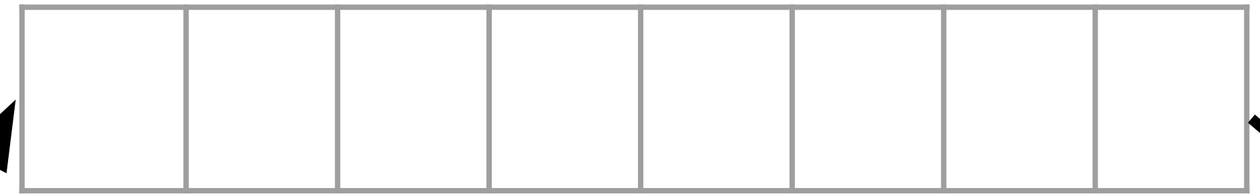


Flush

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

Console

Zooming in on Output Streams!



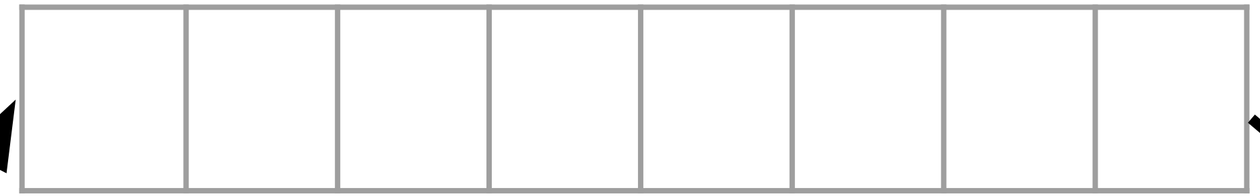
Flush

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

“6.28”

Console

Zooming in on Output Streams!



Flush

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

“6.28”

Console

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"

Recall

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

  `cerr`: used to output errors (unbuffered)

  - sends errors out **IMMEDIATELY**

  `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.

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.

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

Another Caveat

This only works if your output stream is non-interactive!

We tested this ‘std::ios::sync_with_stdio(false)’ proposed solution on various output streams, and found out that it only stopped flushing ‘\n’s when the output stream was non-interactive (i.e. file, Unix pipe).

However, if the output stream was interactive (i.e. terminal), the output stream still interpreted it as a line buffer, resulting in an immediate flush when ‘\n’ was pushed to the stream.

Yeah… it’s weird

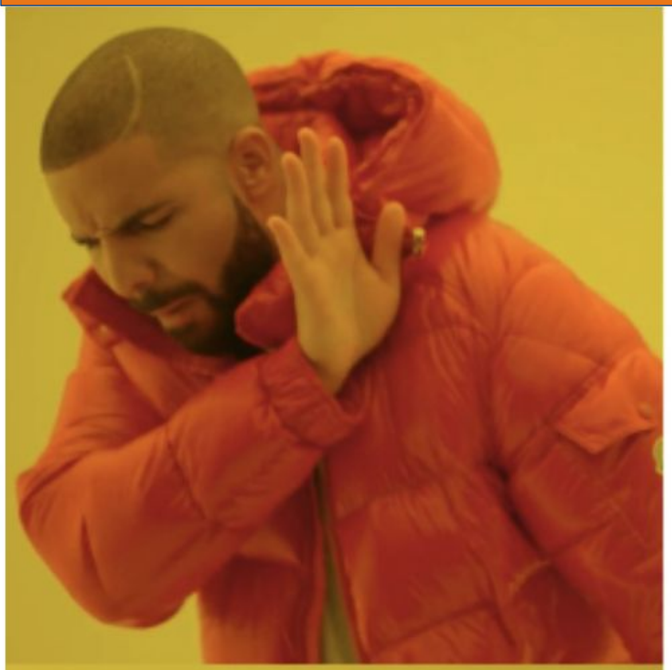
Sometimes you have to do some digging around to figure out what works and what doesn’t :)

Part of working with this language.


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!

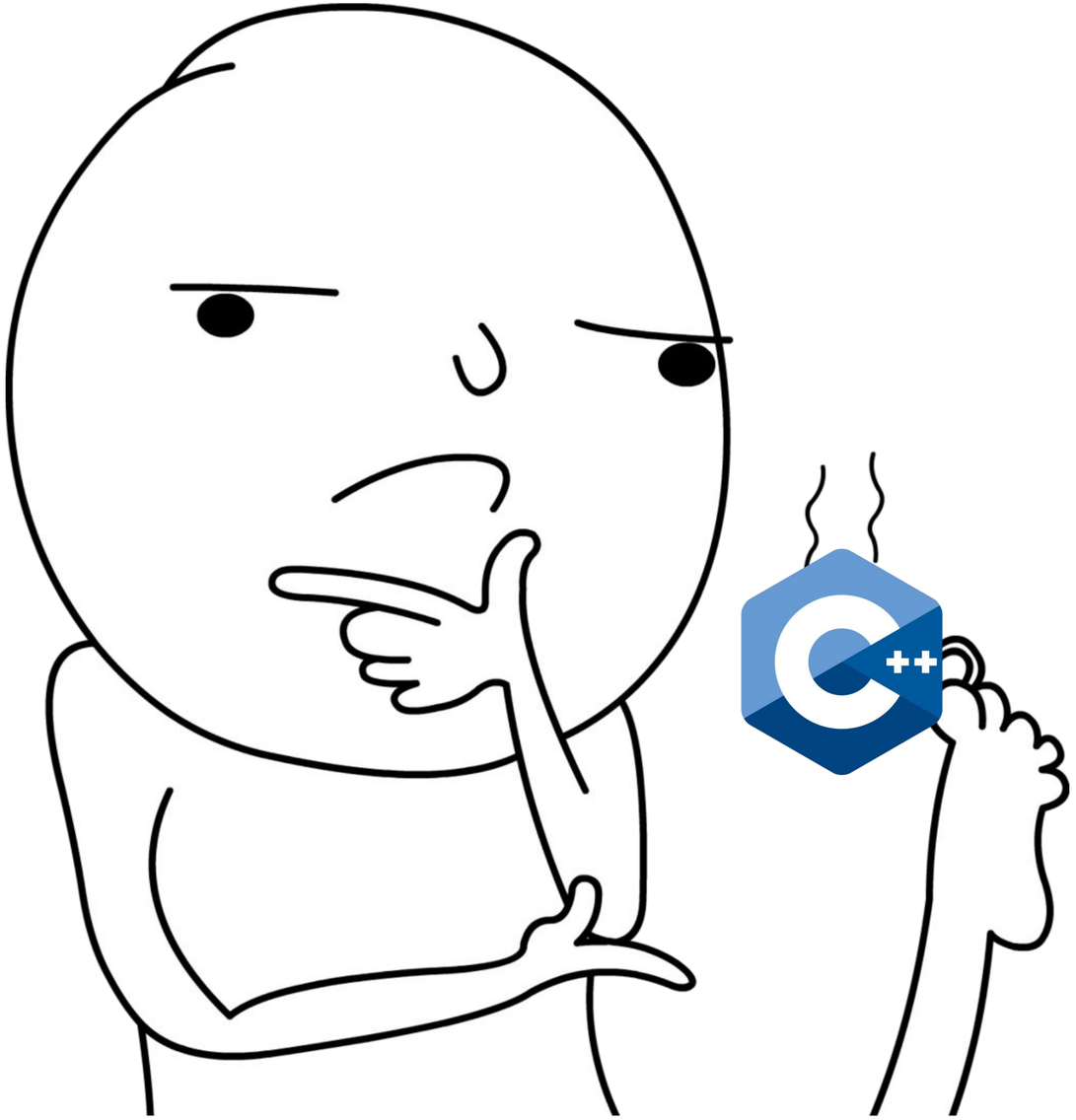
Use ‘\n’!



```
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");
```

Output File Streams

```
int main() {
  /// associating file on construction
  std::ofstream ofs("hello.txt");
```

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';
  }
```

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();
```

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";
```

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

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

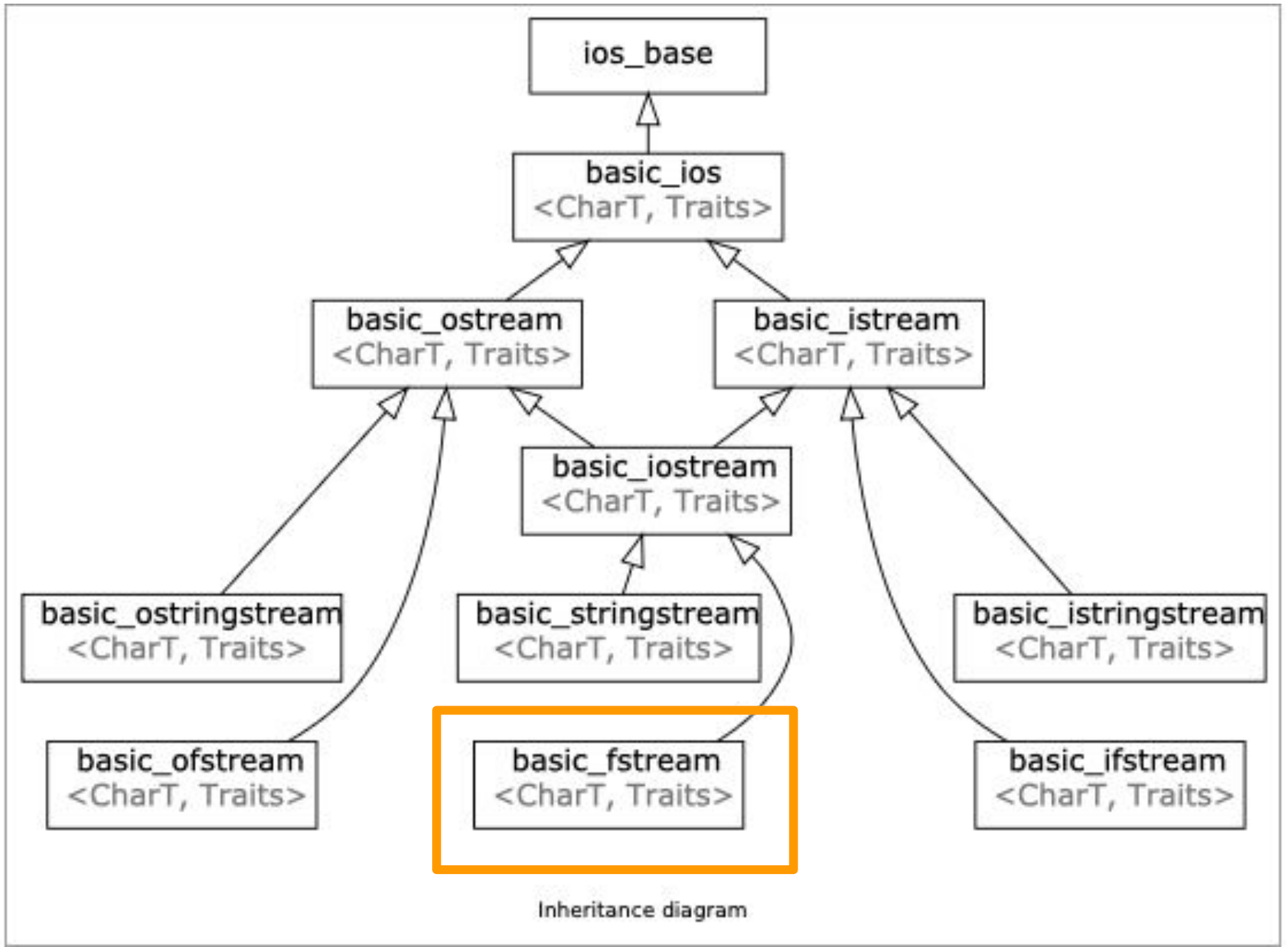
```
int inputFileStreamExample() {
  std::ifstream ifs("input.txt");
  if (ifs.is_open()) {
      std::string line;
      std::getline(ifs, line);
      std::cout << "Read from the file: " << line << '\n';
  }
  if (ifs.is_open()) {
      std::string lineTwo;
      std::getline(ifs, lineTwo);
      std::cout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

Input File Streams

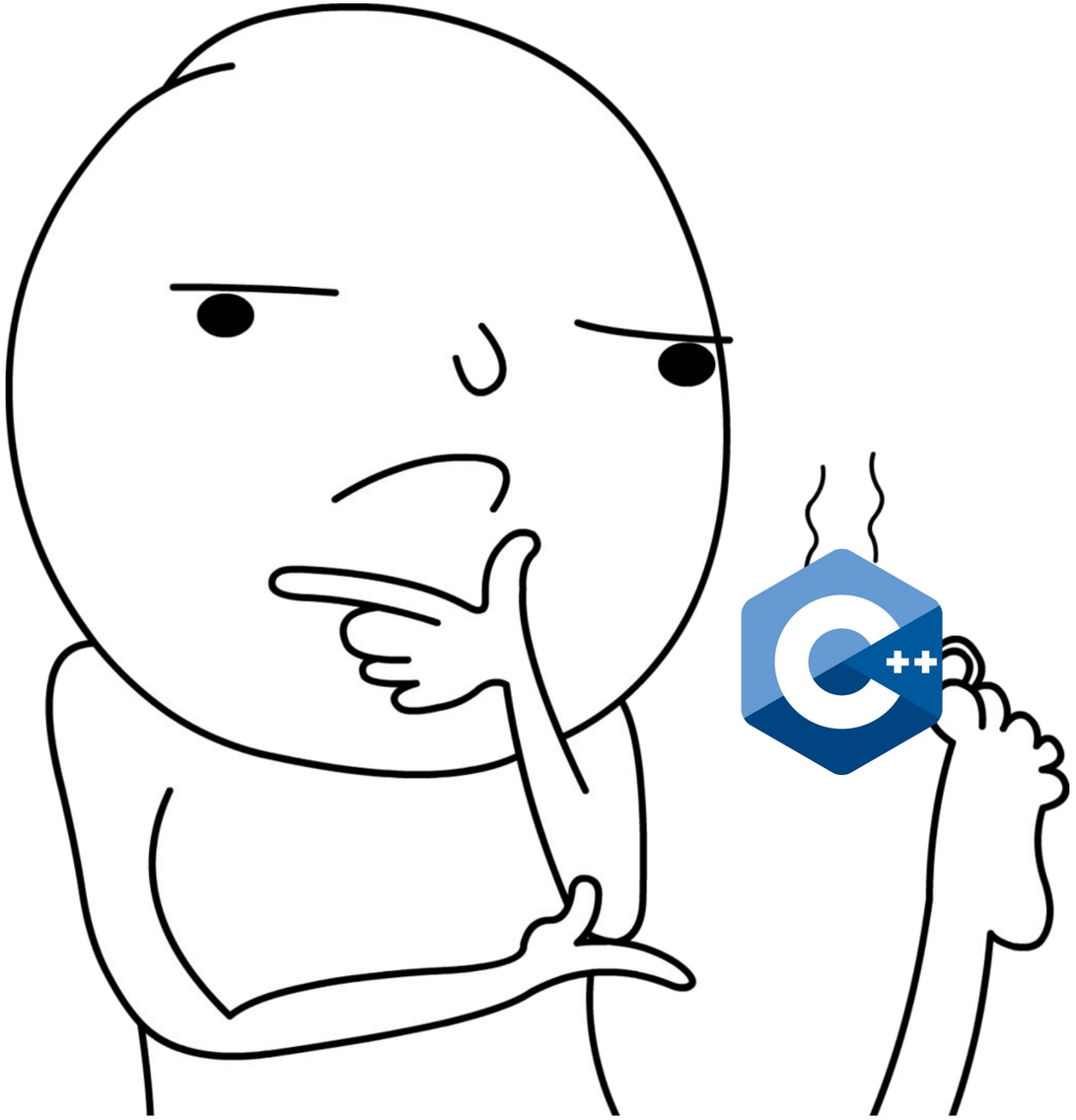
```
int inputFileStreamExample() {
  std::ifstream ifs("input.txt");
  if (ifs.is_open()) {
      std::string line;
      std::getline(ifs, line);
      std::cout << "Read from the file: " << line << '\n';
  }
  if (ifs.is_open()) {
      std::string lineTwo;
      std::getline(ifs, lineTwo);
      std::cout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


```
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”

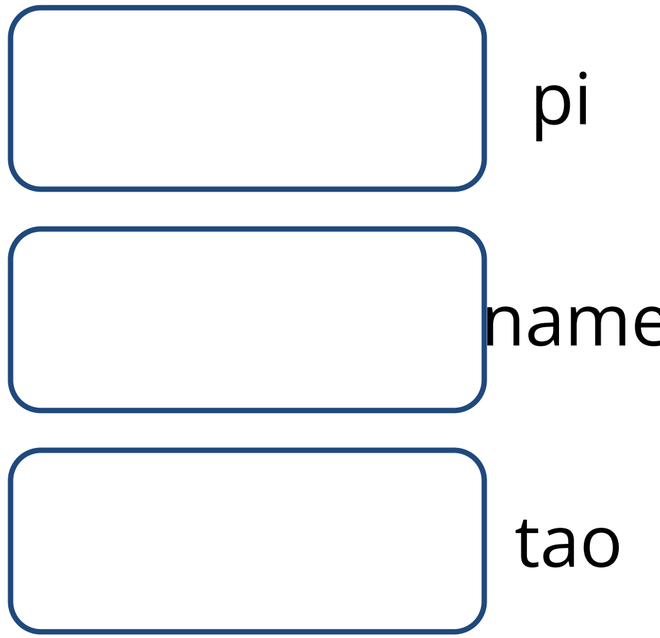
Alternatively


```
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!

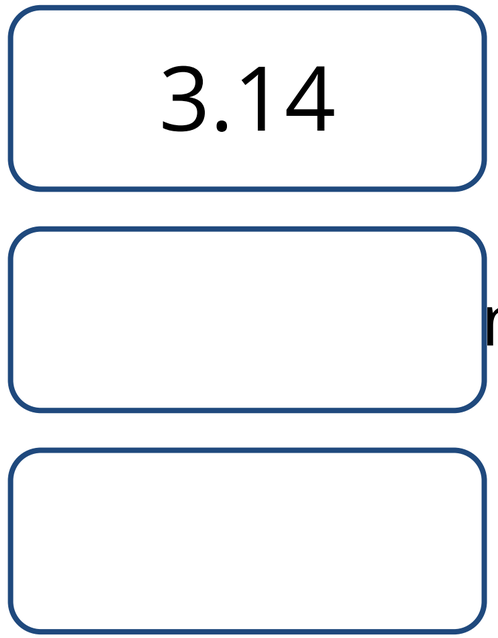

```
int main()
{
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::cin >> name;
  std::cin >> tao;
  std::cout << “my name is: “ << name <<
  “ tao is: “ << tao << “ pi is: “ << pi << ‘\n’;
  return 0;
}
```



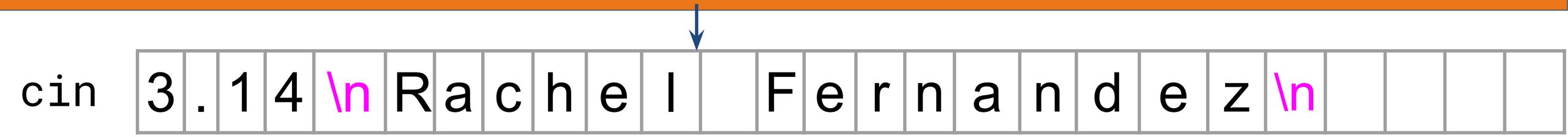
When std::cin fails!


```
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**

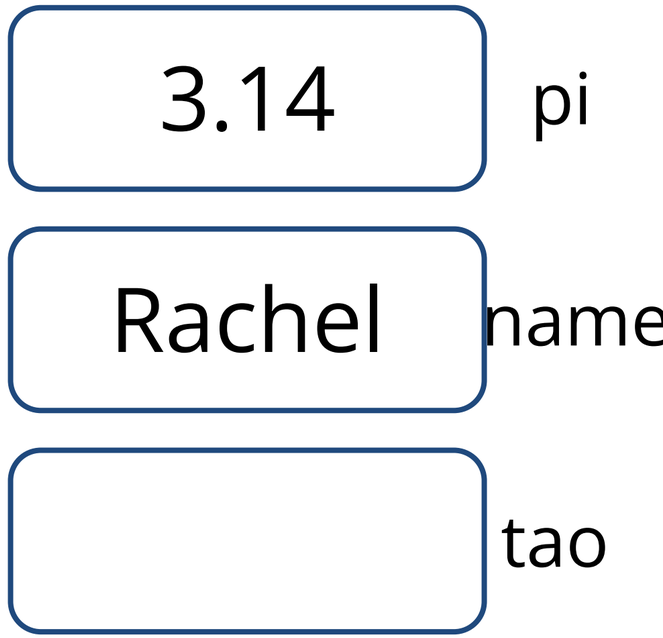


When std::cin fails!




```
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**



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;
}
```

Notice that cin ***only*** reads until the next whitespace

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

3.14	pi
Rachel	name
	tao

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: “
  “ tao is: “ << tao << “ pi is: “ << pi << ‘\n’;
  return 0;
}
```

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

3.14	pi
Rachel	name
?	tao

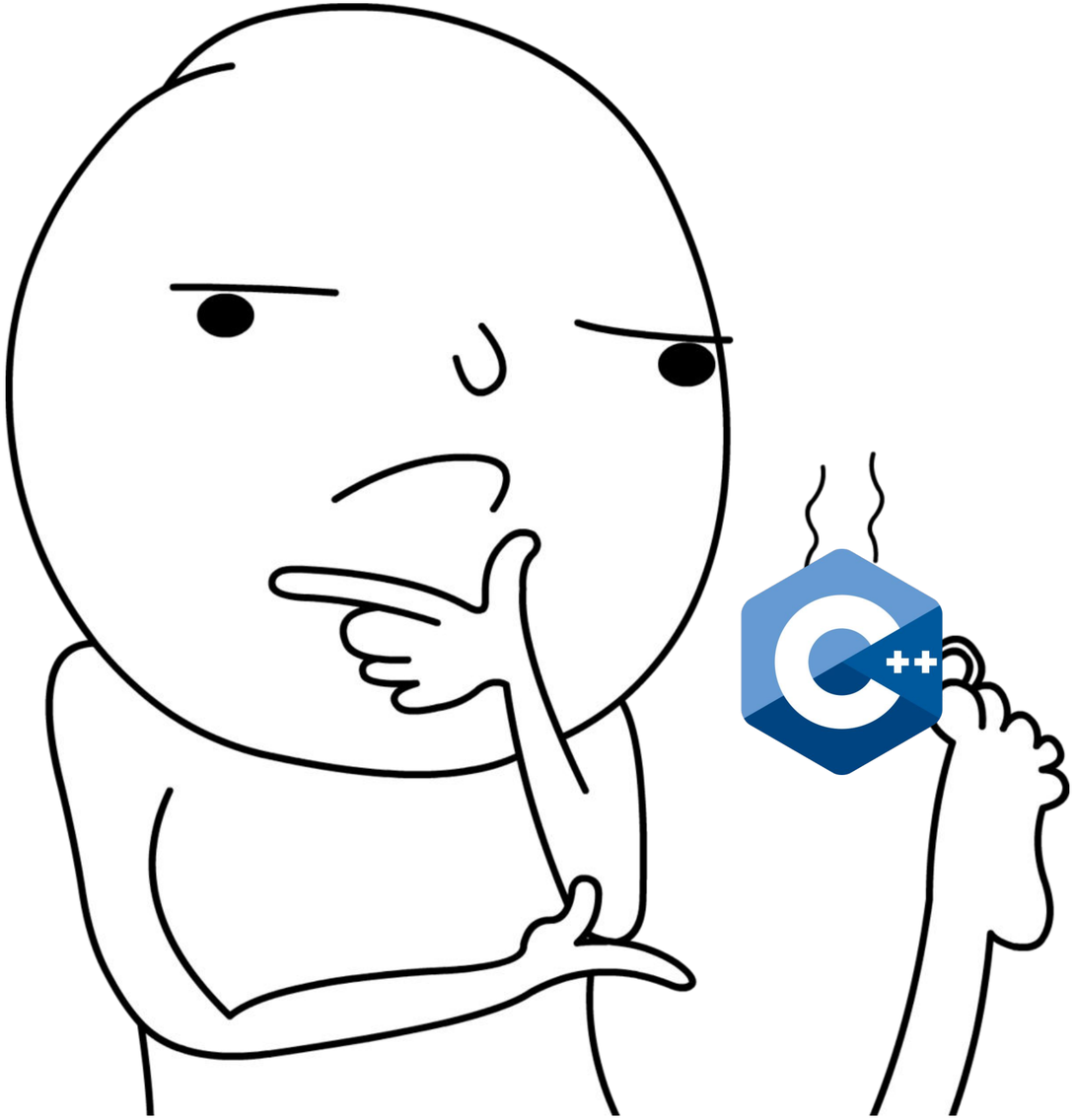
When std::cin fails!


```
void cinFailure()
{
  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
Rachel	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';
}
```

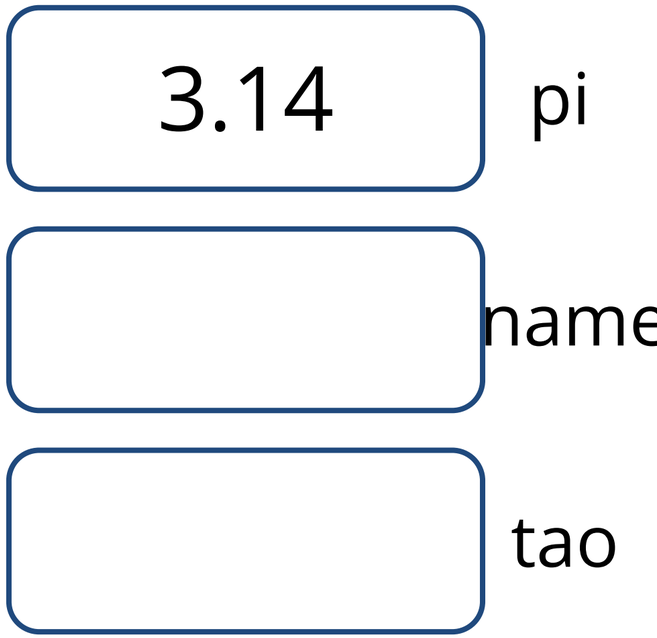
3.14 pi

Rachel 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';
}
```



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';
}
```

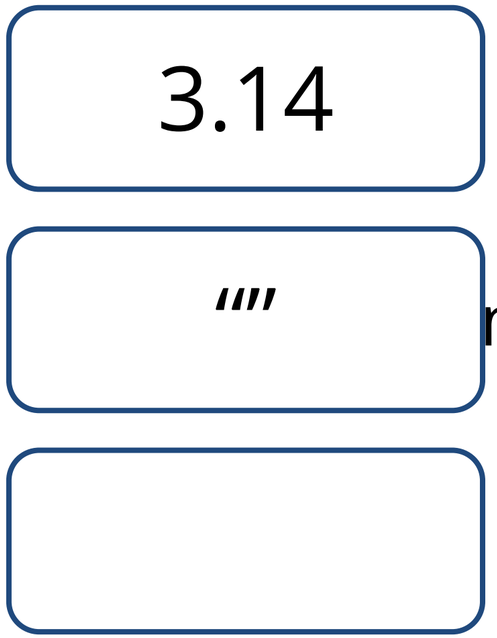
Any guesses for what happens here?

3.14	pi
	name
	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

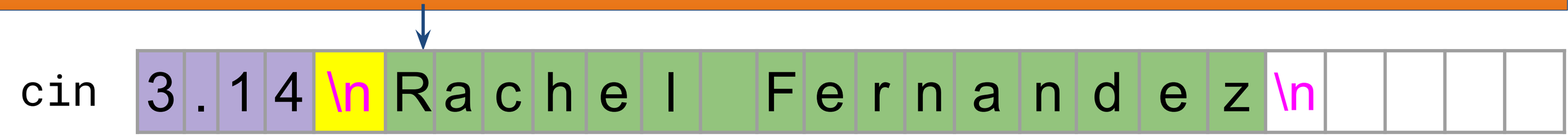


Fix?


```
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';
}
```

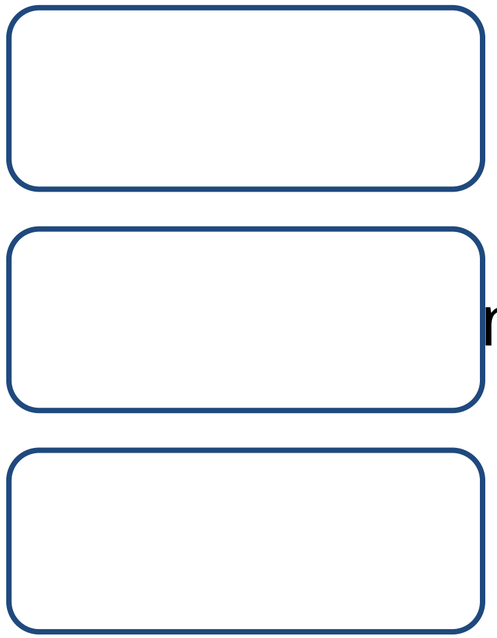
It's going to try to read the green stuff (name). But tao is a **double**!

How do we fix this?

Anyone want to take another guess?

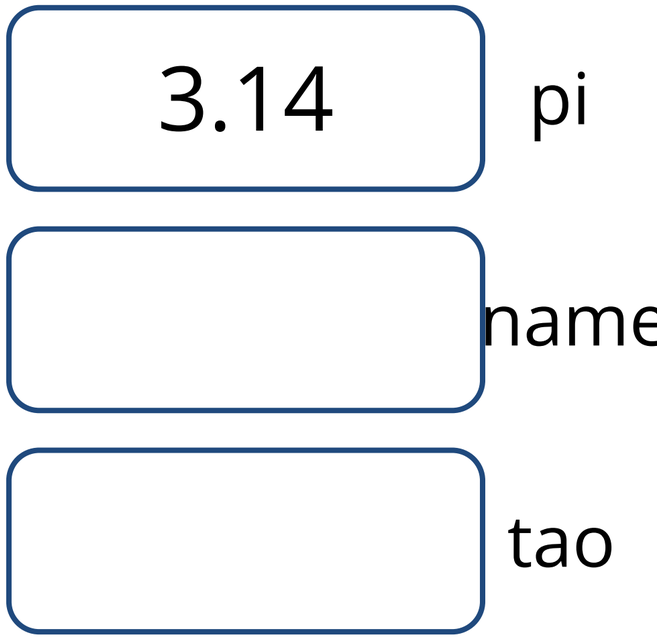
Fix?


```
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';
}
```



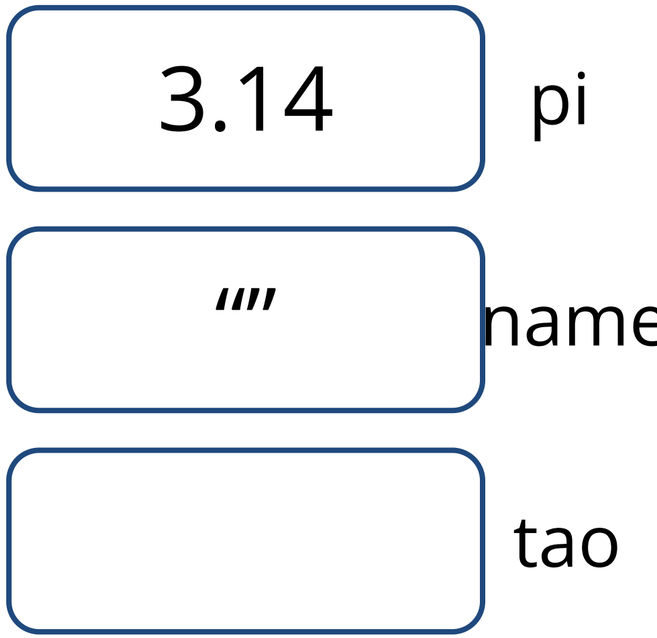
Fix?


```
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';
}
```

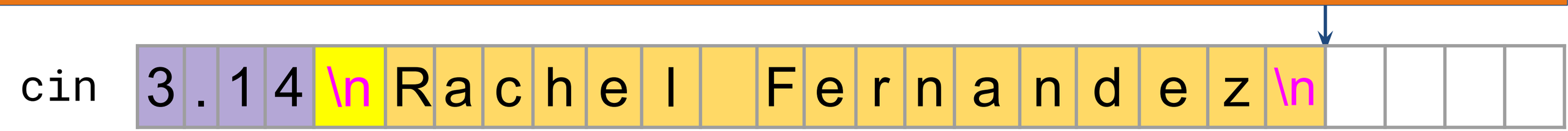


Fix ✅

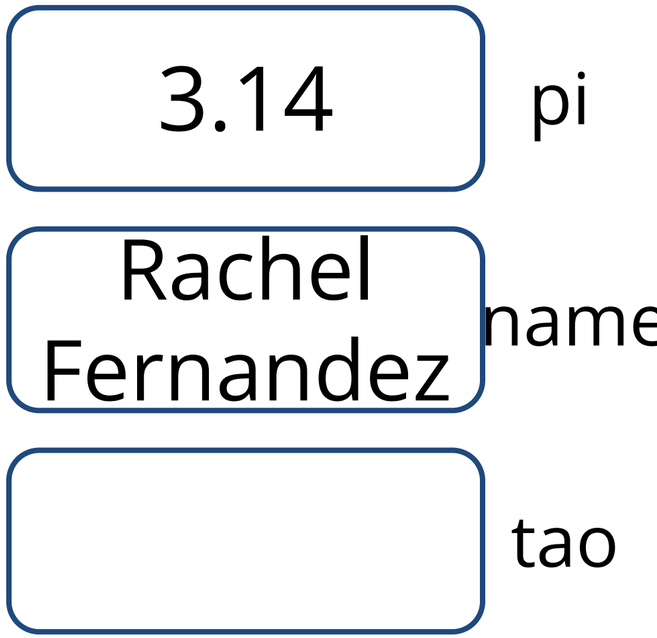

```
void cinGetline() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << name << " tao is :
" << tao << " pi is : " << pi << '\n';
}
```



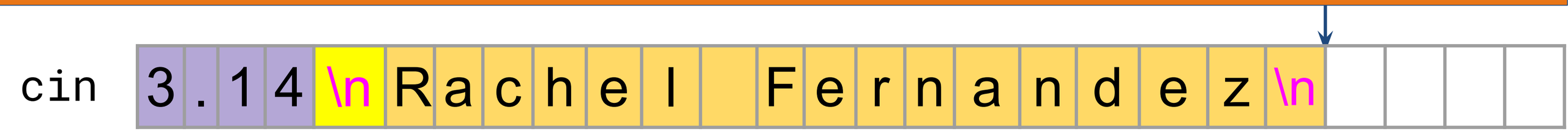
Fix ✅




```
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';
}
```

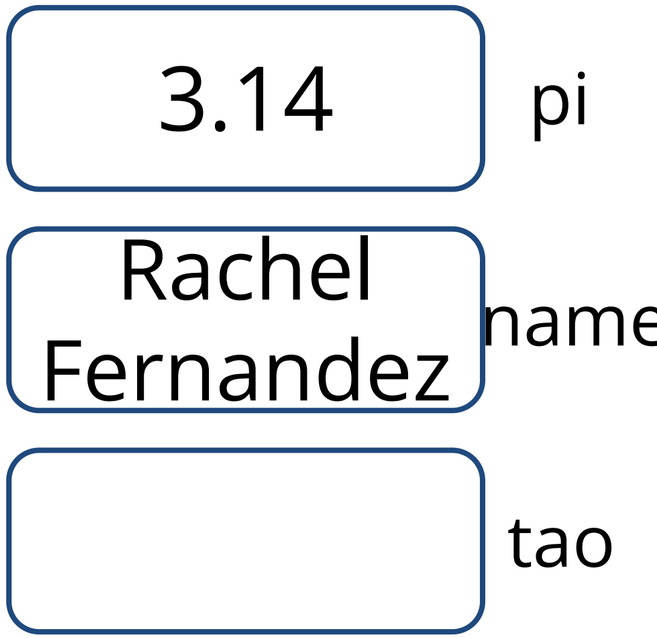


Fix ✅

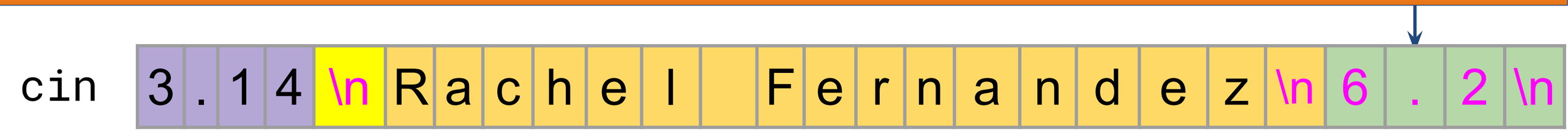



```
void cinGetline() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;
  std::getline(std::cin, name);
  std::getline(std::cin, name);
  std::cin >> tao;
  std::cout << "my name is : " << name << " tao is :
" << tao << " pi is : " << pi << '\n';
}
```

The stream is empty! So it is going to prompt a user for input



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';
}
```

3.14 pi

Rachel Fernandez name

6.2 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.