CS106L Lecture 4:

Streams 🏞️

Autumn 2024

Fabio Ibanez, Jacob Roberts-Baca
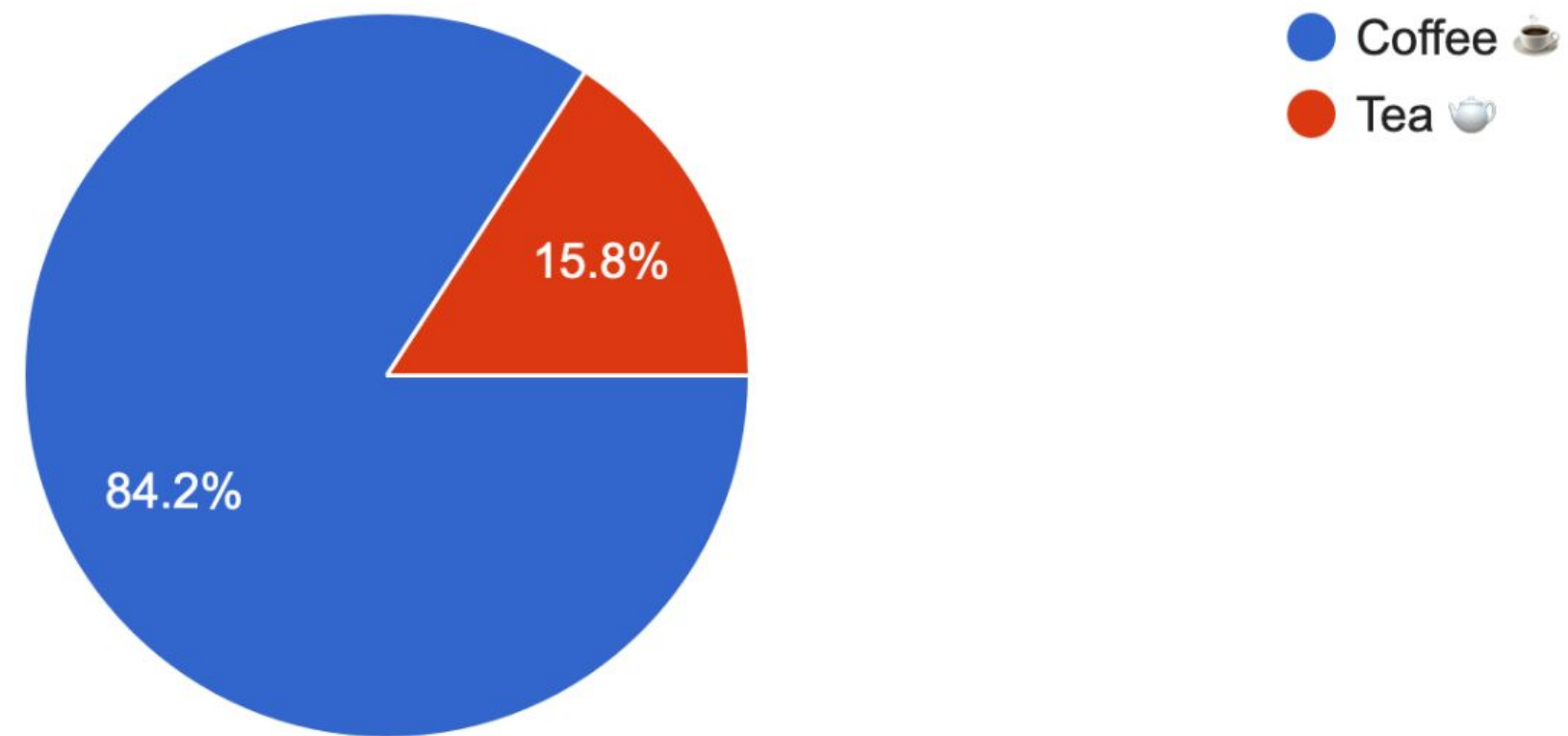
Attendance 🪄

https://tinyurl.com/streamsF24

Interesting Stats 🪄

Coffee or Tea? (There is one right answer)
38 responses

- Coffee ☕
- Tea 🫖

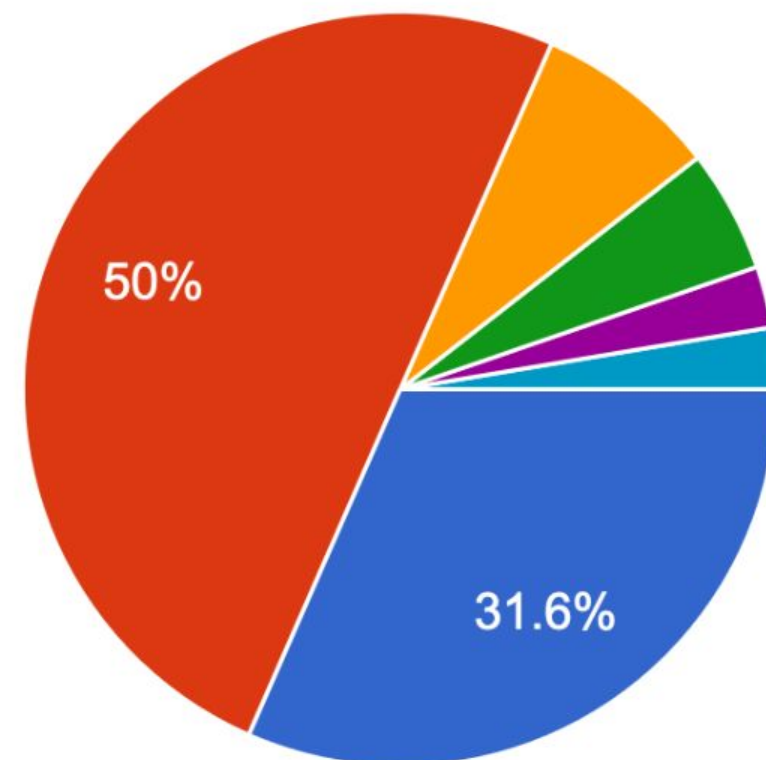84.2%

15.8%

Personally

Interesting Stats 🪄

Which describes your current status?
38 responses

Legend:
- 🔵 Present and caffeinated!! ☕
- 🔴 Here, but I would like caffeine 😪
- 🟠 My cat is attending on my behalf 🐱
- 🟢 Physically here, mentally in Hogwarts 🪄
- 🟣 here
- 🔵 no caffiene

Pie chart values: 50%, 31.6%

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) tho?

No but actually

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

- *Bjarne Stroustrup*

Streams

"~~Designing and implementing~~ a general **<u>input/output</u>** 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)** <mark>abstraction</mark> for C++

Abstractions

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

cout and cin

Known as the standard `iostreams`

- **cerr and clog**

  cerr: used to output errors

  clog: used for non-critical event logging

read more here: [GeeksForGeeks](#)

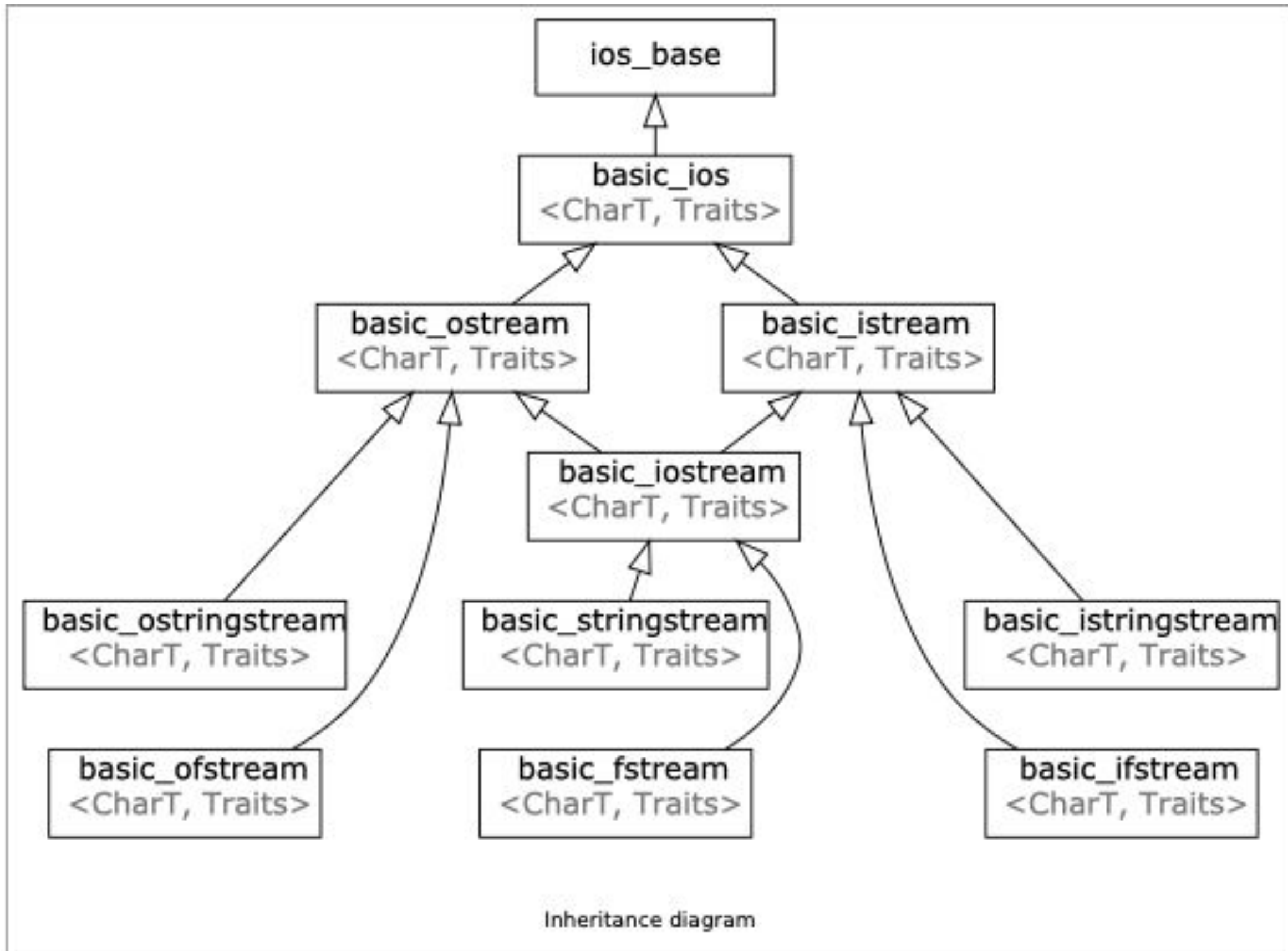# cout and cin

Known as the standard `iostreams`

- ~~**cerr and clog**~~ **cout** and **cin**

cerr: used to output errors

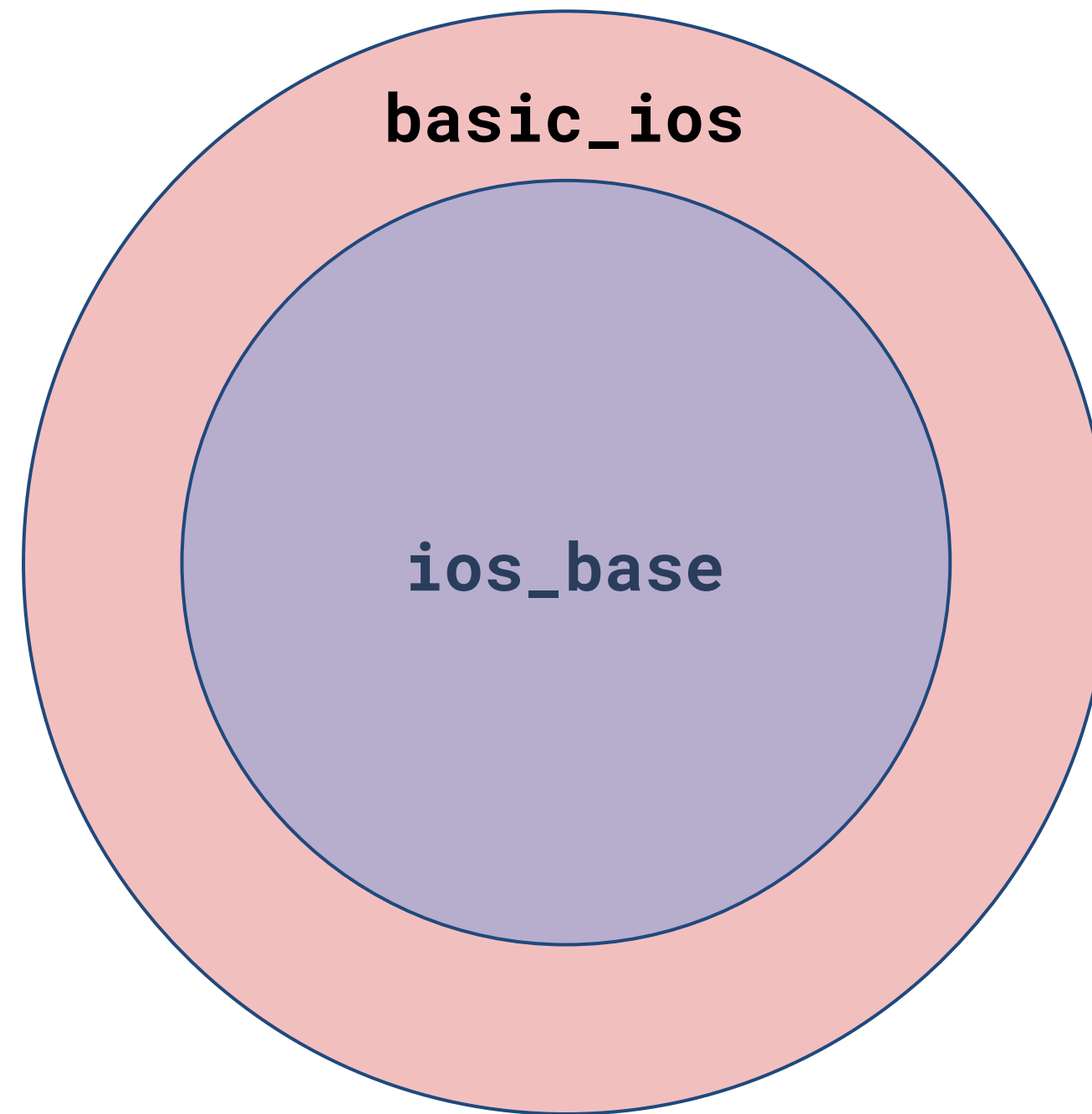**clog:** used for non-critical event logging

read more here: GeeksForGeeks

What streams actually are

Inheritance diagram

streams and types

basic_ios

ios_base

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

A familiar stream!

```cpp
std::cout << "Hello, World" << std::endl;
```

A familiar stream!

```
std::cout << "Hello, World" << std::endl;
```
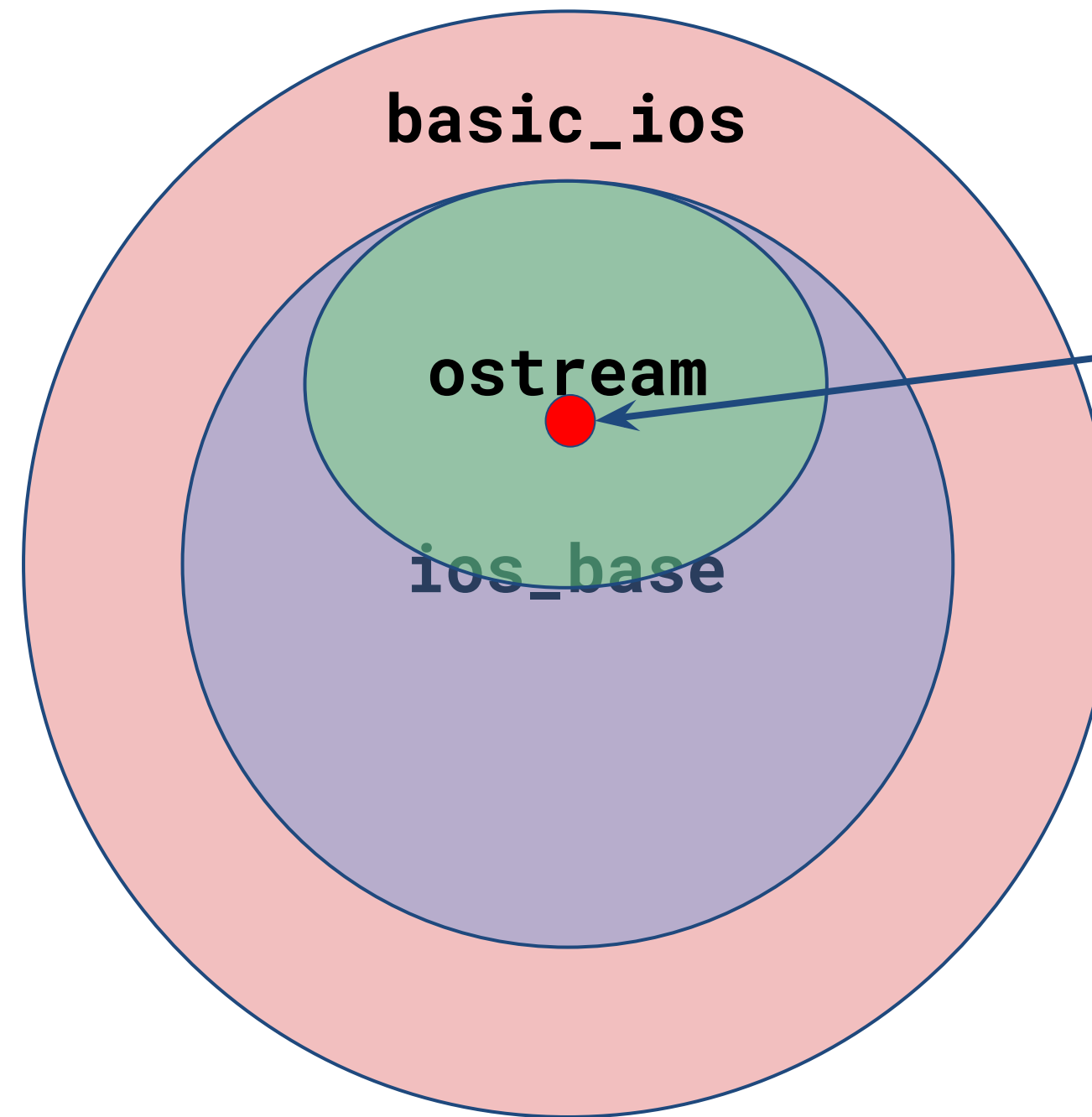
This is a stream

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!

streams and types

basic_ios

ostream

ios_base

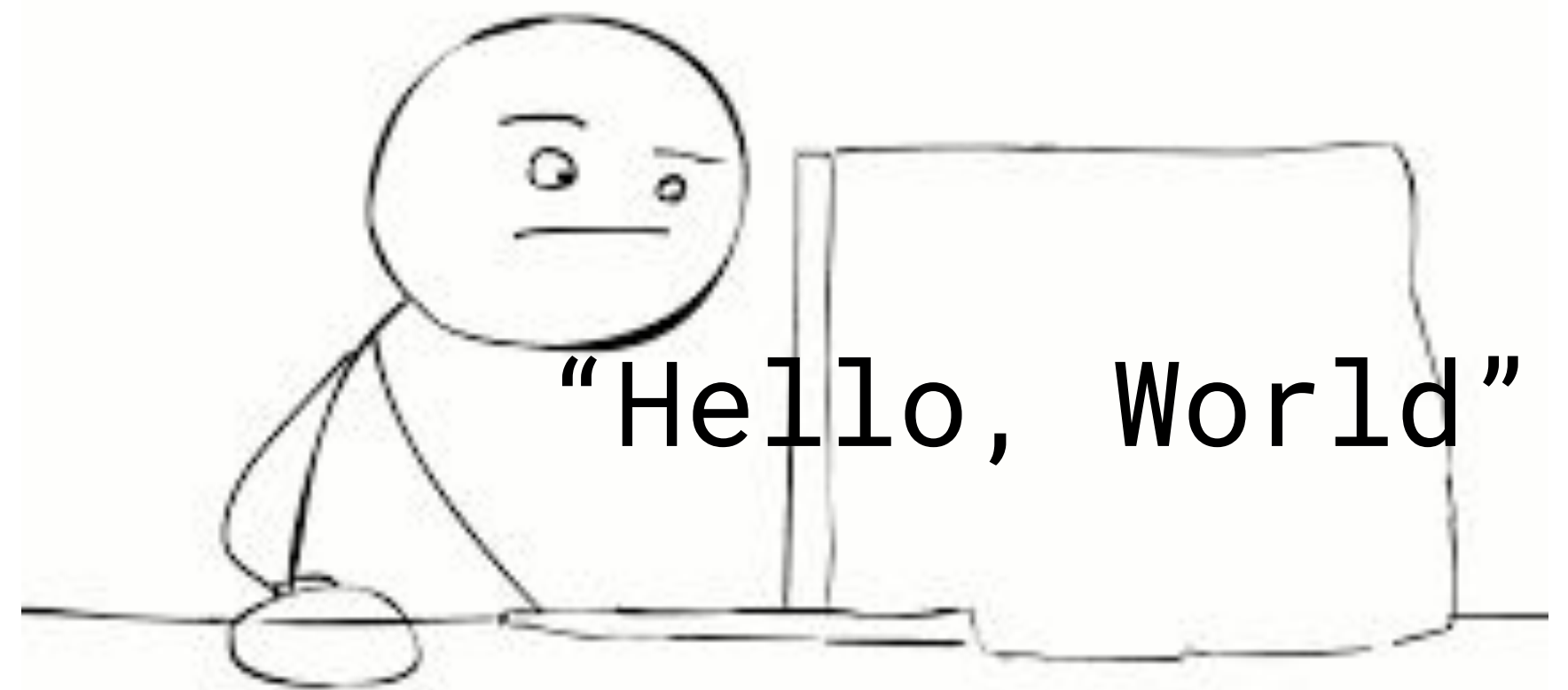std::cout has the property of the std::basic_ostream type

std::cout

std::cout << "Hello, World" << std::endl;

std::cout
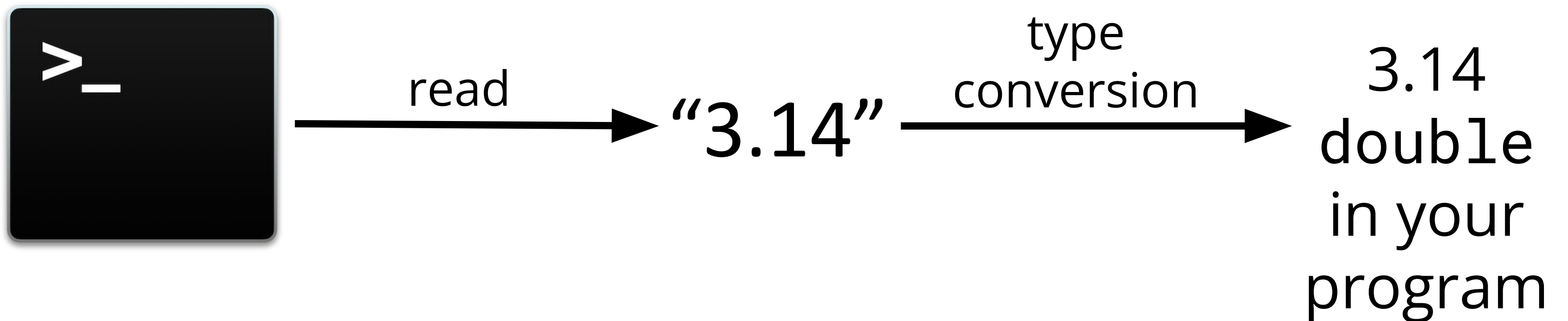
"Hello, World"

But how do we go from external source to program?

An Input Stream

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

read → "3.14" → type conversion → 3.14 double in your program

An Input Stream

How do you read a `double` from your console?
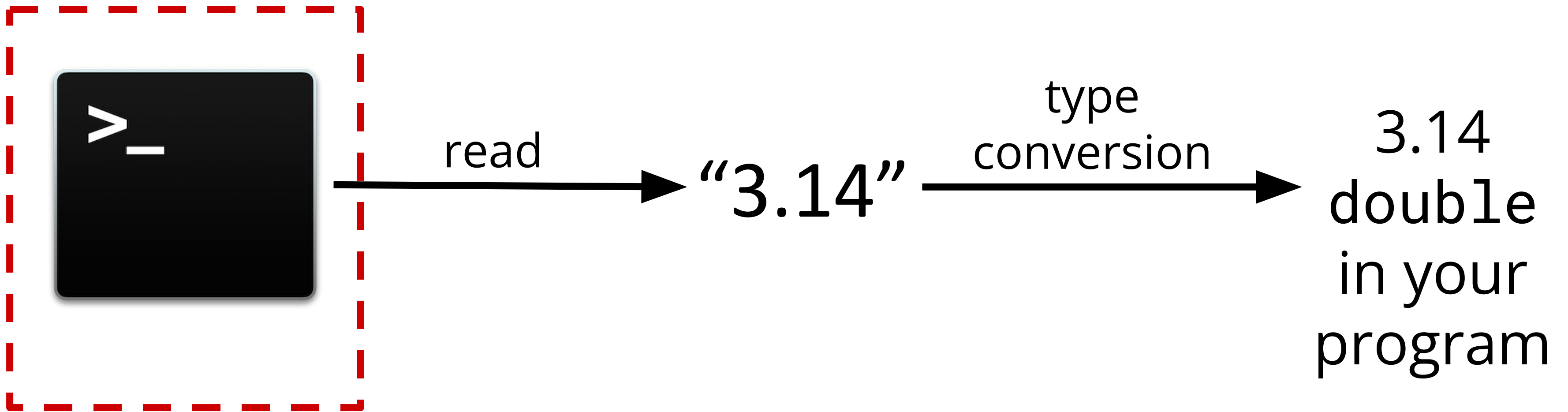
read → "3.14" → type conversion → 3.14 double in your program

An Input Stream

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

An Input Stream
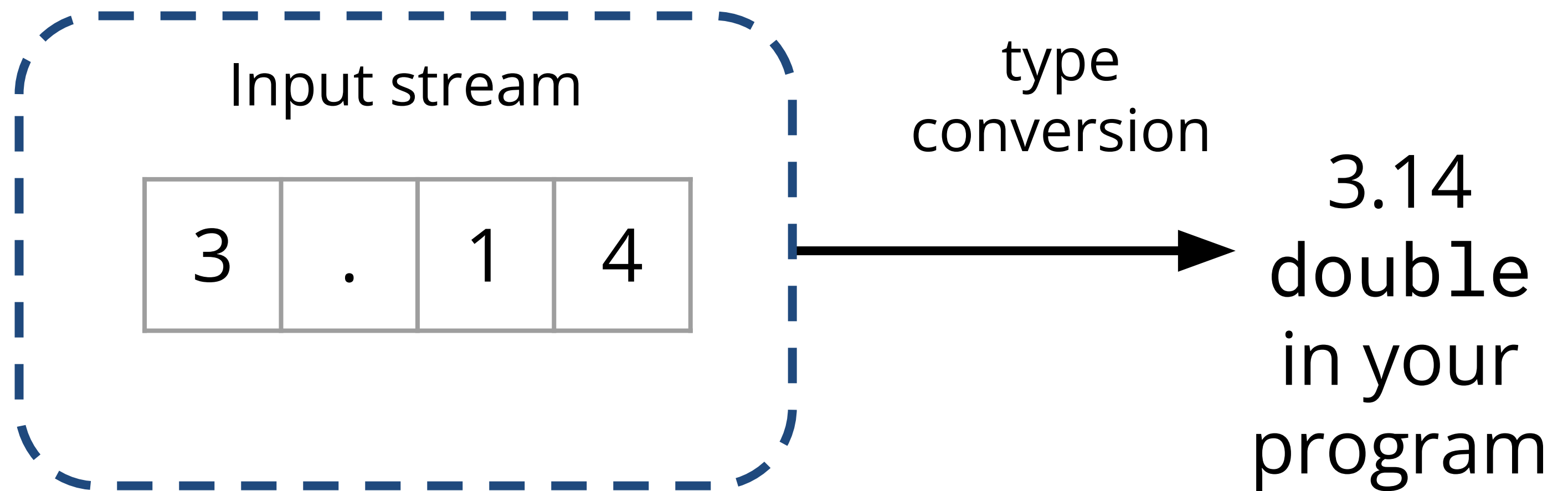
How do you read a double from your console?

std::cin is the console input stream!

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

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

streams and types

basic_ios

ostream

ios_base

istream

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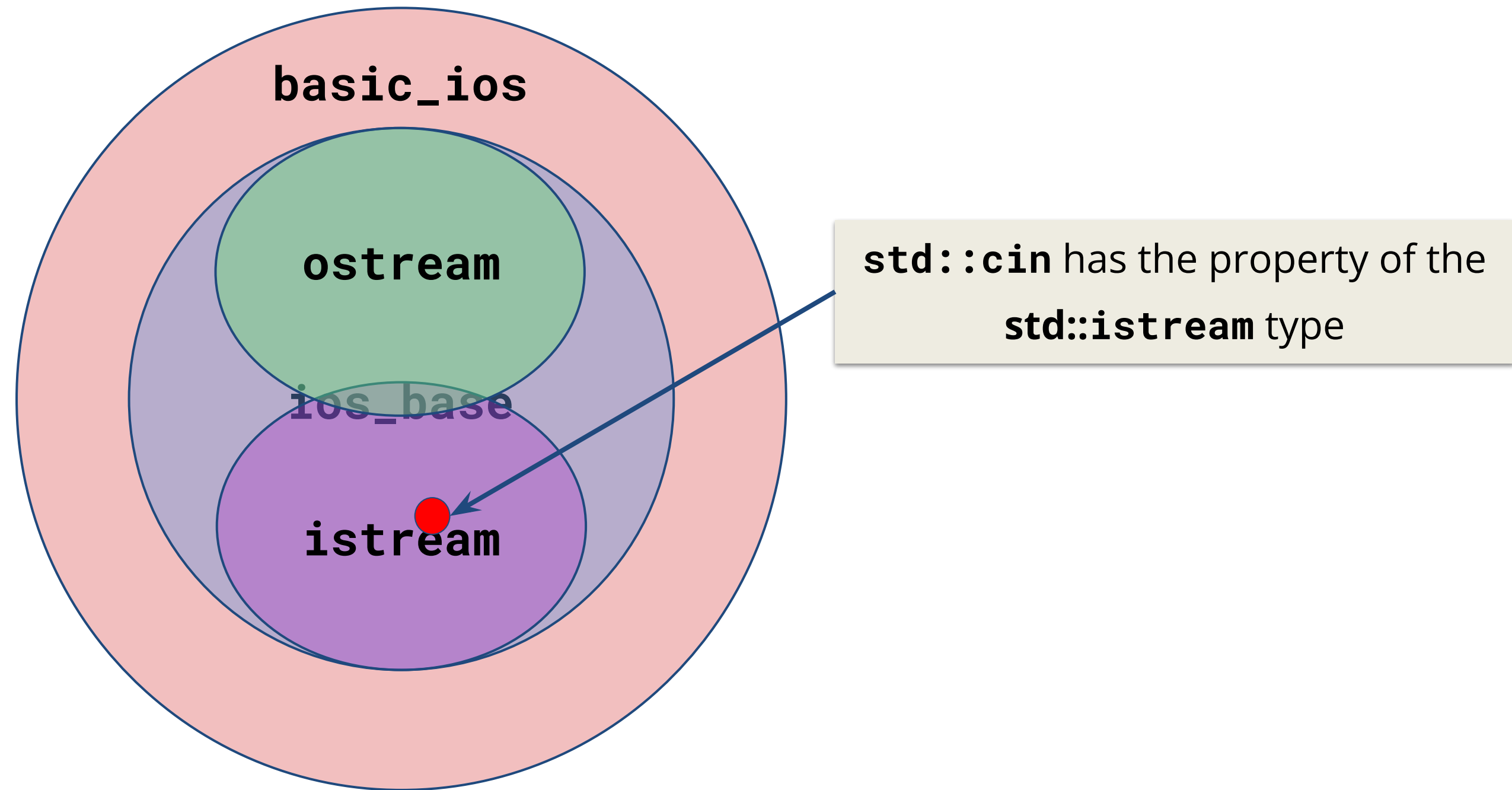
std::cin

```cpp
int main()
{

  double pi;
  std::cin >> pi;
  /// verify the value of pi!
  std::cout << pi / 2 << '\n';


  return 0;
}
```

"1.57" Console

Generalizing the Stream

? — external source ←→ stream

"3.14"

String representation

type conversion ←→ 3.14 double in your program

Generalizing the Stream

external source

stream

"3.14"

String representation

type conversion
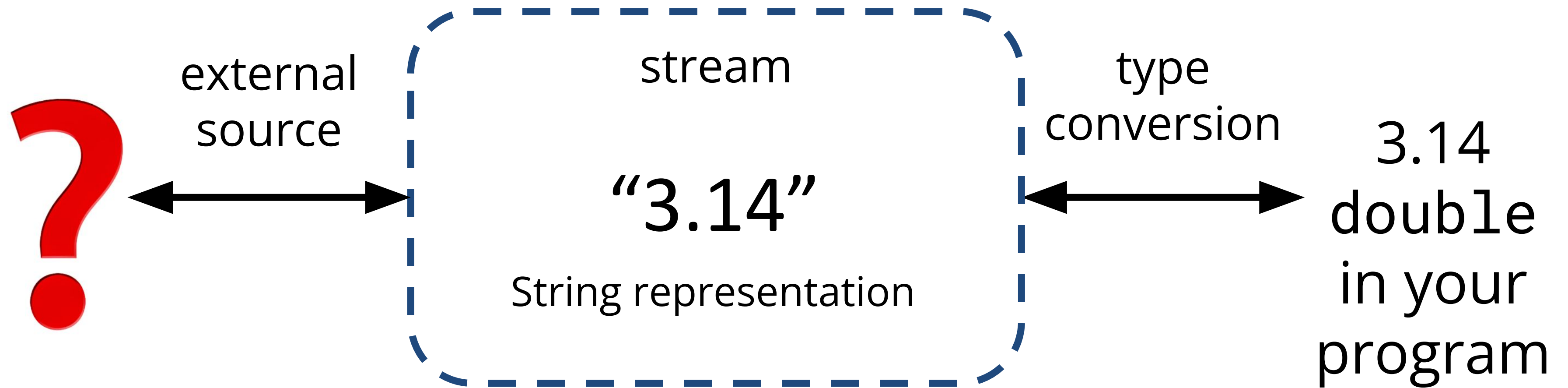
3.14
double
in your
program
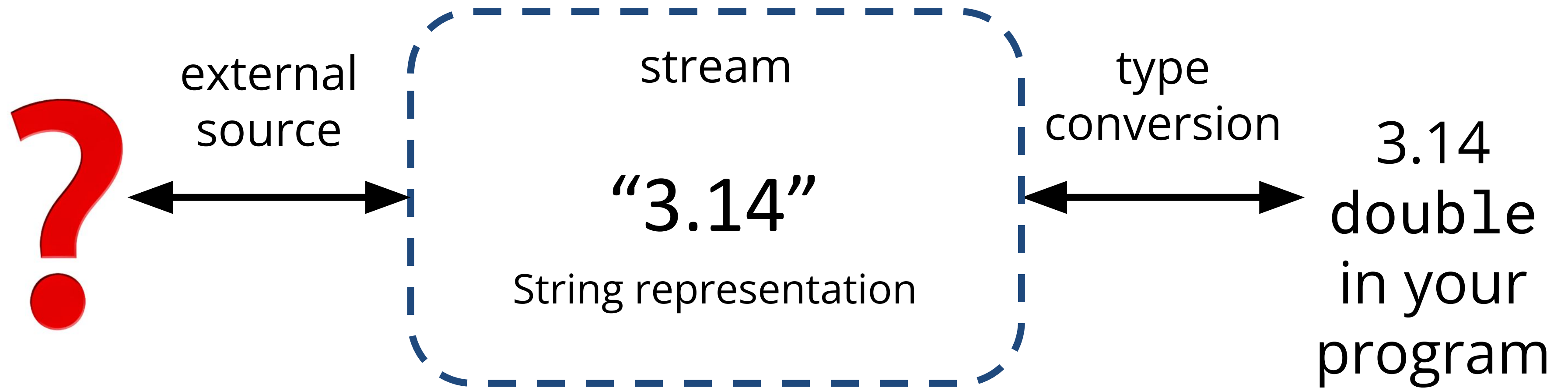
Implementation vs Abstraction

external
source

stream

"?"

String representation

type
conversion

?
`fill_in_type`
in your
program
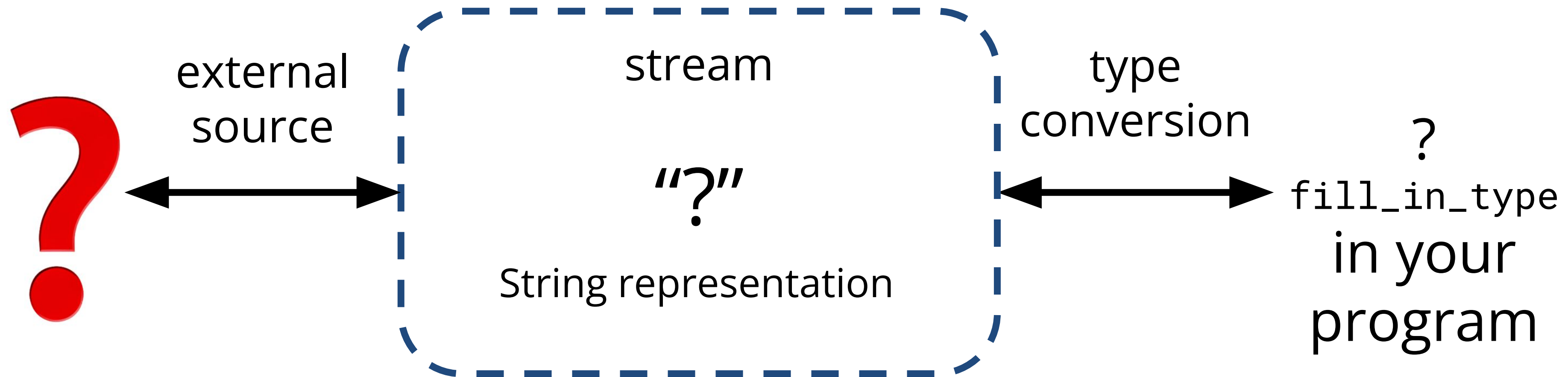
Why is this even useful?

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)

What streams actually are
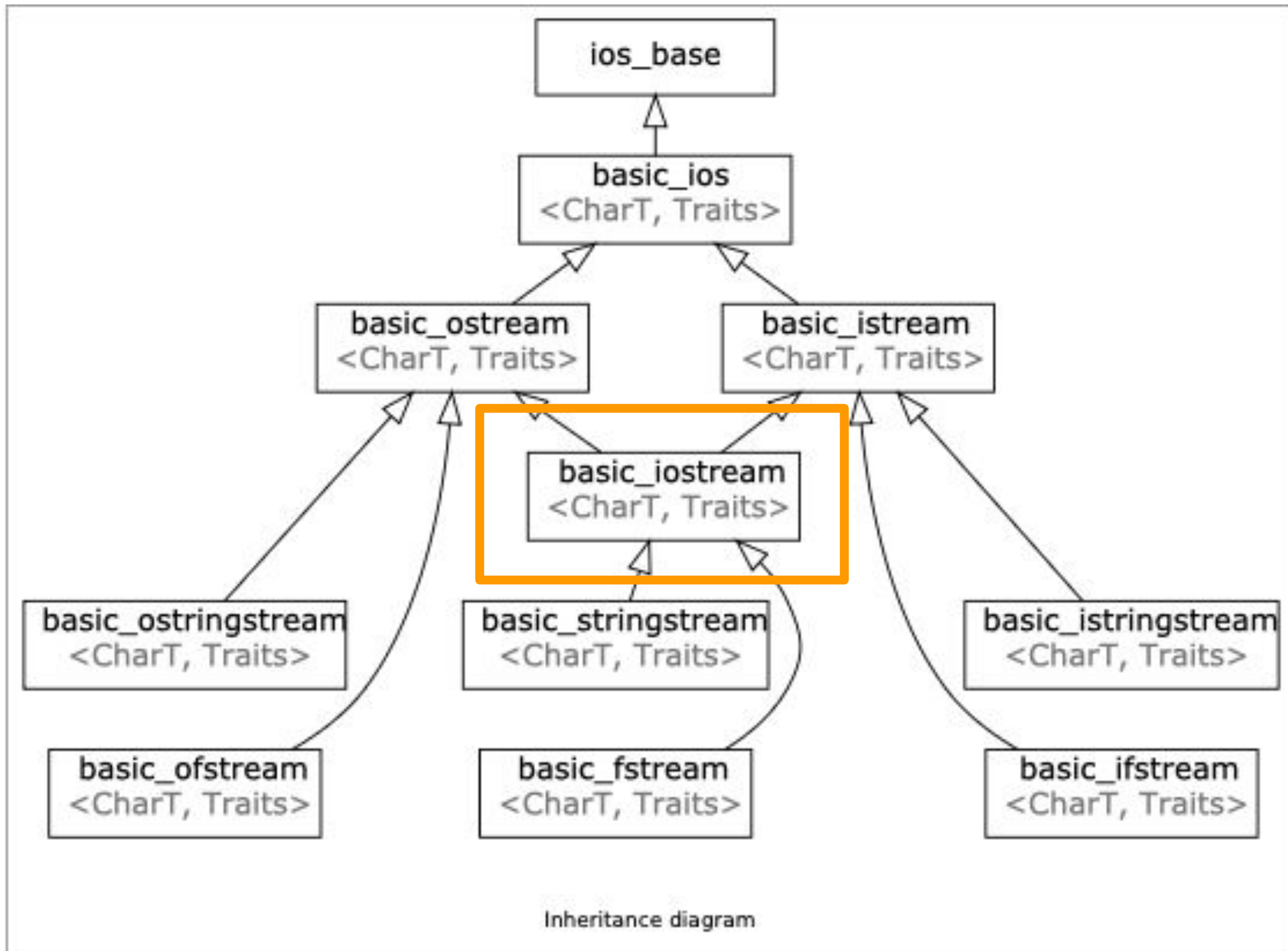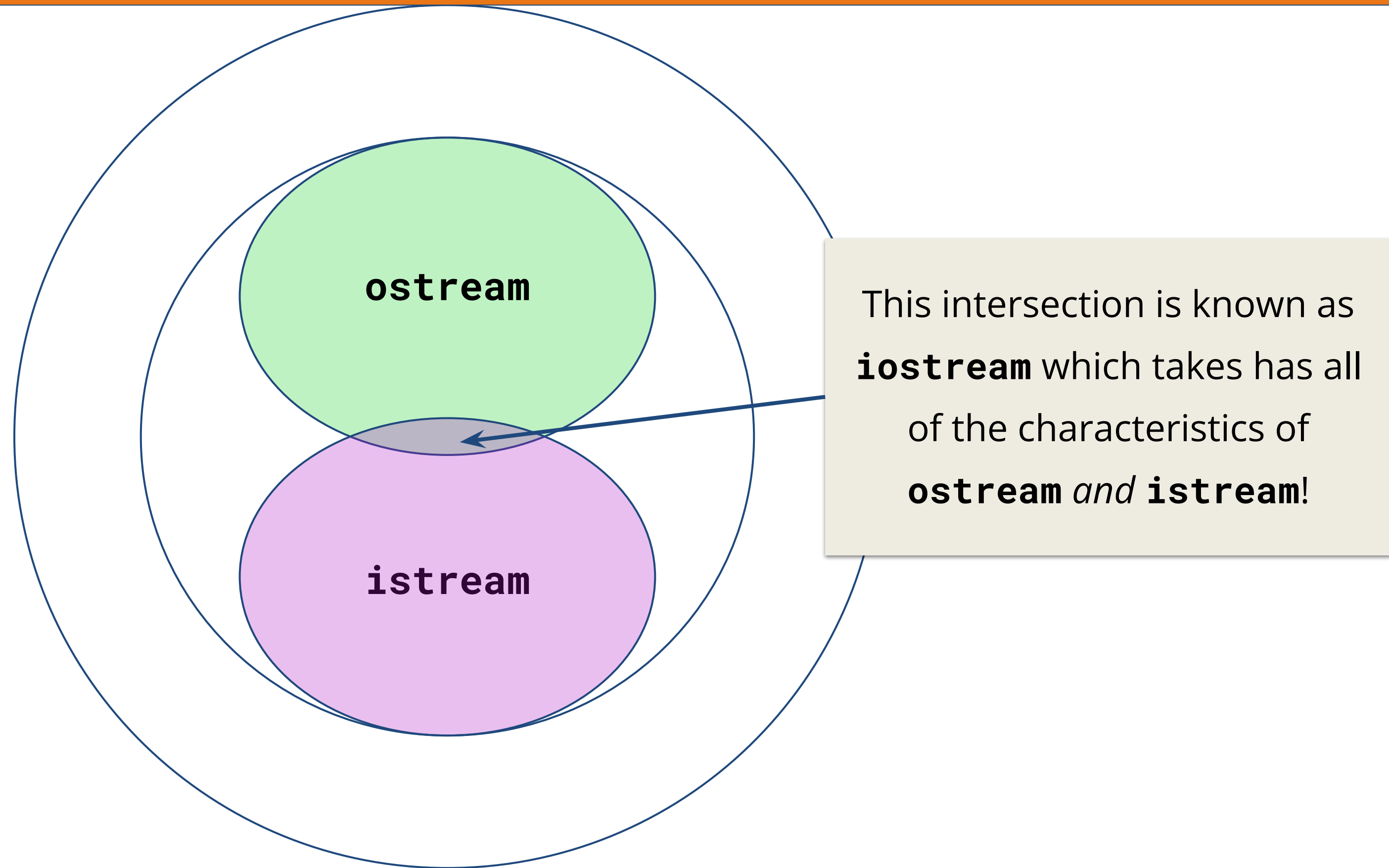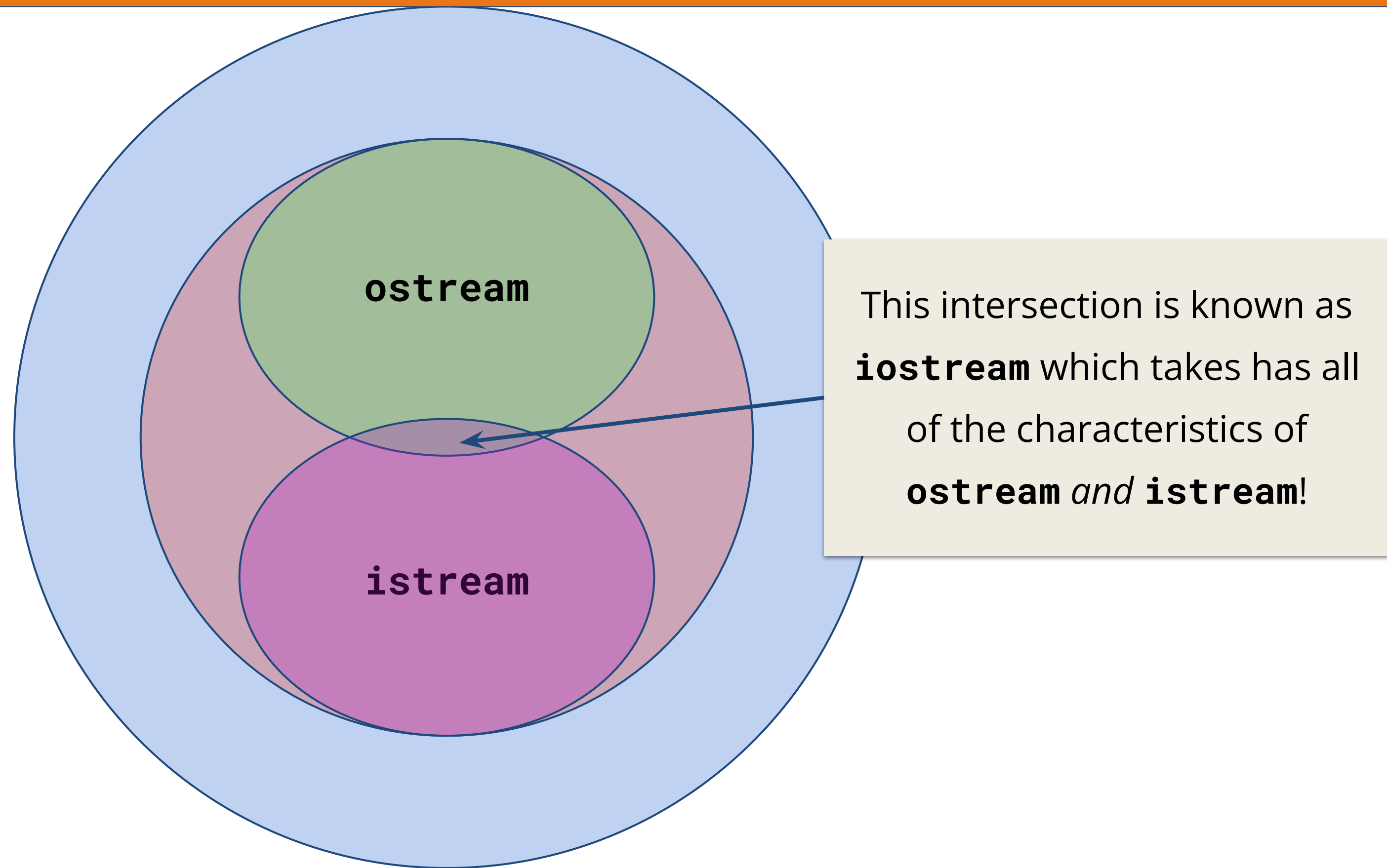
Inheritance diagram

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
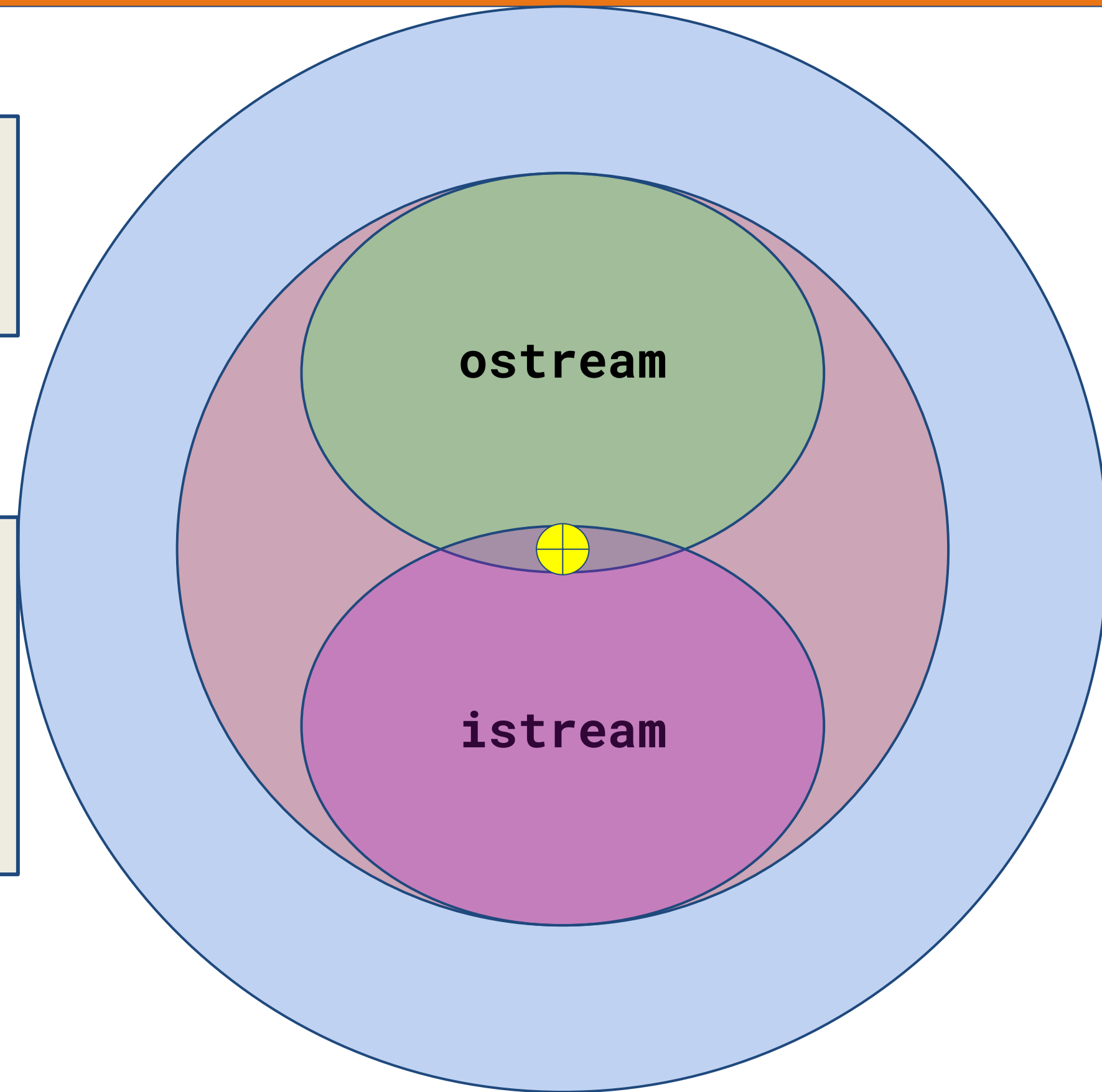
What streams actually are
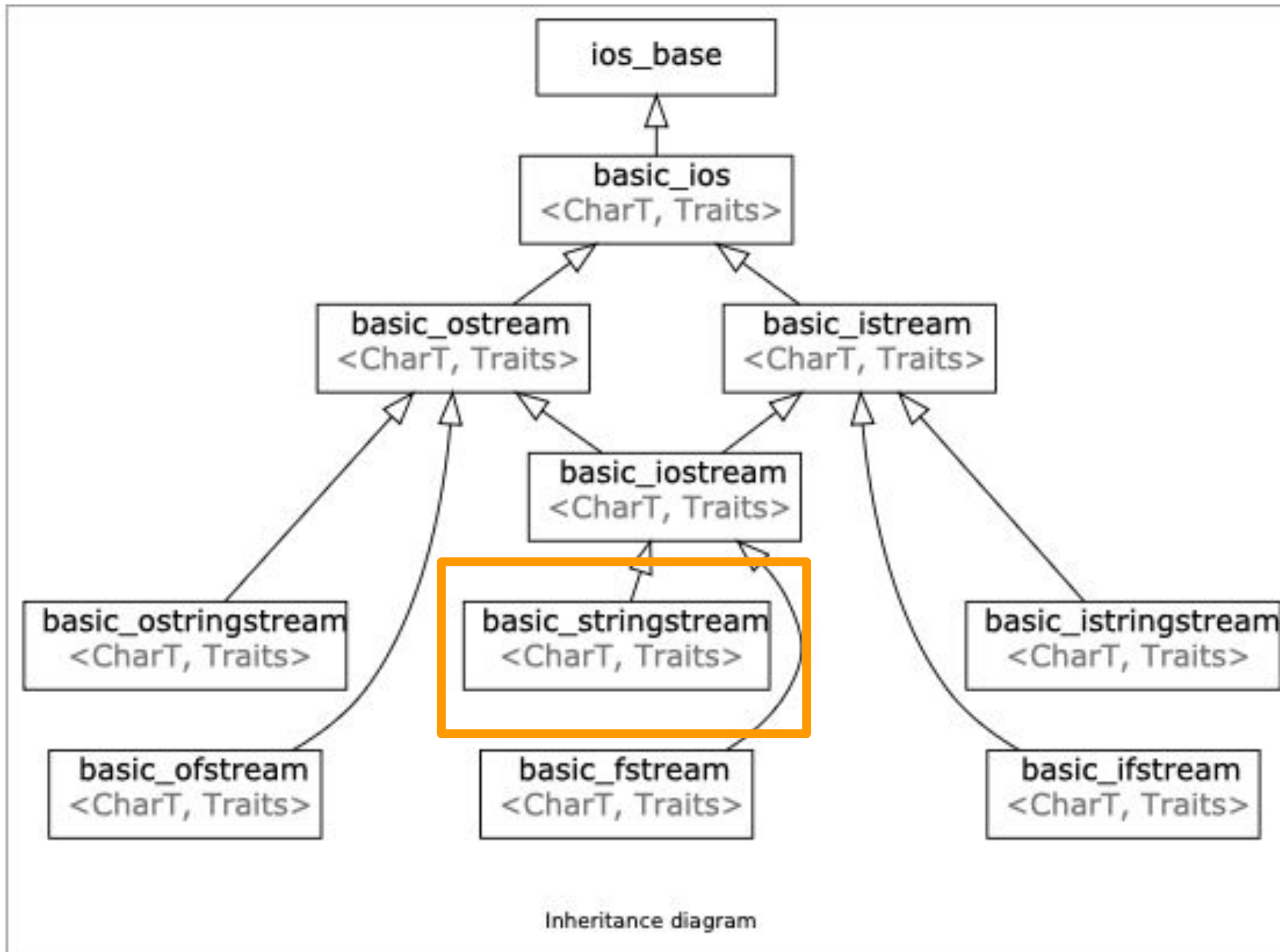
Inheritance diagram

std::stringstream example

```cpp
int main() {
  /// 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

```cpp
int main() {
  /// 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_string` like this!

what the stream looks like!

Start

End of stream

Bjarne Stoustrup C makes it easy to shoot yourself in the foot \n

std::stringstream example

```cpp
int main() {
  /// 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

```cpp
int main() {
  /// 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;
```

B	j	a	r	n	e		S	t	o	u	s	t	r	u	p		C		m	a	k	e	s		i	t		e	a	s	y		t	o		s	h	o	o	t

y	o	u	r	s	e	l	f		i	n		t	h	e		f	o	o	t	\n

First

Last

Language

what the stream looks like!

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

extracts from the stream!

B	j	a	r	n	e		S	t	o	u	s	t	r	u	p		C		m	a	k	e	s		i	t		e	a	s	y		t	o		s	h	o	o	t

y	o	u	r	s	e	l	f		i	n		t	h	e		f	o	o	t	\n

First

Last

Language

what the stream looks like!

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

| B | j | a | r | n | e | | S | t | o | u | s | t | r | u | p | | C | | m | a | k | e | s | | i | t | | e | a | s | y | | t | o | | s | h | o | o | t |
| y | o | u | r | s | e | l | f | | i | n | | t | h | e | | f | o | o | t | \n |

Bjarne

First

Last

Language

what the stream looks like!

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

Bjarne Stoustrup C makes it easy to shoot yourself in the foot \n

Bjarne

Stroustrup

First

Last

Language

what the stream looks like!

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

B	j	a	r	n	e		S	t	o	u	s	t	r	u	p		C		m	a	k	e	s		i	t		e	a	s	y		t	o		s	h	o	o	t
y	o	u	r	s	e	l	f		i	n		t	h	e		f	o	o	t	\n

Bjarne	Stroustrup	C
First	Last	Language

std::stringstream example

```cpp
int main() {
  /// 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;
}
```
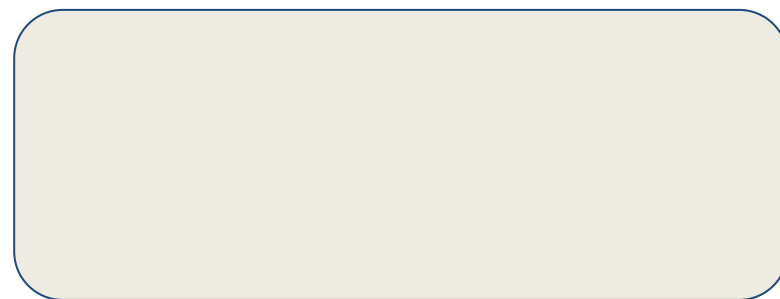
We want to extract the quote!

what the stream looks like!

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

B	j	a	r	n	e		S	t	o	u	s	t	r	u	p		C		m	a	k	e	s		i	t		e	a	s	y		t	o		s	h	o	o	t

y	o	u	r	s	e	l	f		i	n		t	h	e		f	o	o	t	\n

Bjarne

Stroustrup

C

First

Last

Language

what the stream looks like!

Problem:
?

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

B	j	a	r	n	e		S	t	o	u	s	t	r	u	p		C		m	a	k	e	s		i	t		e	a	s	y		t	o		s	h	o	o	t

y	o	u	r	s	e	l	f		i	n		t	h	e		f	o	o	t	\n

Bjarne

Stroustrup

C

First

Last

Language

what the stream looks like!

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

B	j	a	r	n	e		S	t	o	u	s	t	r	u	p		C		m	a	k	e	s		i	t		e	a	s	y		t	o		s	h	o	o	t

y	o	u	r	s	e	l	f		i	n		t	h	e		f	o	o	t		\n

Bjarne	Stroustrup	C
First	Last	Language

what the stream looks like!

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

| B | j | a | r | n | e | | S | t | o | u | s | t | r | u | p | | C | | m | a | k | e | s | | i | t | | e | a | s | y | | t | o | | s | h | o | o | t |

| y | o | u | r | s | e | l | f | | i | n | | t | h | e | | f | o | o | t | \n |

| 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;
```

B j a r n e   S t o u s t r u p   C   makes it easy to shoot
yourself in the foot \n

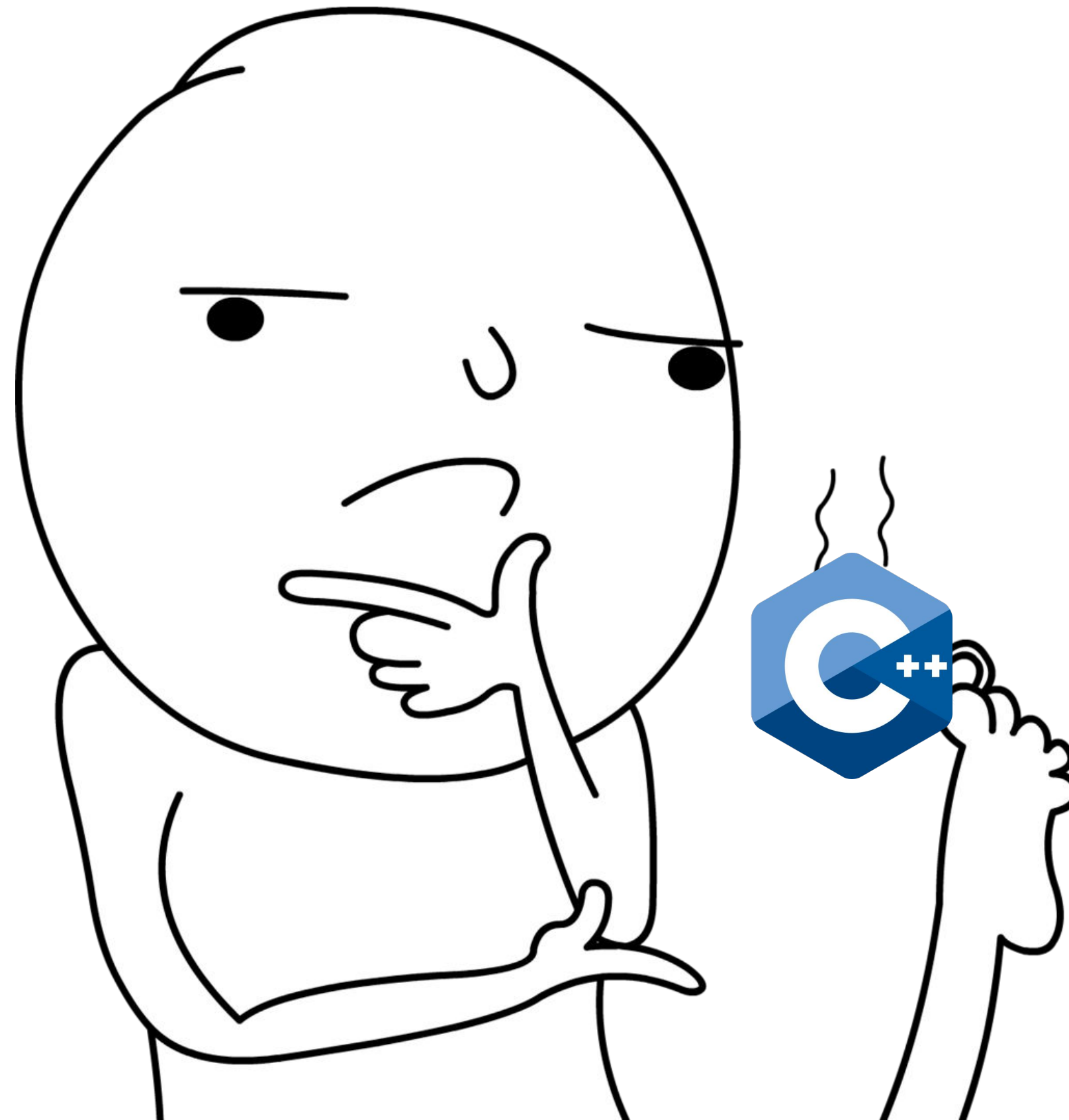| Bjarne	Stroustrup	C
First	Last	Language

std::stringstream example

```cpp
int main() {
  /// 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;
}
```

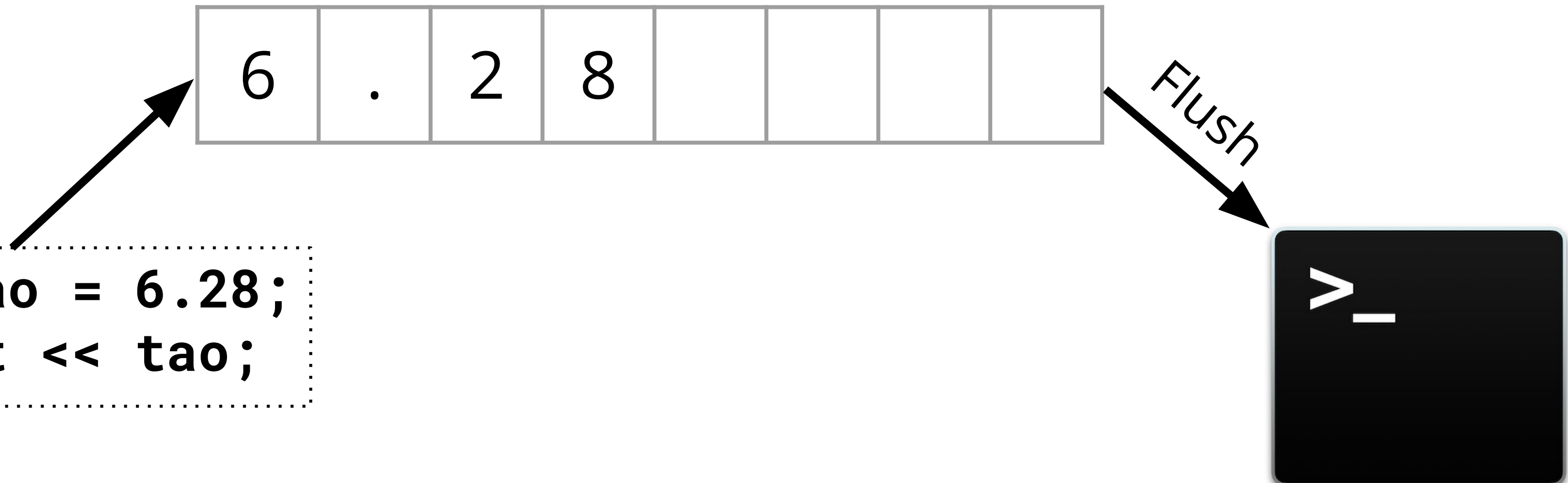# What questions do we have?

Plan

1. Quick recap

2. What are streams??!!

3. `stringstreams!`

4. **`cout and cin`**

5. Output streams

6. Input streams
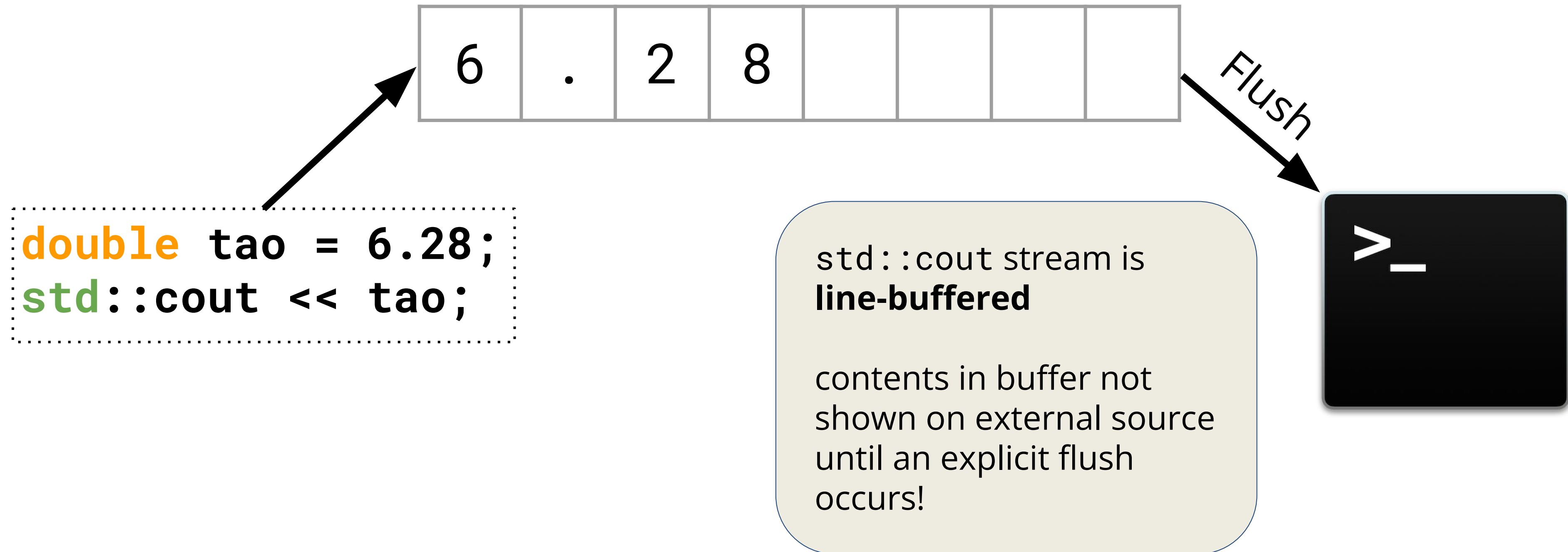
Output Streams

- 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!

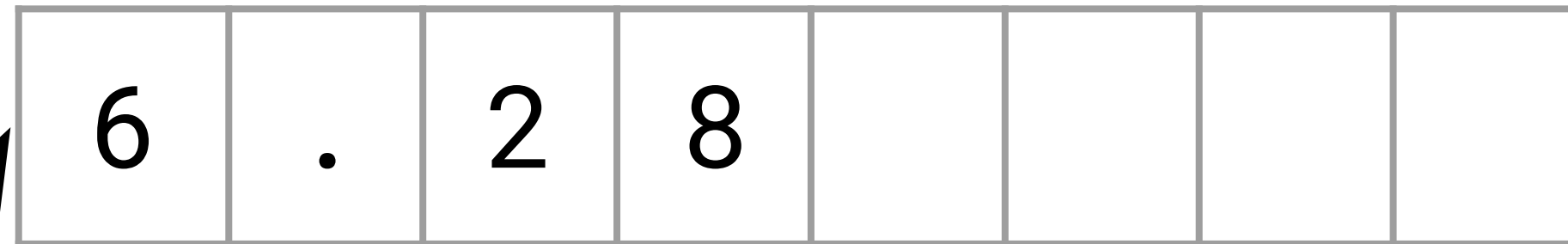Character in output streams are stored in an intermediary buffer before being flushed to the destination

Zooming in on Output Streams!

6	.	2	8				

Flush

```
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!

Zooming in on Output Streams!


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

"6.28"
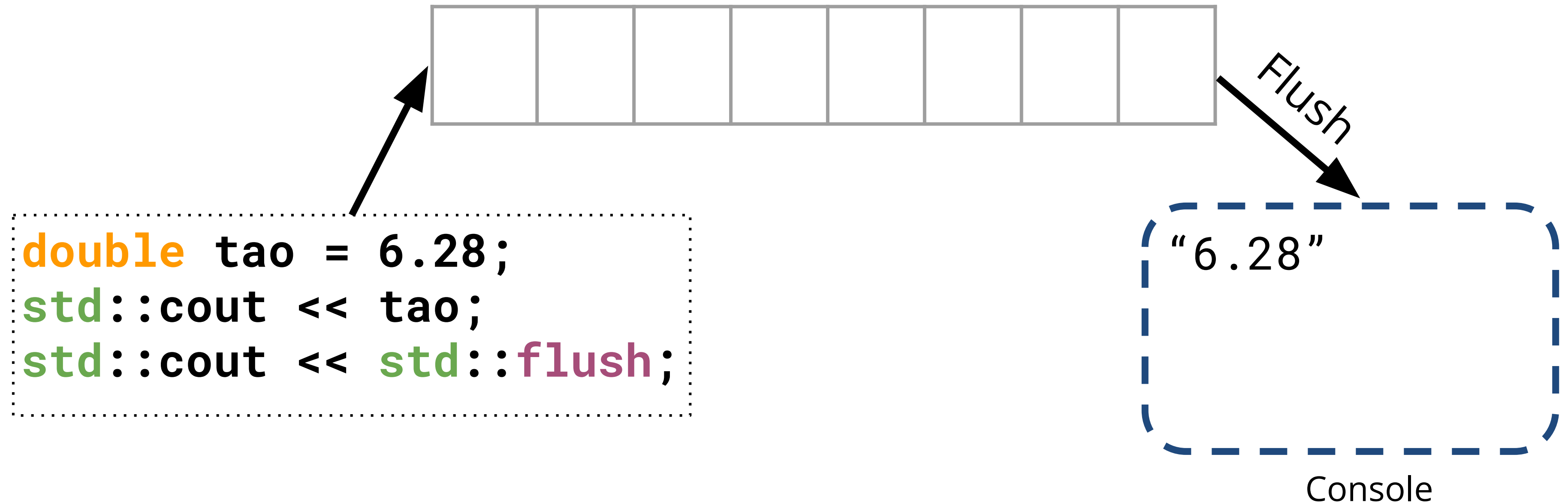
Flush

Console

Zooming in on Output Streams!

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

Flush

"6.28"

Console

std::endl

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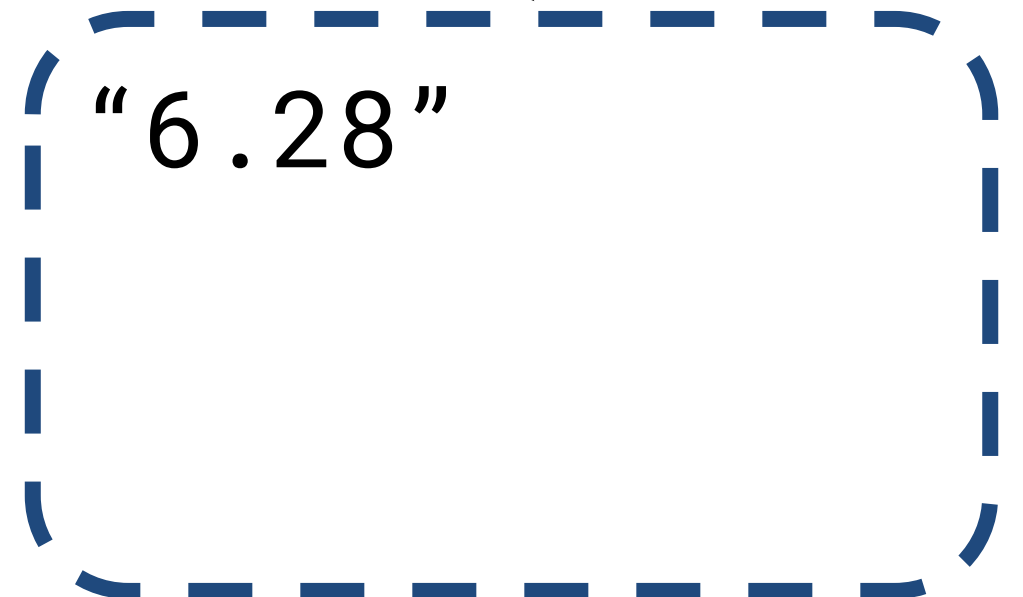
Output:

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

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

Here's without `std::endl`

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

Output:

"12345"

std::endl

```cpp
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

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

intermediate buffer

Output:

1

i

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'						

1

i

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;
}
```

intermediate buffer

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

| 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

```cpp
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

5

i

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

5

i

flushing is an expensive operation!

Output:

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

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

'\n' 🤠

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

intermediate buffer

1

i

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

Output:

Let's try just adding the **'\n'** character

' \n ' 🤠

```cpp
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' 🤠

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

i = 2

Let's try just adding the '\n' character

intermediate buffer

1	'\n'						

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

Output:

'\n' 🤠

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

intermediate buffer

1	'\n'	2	'\n'				

2
i

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

Output:

Let's try just adding the '\n' character

'\n' 🤠

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

intermediate buffer

1	'\n'	2	'\n'				

3

i

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

Output:

Let's try just adding the '\n' character

'\n' 🤠

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

intermediate buffer

1	'\n'	2	'\n'	3	'\n'		

3
i

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

Let's try just adding the '\n' character

Output:

'\n' 🤠

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

intermediate buffer

1	'\n'	2	'\n'	3	'\n'		

4

i

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

Output:

Let's try just adding the '\n' character

'\n' 🤠

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

intermediate buffer

1	'\n'	2	'\n'	3	'\n'	4	'\n'

4
i

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

Output:

Let's try just adding the '\n' character

'\n' 🤠

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

intermediate buffer

1	'\n'	2	'\n'	3	'\n'	4	'\n'

5
i

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

Output:

Let's try just adding the '\n' character

'\n' 🤠

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

intermediate buffer

5
i

C++: FLUSH

Output:
"1"
"2"
"3"
"4"

Let's try just adding the '\n' character

'\n' 🤠

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

intermediate buffer

5	'\n'						

i: 5

Yay!

Output:
"1"
"2"
"3"
"4"

Let's try just adding the '\n' character

'\n' 🤠

```cpp
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

```cpp
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 << "Draaaakkkkeeeeeeeeee" << std::endl;

std::cout << "Draaaakkkkeeeeeeeeee" << '\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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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 go on Replit :))

Output File Streams

```cpp
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

```cpp
int inputFileStreamExample() {
  std::ifstream ifs("append.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

```cpp
int inputFileStreamExample() {
  std::ifstream ifs("append.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!
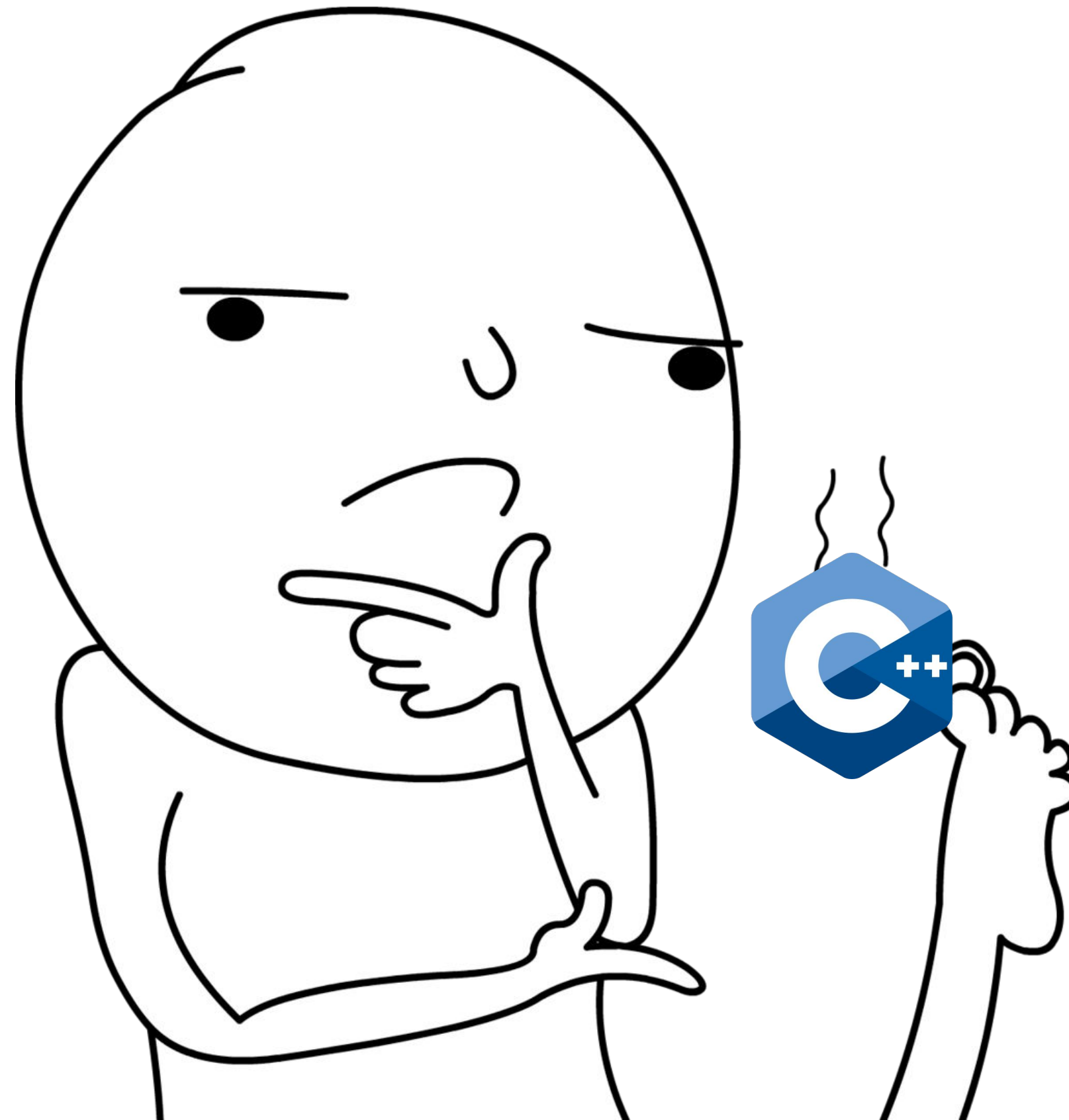
IO File Streams

Inheritance diagram
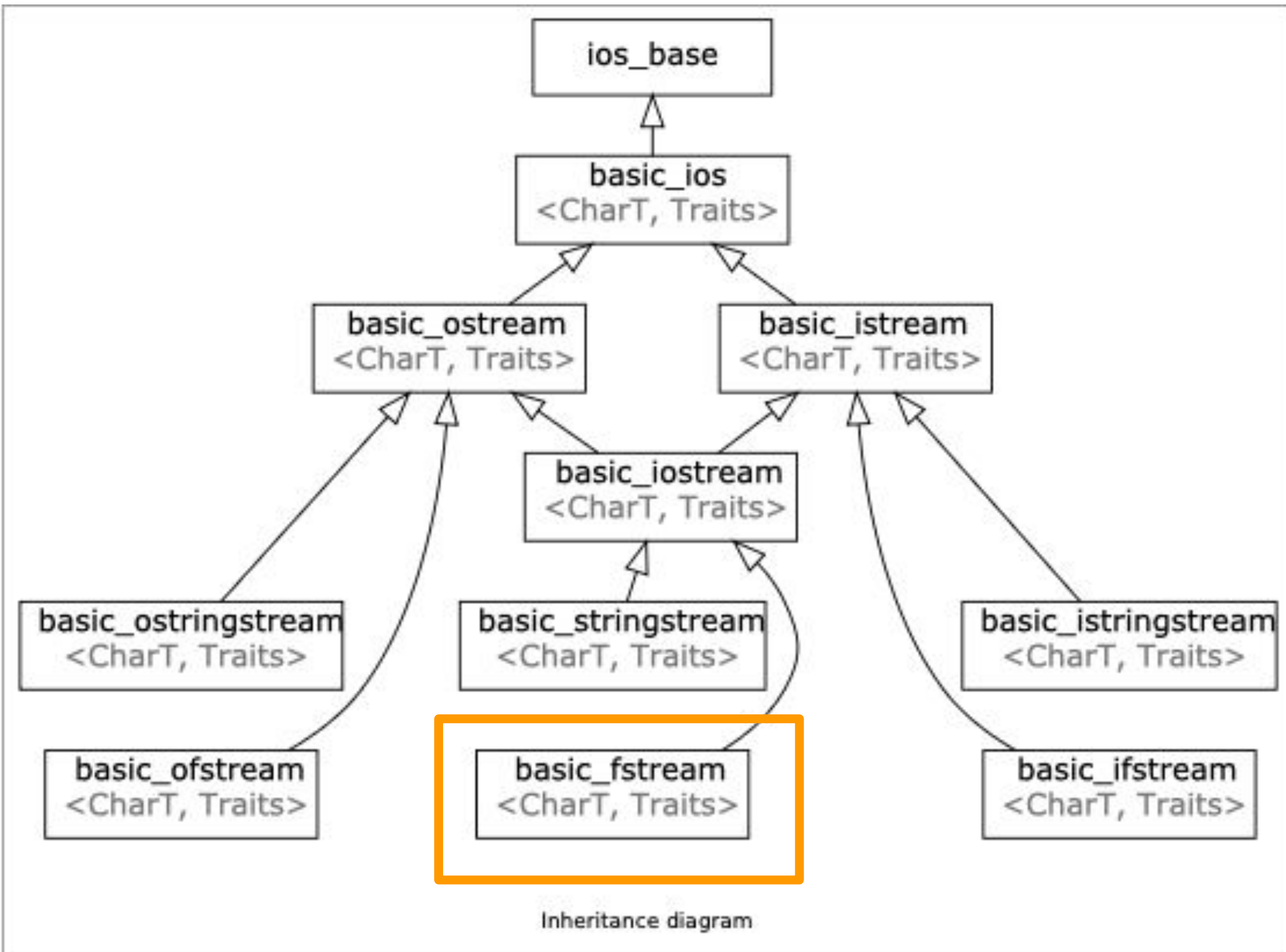
Check out the Replit!

🤠Checkout the function **`testFstream()`**
🤠

With your knowledge of how output and input streams independently work you can make great use of their combined implementation

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

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

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

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

cin | 3 | . | 1 | 4 | '\n' | | | | | | | | | | | | |

```cpp
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' | | | | | | | | | | | | |

```cpp
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

cin | 3 | . | 1 | 4 | '\n' | | | | | | | | | | | | |

```cpp
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' | | | | | | | | | | | | |

```cpp
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!

cin

```
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;
}
```

pi

name

tao

When `std::cin` fails!

```
cin   3 . 1 4 \n
```

```cpp
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**

3.14 pi

name

tao

When `std::cin` fails!

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

```cpp
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!

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

```cpp
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

Fabio name

tao

When `std::cin` fails!

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

```cpp
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;
}
```

3.14 — pi

Fabio — name

? — tao

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

When `std::cin` fails!

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

```cpp
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
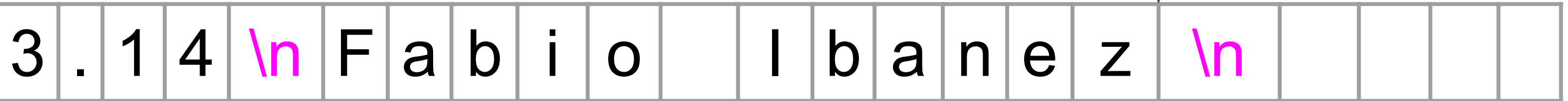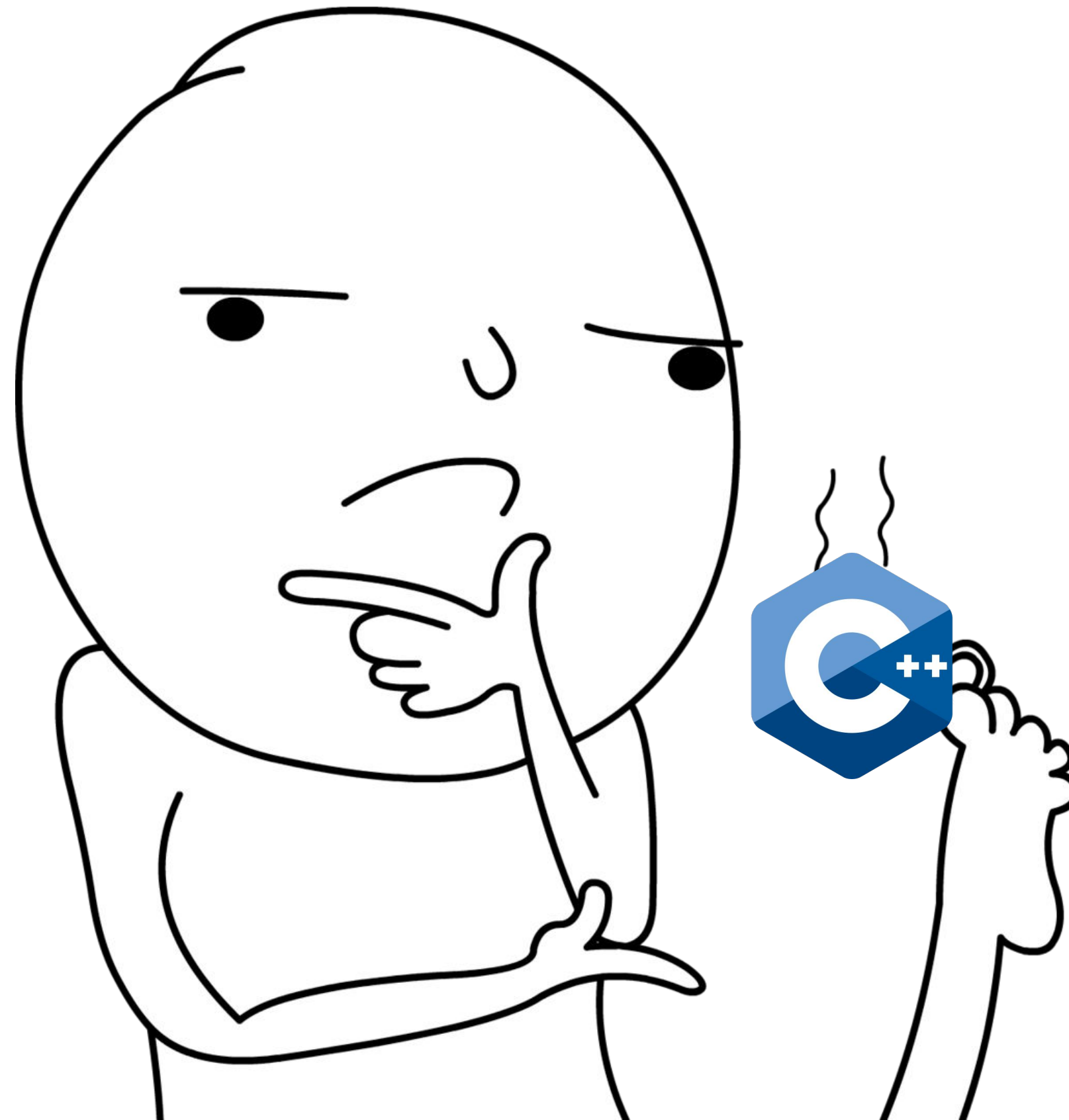
What questions do we have?

How do we fix this?

Anyone want to take a guess?

Fix?

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

```cpp
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?

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

```cpp
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?

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

```cpp
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?

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

```cpp
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

3.14   pi

""   name

🗑   tao

Fix?

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

```cpp
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';
}
```

3.14 pi

"" name

🗑 tao

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

Fix?

cin | 3 | . | 1 | 4 | \n | F | a | b | i | o | | I | b | a | n | e | z | | \n | | | |
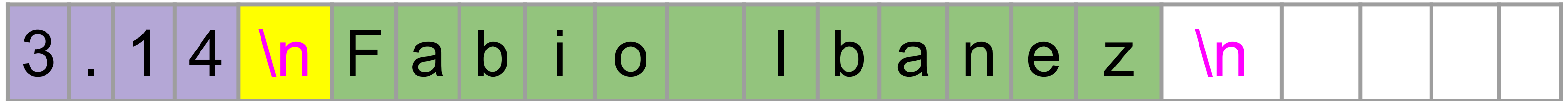
```cpp
void cinGetlineBug() {
  double pi;
  double tao;
  std::string name;
  std::cin >> pi;

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

3.14 — pi

"" — name

🗑 — tao

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?

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

```cpp
void cinGetlineBug() {
  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';
}
```

pi

name

tao

Fix?

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

```cpp
void cinGetlineBug() {
  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   I b a n e z \n
```

```cpp
void cinGetlineBug() {
  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 | | I | b | a | n | e | z | \n

```cpp
void cinGetlineBug() {
  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

— tao

Fix ✅

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

```cpp
void cinGetlineBug() {
  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 cinGetlineBug() {
  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.

If you really do need to though, it *is* possible, but not recommended.

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!

BYE, I'M OFF TO HOGWARTS

Acknowledgements

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

Thank you Aolin Zhang for pointing out the nuance about buffering!