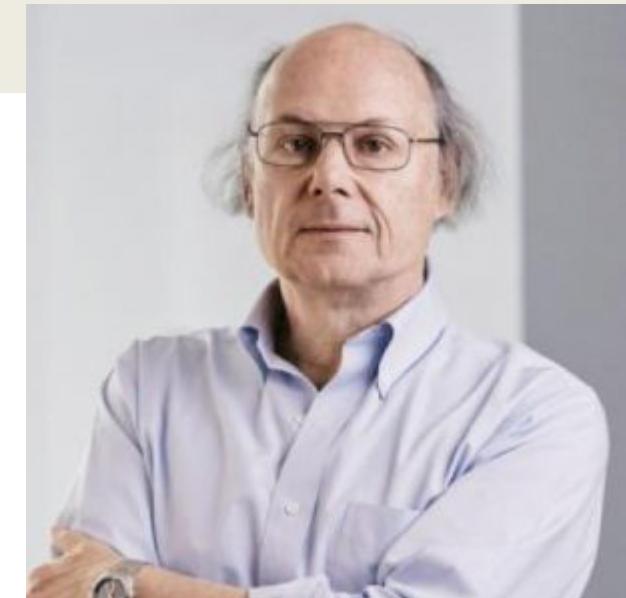


Streams

a general input/output facility for C++

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



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

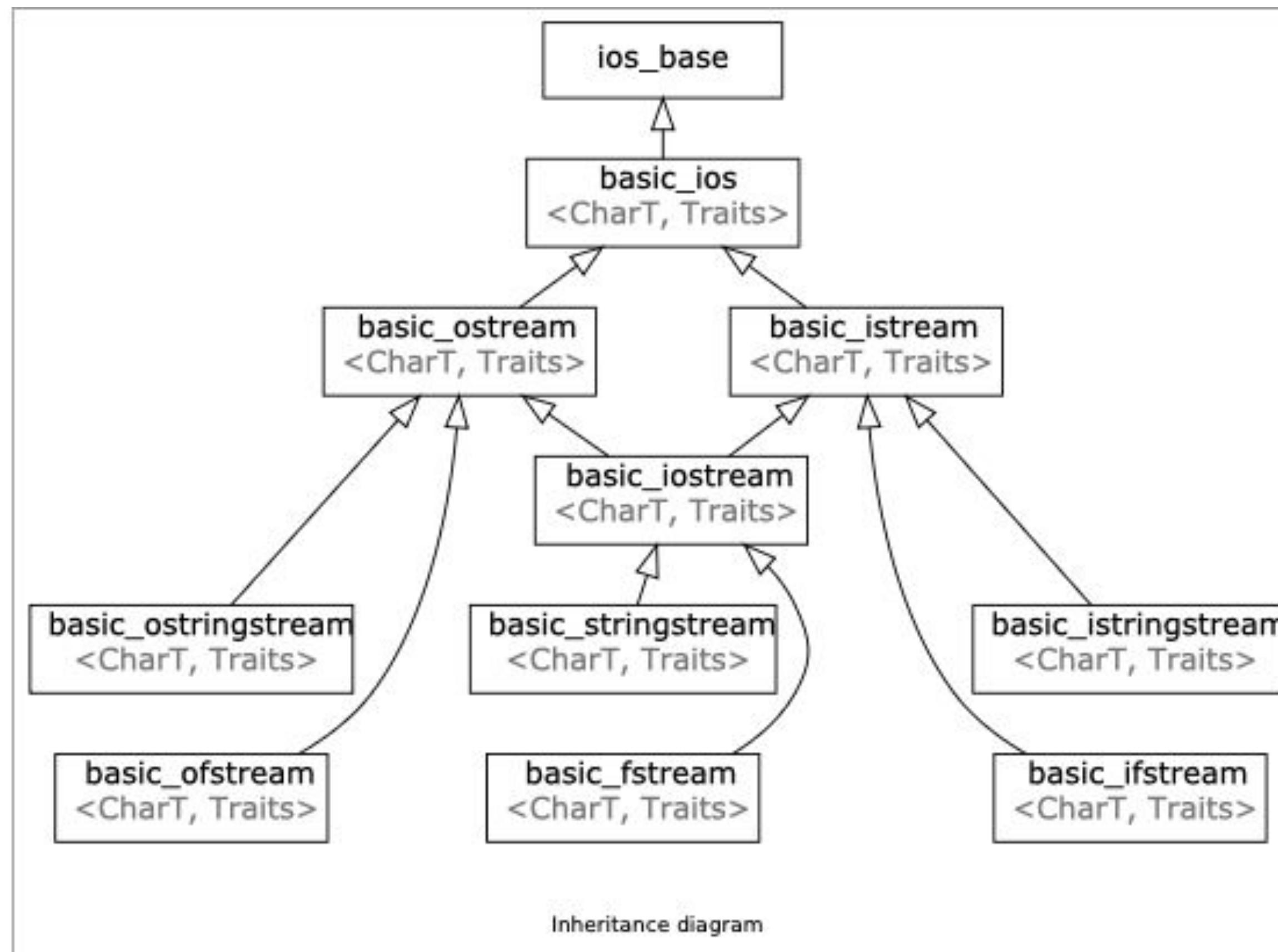
~~• cerr and clog~~ **cout and cin**

cerr: used to output errors

clog: used for non-critical event logging

read more here: [GeeksForGeeks](#)

What streams actually are



Inheritance diagram

A familiar stream!

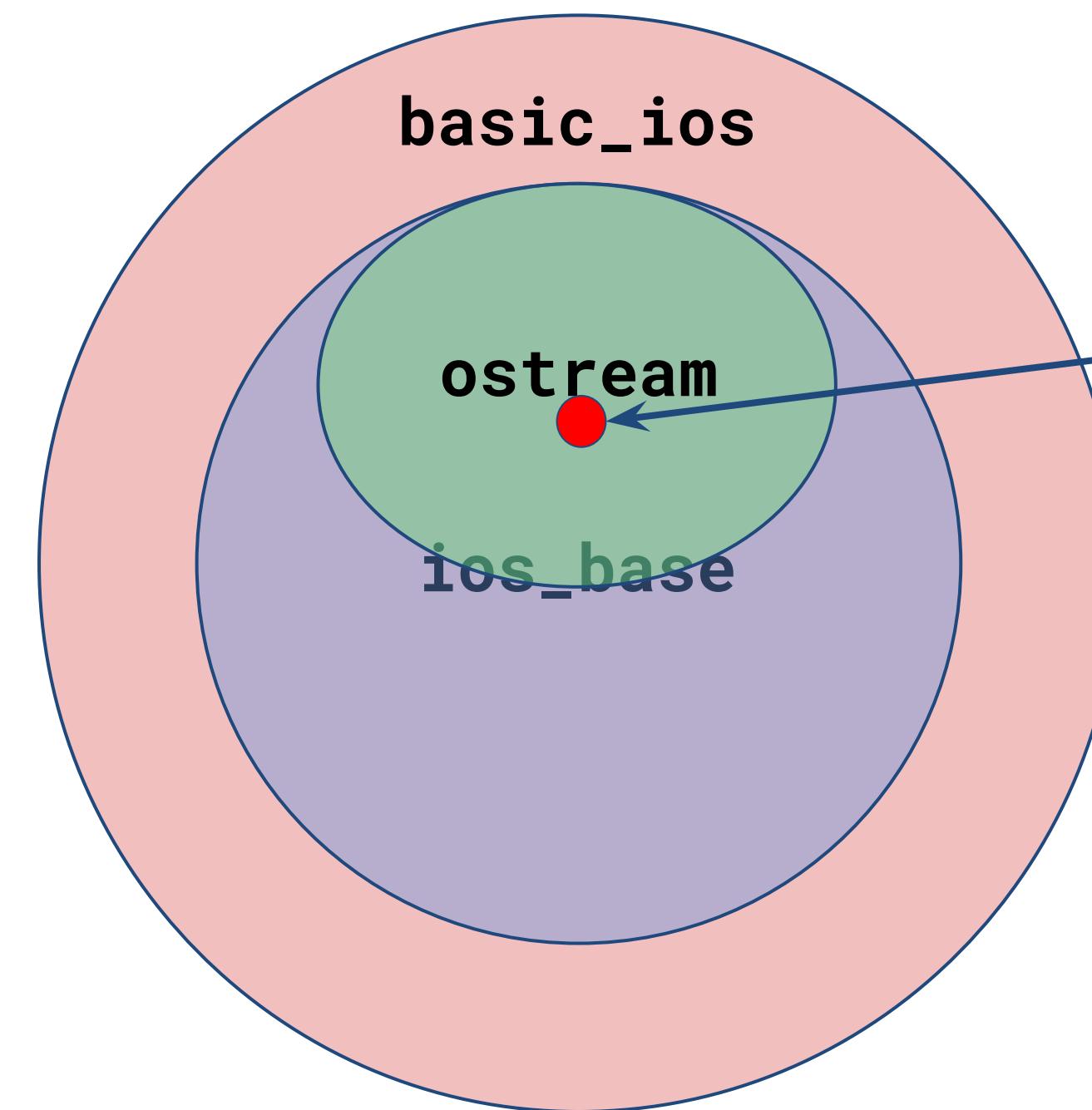
```
std::cout << "Hello, World" << std::endl;
```

This is a stream



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

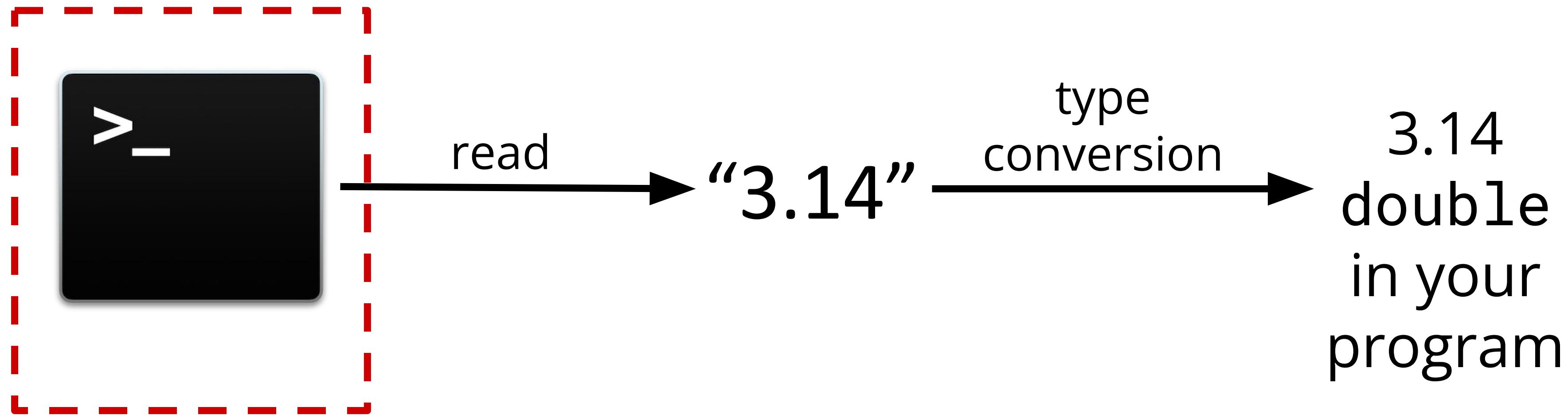
streams and types



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

An Input Stream

How do you read a double from your console?



An Input Stream

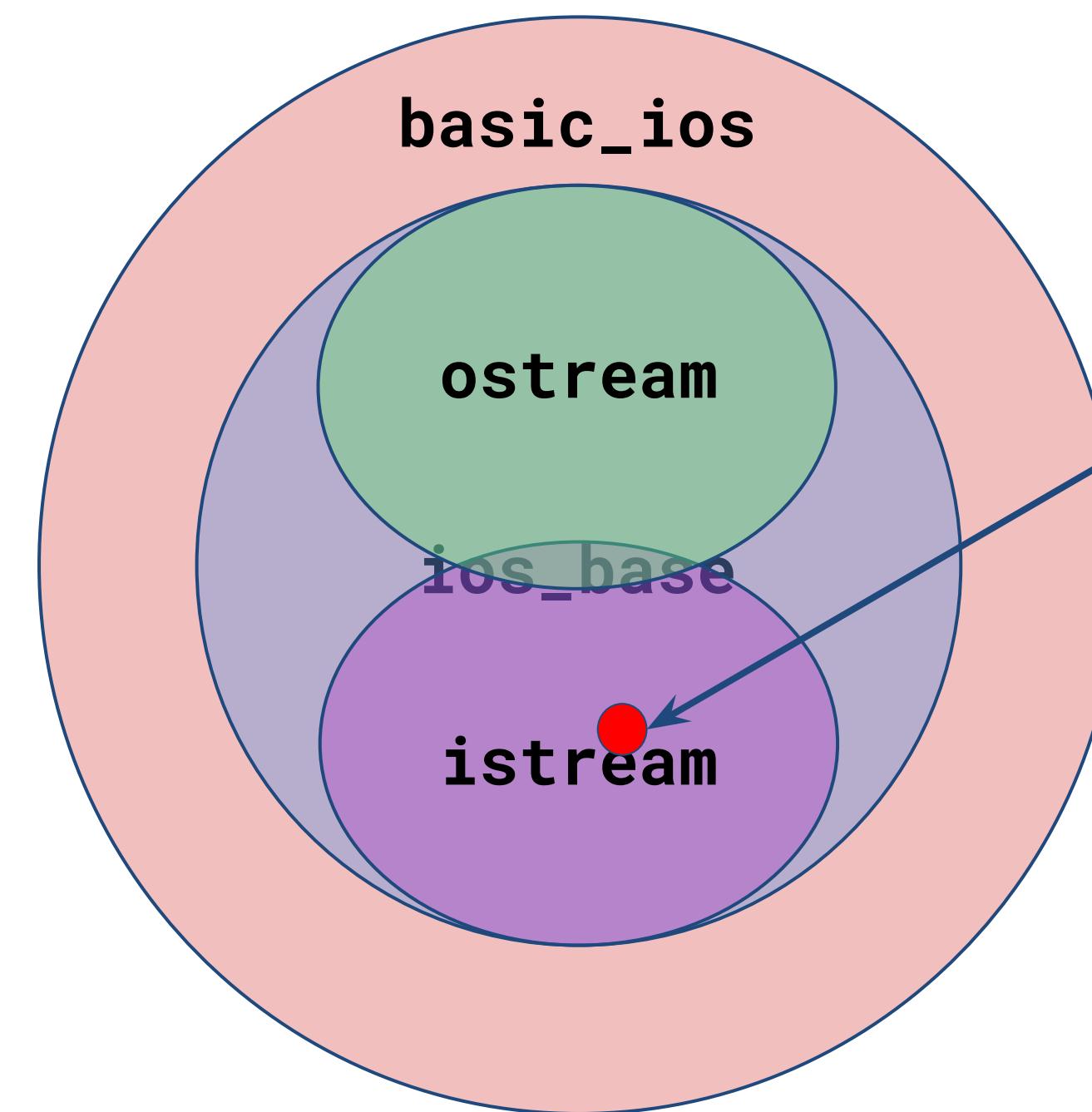
How do you read a **double** from your console?

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

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

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

streams and types



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

What streams actually are

Classifying different types of streams

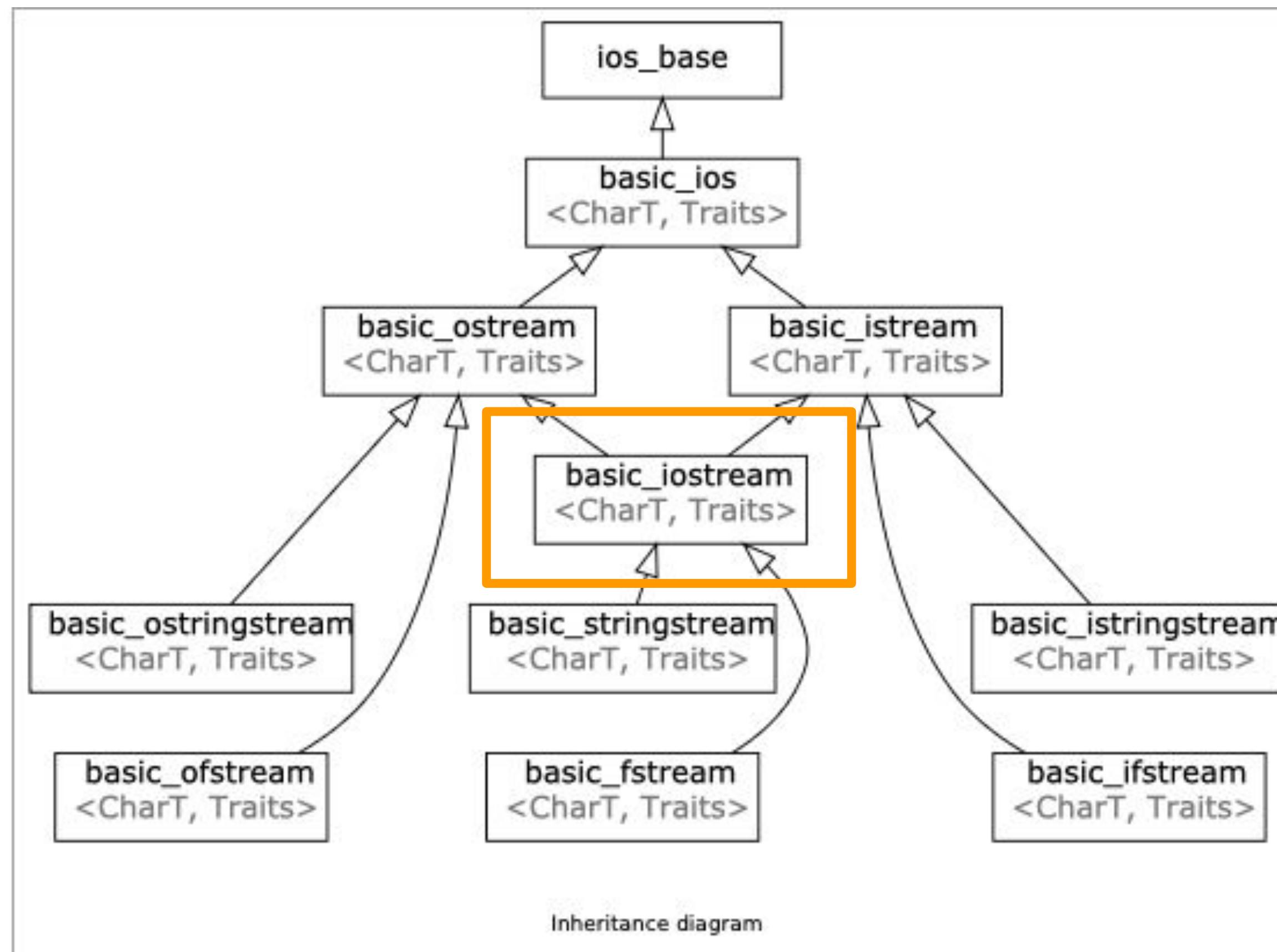
Input streams (I)

- 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)

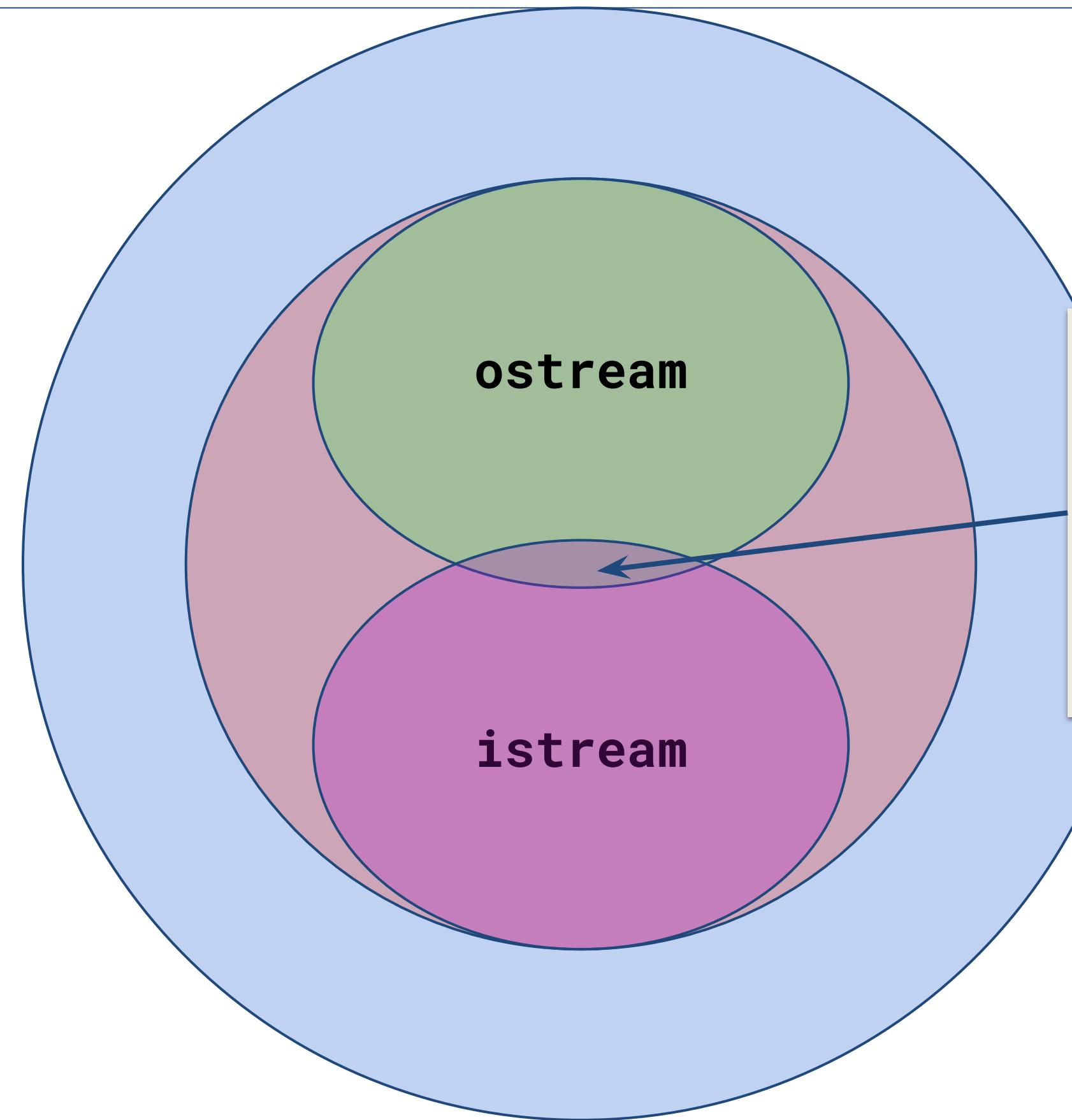
Output streams (O)

- 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



streams and types



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

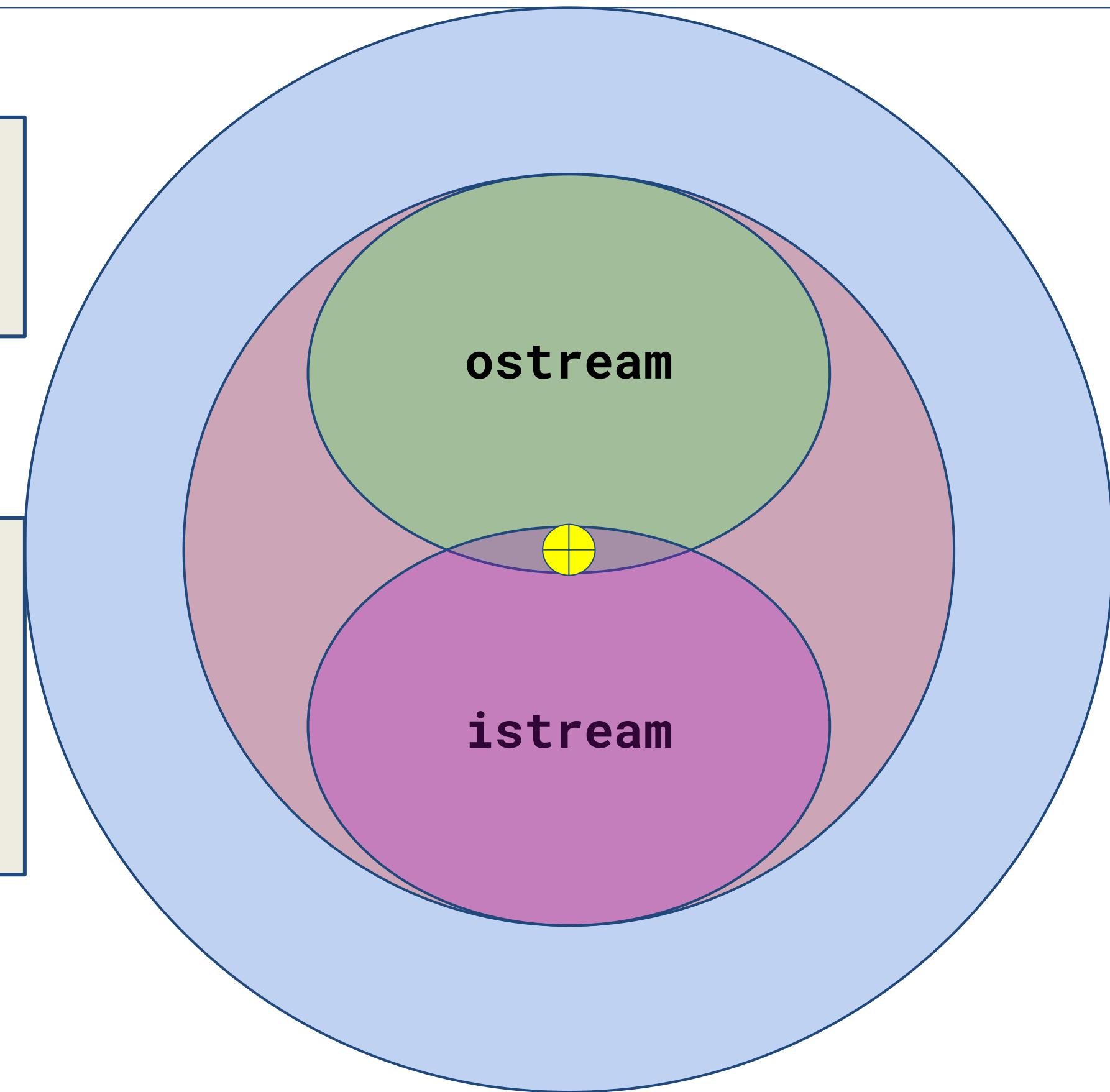
`std::stringstream`

What?

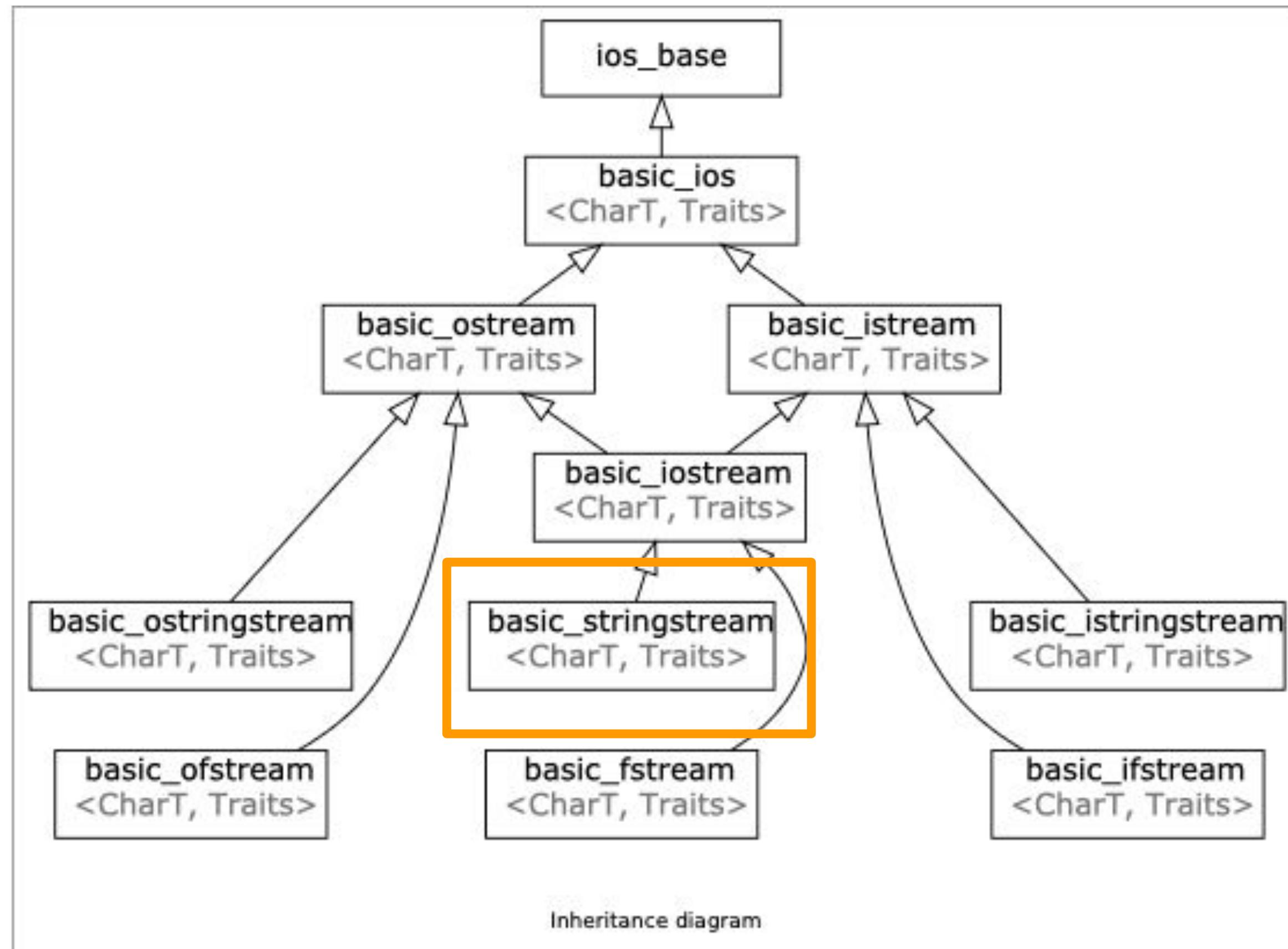
a way to treat strings as streams

Utility?

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



What streams actually are



std::stringstream example

```
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); ←
    // initialize
    // stringstream with
    // string constructor

    /// 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

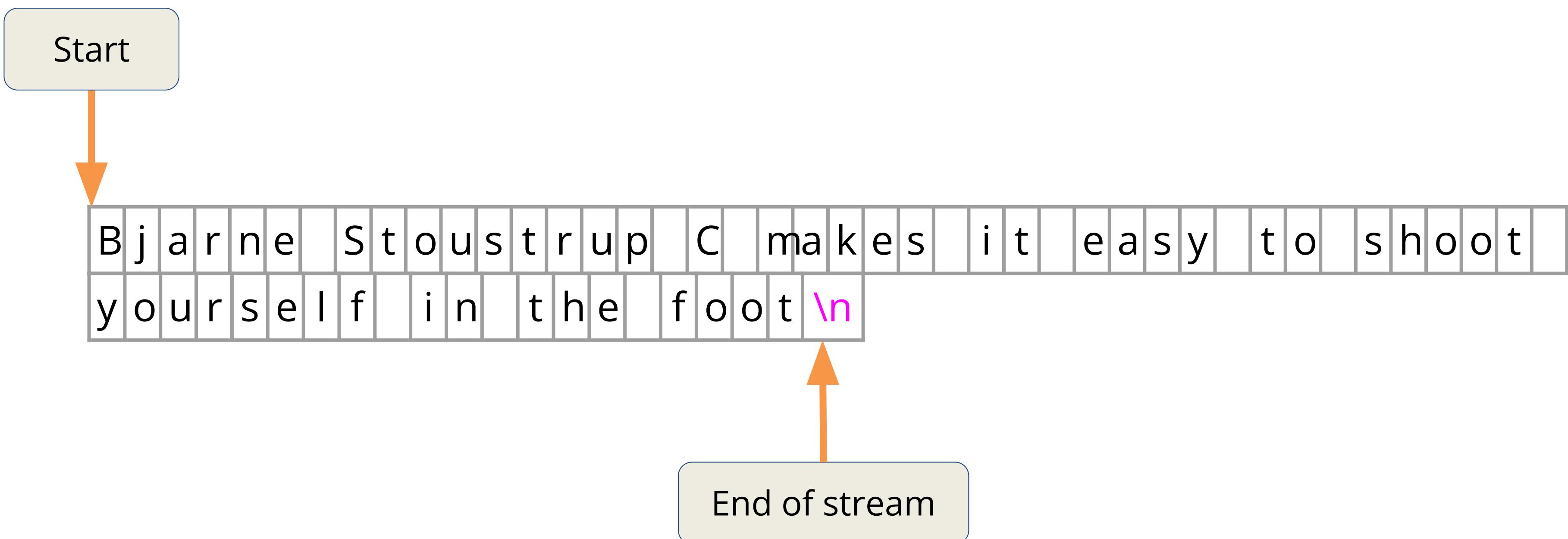
```
int main() {
    /// partial Bjarne Quote
    std::string initial_quote = "Bjarne Stroustrup C makes it easy to shoot
    yourself in the foot";

    /// create a stringstream
    std::stringstream ss;
    ss << initial_quote; ← since this is a stream we can
                            also insert the
                            initial_string like this!

    /// 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;
}
```

what the stream looks like!



std::stringstream example

```
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);

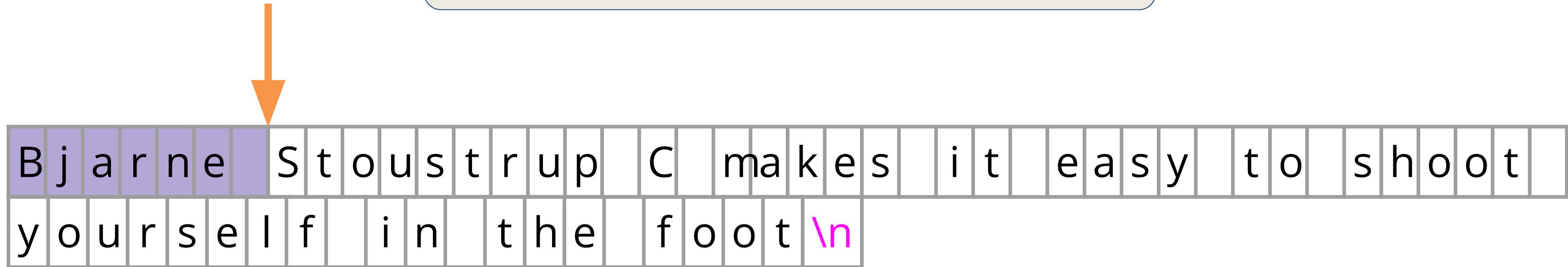
    /// data destinations
    std::string first;
    std::string last;
    std::string language, extracted_quote;

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

Remember! Streams
move data from one
place to another

what the stream looks like!

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



Bjarne

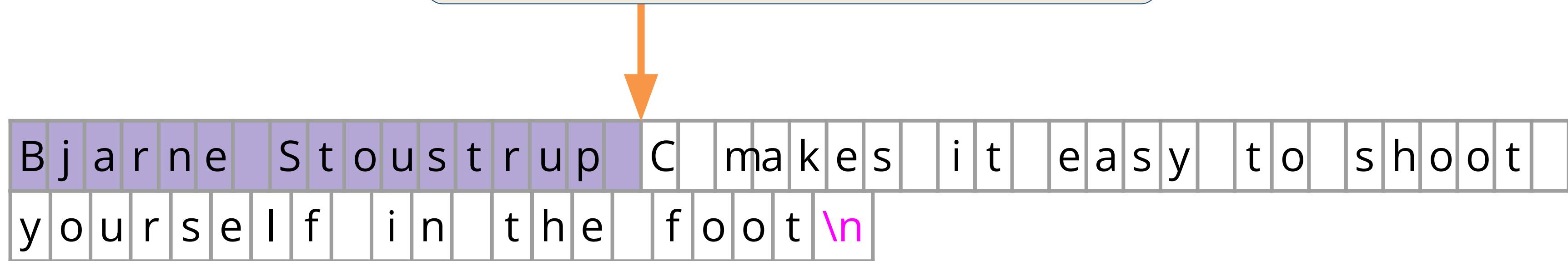
First

Last

Language

what the stream looks like!

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



Bjarne

Stroustrup

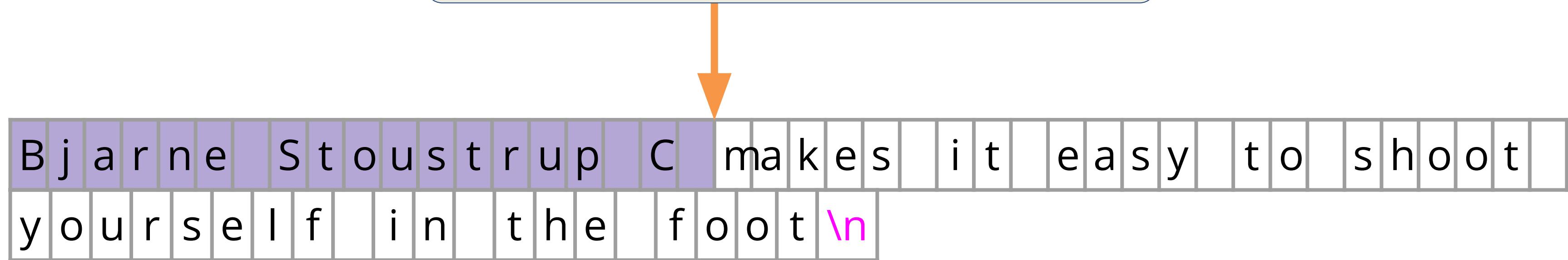
First

Last

Language

what the stream looks like!

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



Bjarne

First

Stroustrup

Last

C

Language

std::stringstream example

```
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; ← We want to extract the quote!

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

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**.
- The **delim** char is by default ‘ \n ’.
- **getline()** ***consumes*** the **delim** character!
 - PAY ATTENTION TO THIS :)

std::stringstream example

```
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);

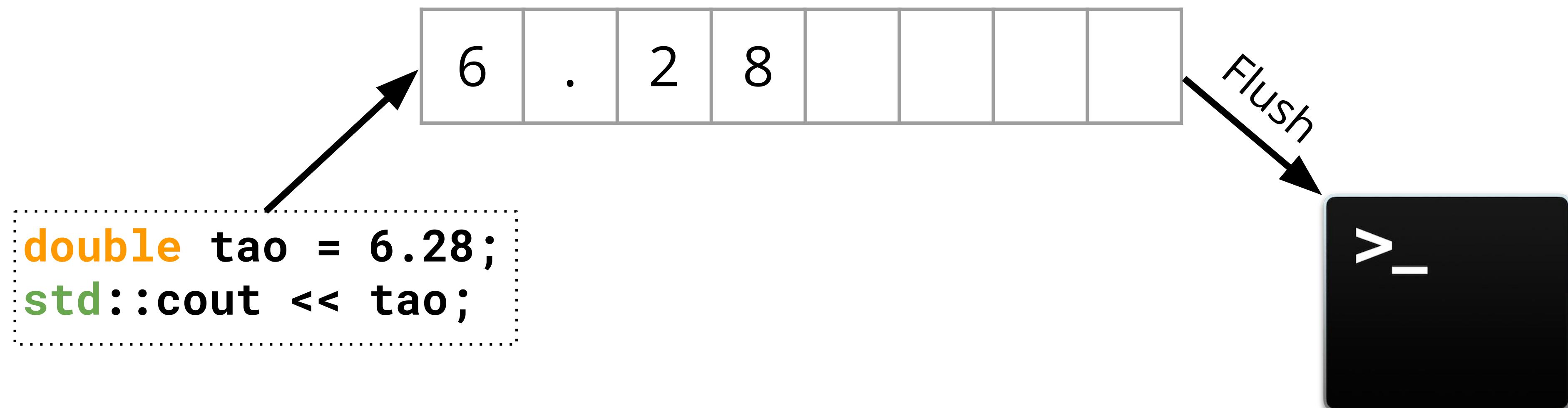
    /// data destinations
    std::string first;
    std::string last;
    std::string language, extracted_quote;
    ss >> first >> last >> language;
    std::getline(ss, extracted_quote);
    std::cout << first << " " << last << " said this: '" << language << " "
    extracted_quote + "'" << std::endl;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

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

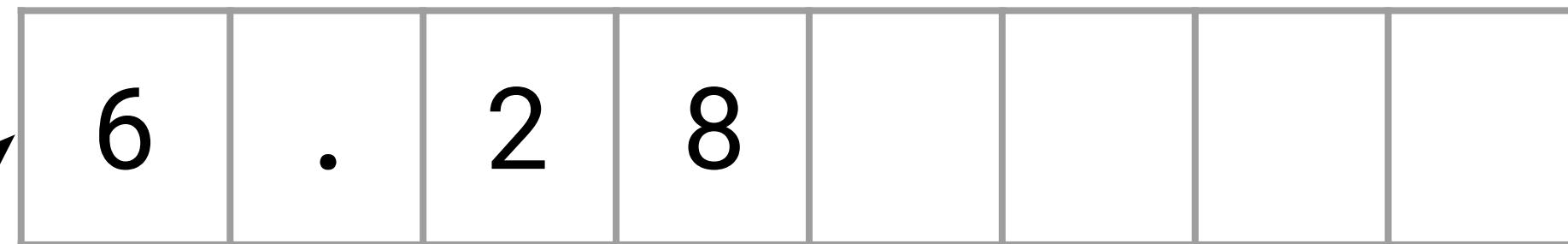
Zooming in on Output Streams!

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

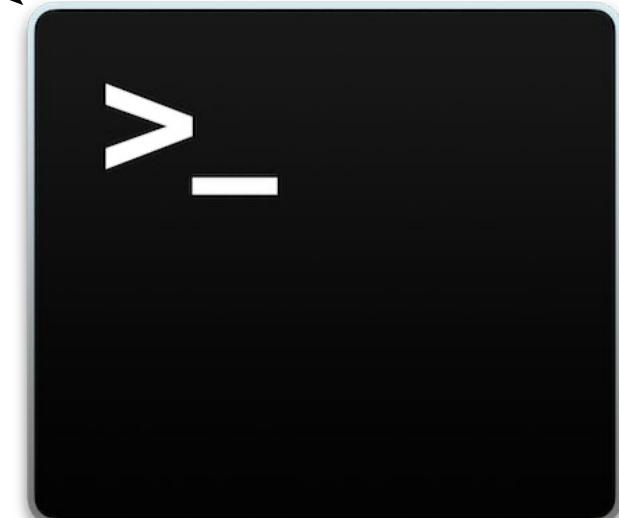


Zooming in on Output Streams!

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



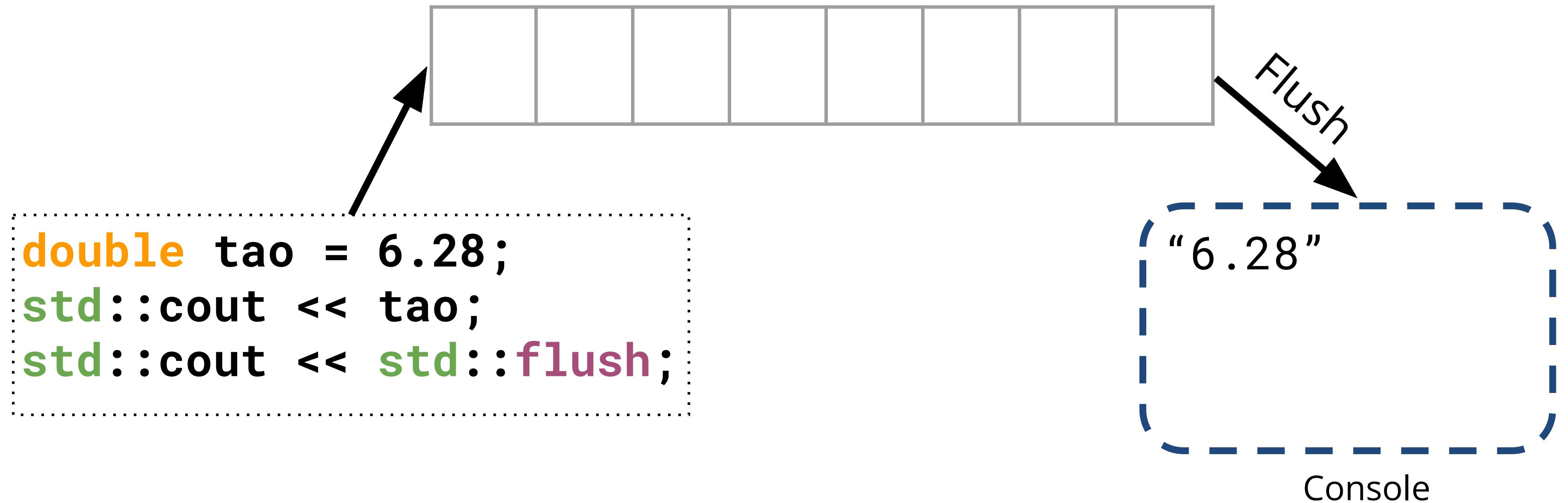
Flush



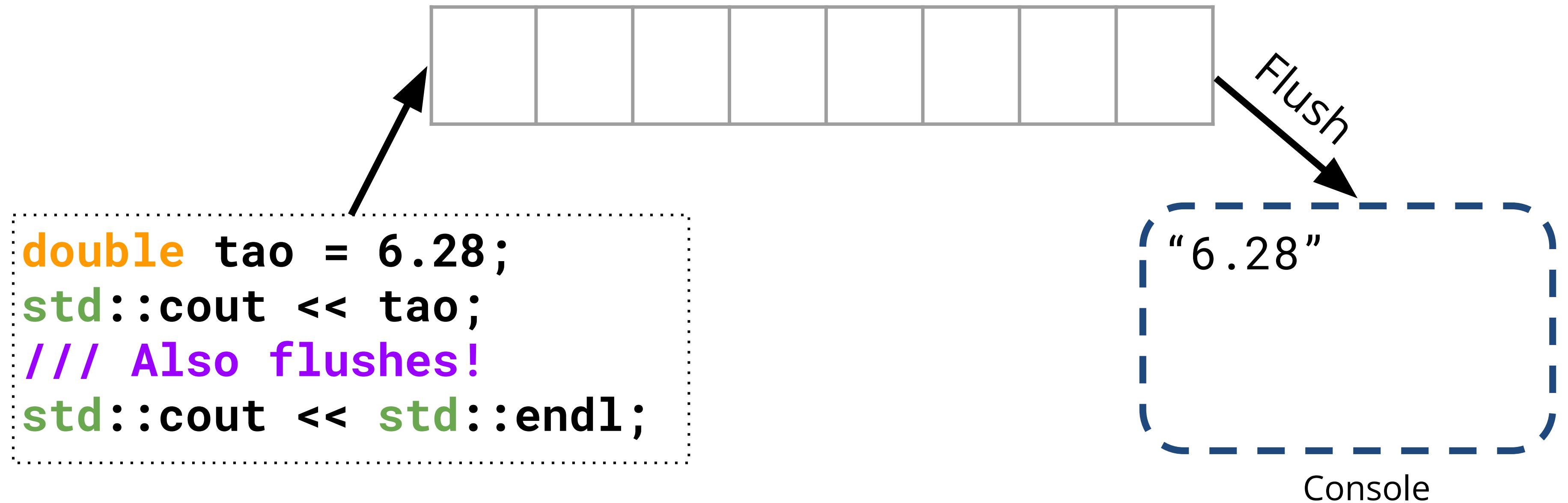
std::cout stream is
line-buffered

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

Zooming in on Output Streams!



Zooming in on Output Streams!



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**

' \n '

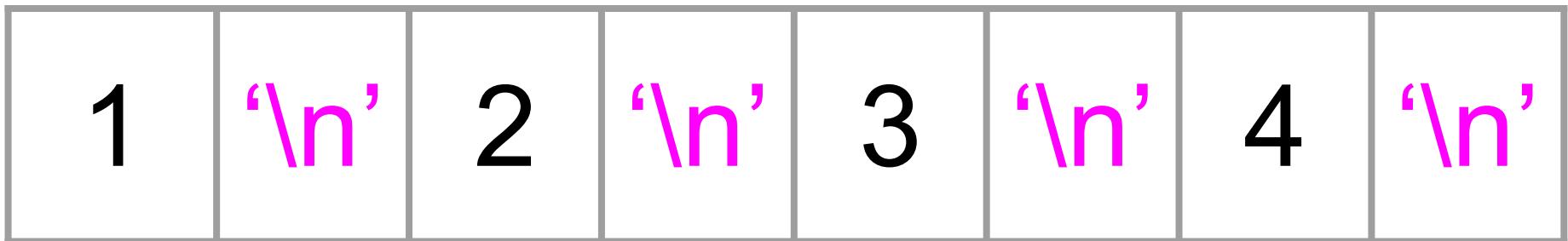


```
int main()
{
    for (int i=1; i <= 5; ++i) {
        std::cout << i << '\n';
    }
    return 0;
}
```

4
i

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

intermediate buffer



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

Output:

' \n '



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

5
i

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

intermediate buffer



😱😱 Our
intermediate buffer
is full!

Output:

' \n '



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

5
i

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

intermediate buffer



C++: FLUSH

Output:

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

' \n '



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

5
i

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

intermediate buffer



Yay!

Output:

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

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

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!

Use ‘\n’!



```
std::cout << "Draaaakkkeeeeeeee" << std::endl;
```



```
std::cout << "Draaaakkkeeeeeeee" << '\n';
```

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;
}
```

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;
}
```

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;
}
```

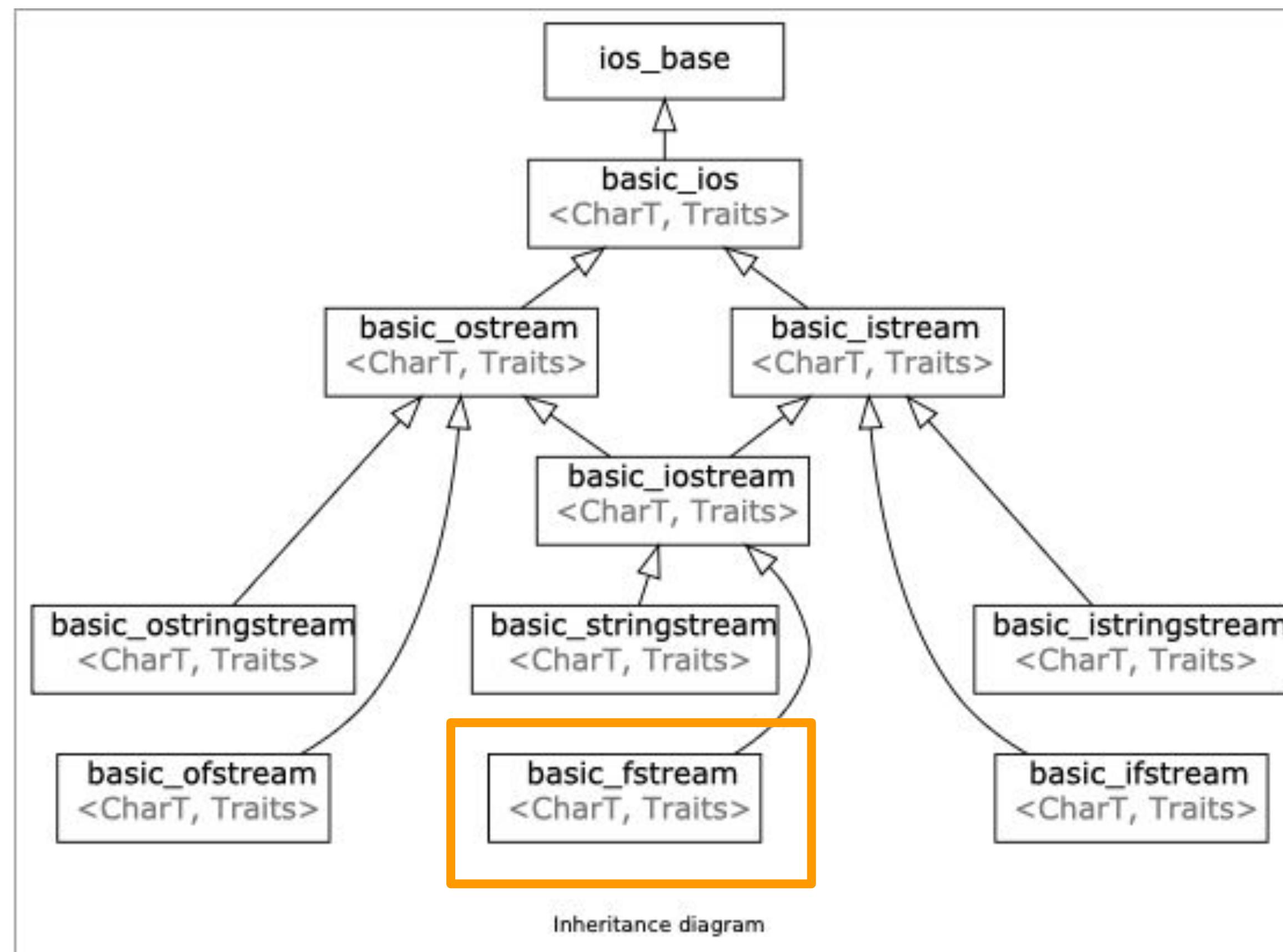
Flag specifies you want to
append, not truncate!

Input File Streams

```
int inputStreamExample() {
    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



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

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
- 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



```
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

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!

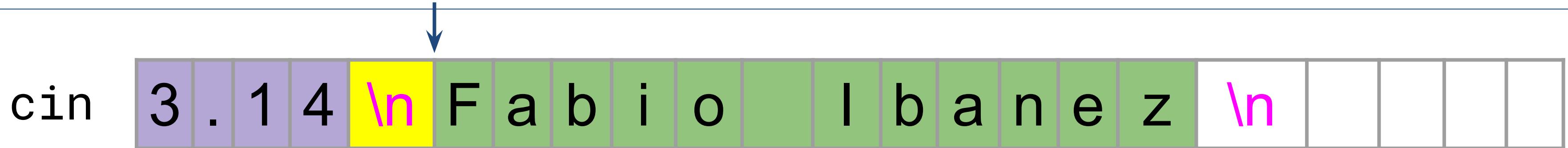
cin 3 . 1 4 \n F a b i o I b a n e z \n

```
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

Fix?

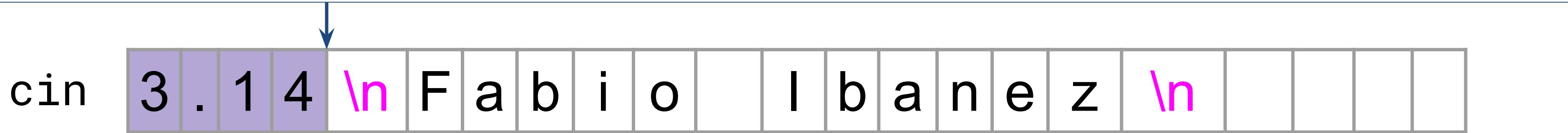


```
void cinGetlineBug() {  
    double pi;  
    double tao;  
    std::string name;  
    std::cin >> pi;  
    std::getline(std::cin, name);  
    std::cin >> tao;  
    std::cout << "my name is : " << na  
" << tao  
        << " pi is : " << pi << '\n';  
}
```

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

3.14 pi
"" name
tao

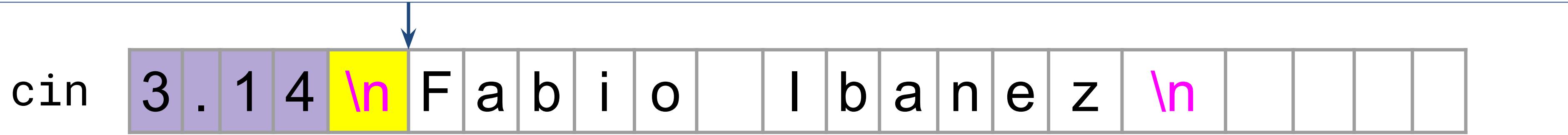
Fix?



```
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 



```
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

```
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 name
Ibanez
tao

Fix

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

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

3.14

Fabio
Ibanez

pi

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