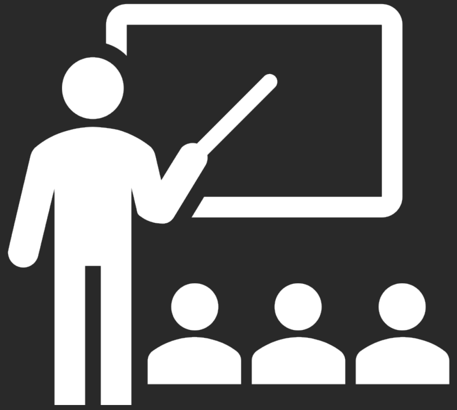
Streams II

CS 106L Fall 2019 – Avery Wang and Anna Zeng

Game Plan



- iostreams
- implementing getInteger
- interlude: C++ types
- file streams
- manipulators
- overloading << and >>

When state is not good, streams do not work!

G Good bit: ready for read/write.

F Fail bit: previous operation failed, future operations fail.

E EOF bit: reached end of buffer content, future operations fail.

B Bad bit: external error, future operations fails.

Third attempt: complete error-checking.

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

     int result;
     iss >> result;
     if (iss.fail()) throw domain_error(…);

     char remain;
     iss >> remain;
     if (!iss.fail()) throw domain_error(…);
     return result;
}
```

Check if the operation failed (due to type mismatch).

Very helpful shortcut.

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

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

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

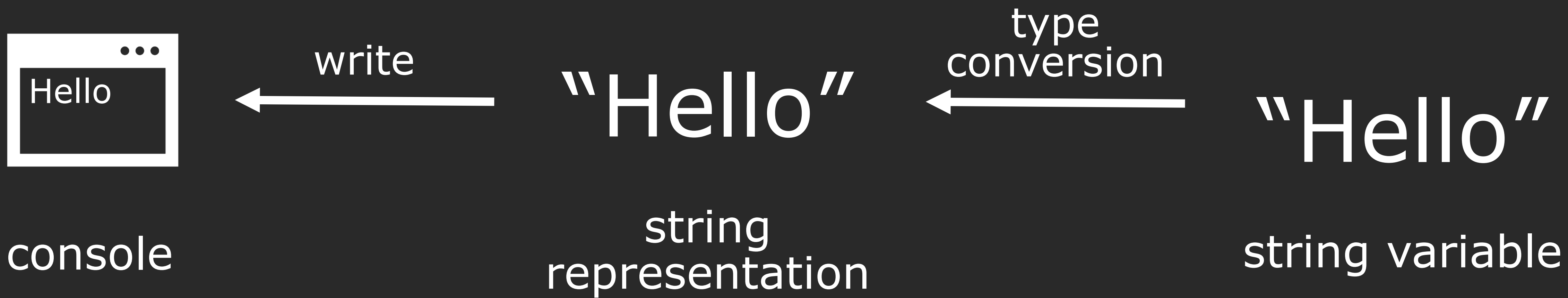
Third attempt: complete error-checking.

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

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

    return result;
}
```

Notice the short circuiting!

cout and cin

Key difference: there is an external source.



Hello

console

write

“Hello”

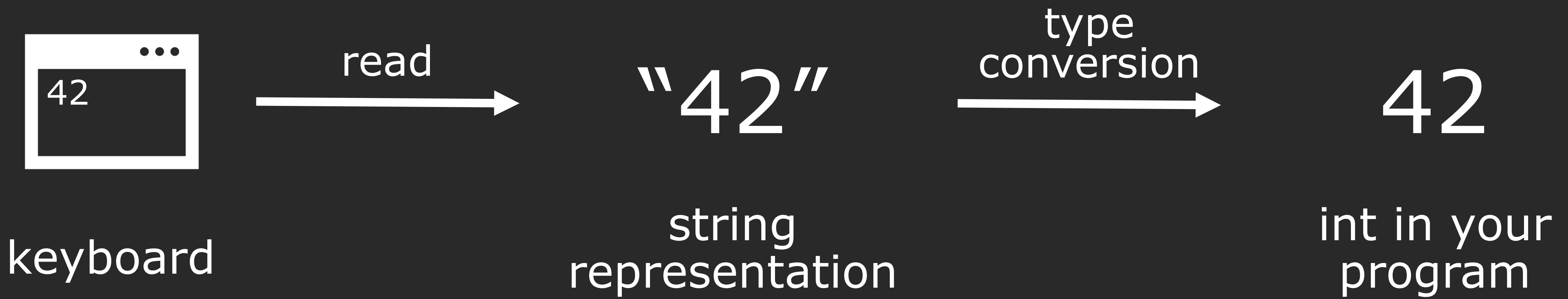
string representation

type conversion

“Hello”

string variable

Data is sent between the external source and the buffer.



42

keyboard

read

“42”

string representation

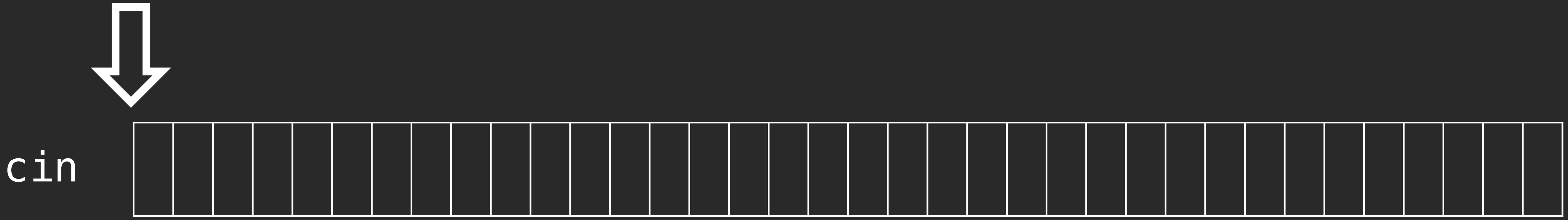
type conversion

42

int in your program

There are four standard iostreams.

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



G F E B

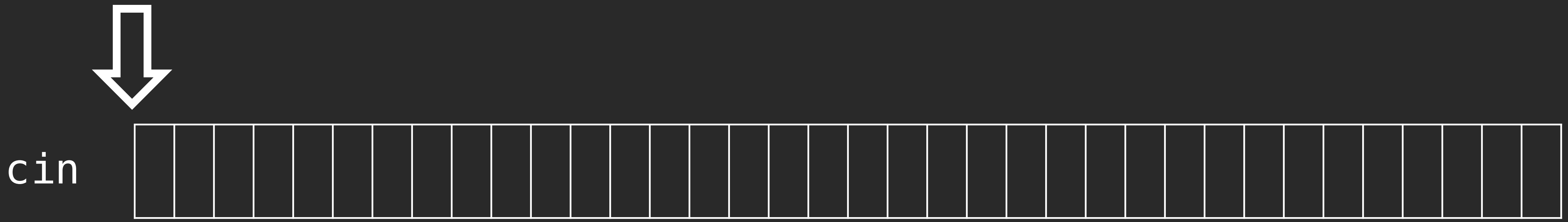
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name
(string)
???

response
(string)
???

age
(int)
???

The lecture code included cout statements.



G F E B

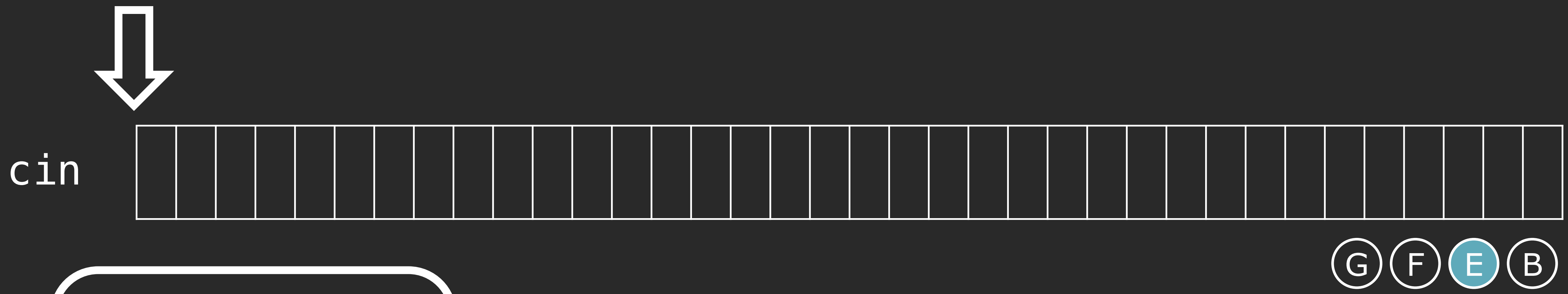
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string) ???

response (string) ???

age (int) ???

The lecture code included cout statements.



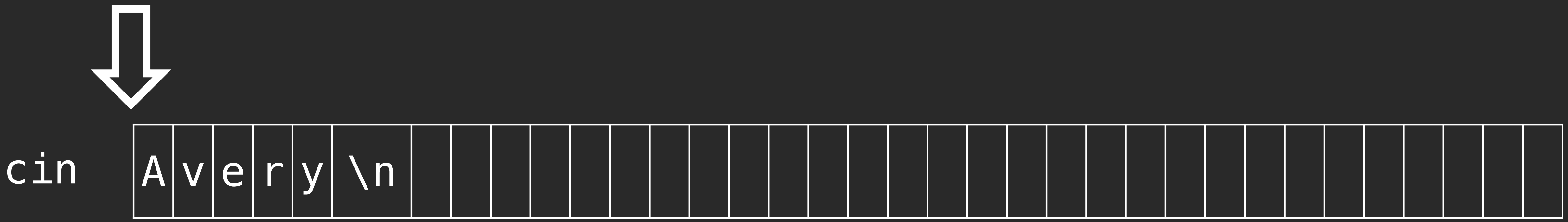
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): ???

response (string): ???

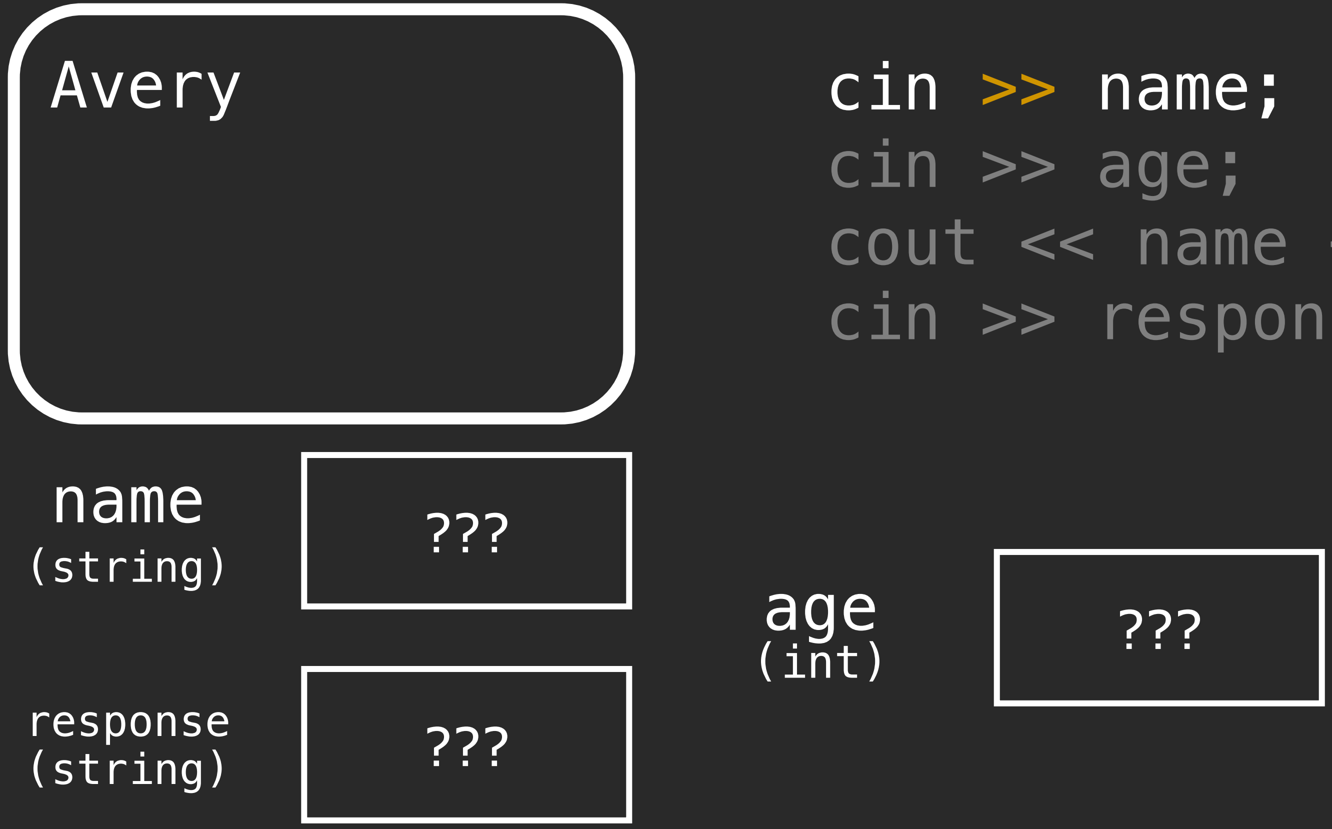
age (int): ???

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

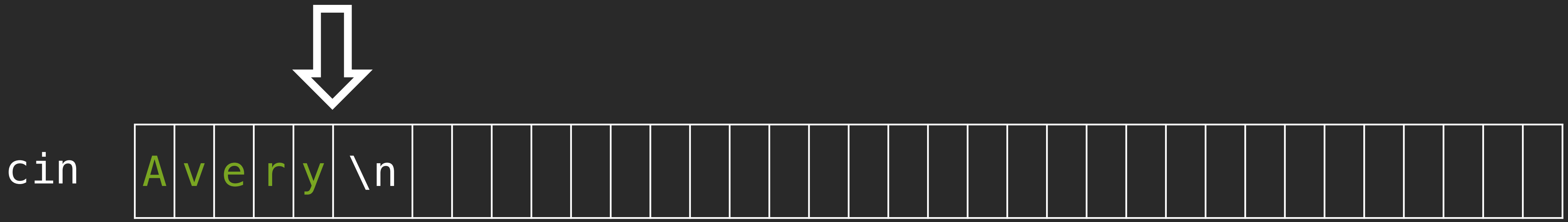


G F E B

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



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



G F E B

Avery

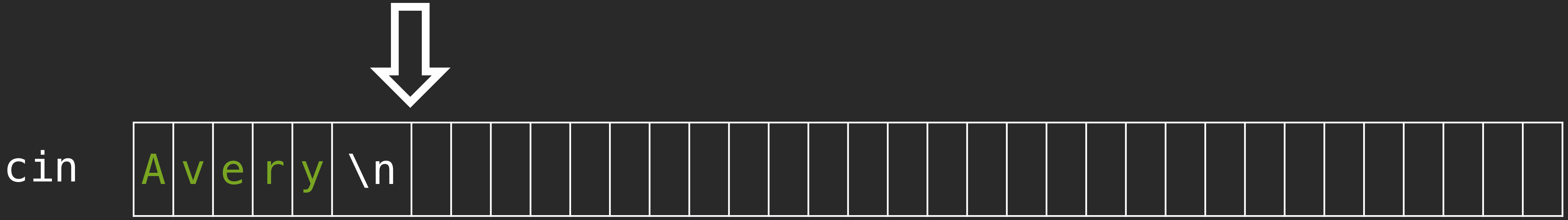
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string) "Avery"

response (string) ???

age (int) ???

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



cin A v e r y \n

G F E B

```
Avery
|
```

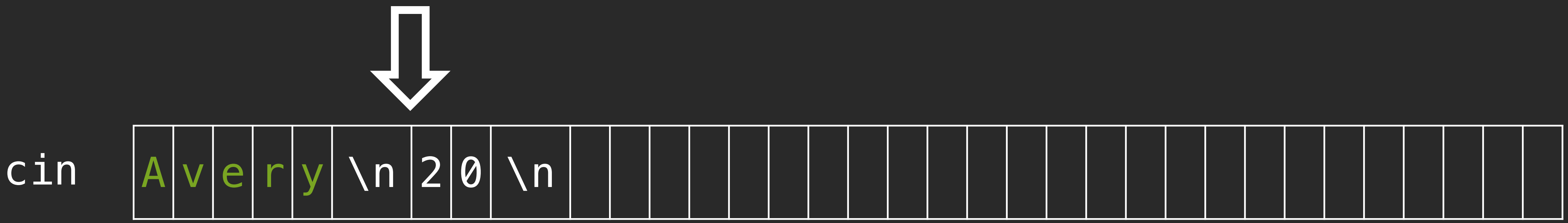
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string) “Avery”

response (string) ???

age (int) ???

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



cin

```
A v e r y \n 2 0 \n
```

G F E B

```
Avery
20
```

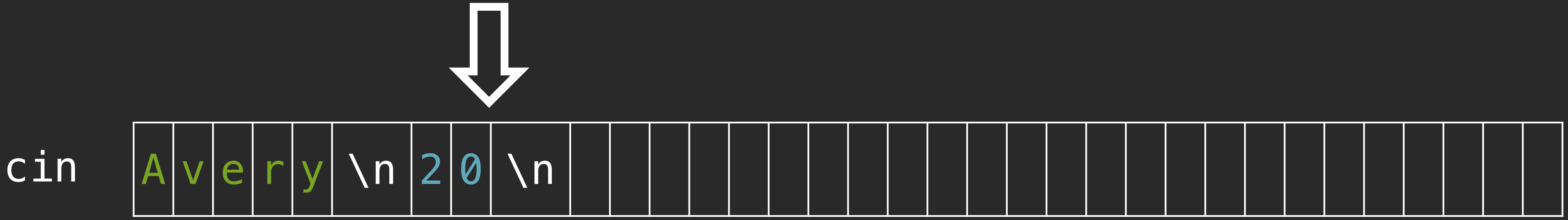
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): "Avery"

response (string): ???

age (int): ???

Everything I type is transferred to the buffer.



cin	A	v	e	r	y	\n	2	0	\n																									

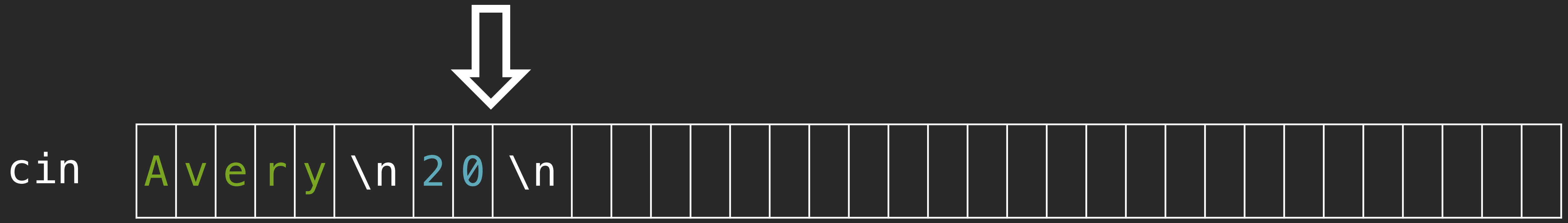
G F E B

```
Avery
20
```

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

name (string)	"Avery"
response (string)	???
age (int)	20

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



cin

A	v	e	r	y	\n	2	0	\n

G F E B

```
Avery
20
```

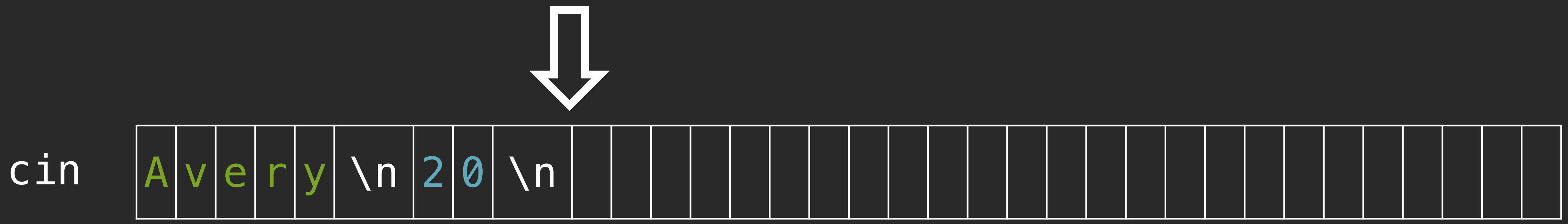
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): “Avery”

response (string): ???

age (int): 20

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



cin

A	v	e	r	y	\n	2	0	\n

G F E B

```
Avery
20
Avery20
```

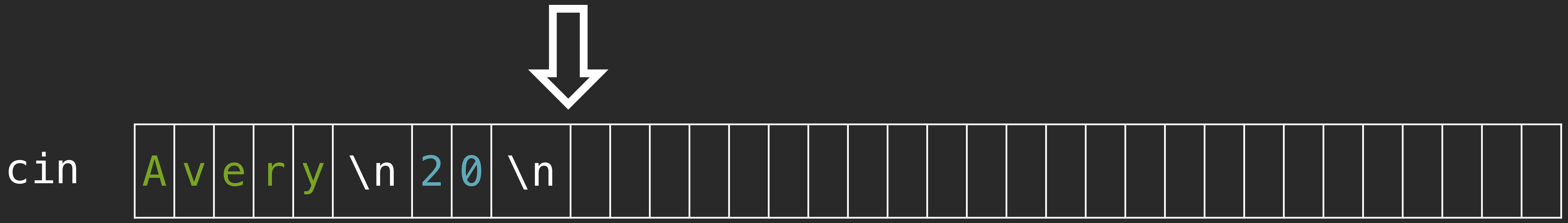
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): “Avery”

response (string): ???

age (int): 20

But attempting reading again will flush cout.



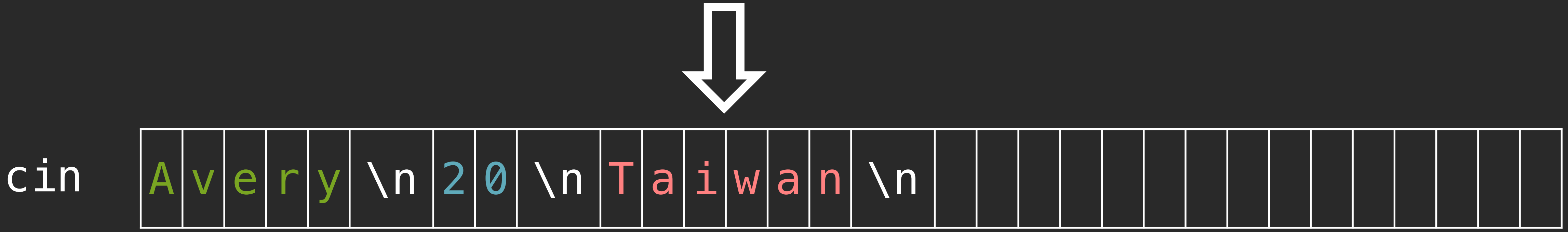
G F E B

```
Avery
20
Avery20|
```

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

name (string)	"Avery"
response (string)	???
age (int)	20

We prompt the user again.



cin

A	v	e	r	y	\n	2	0	\n	T	a	i	w	a	n	\n															

G F E B

```
Avery
20
Avery20Taiwan
```

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

name
(string)

"Avery"

response
(string)

"Taiwan"

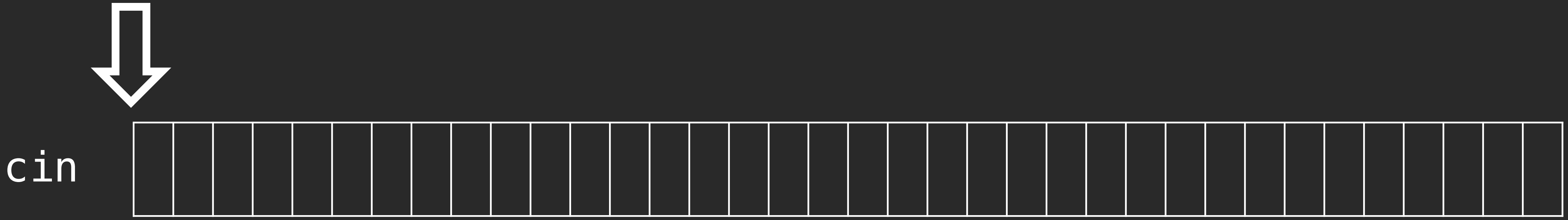
age
(int)

20

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

Example

when input streams go wrong



G F E B

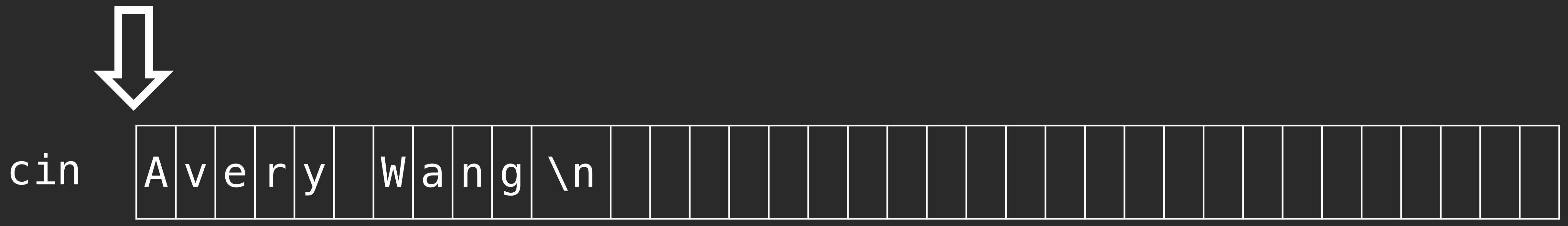
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name
(string) ???

response
(string) ???

age
(int) ???

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



cin	A	v	e	r	y		W	a	n	g	\n																								

G F E B

```
Avery Wang
```

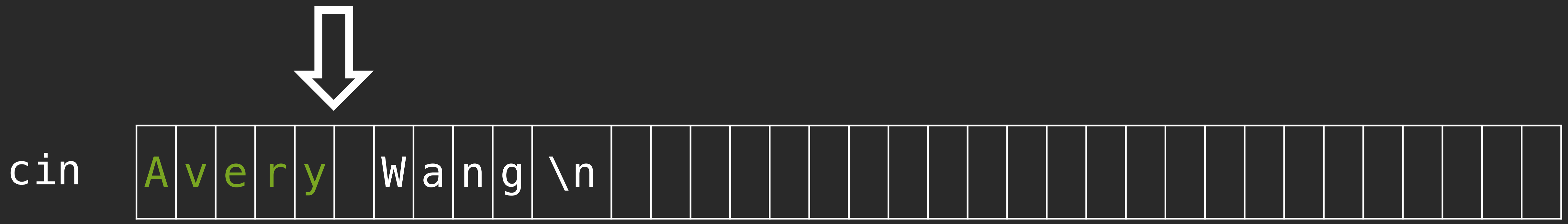
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): ???

age (int): ???

response (string): ???

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



G F E B

```
Avery Wang
```

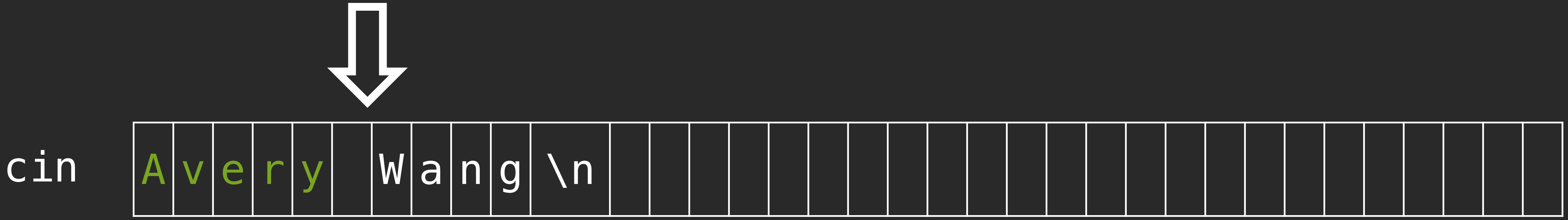
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): "Avery"

response (string): ???

age (int): ???

Remember cin reads up to a whitespace.



cin

A v e r y   W a n g \n

G F E B

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

Avery Wang

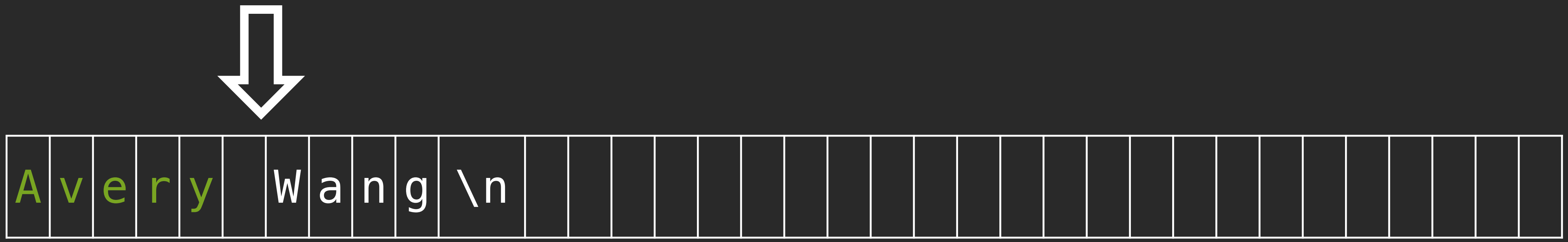
name (string) “Avery”

response (string) ???

age (int) ???

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

cin



G F E B

Avery Wang

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

name
(string)

“Avery”

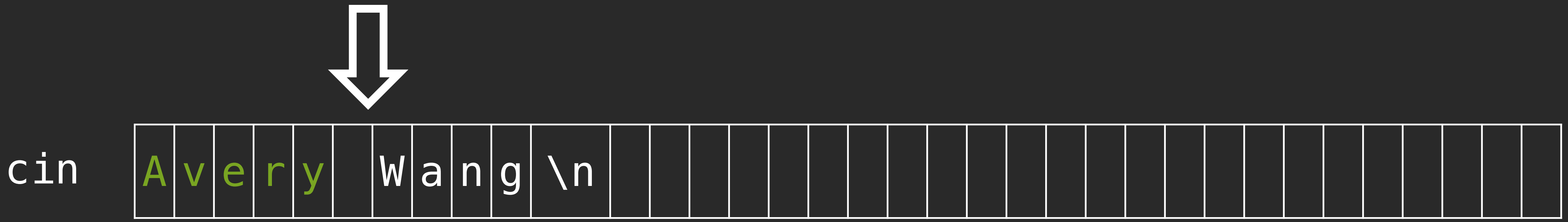
response
(string)

???

age
(int)

???

It tries to read in an int, but fails.



cin

A	v	e	r	y		W	a	n	g	\n

G F E B

Avery Wang

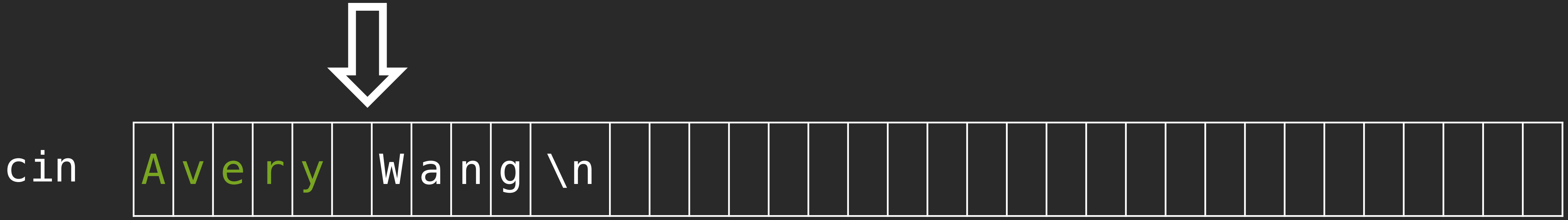
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): “Avery”

response (string): ???

age (int): ???

It tries to read in an int, but fails.



cin																																				
	A	v	e	r	y		W	a	n	g	\n																									

G F E B

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

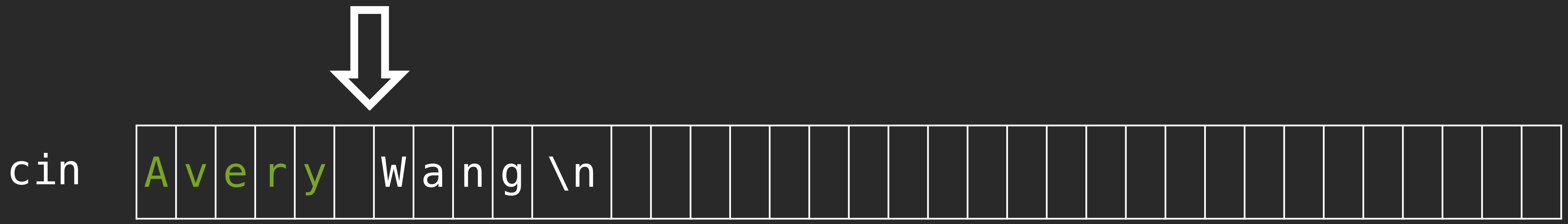
```
Avery Wang
```

name
(string)
"Avery"

response
(string)
???

age
(int)
???

The fail bit is turned on.



```
cin  A v e r y   W a n g \n
```

G F E B

```
Avery Wang
Avery -2736262
```

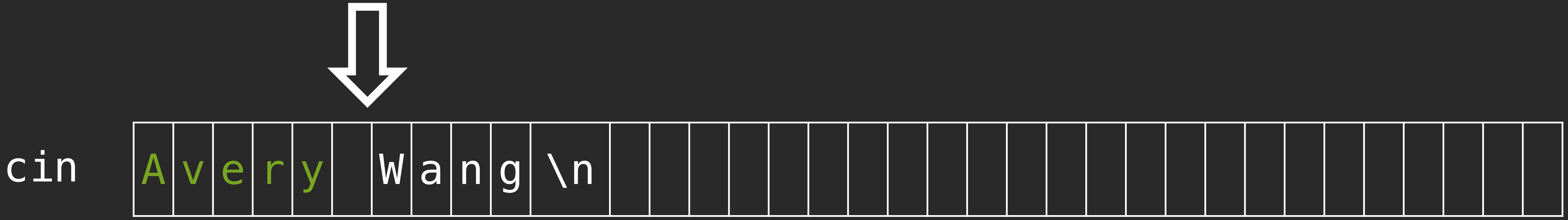
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string): "Avery"

response (string): ???

age (int): ???

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



cin A v e r y   W a n g \n

G F E B

```
Avery Wang
Avery -2736262
```

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

name (string) "Avery"

response (string) ???

age (int) ???

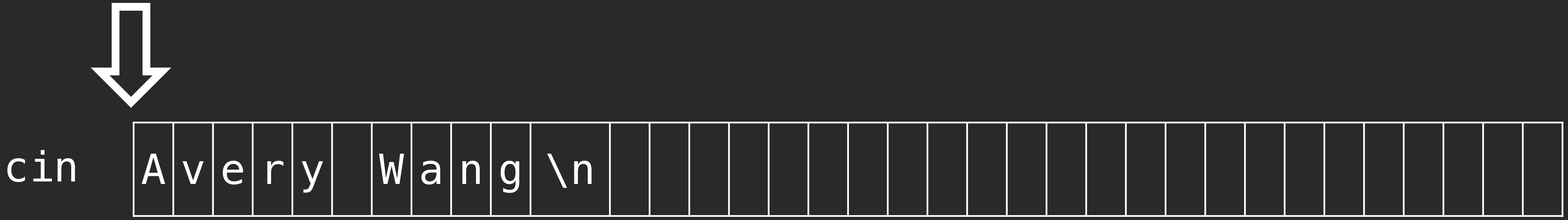
Worst part, since the fail bit is on, all future cin operations fail.

There are 3 reason why >> with cin is a nightmare.

1. cin reads the entire line into the buffer but gives you whitespace-separated tokens.

2. Trash in the buffer will make cin not prompt the user for input at the right time.

3. When cin fails, all future cin operations fail too.



cin

A	v	e	r	y		W	a	n	g	\n																									

G F E B

```
Avery Wang
```

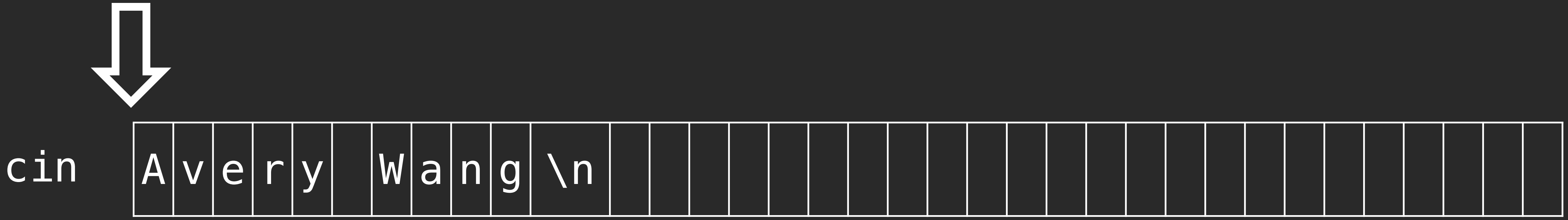
```
cin >> name;
cin >> age;
cout << name << age;
cin >> response;
```

name (string) ???

age (int) ???

response (string) ???

We want to read the entire line the user typed in into name.



cin

A	v	e	r	y		W	a	n	g	\n																									

G F E B

Avery Wang

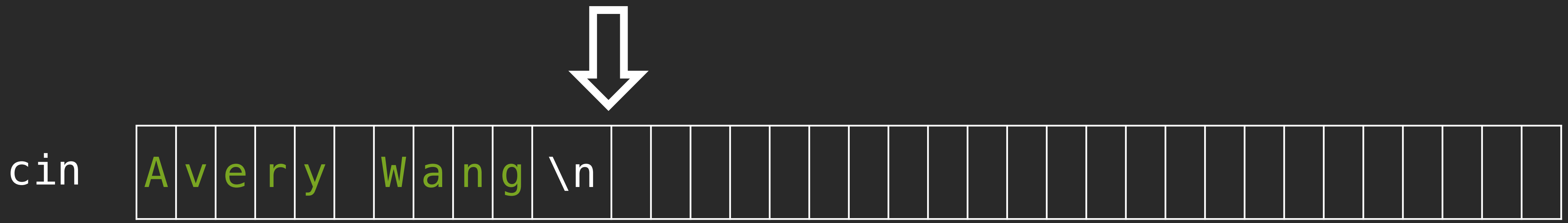
```
getline(cin, name, '\n');
cin >> age;
cout << name << age;
cin >> response;
```

name
(string)
???

response
(string)
???

age
(int)
???

This reads up to the \n.



G F E B

```
Avery Wang
```

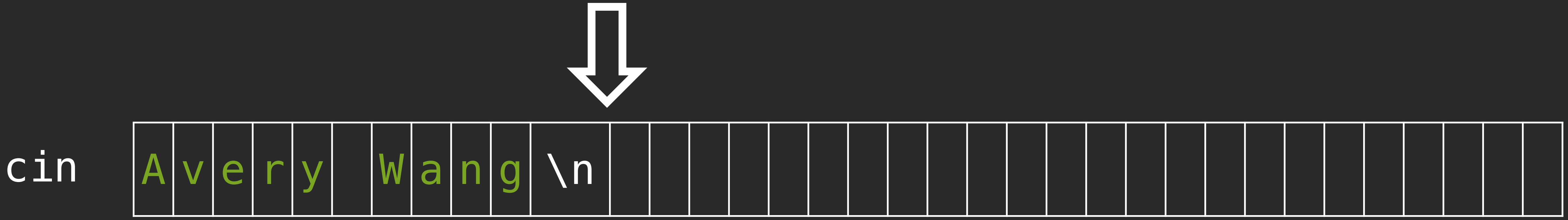
```
getline(cin, name, '\n');
cin >> age;
cout << name << age;
cin >> response;
```

name (string): "Avery Wang"

age (int): ???

response (string): ???

Note that getline will skip the delimiter (in this case, '\n')



cin

A	v	e	r	y		W	a	n	g	\n

G F E B

```
Avery Wang
|
```

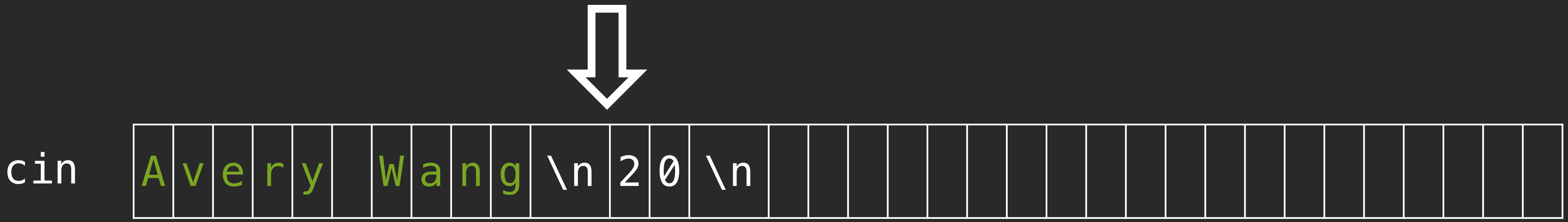
```
getline(cin, name, ‘\n’);
cin >> age;
cout << name << age;
cin >> response;
```

name (string): “Avery Wang”

response (string): ???

age (int): ???

Prompt user again!



cin	A	v	e	r	y		W	a	n	g	\n	2	0	\n

G F E B

```
Avery Wang
20
```

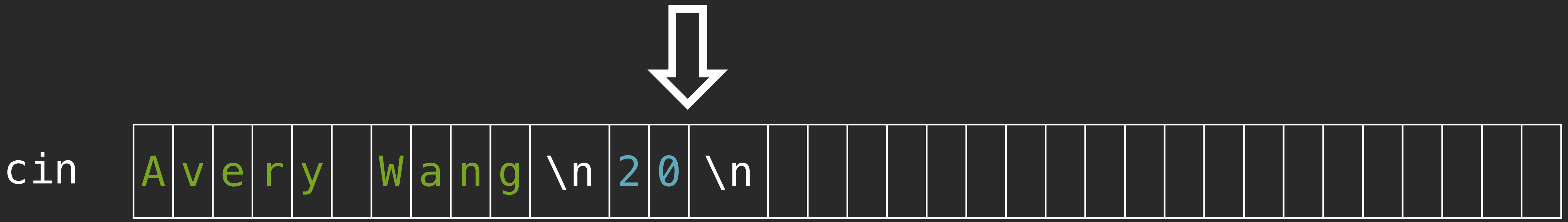
```
getline(cin, name, ‘\n’);
cin >> age;
cout << name << age;
cin >> response;
```

name (string): “Avery Wang”

response (string): ???

age (int): ???

Everything copied over.



cin

A	v	e	r	y		W	a	n	g	\n		2	0	\n																					

G F E B

```
Avery Wang
20
```

```
getline(cin, name, '\n');
cin >> age;
cout << name << age;
cin >> response;
```

name
(string)

"Avery Wang"

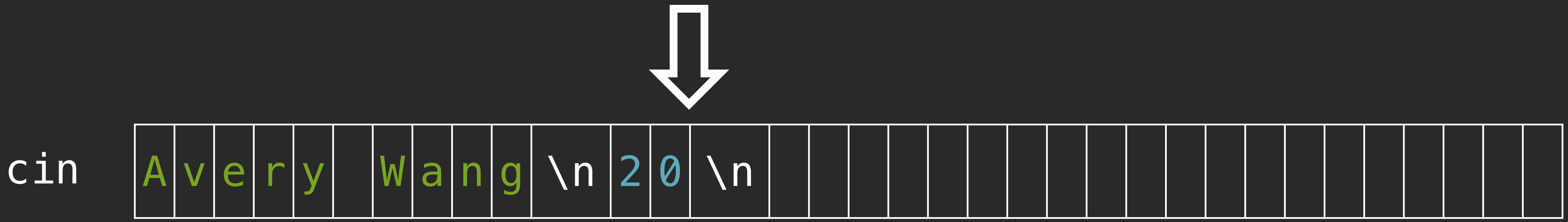
response
(string)

???

age
(int)

20

Everything copied over.



G F E B

```
Avery Wang
20
```

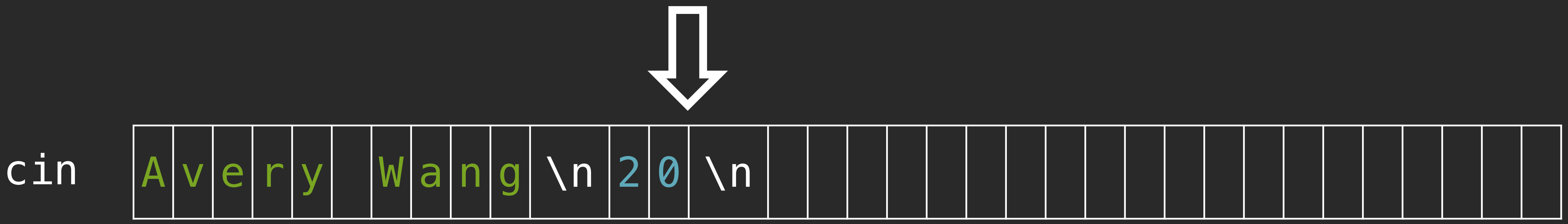
```
getline(cin, name, ‘\n’);
cin >> age;
cout << name << age;
cin >> response;
```

name (string): “Avery Wang”

response (string): ???

age (int): 20

Everything fine so far
(don’t forget cout is buffered!)



cin	A	v	e	r	y		W	a	n	g	\n	2	0	\n																					

G F E B

```
Avery Wang
20
Avery Wang20
```

```
getline(cin, name, ‘\n’);
cin >> age;
cout << name << age;
cin >> response;
```

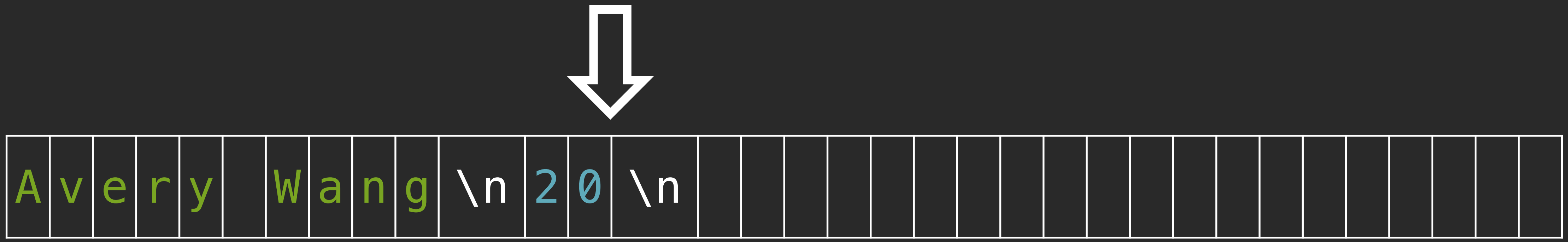
name (string) “Avery Wang”

response (string) ???

age (int) 20

Now cout is flushed when we attempt to read from cin.

cin



G F E B

```
Avery Wang
20
Avery Wang20
```

```
getline(cin, name, ‘\n’);
cin >> age;
cout << name << age;
cin >> response;
```

name
(string)

“Avery Wang”

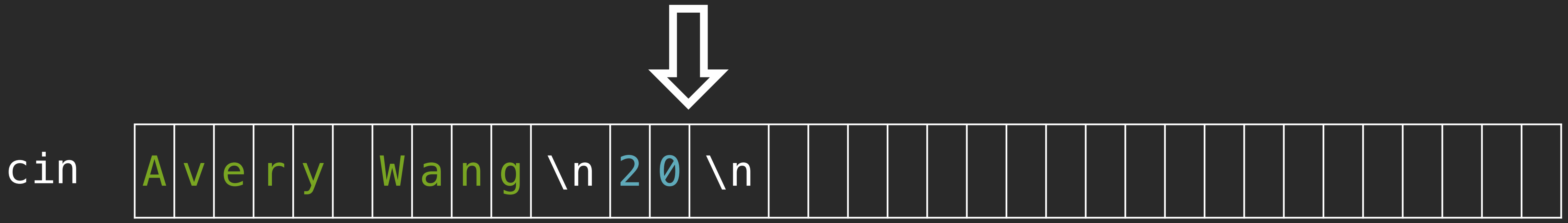
response
(string)

???

age
(int)

20

We should also switch this to getline in case the user enters multiple words.



cin	A	v	e	r	y		W	a	n	g	\n	2	0	\n																						

G F E B

```
Avery Wang
20
Avery Wang20
```

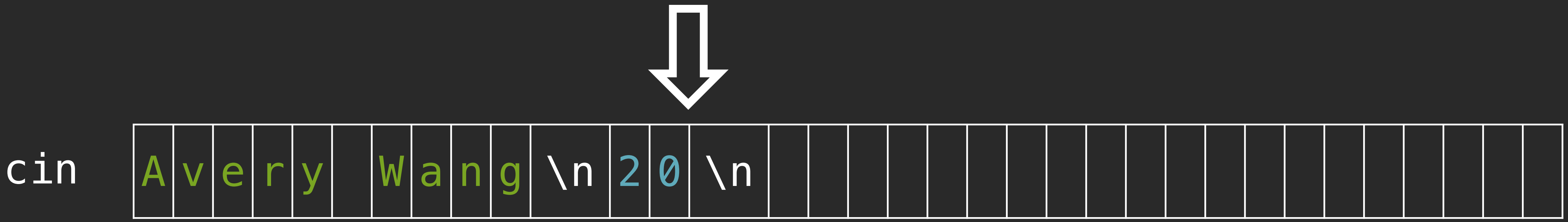
```
getline(cin, name, '\n');
cin >> age;
cout << name << age;
getline(cin, response, '\n');
```

name (string): "Avery Wang"

response (string): ???

age (int): 20

We should also switch this to getline in case the user enters multiple words.



cin

A	v	e	r	y		W	a	n	g	\n	2	0	\n																				

G F E B

```
Avery Wang
20
Avery Wang20
```

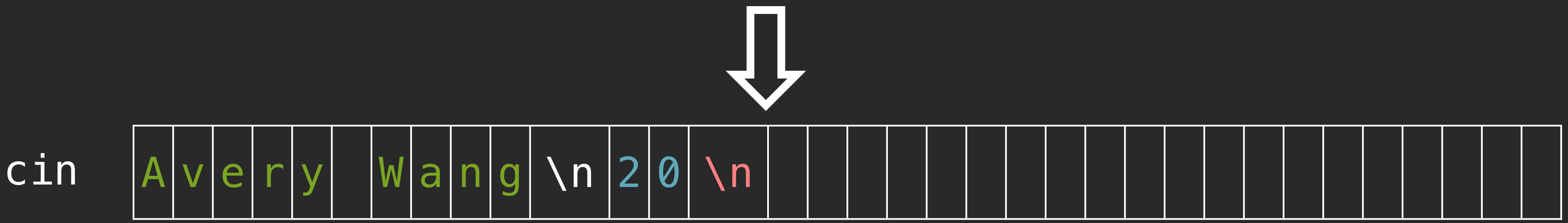
```
getline(cin, name, ‘\n’);
cin >> age;
cout << name << age;
getline(cin, response, ‘\n’);
```

name (string): “Avery Wang”

response (string): ???

age (int): 20

But notice what happens.



cin	A	v	e	r	y		W	a	n	g	\n	2	0	\n																				

G F E B

```
Avery Wang
20
Avery Wang20
```

```
getline(cin, name, '\n');
cin >> age;
cout << name << age;
getline(cin, response, '\n');
```

name (string) “Avery Wang”

response (string) “\n”

age (int) 20

getline does not skip a leading newline character!

cin	A	v	e	r	y		W	a	n	g	\n	2	0	\n																				

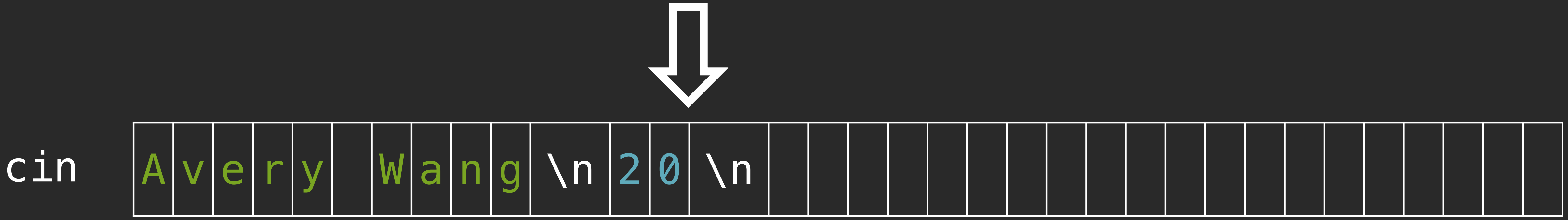
G F E B

```
Avery Wang
20
Avery Wang20
```

```
getline(cin, name, '\n');
cin >> age;
cout << name << age;
getline(cin, response, '\n');
```

name (string)	"Avery Wang"
response (string)	"\n"
age (int)	20

In 106B you are told not to mix >> with getline.
This is the reason.



cin															
	A	v	e	r	y		W	a	n	g	\n	2	0	\n	

G F E B

```
Avery Wang
20
Avery Wang20
```

```
getline(cin, name, '\n');
cin >> age;
cout << name << age;
cin.ignore();
getline(cin, response, '\n');
```

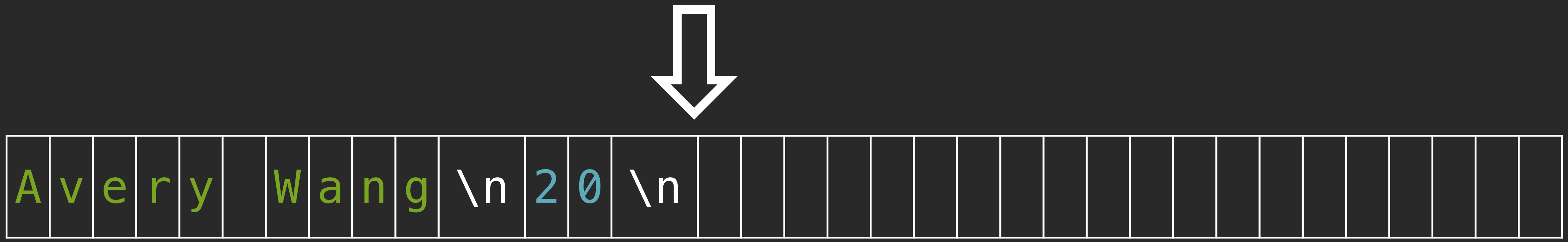
name (string): "Avery Wang"

response (string): ???

age (int): 20

Here is the fix.

cin



```
A v e r y   W a n g \n 2 0 \n
```

G F E B

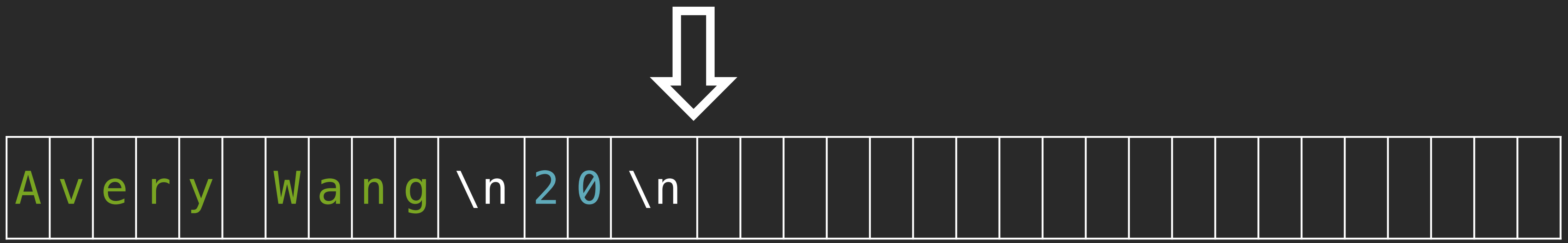
```
Avery Wang
20
Avery Wang20
```

```
getline(cin, name, ‘\n’);
cin >> age;
cout << name << age;
cin.ignore();
getline(cin, response, ‘\n’);
```

name (string)	“Avery Wang”
response (string)	???
age (int)	20

cin ignores one character in the buffer.

cin



A	v	e	r	y		W	a	n	g	\n	2	0	\n																						

G F E B

```
Avery Wang
20
Avery Wang20
|
```

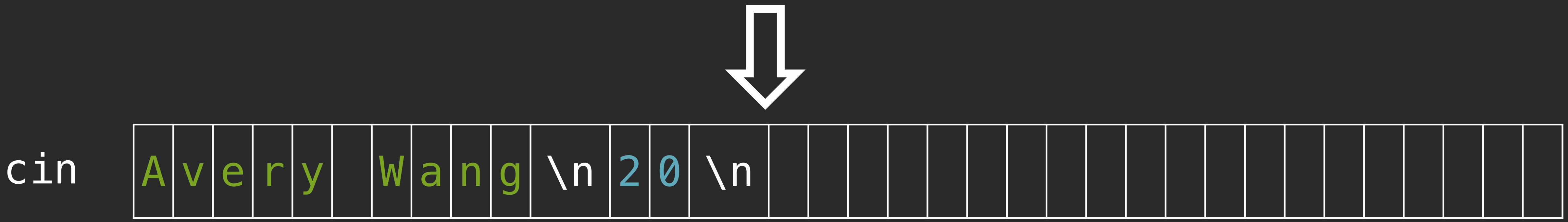
```
getline(cin, name, '\n');
cin >> age;
cout << name << age;
cin.ignore();
getline(cin, response, '\n');
```

name (string): "Avery Wang"

response (string): ???

age (int): 20

The user is now prompted for input.



cin

A	v	e	r	y		W	a	n	g	\n	2	0	\n																						

G F E B

```
Avery Wang
20
Avery Wang20
|
```

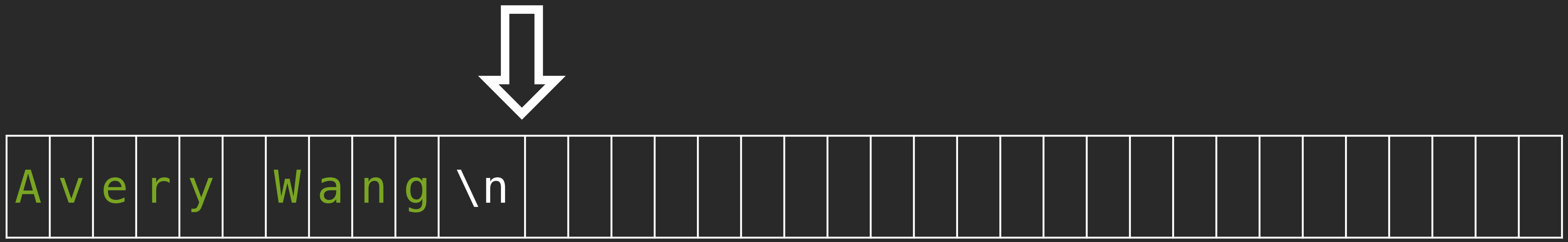
```
getline(cin, name);
cin >> age;
cout << name << age;
cin.ignore();
getline(cin, response);
```

name
(string)
"Avery Wang"

response
(string)
???

age
(int)
20

Also, note that the '\n' is optional, though you can use other delimiters.



cin

A	v	e	r	y		W	a	n	g	\n																								

G F E B

```
Avery Wang
|
```

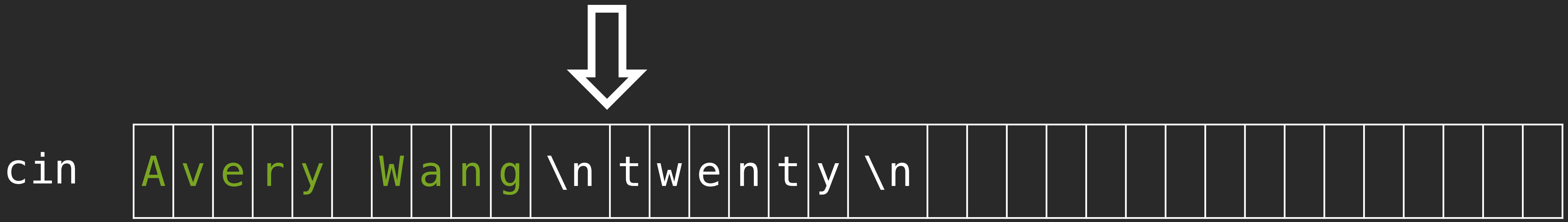
```
getline(cin, name);
cin >> age;
cout << name << age;
cin.ignore();
getline(cin, response);
```

name (string): "Avery Wang"

response (string): ???

age (int): ???

Let's back up one sec.
This line is also a problem.



cin

A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																		

G F E B

```
Avery Wang
twenty
```

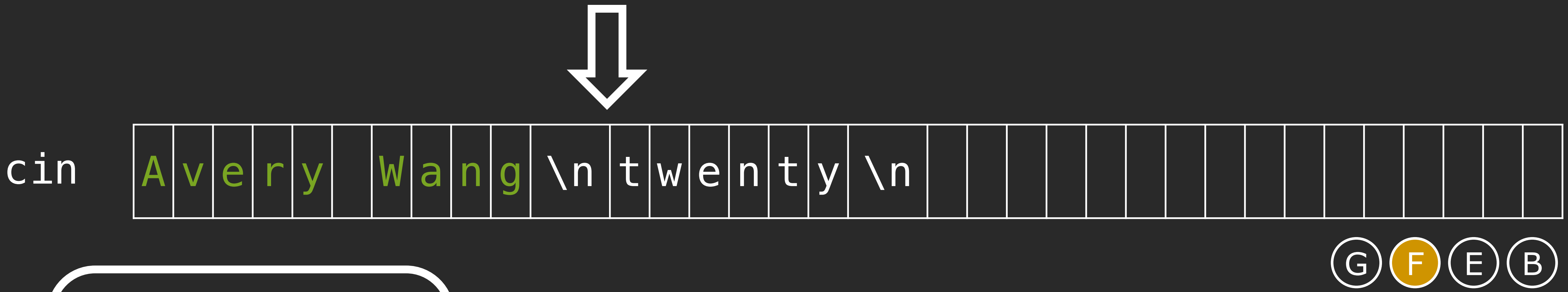
```
getline(cin, name);
cin >> age;
cout << name << age;
cin.ignore();
getline(cin, response);
```

name (string): "Avery Wang"

age (int): ???

response (string): ???

This code does not do any error checking at all.



cin	A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																

```
Avery Wang
twenty
```

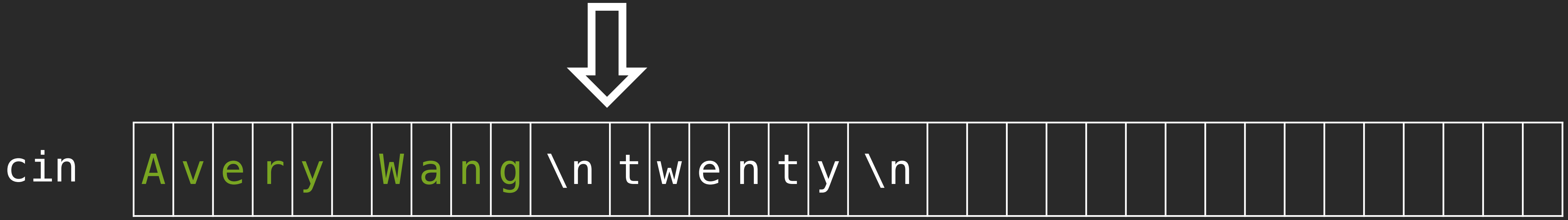
```
getline(cin, name);
cin >> age;
cout << name << age;
cin.ignore();
getline(cin, response);
```

name (string): “Avery Wang”

response (string): ???

age (int): ???

As before, fail bit is on, everything else fails too.



cin

A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																

G F E B

```
Avery Wang
twenty
```

```
getline(cin, name);
age = getInteger();
cout << name << age;
cin.ignore();
getline(cin, response);
```

name
(string)

"Avery Wang"

response
(string)

???

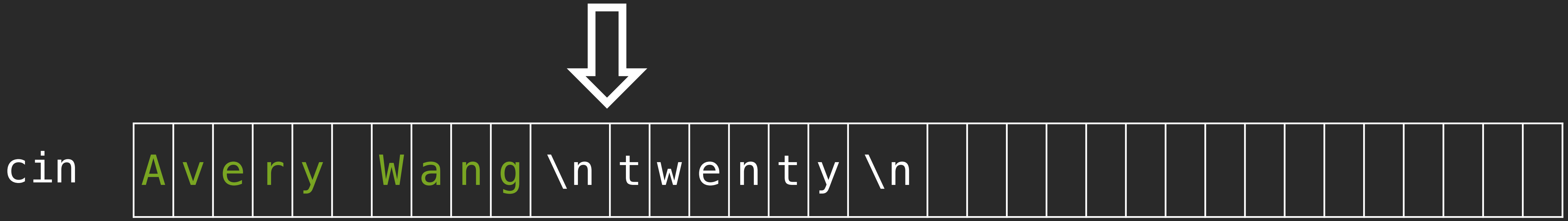
age
(int)

???

If only we could do this...

Example

implementing getInteger



cin	A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																	

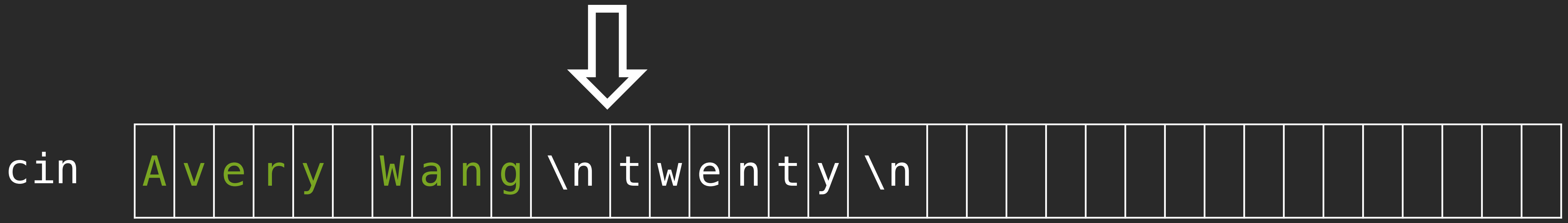
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
  }
}
```

result
(int)

???

The lecture code also had prompting and reprompting.



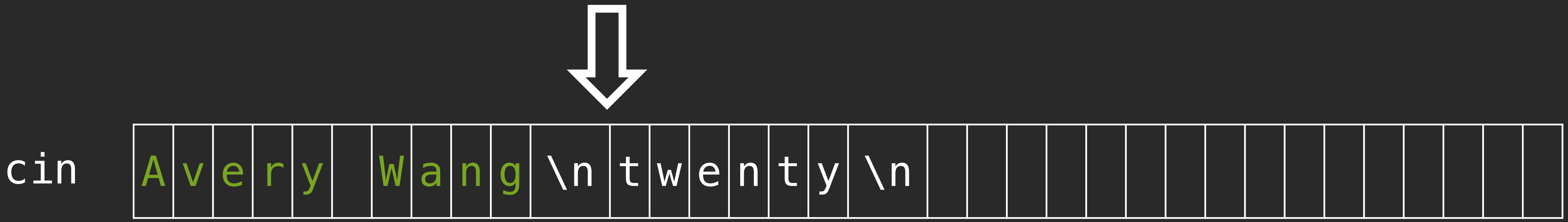
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result
(int)

???

Keep looping until the user gets it right.



cin	A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																

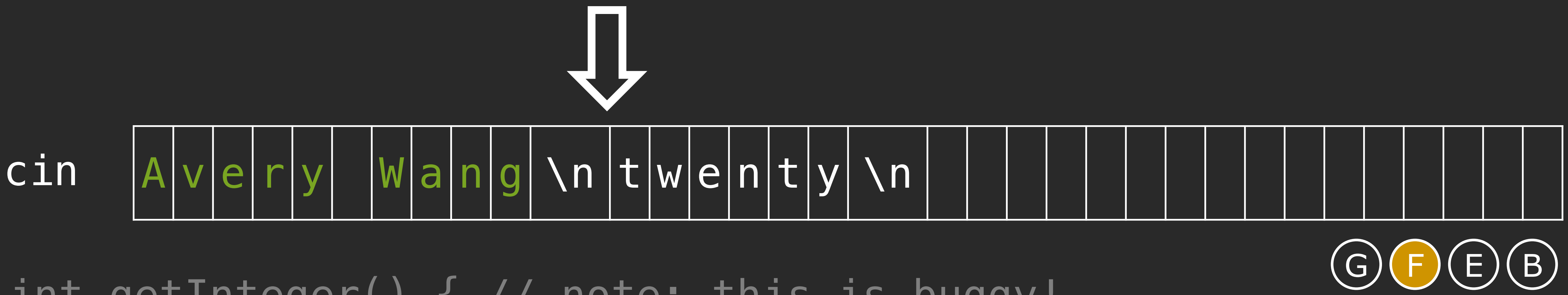
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result
(int)

???

Try reading in an int. It fails.



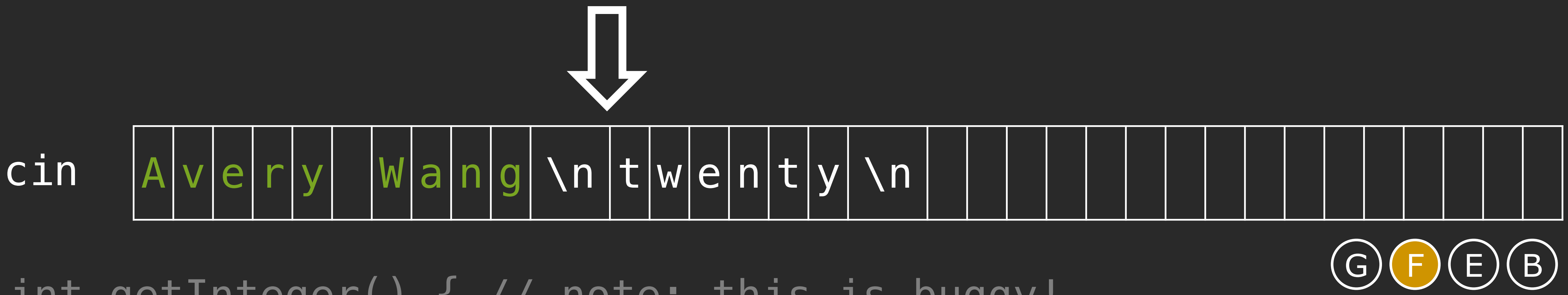
cin	A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																	

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
  }
}
```

result
(int)

???

Fail bit on.

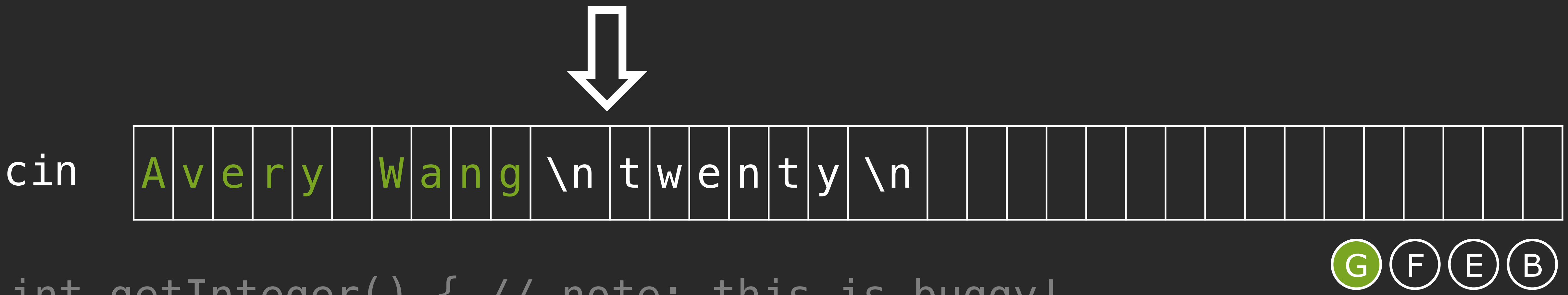


cin	A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int) ???

This clears the state, restoring the state to good.



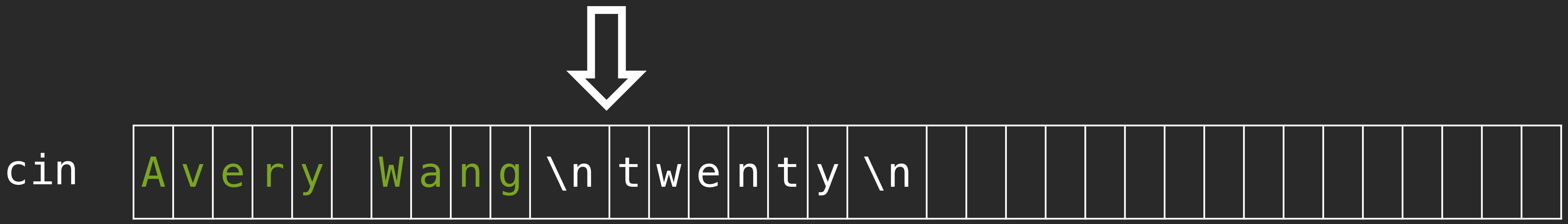
```
cin   A v e r y   W a n g \n t w e n t y \n
```

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result
(int)

???

This clears the state, restoring the state to good.



cin	A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																	

G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
  }
}
```

result
(int)

???

Keep moving position until you reach a ‘\n’ or you run out of buffer room.

cin

																		⇩															
A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																

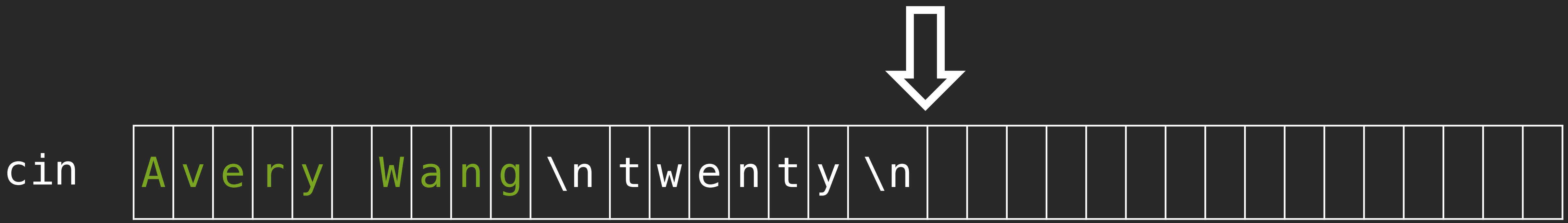
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
  }
}
```

result
(int)

???

Keep moving position until you reach a '\n' or you run out of buffer room.



cin

A	v	e	r	y		W	a	n	g	\n	t	w	e	n	t	y	\n																

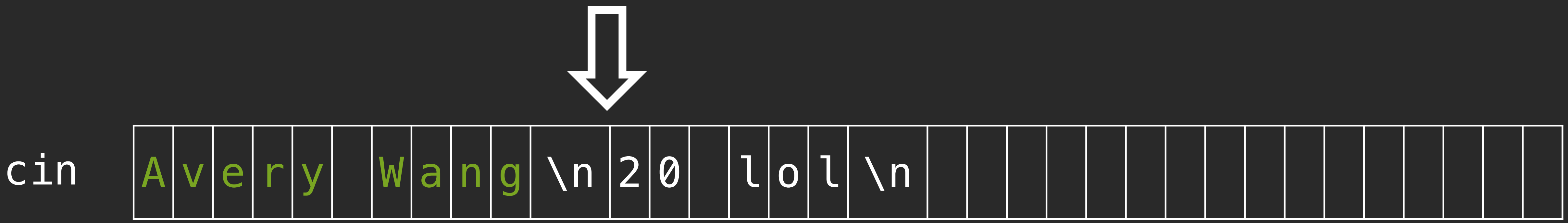
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result
(int)

???

And then try again.



cin

A	v	e	r	y		W	a	n	g	\n	2	0		l	o	l	\n																

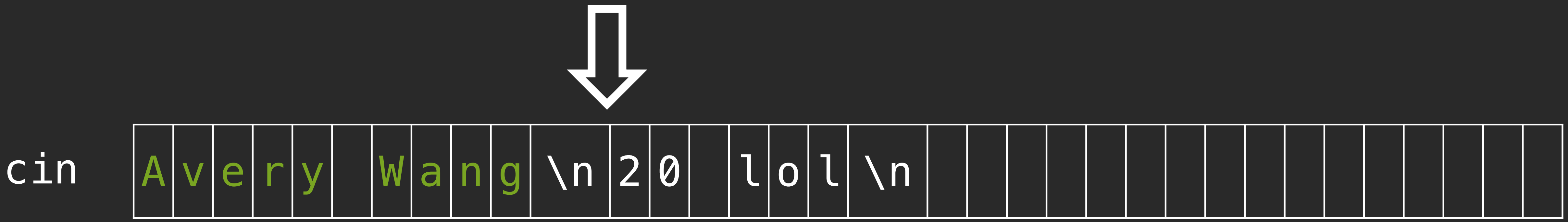
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
  }
}
```

result
(int)

???

This code is wrong.
Try this new input.



cin

A	v	e	r	y		W	a	n	g	\n	2	0		l	o	l	\n																

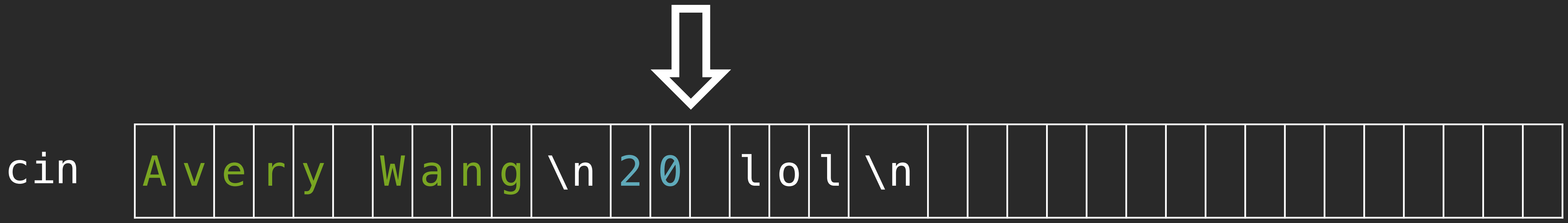
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result
(int)

???

This code is wrong.
Try this new input.



cin	A	v	e	r	y		W	a	n	g	\n	2	0		l	o	l	\n																	

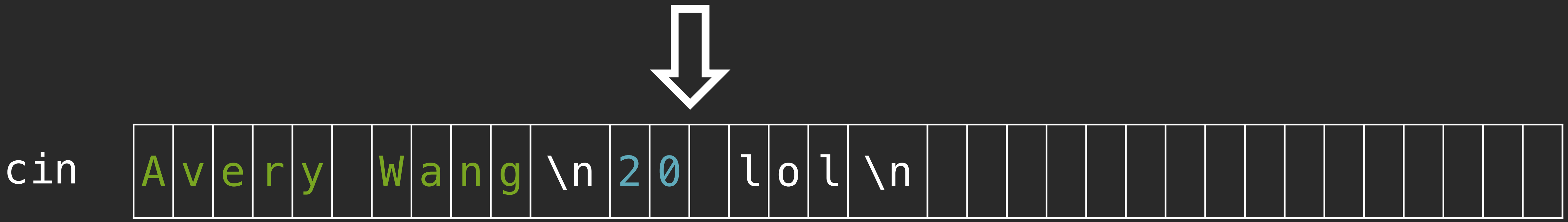
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result
(int)

20

This code is wrong.
Try this new input.



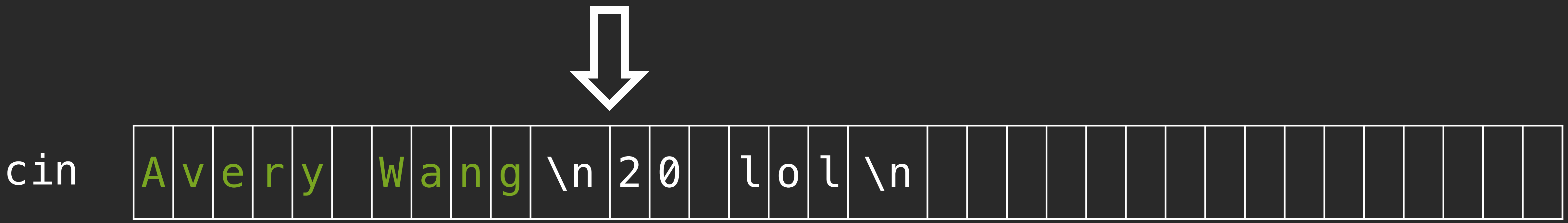
cin	A	v	e	r	y		W	a	n	g	\n	2	0		l	o	l	\n																	

G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result;
    if (cin >> result) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int)	20

This returns when it should not have!



cin	A	v	e	r	y		W	a	n	g	\n	2	0		l	o	l	\n

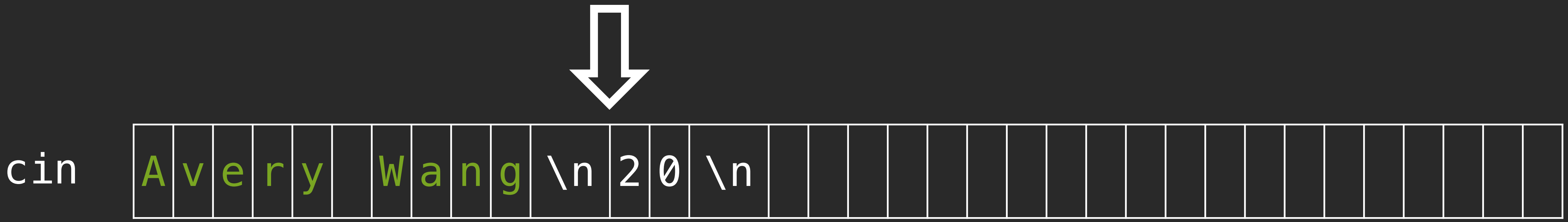
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result; char trash;
    if (cin >> result && !(cin >> trash)) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result
(int)

???

Another attempt:
if there are more characters,
then this is not a valid int.



cin	A	v	e	r	y		W	a	n	g	\n	2	0	\n

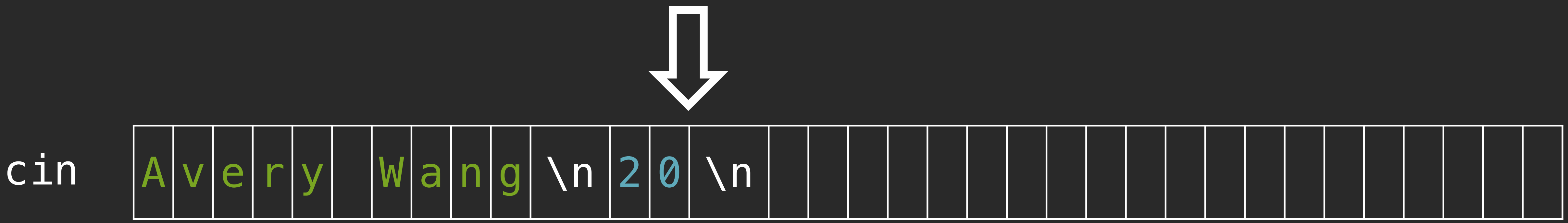
G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result; char trash;
    if (cin >> result && !(cin >> trash)) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result
(int)

???

But this fails valid input then.



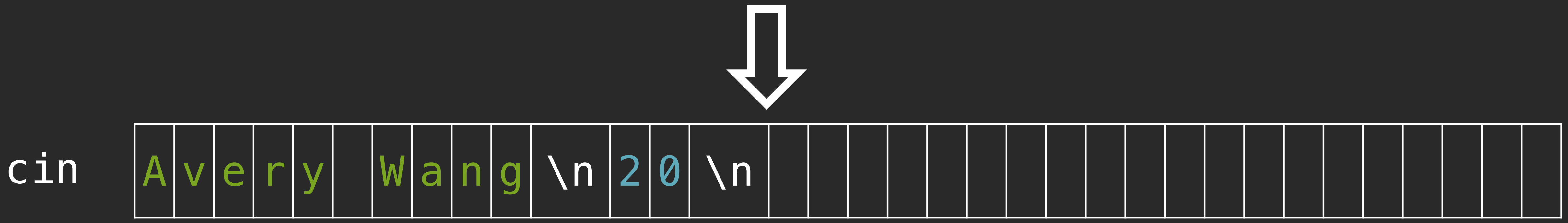
```
cin  A v e r y   W a n g \n 2 0 \n
```

G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result; char trash;
    if (cin >> result && !(cin >> trash)) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int)	20

But this fails valid input then.



cin	A	v	e	r	y		W	a	n	g	\n	2	0	\n	

G F E B

```
int getInteger() { // note: this is buggy!
  while (true) {
    int result; char trash;
    if (cin >> result && !(cin >> trash)) return result;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result
(int)

20

Now the program hangs since there is no more input.

Key Takeaway

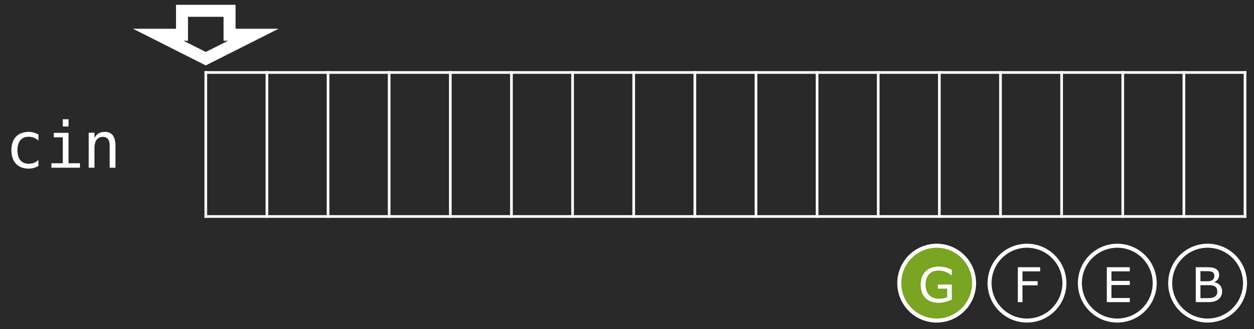
cin hangs while it waits for user input.

That makes any attempt to parse the buffer for bad input very hard.

Key Takeaway

Copy everything to a stream that is not connected to an external source…

stringstream!

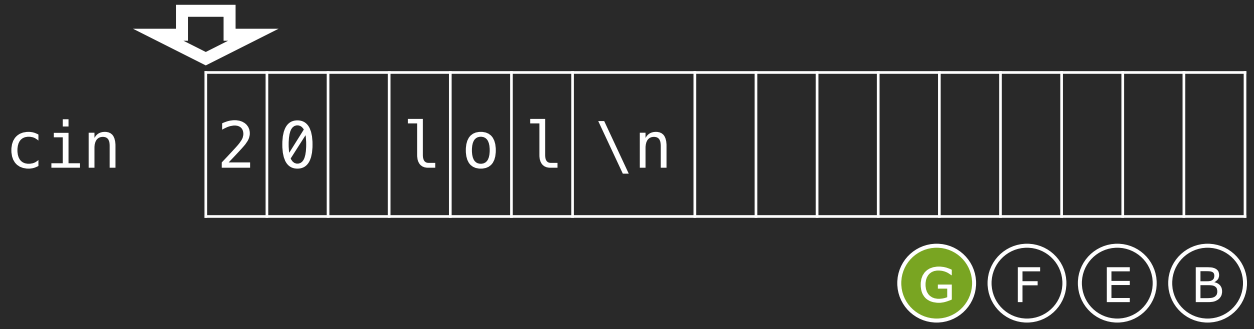


```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int) ???

trash (char) ???

This code is wrong.
Try this new input.



cin

2	0		l	o	l	\n									

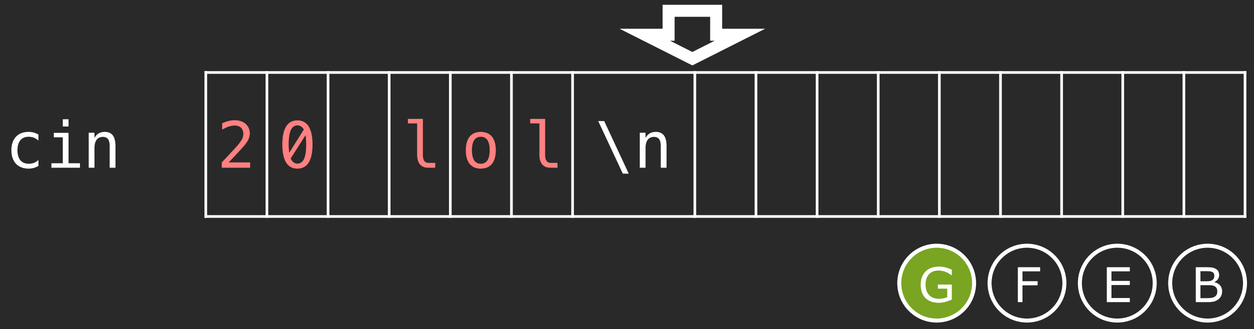
G F E B

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

line
(string)

???

Try reading a line.
Since the buffer is empty, it prompts the user.

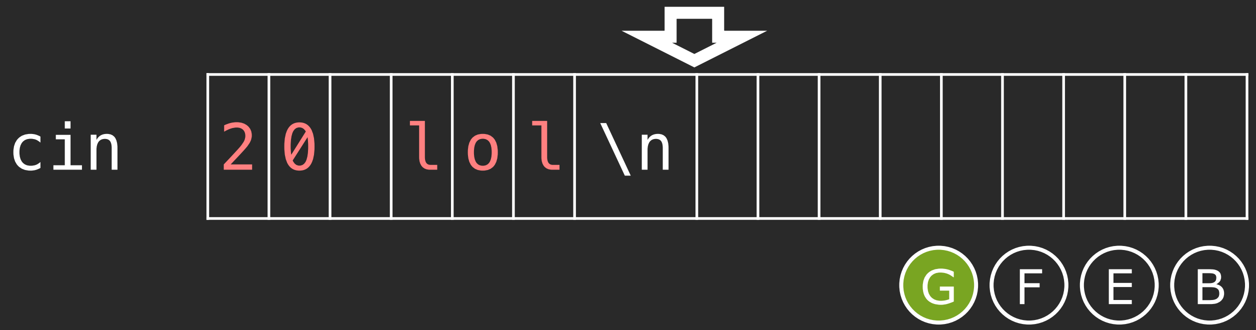


```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

line
(string)

“20 lol”

Try reading a line.
Since the buffer is empty, it prompts the user.

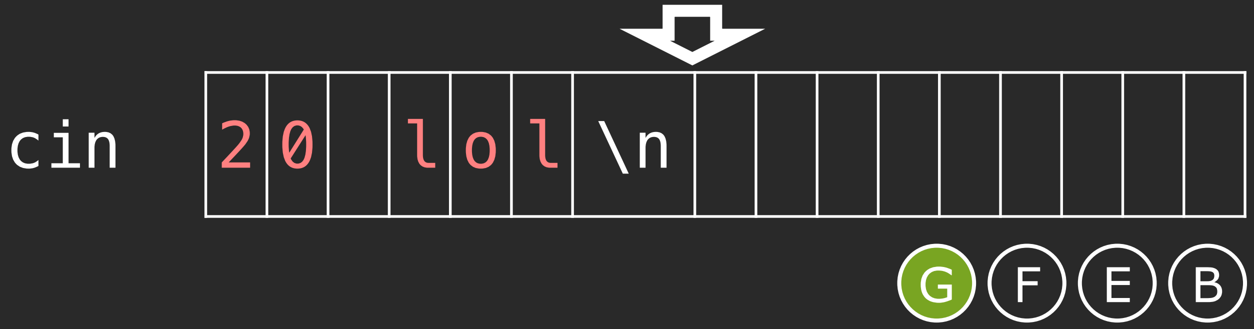


```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

line
(string)

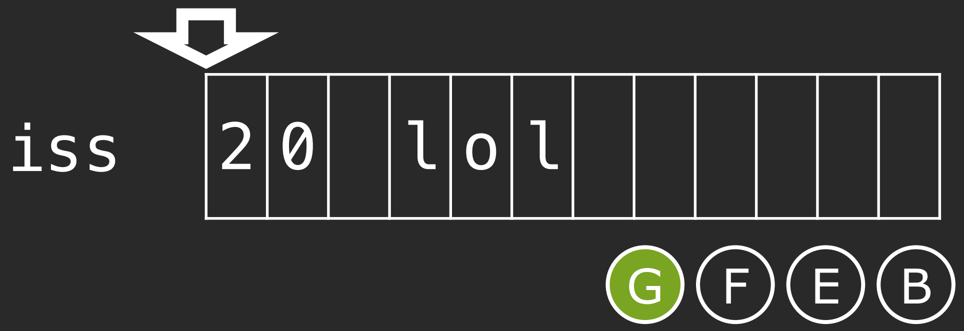
“20 lol”

Create an istringstream using that line.


cin

2 0   l o l \n

G F E B


iss

2 0   l o l

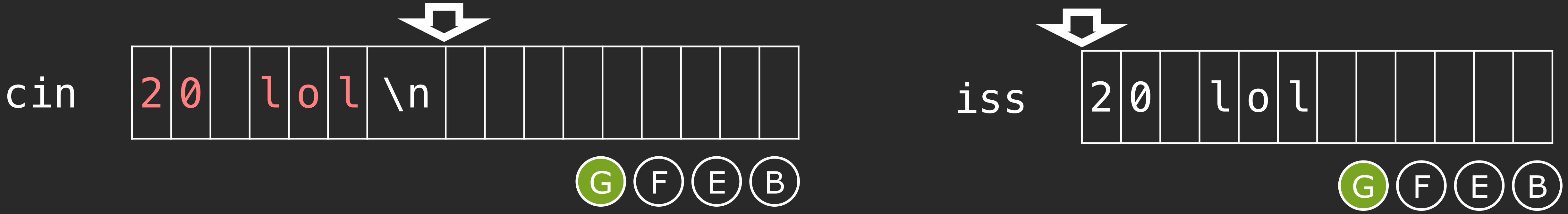
G F E B

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

line
(string)

“20 lol”

Create an istringstream using that line.

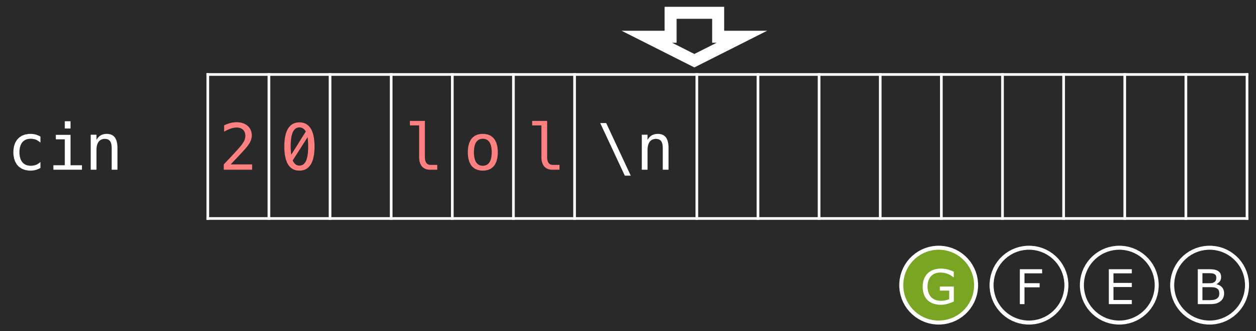


```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int) ???

trash (char) ???

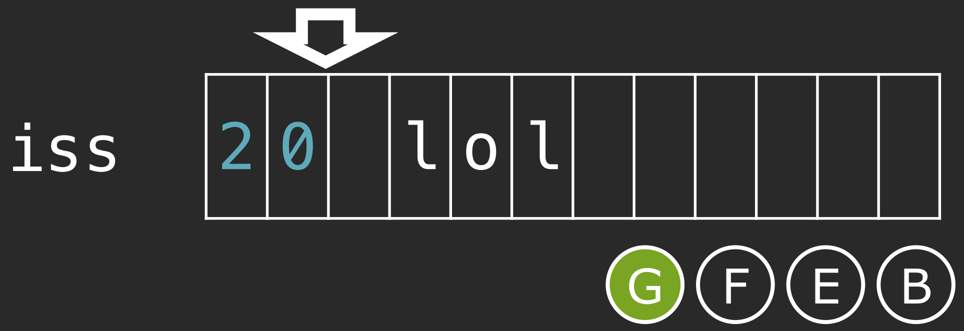
Try reading an int.



cin

2 0   l o l \n

G F E B



iss

2 0   l o l

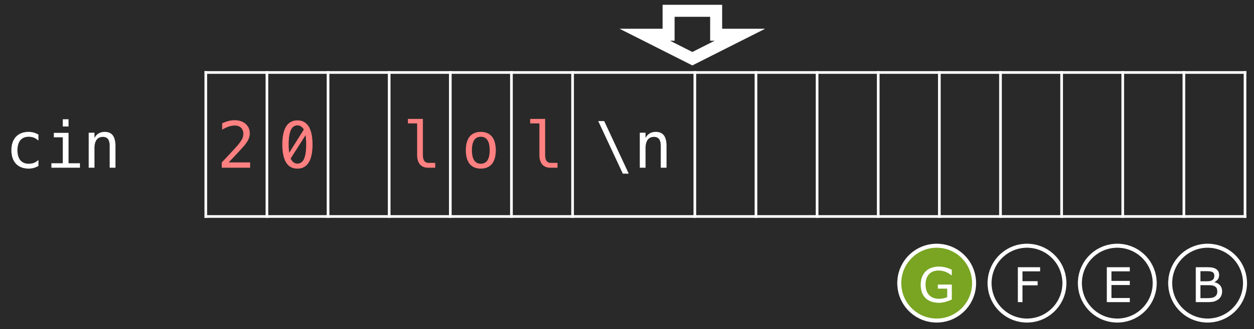
G F E B

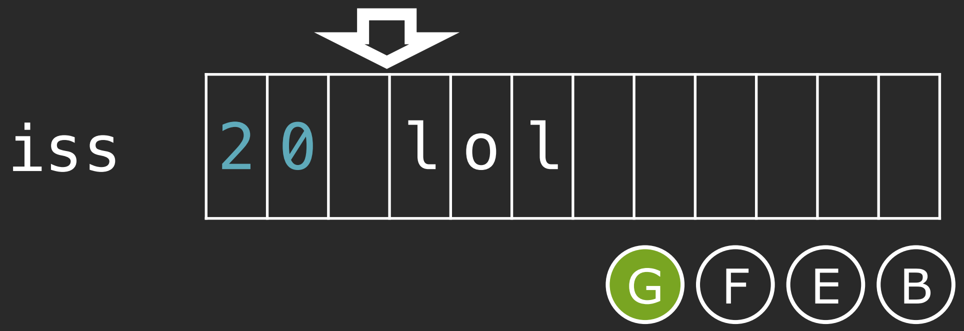
```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int): 20

trash (char): ???

Try reading an int.

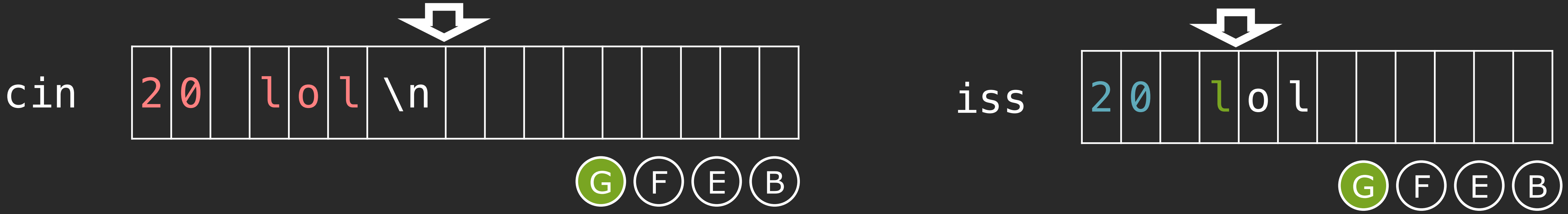




```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int)	20	trash (char)	???

Now try reading in any trash.

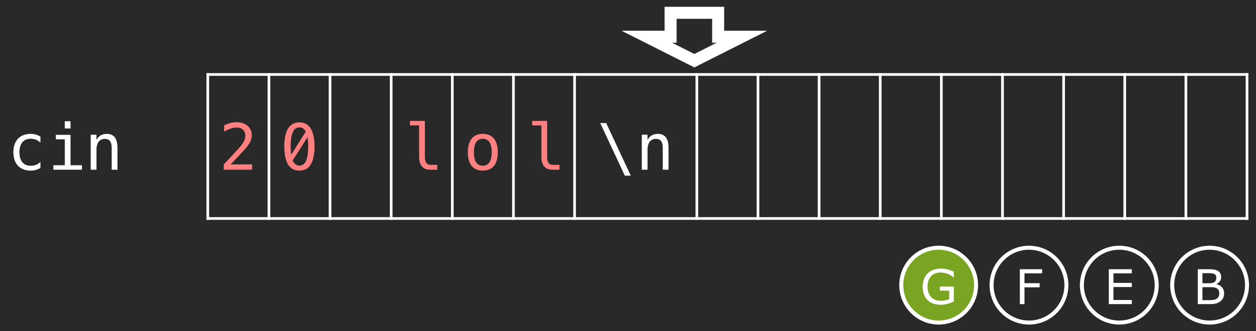


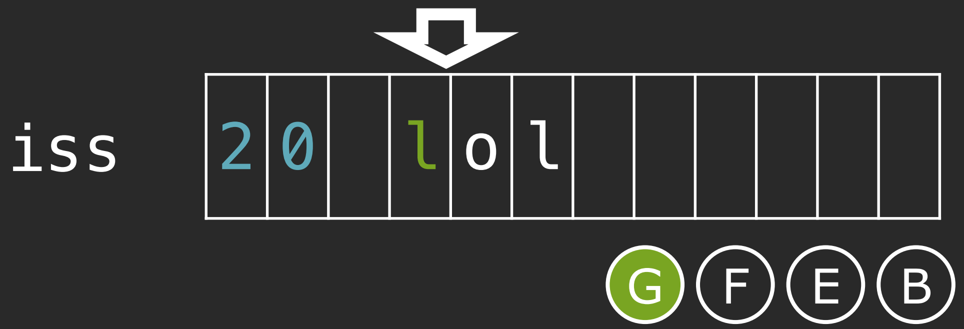
```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int): 20

trash (char): ‘l’

Now try reading in any trash.

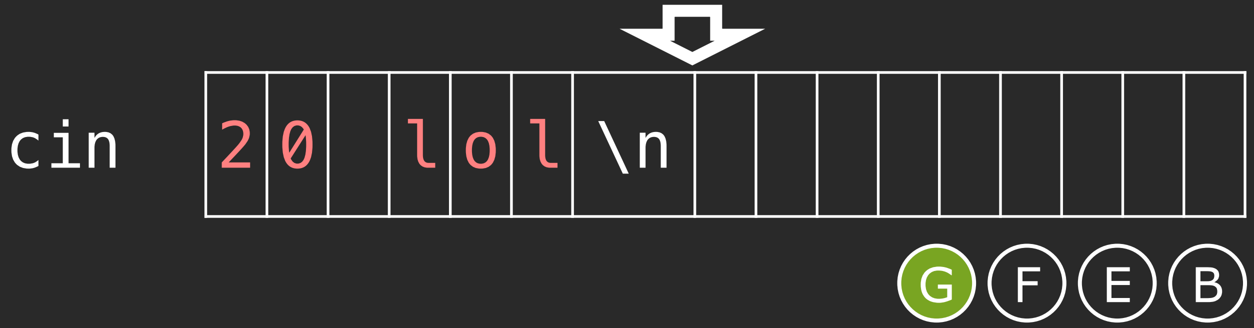


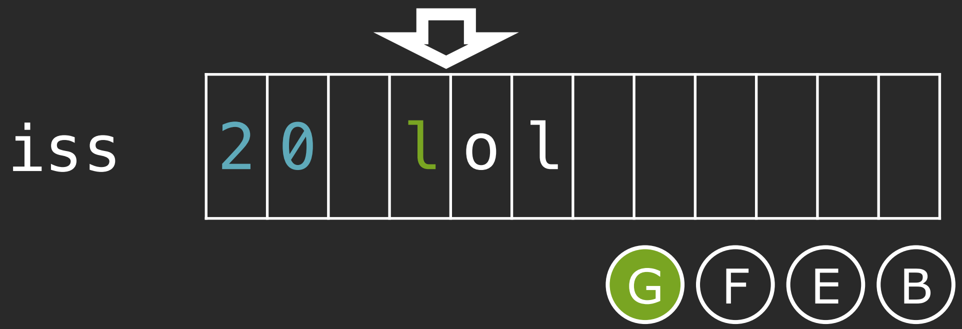


```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int)	trash (char)
20	‘l’

That succeeded, which is not what we wanted.



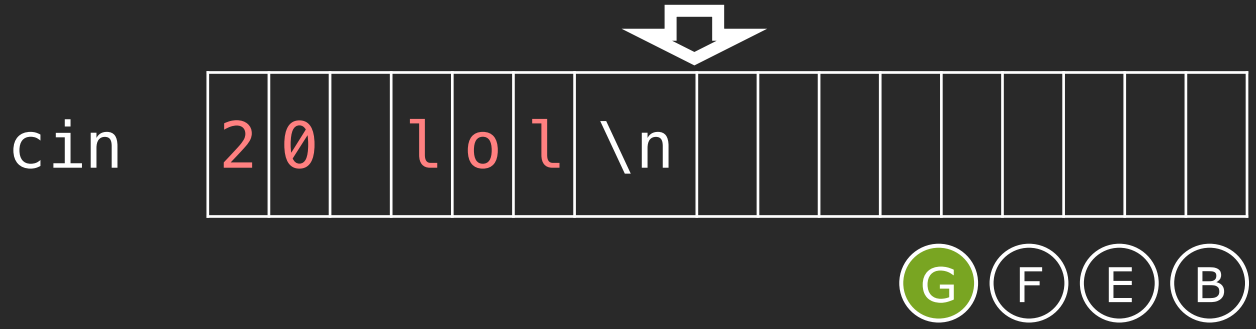


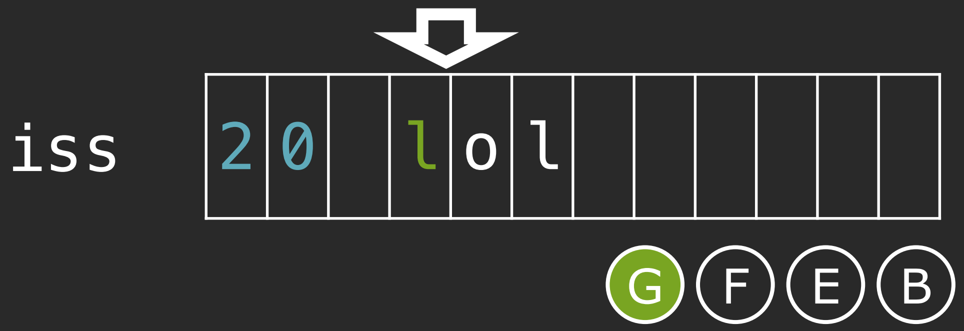
```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int): 20

trash (char): ‘l’

Forget everything,
and try again.



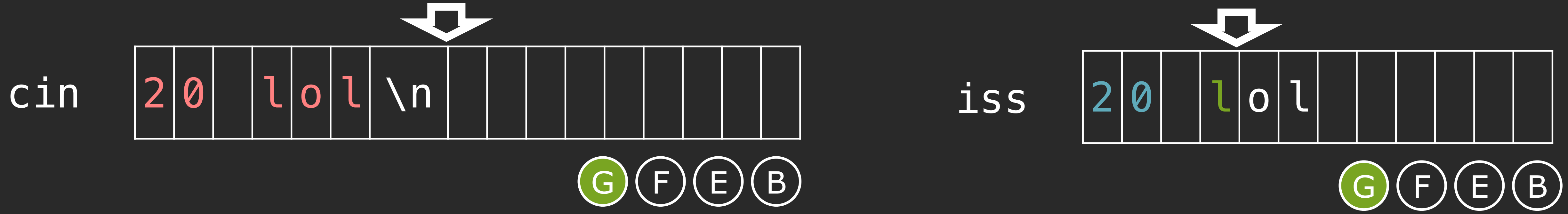


```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
    iss.clear();
    iss.ignore(numeric_limits<streamsize>::max(), '\n’);
  }
}
```

result (int): 20

trash (char): ‘l’

Actually, since we create a new stringstream each time, we can remove these lines.

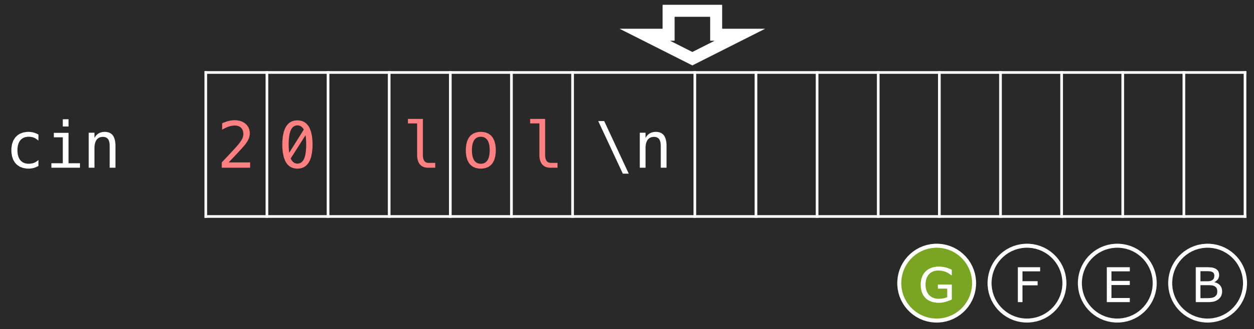


```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

result (int): 20

trash (char): ‘l’

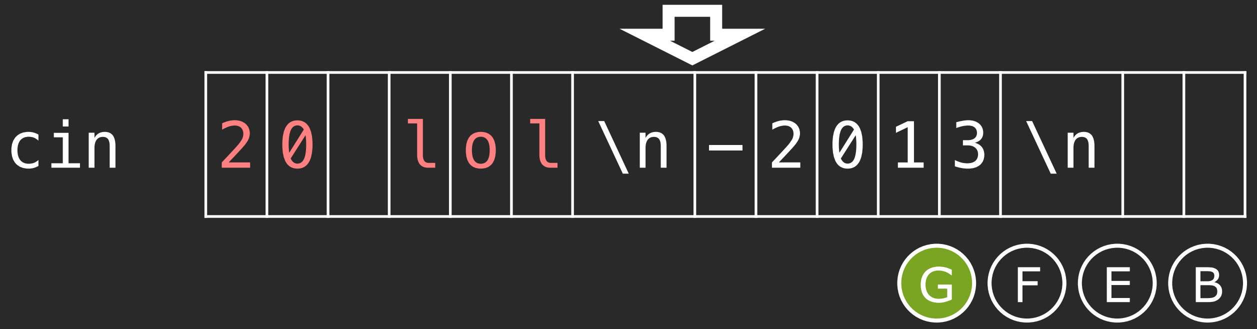
Actually, since we create a new stringstream each time, we can remove these lines.



```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

line
(string)
???

Try reading another int. This waits for the user to enter something.



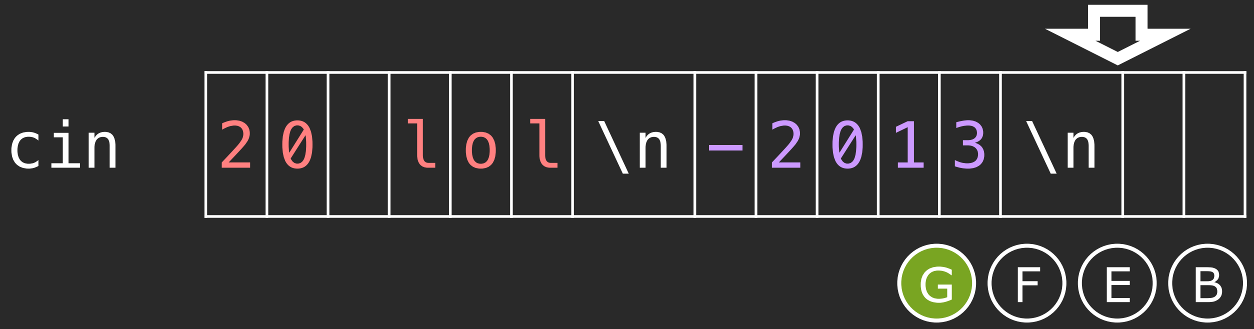
cin	2	0		l	o	l	\n	-	2	0	1	3	\n		

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

line
(string)

???

Try reading another int. This waits for the user to enter something.

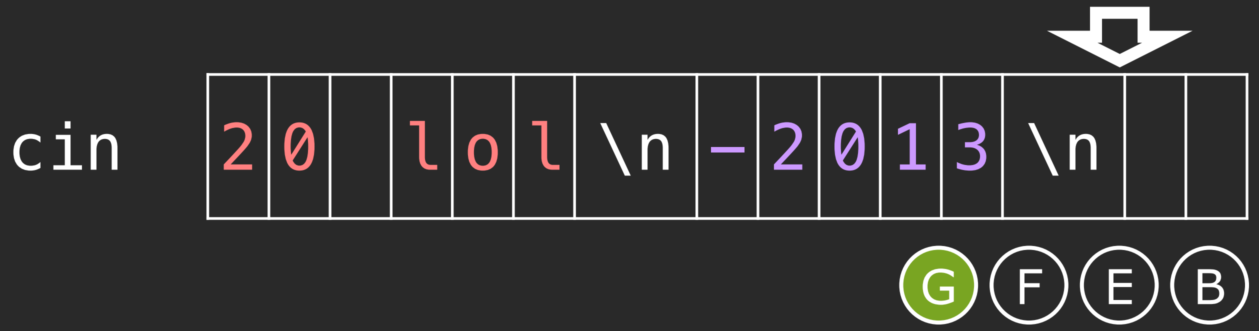


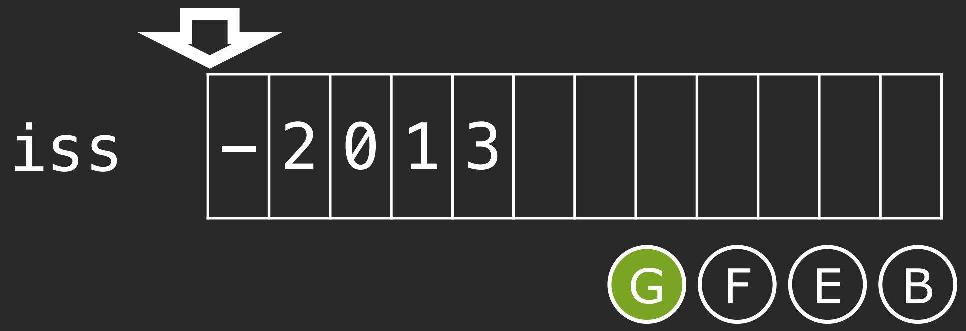
cin	2	0		l	o	l	\n	-	2	0	1	3	\n		

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

line (string)	"-2013"

Try reading another int. This waits for the user to enter something.



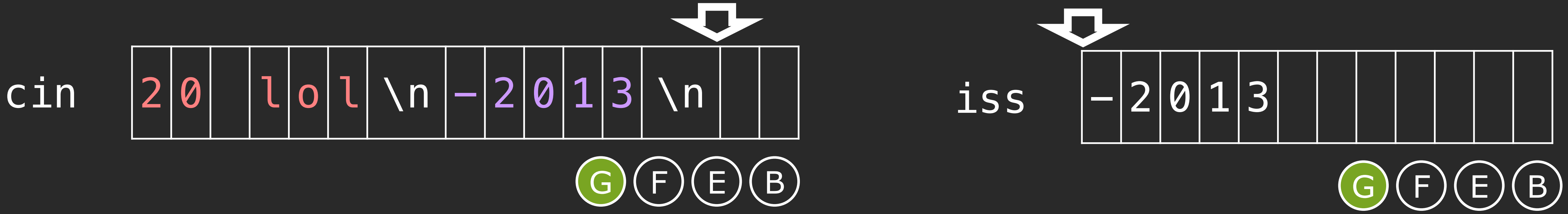


```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

line
(string)

"-2013"

Create a separate stringstream with the line we just read.



cin	2	0		l	o	l	\n	-	2	0	1	3	\n		

G F E B

iss	-	2	0	1	3							

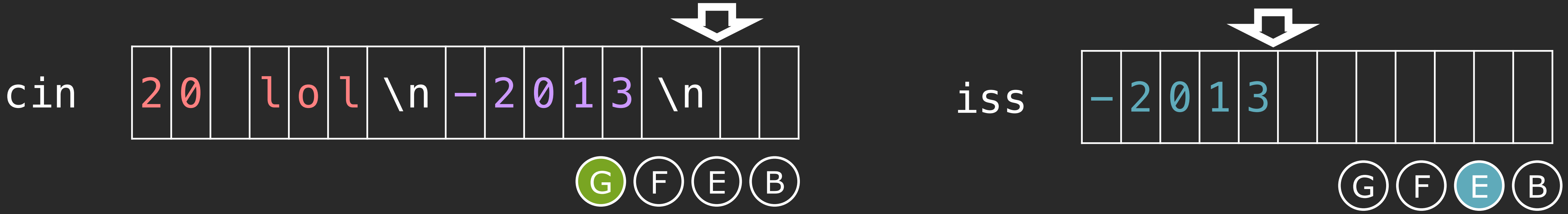
G F E B

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

result (int): ???

trash (char): ???

Try reading an int from the stringstream.



cin

2	0		l	o	l	\n	-	2	0	1	3	\n		

G F E B

iss

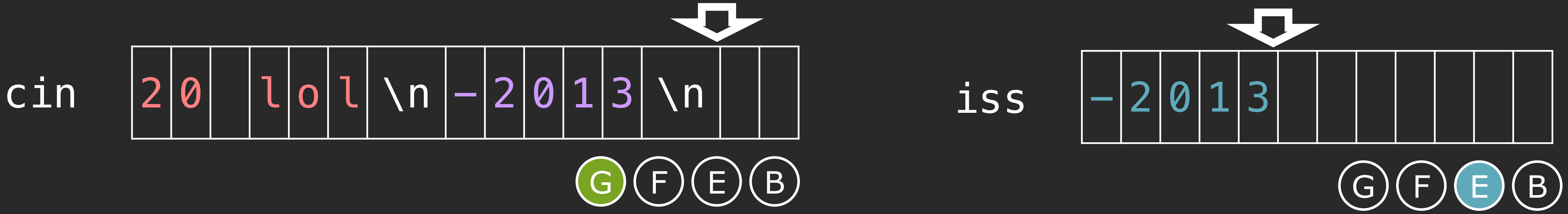
-	2	0	1	3							

G F E B

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

result (int)	-2013	trash (char)	???

Try reading an int from the stringstream.



cin

2	0		l	o	l	\n	-	2	0	1	3	\n		

G F E B

iss

-	2	0	1	3							

G F E B

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

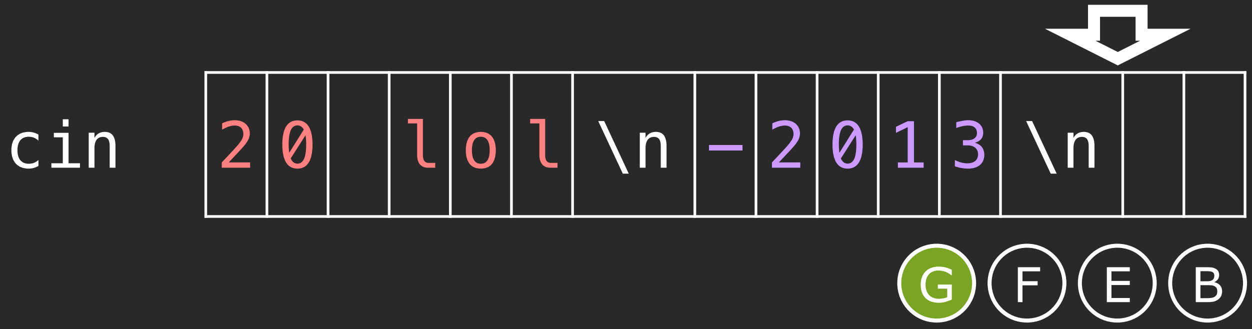
result
(int)

-2013

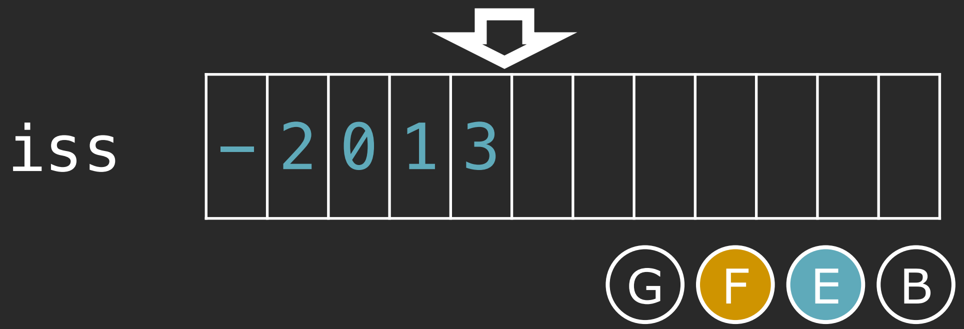
trash
(char)

???

Try reading any remaining characters from the buffer.



cin	2	0		l	o	l	\n	-	2	0	1	3	\n	

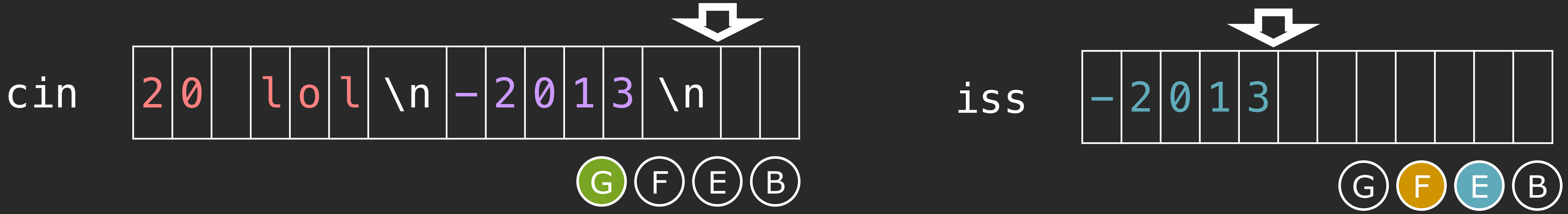


iss	-	2	0	1	3							

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

result (int)	-2013	trash (char)	???

There are no more characters, so that fails.



```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

result (int): -2013

trash (char): ???

Both conditions are true, so we return the result.

One of the toughest functions to implement in the Stanford library.

```
int getInteger(const string& prompt = “[shortened]",
               const string& reprompt = “[shortened]") {
  while (true) {
    cout << prompt;
    string line; int result; char extra;
    if (!getline(cin, line))
       throw domain_error(“[shortened]”);
    istringstream iss(line);
    if (cin >> result && !(cin >> extra)) return result;
    cout << reprompt << endl;
  }
}
```

Types

Game Plan

- type deduction
- structures
- initialization

This is a ~20 minute crash course on modern C++ types.

type deduction

The STL sometimes uses very long types.

```
std::unordered_map<forward_list<Student>,
     unordered_set>::iterator begin = studentMap.cbegin();

std::unordered_map<forward_list<Student>,
     unordered_set>::iterator end = studentMap.cend();
```

We can fix that using a type alias.

```
using map_iterator = std::unordered_map<forward_list<Student>,
                    unordered_set>::iterator;

std::unordered_map<forward_list<Student>,
     unordered_set>::iterator begin = studentMap.cbegin();

std::unordered_map<forward_list<Student>,
     unordered_set>::iterator end = studentMap.cend();
```

We can fix that using a type alias.

```
using map_iterator = std::unordered_map<forward_list<Student>,
                                  unordered_set>::iterator;

map_iterator begin = studentMap.cbegin();


map_iterator end = studentMap.cend();
```

This is still a bit error prone, since you have to figure out the correct type.

```
using map_iterator = std::unordered_map<forward_list<Student>,
                    unordered_set>::const_iterator;

map_iterator begin = studentMap.cbegin();


map_iterator end = studentMap.cend();
```

We made a typo! The types are actually const iterators.

This is still a bit error prone, since you have to figure out the correct type.

```
using map_iterator = std::unordered_map<forward_list<Student>,
                                unordered_set>::const_iterator;

auto begin = studentMap.cbegin();


auto end = studentMap.cend();
```

An idea: let the compiler figure out the type for us.

C++11 supports automatic type inference: have the compiler figure out the type for you.

```
auto begin = studentMap.cbegin();

auto end = studentMap.cend();
```

auto can be used in almost all places!

```
auto calculateSum(const vector<string>& v) {
  auto copy = v;
  auto multiplier = 2.4;
  auto name = "Avery";
  auto betterName = string{"Avery“};
  auto& refMult = multiplier;
  auto func = [](auto i) {return i*2};

  return betterName;
}
```

This code is wrong, since the types are const iterators.

auto can be used in almost all places!

```
auto calculateSum(const vector<string>& v) {
   auto copy = v;
   auto multiplier = 2.4;
   auto name = "Avery";
   auto betterName = string{"Avery“};
   auto& refMult = multiplier;
   auto func = [](auto i) {return i*2};

   return betterName;
}
```

Local variables!
copy is of type vector<string>.

auto can be used in almost all places!

```
auto calculateSum(const vector<string>& v) {
  auto copy = v;
  auto multiplier = 2.4;
  auto name = "Avery";
  auto betterName = string{"Avery“};
  auto& refMult = multiplier;
  auto func = [](auto i) {return i*2};

  return betterName;
}
```

multiplier is a double.

auto can be used in almost all places!

```
auto calculateSum(const vector<string>& v) {
  auto copy = v;
  auto multiplier = 2.4;
  auto name = "Avery";
  auto betterName = string{"Avery“};
  auto& refMult = multiplier;
  auto func = [](auto i) {return i*2};

  return betterName;
}
```

name is a char* because the literal forms a C-string.

auto can be used in almost all places!

```
auto calculateSum(const vector<string>& v) {
  auto copy = v;
  auto multiplier = 2.4;
  auto name = "Avery";
  auto betterName = string{"Avery“};
  auto& refMult = multiplier;
  auto func = [](auto i) {return i*2};

  return betterName;
}
```

The fix is to call the string constructor to create a string.

auto can be used in almost all places!

```
auto calculateSum(const vector<string>& v) {
  auto copy = v;
  auto multiplier = 2.4;
  auto name = "Avery";
  auto betterName = string{"Avery“};
  auto& refMult = multiplier;
  auto func = [](auto i) {return i*2};

  return betterName;
}
```

refMult is a reference to a double.

auto can be used in almost all places!

```
auto calculateSum(const vector<string>& v) {
  auto copy = v;
  auto multiplier = 2.4;
  auto name = "Avery";
  auto betterName = string{"Avery“};
  auto& refMult = multiplier;
  auto func = [](auto i) {return i*2};

  return betterName;
}
```

Sometimes you don’t know the type, and need to ask the compiler for it.

auto can be used in almost all places!

```
auto calculateSum(const vector<string>& v) {
  auto copy = v;
  auto multiplier = 2.4;
  auto name = "Avery";
  auto betterName = string{"Avery“};
  auto& refMult = multiplier;
  auto func = [](auto i) {return i*2};

  return betterName;
}
```

The compiler can figure out the return type too!
Here it’s a string.

auto can be used in almost all places!

```
auto calculateSum(const vector<string>& v) {
  auto copy = v;
  auto multiplier = 2.4;
  auto name = "Avery";
  auto betterName = string{"Avery“};
  auto& refMult = multiplier;
  auto func = [](auto i) {return i*2};

  return betterName;
}
```

One place where you can’t use auto: function parameters.

auto offers many benefits.

Correctness
avoid bug such as implicit conversions and uninitialized variables.

Flexibility
code more flexible if you later change the types of the variables.

When auto should be used is a pretty contentious topic.

Good guidelines to follow:

- Use it when the type is clear from context.
- Use it when the exact type is unimportant.
- Don’t use it when it obviously hurts readability.
- Don’t use auto in CS 106B.

When auto should be used is a pretty contentious topic.

Good guidelines to follow:

- Use it when the type is clear from context.
- Use it when the exact type is unimportant.
- Don’t use it when it obviously hurts readability.
- Don’t use auto in CS 106B.

```
auto spliceString(const string& s);
```

Can you guess what this function returns? Not really.

structures

Reference parameters are the classic CS 106B solution to return multiple values.

```
void findPriceRange(int dist, int& min, int& max) {
  min = static_cast<int>(dist * 0.08 + 100);
  max = static_cast<int>(dist * 0.36 + 750);
}

int main() {
  int dist = 6452;
  int min, max;
  findPriceRange(age, min, max);
  cout << “You can find prices between: ”
       << min << “ and “ << max;
}
```

Reference parameters are the classic CS 106B solution to return multiple values.

```
void findPriceRange(int dist, int& min, int& max) {
  min = static_cast<int>(dist * 0.08 + 100);
  max = static_cast<int>(dist * 0.36 + 750);
}

int main() {
  int dist = 6452;
  int min, max;
  findPriceRange(age, min, max);
  cout << “You can find prices between: ”
       << min << “ and “ << max;
}
```

It’s not clear that min and max are reference parameters.

Returning the output is more natural.

```
pair<int, int> findPriceRange(int dist) {
  int min = static_cast<int>(dist * 0.08 + 100);
  int max = static_cast<int>(dist * 0.36 + 750);
  return make_pair(min, max);
}

int main() {
  int dist = 6452;
  pair<int, int> p = findPriceRange(dist);
  cout << “You can find prices between: ”
       << p.first << “ and “ << p.second;
}
```

More natural: return a pair of values.

Returning the output is more natural.

```
pair<int, int> findPriceRange(int dist) {
  int min = static_cast<int>(dist * 0.08 + 100);
  int max = static_cast<int>(dist * 0.36 + 750);
  return make_pair(min, max);
}

int main() {
  int dist = 6452;
  auto p = findPriceRange(dist);
  cout << “You can find prices between: ”
       << p.first << “ and “ << p.second;
}
```

Even better: use auto!

C++17 allows structured bindings, allowing you to unpack the variables in a pair.

```
pair<int, int> findPriceRange(int dist) {
  int min = static_cast<int>(dist * 0.08 + 100);
  int max = static_cast<int>(dist * 0.36 + 750);
  return make_pair(min, max);
}

int main() {
  int dist = 6452;
  auto [min, max] = findPriceRange(dist);
  cout << “You can find prices between: ”
       << min << “ and “ << max;
}
```

Notice the use of auto here – automatically infer the type of each variable.

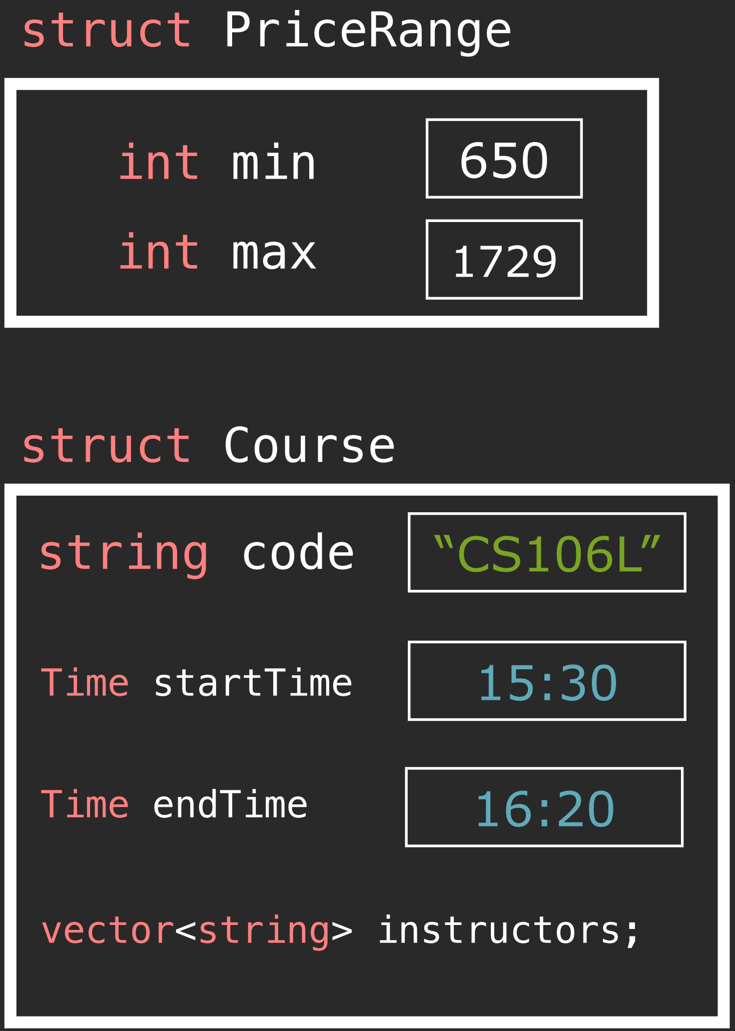
C++17 allows structured bindings, allowing you to unpack the variables in a pair.

```
pair<int, int> findPriceRange(int dist) {
  int min = static_cast<int>(dist * 0.08 + 100);
  int max = static_cast<int>(dist * 0.36 + 750);
  return make_pair(min, max);
}

int main() {
  int dist = 6452;
  auto [min, max] = findPriceRange(dist);
  cout << “You can find prices between: ”
       << min << “ and “ << max;
}
```

What drawbacks of this approach can you see?

A struct is a collection of named variables grouped together.

```
struct PriceRange {
  int min;
  int max;
}

struct Course {
  string code;
  Time startTime; Time endTime;

  vector<string> instructors;
}
```



struct PriceRange

int min 650

int max 1729

struct Course

string code “CS106L”

Time startTime 15:30

Time endTime 16:20

vector<string> instructors;

A struct is a collection of named variables grouped together.

```
struct PriceRange {
  int min;
  int max;
}

struct Course {
  string code;
  Time startTime; Time endTime;

  vector<string> instructors;
}
```

Same as a pair<int, int>, but the ints are named.

Structs offer the benefit that the struct itself and the variables inside are named.

```
PriceRange findPriceRange(int dist) {
  int min = static_cast<int>(dist * 0.08 + 100);
  int max = static_cast<int>(dist * 0.36 + 750);
  return PriceRange{min, max};
}

int main() {
  int dist = 6452;
  PriceRange p = findPriceRange(dist);
  cout << “You can find prices between: ”
       << p.min << “ and “ << p.max;
}
```

This is very readable: result is a DatingRange, and you are printing its min and max.

Structs offer the benefit that the struct itself and the variables inside are named.

```
PriceRange findPriceRange(int dist) {
  int min = static_cast<int>(dist * 0.08 + 100);
  int max = static_cast<int>(dist * 0.36 + 750);
  return PriceRange{min, max};
}

int main() {
  int dist = 6452;
  PriceRange p = findPriceRange(dist);
  cout << “You can find prices between: ”
       << p.min << “ and “ << p.max;
}
```

To access a member inside the struct, use the . notation.

You can also use structured bindings on structs.

```
PriceRange findPriceRange(int dist) {
  int min = static_cast<int>(dist * 0.08 + 100);
  int max = static_cast<int>(dist * 0.36 + 750);
  return PriceRange{min, max};
}

int main() {
  int dist = 6452;
  auto [min, max] = findPriceRange(dist);
  cout << “You can find prices between: ”
       << min << “ and “ << max;
}
```

The order the binding occurs is the same order as the variables are laid in the struct.

Structures are frequently used in practice.

```
std::tuple<bool, Time, Time> findCourseTime([omitted]) {
  for (int i = 0; i < courseDatabase.size(); ++i) {
    if (courseCode == course.code) {
      return make_tuple(true, course.startTime,
                              course.endTime);
    }
  }

  return make_tuple(false, Time{}, Time{});
}
```

Note: we’ll clean this up using uniform initialization later!

Often times the first component is a boolean indicating if a query was successful.

```
std::tuple<bool, Time, Time> findCourseTime([omitted]) {
  for (int i = 0; i < courseDatabase.size(); ++i) {
    if (courseCode == course.code) {
      return make_tuple(true, course.startTime,
                              course.endTime);
    }
  }

  return make_tuple(false, Time{}, Time{});
}
```

First component is true if we found the course, false otherwise.

Often times the first component is a boolean indicating if a query was successful.

```
std::tuple<bool, Time, Time> findCourseTime([omitted]) {
  for (int i = 0; i < courseDatabase.size(); ++i) {
    if (courseCode == course.code) {
      return make_tuple(true, course.startTime,
                              course.endTime);
    }
  }

  return make_tuple(false, Time{}, Time{});
}
```

If first component is false, the latter components aren’t used.

Often times the a component is a boolean indicating if a query was successful.

```
pair<iterator, bool> insert (const value_type& val);
```

A major difference between
Stanford vs. STL.

references

In CS 106B: you learned about reference parameters.

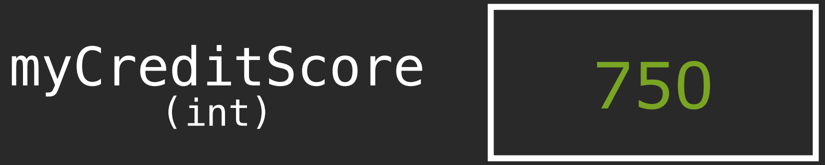
```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
```

```
int main() {
  int myCreditScore = 750;
  getOutrageousLoan(myCreditScore);
  cout << myCreditScore << endl;
}
```

In CS 106B: you learned about reference parameters.

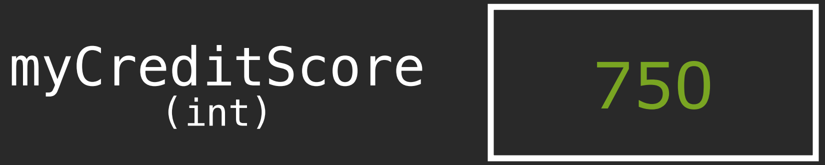
```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
```

```
int main() {
   int myCreditScore = 750;
   getOutrageousLoan(myCreditScore);
   cout << myCreditScore << endl;
}
```



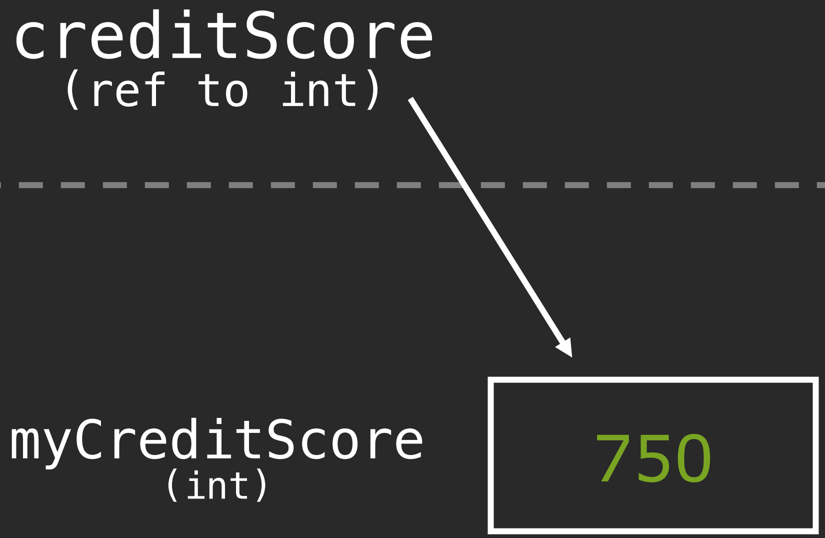
In CS 106B: you learned about reference parameters.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
--------------------------------------------------------------
int main() {
  int myCreditScore = 750;
  getOutrageousLoan(myCreditScore);
  cout << myCreditScore << endl;
}
```



In CS 106B: you learned about reference parameters.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
-----------------------------------------------------------
int main() {
   int myCreditScore = 750;
   getOutrageousLoan(myCreditScore);
   cout << myCreditScore << endl;
}
```



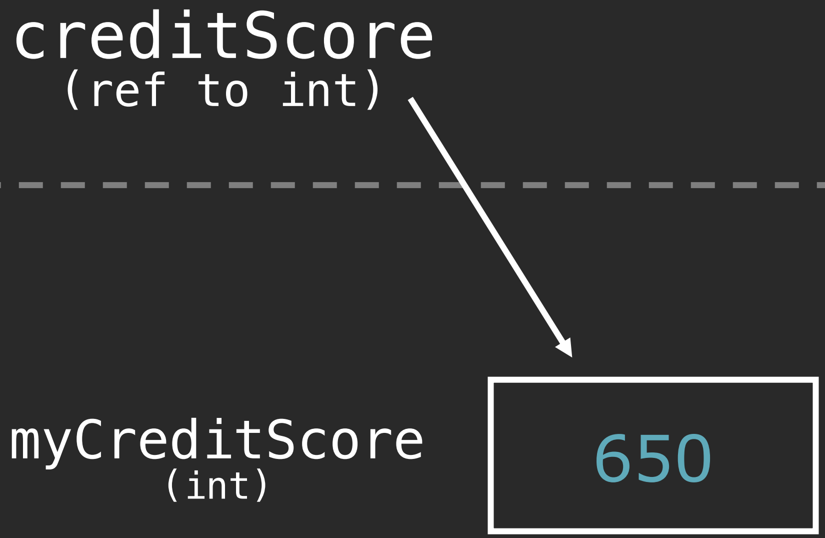
creditScore
(ref to int)

myCreditScore
(int)

750

In CS 106B: you learned about reference parameters.

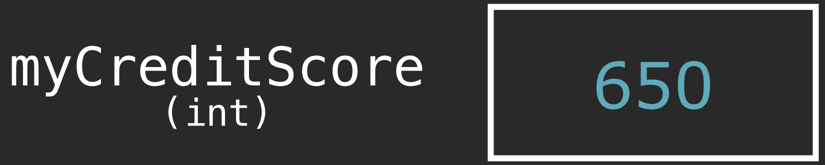
```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
--------------------------------------------------------------
int main() {
  int myCreditScore = 750;
  getOutrageousLoan(myCreditScore);
  cout << myCreditScore << endl;
}
```



In CS 106B: you learned about reference parameters.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
```

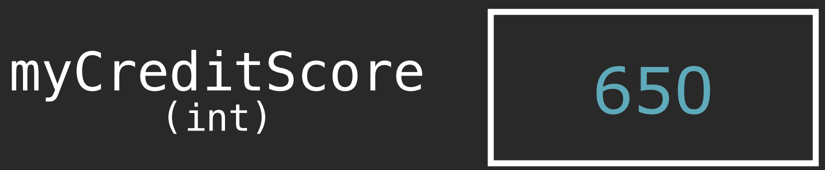
```
int main() {
  int myCreditScore = 750;
  getOutrageousLoan(myCreditScore);
  cout << myCreditScore << endl;
}
```



In CS 106B: you learned about reference parameters.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
```

```
int main() {
  int myCreditScore = 750;
  getOutrageousLoan(myCreditScore);
  cout << myCreditScore << endl;
}
```



myCreditScore
(int)

650

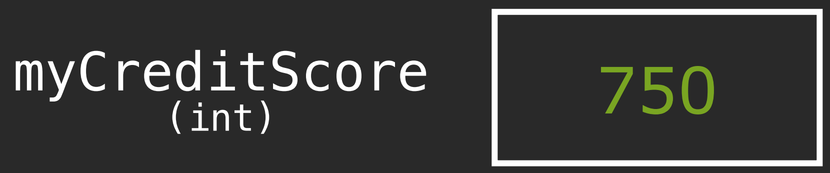
Local variables work the same way.
You can make a copy of a variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
```

```
int main() {
  int myCreditScore = 750;
  int copyCreditScore = myCreditScore;
  getOutrageousLoan(copyCreditScore);
  cout << myCreditScore << endl;
}
```

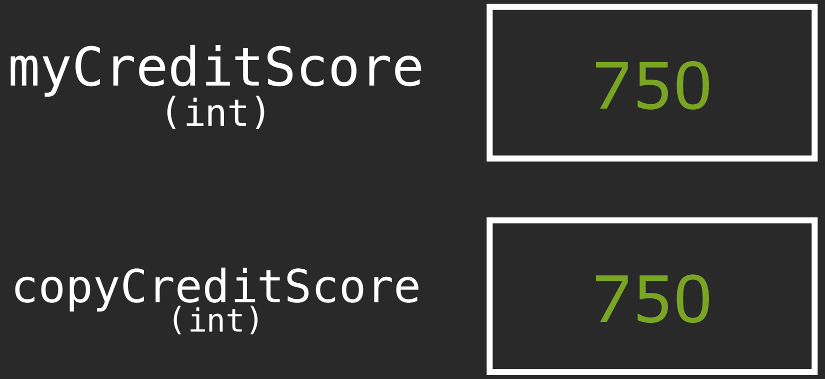
Local variables work the same way.
You can make a copy of a variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
----------------------------------------------------------------
int main() {
  int myCreditScore = 750;
  int copyCreditScore = myCreditScore;
  getOutrageousLoan(copyCreditScore);
  cout << myCreditScore << endl;
}
```



Local variables work the same way.
You can make a copy of a variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
----------------------------------------------------------------
int main() {
   int myCreditScore = 750;
   int copyCreditScore = myCreditScore;
   getOutrageousLoan(copyCreditScore);
   cout << myCreditScore << endl;
}
```

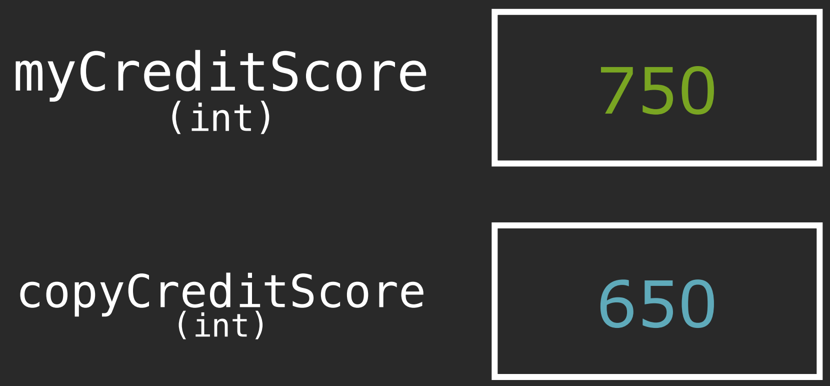


myCreditScore (int) 750

copyCreditScore (int) 750

Local variables work the same way. You can make a copy of a variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}

int main() {
   int myCreditScore = 750;
   int copyCreditScore = myCreditScore;
   getOutrageousLoan(copyCreditScore);
   cout << myCreditScore << endl;
}
```

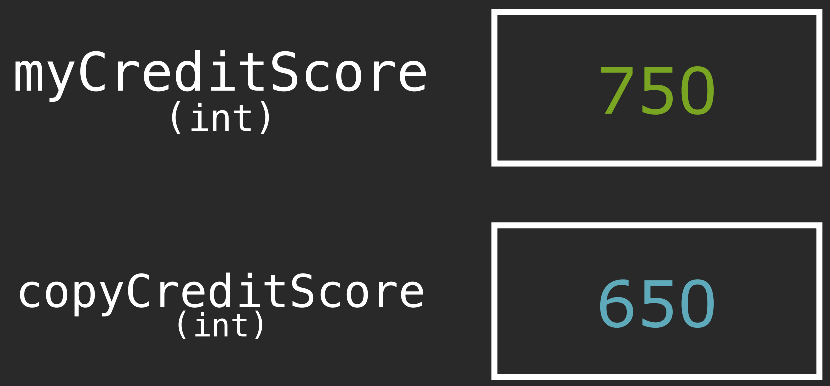


myCreditScore (int) 750

copyCreditScore (int) 650

Local variables work the same way.
You can make a copy of a variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
--------------------------------------------------------------
int main() {
  int myCreditScore = 750;
  int copyCreditScore = myCreditScore;
  getOutrageousLoan(copyCreditScore);
  cout << myCreditScore << endl;
}
```



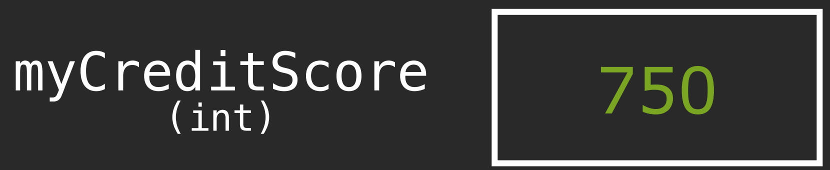
Or you can make a reference to a local variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
```

```
int main() {
  int myCreditScore = 750;
  int& refCreditScore = myCreditScore;
  getOutrageousLoan(refCreditScore);
  cout << myCreditScore << endl;
}
```

Or you can make a reference to a local variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
---------------------------------------------------------------
int main() {
  int myCreditScore = 750;
  int& refCreditScore = myCreditScore;
  getOutrageousLoan(refCreditScore);
  cout << myCreditScore << endl;
}
```

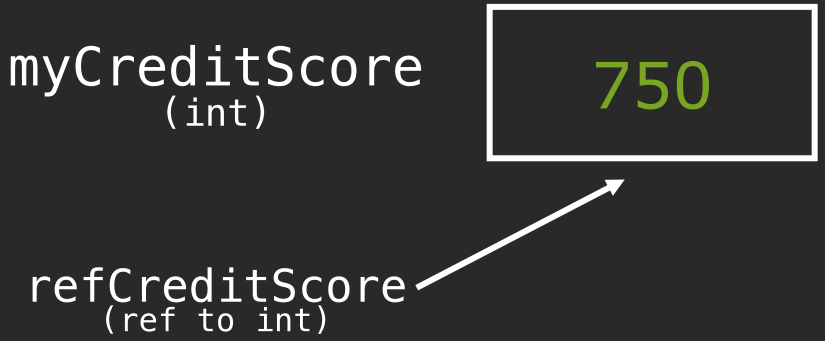


myCreditScore
(int)

750

Or you can make a reference to a local variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
----------------------------------------------------------------
int main() {
  int myCreditScore = 750;
  int& refCreditScore = myCreditScore;
  getOutrageousLoan(refCreditScore);
  cout << myCreditScore << endl;
}
```



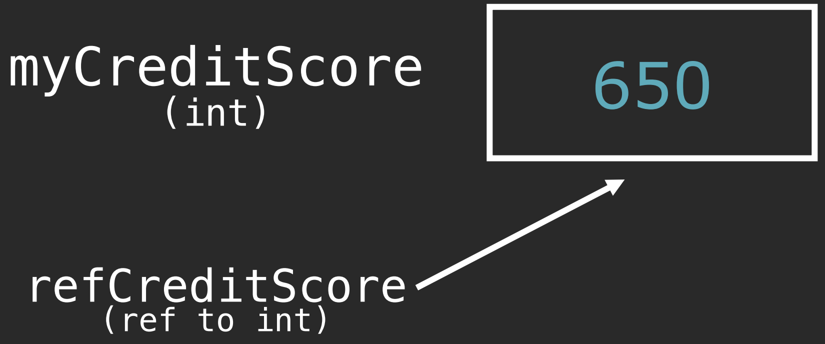
myCreditScore
(int)

750

refCreditScore
(ref to int)

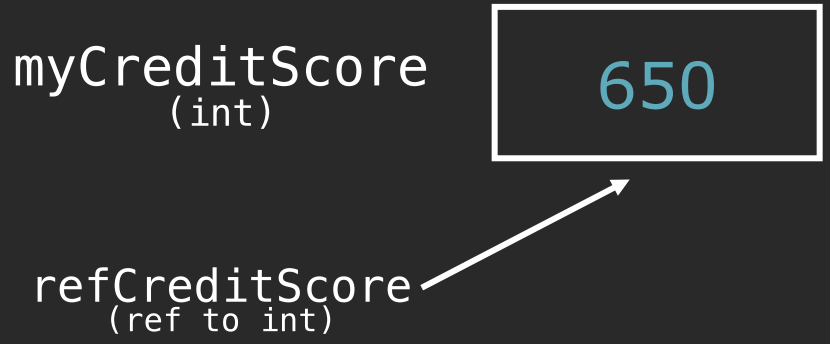
Or you can make a reference to a local variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
--------------------------------
int main() {
  int myCreditScore = 750;
  int& refCreditScore = myCreditScore;
  getOutrageousLoan(refCreditScore);
  cout << myCreditScore << endl;
}
```



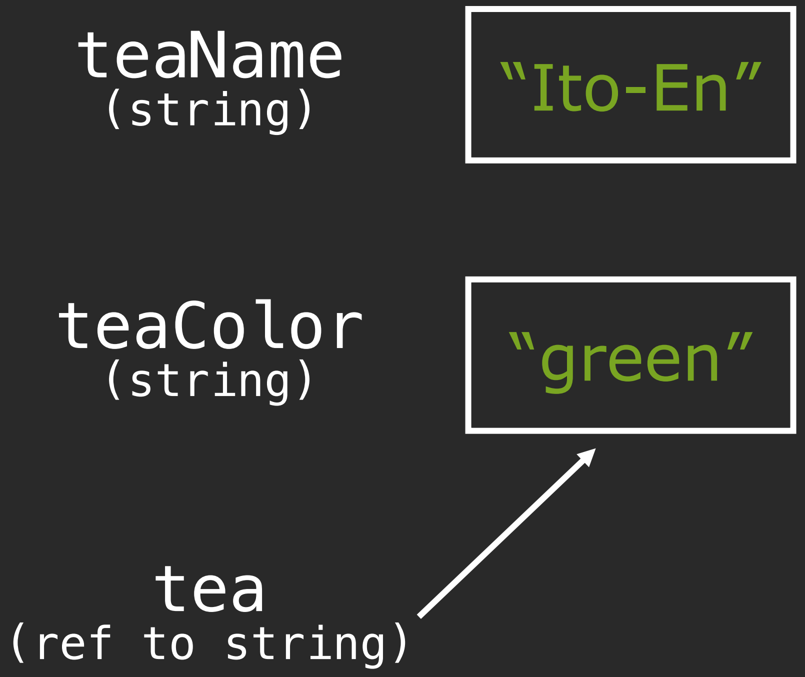
Or you can make a reference to a local variable.

```
void getOutrageousLoan(int& creditScore) {
   creditScore -= 100;
}
--------------------------------------------------------------
int main() {
   int myCreditScore = 750;
   int& refCreditScore = myCreditScore;
   getOutrageousLoan(refCreditScore);
   cout << myCreditScore << endl;
}
```



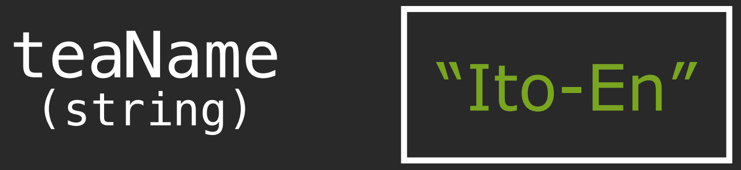
An (l-value) reference is simply another name for a named variable.

```
string teaName = “Ito–En”;
string teaColor = “green”;

string& tea = teaColor;
string& favorite;

tea = teaName;
```



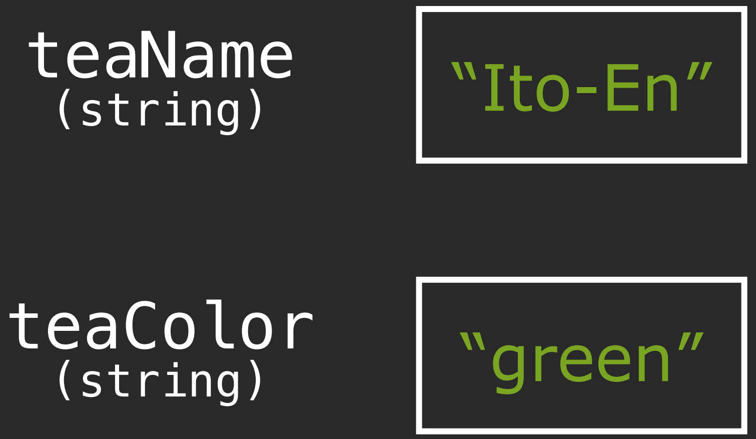
An (l-value) reference is simply another name for a named variable.

```
string teaName = “Ito-En”;
string teaColor = “green”;

string& tea = teaColor;
string& favorite;

tea = teaName;
```



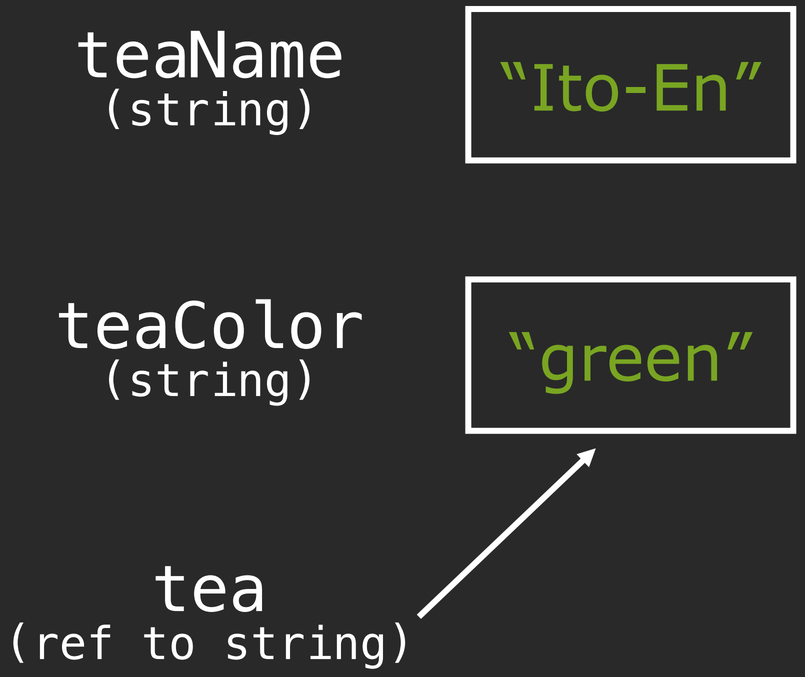
An (l-value) reference is simply another name for a named variable.

```
string teaName = “Ito-En”;
string teaColor = “green”;

string& tea = teaColor;
string& favorite;

tea = teaName;
```



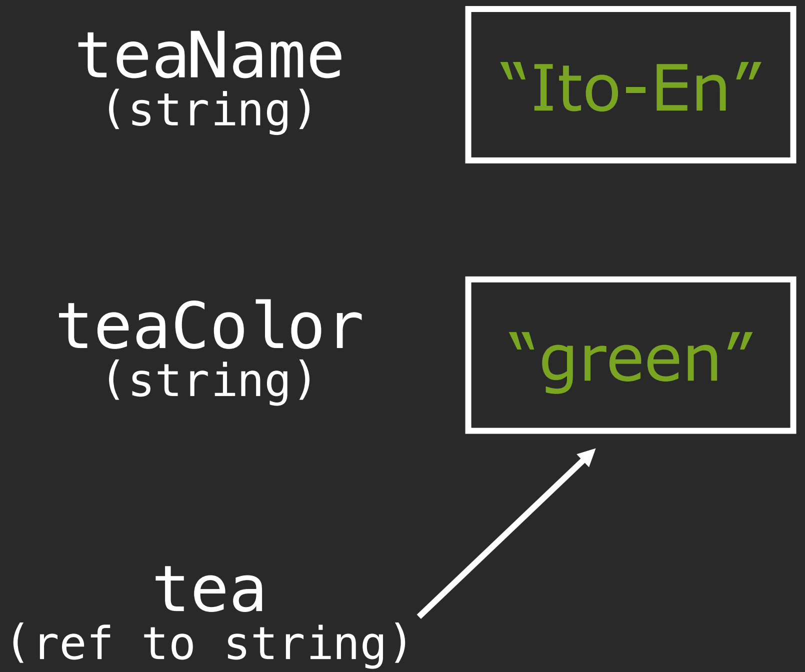
An (l-value) reference is simply another name for a named variable.

```
string teaName = “Ito-En”;
string teaColor = “green”;

string& tea = teaColor;
string& favorite;

tea = teaName;
```



References must be initialized upon declaration.

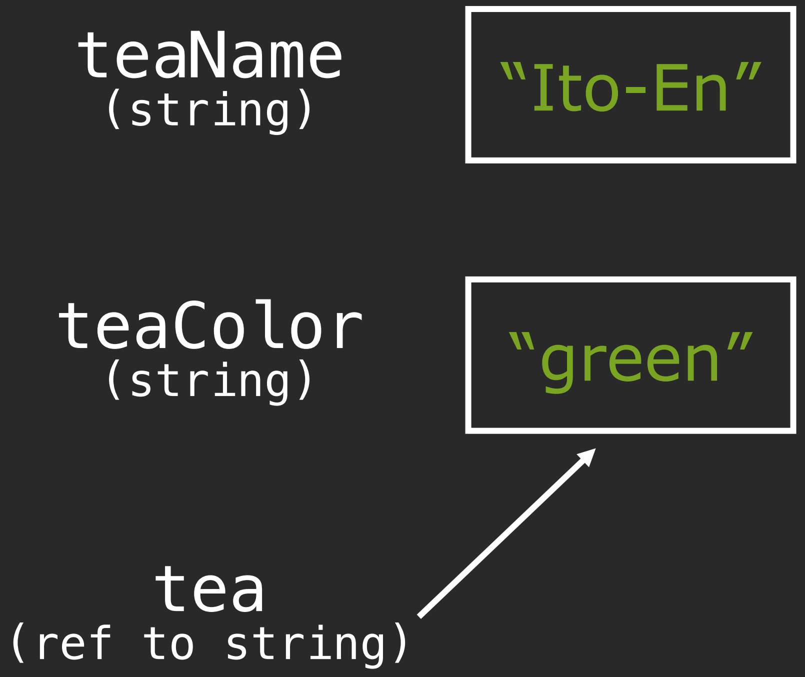
```
string teaName = “Ito-En”;
string teaColor = “green”;

string& tea = teaColor;
string& favorite;

tea = teaName;
```



Error: this line does not compile!

An (l-value) reference is simply another name for a named variable.

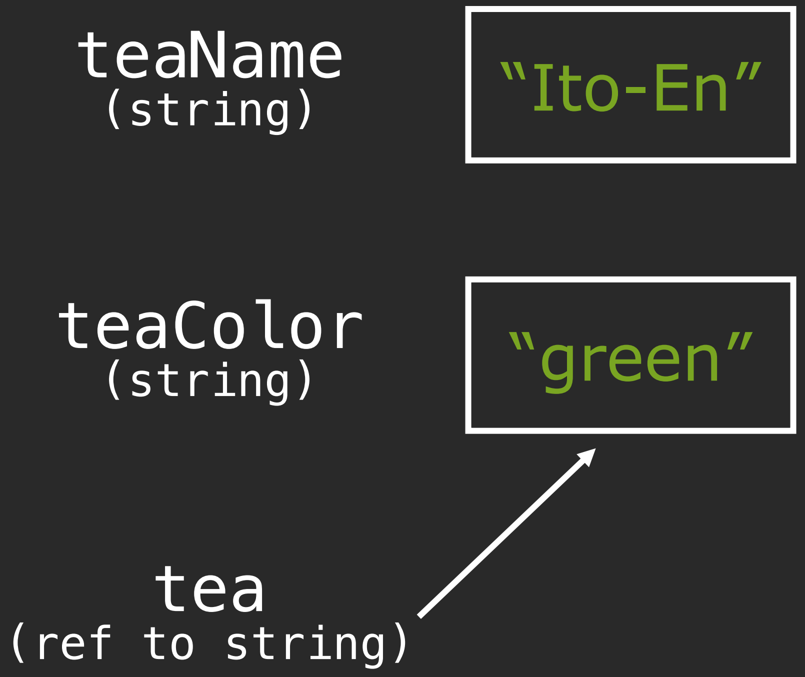
```
string teaName = “Ito-En”;
string teaColor = “green”;

string& tea = teaColor;
string& favorite = “16.9 ounces”;

tea = teaName;
```



Error: the string literal is not a named variable.

An r-value reference is a reference for an unnamed (temporary) value.

```
string teaName = “Ito-En”;
string teaColor = “green”;

string& tea = teaColor;
string&& favorite = “16.9 ounces”;

tea = teaName;
```



teaName
(string)

“Ito-En”

teaColor
(string)

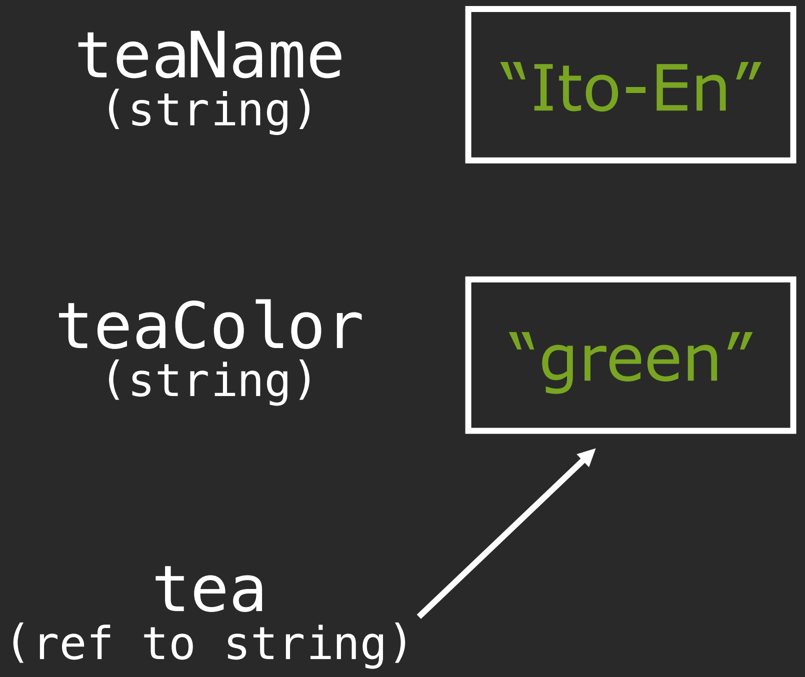
“green”

tea
(ref to string)

We’ll discuss this more
in week 9!

References cannot be reassigned after initialization.

```
string teaName = “Ito-En”;
string teaColor = “green”;

string& tea = teaColor;


tea = teaName;
```



teaName
(string)

“Ito-En”

teaColor
(string)

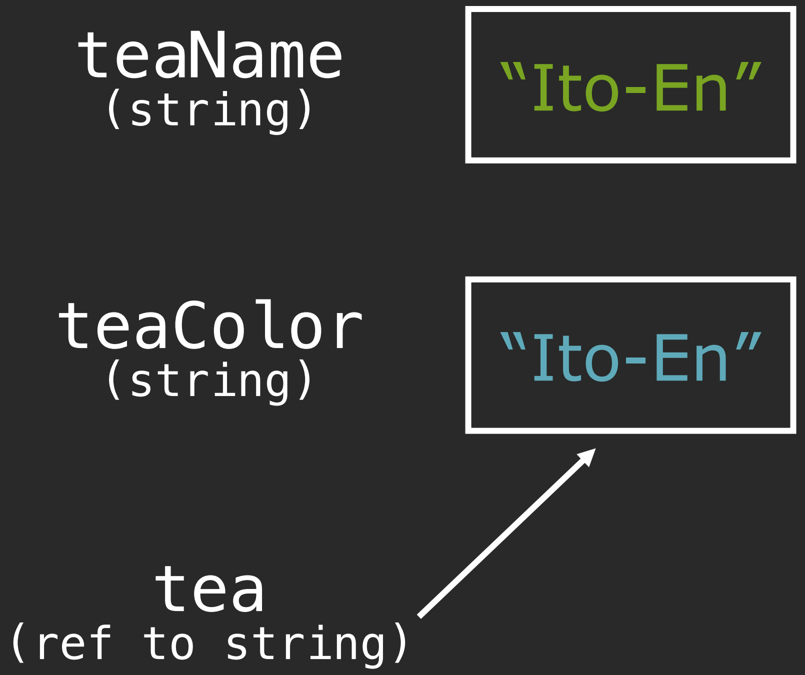
“green”

tea
(ref to string)

Quick quiz: what does this do?

References cannot be reassigned after initialization.

```
string teaName = “Ito-En”;
string teaColor = “green”;

string& tea = teaColor;


tea = teaName;
```



It sets teaColor (what tea is a reference to) to the value inside teaName.

You can also return a reference.

```
char& string::operator[](size_t pos);
```

This method returns a reference to the character in the string.

You can also return a reference.

```
int main() {
  string motto = “Ito-En is Life”;

  motto[12] = ‘t’;
  cout << motto << endl;
}
```

Setting the reference to the char to ‘t’ changes the actual char in the string.

You can also return a reference.

```
int main() {
  string motto = “Ito-En is Life”;

  motto[12] = ‘t’;
  cout << motto << endl;
}
```

Setting the reference to the char to ‘t’ changes the actual char in the string.

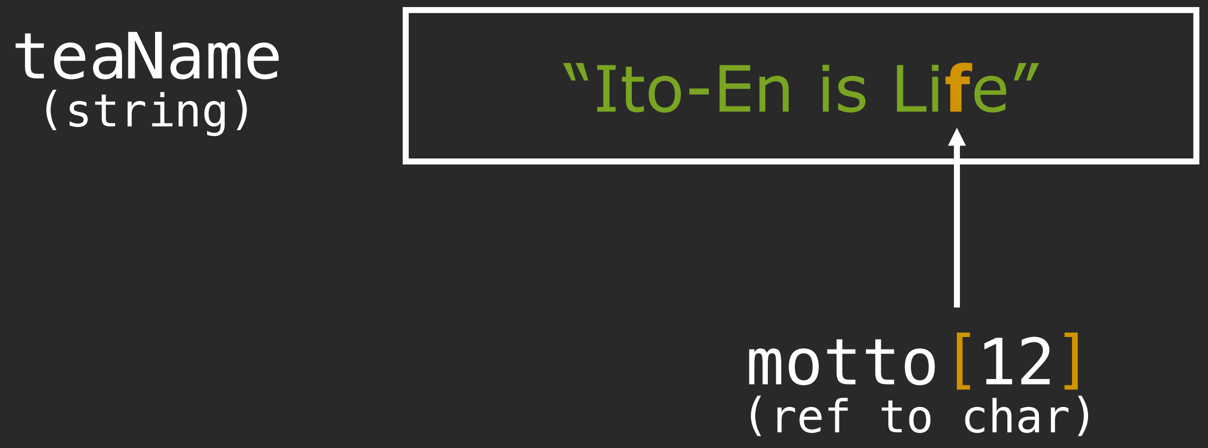
teaName
(string)

“Ito-En is Life”

You can also return a reference.

```
int main() {
  string motto = “Ito-En is Life”;

  motto[12] = ‘t’;
  cout << motto << endl;
}
```

Setting the reference to the char to ‘t’ changes the actual char in the string.



teaName
(string)

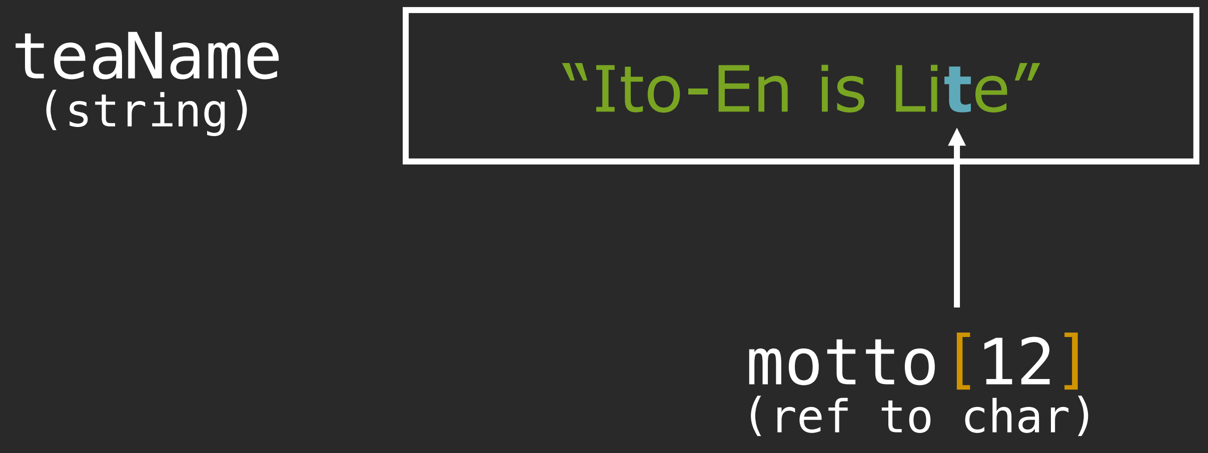
“Ito-En is Life”

motto[12]
(ref to char)

You can also return a reference.

```
int main() {
  string motto = “Ito-En is Life”;

  motto[12] = ‘t’;
  cout << motto << endl;
}
```

Setting the reference to the char to ‘t’ changes the actual char in the string.



teaName
(string)

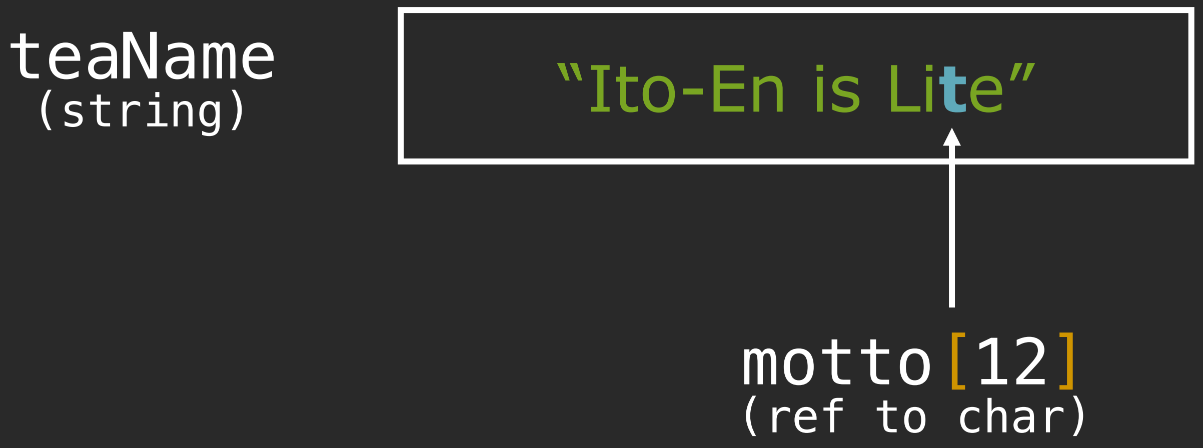
“Ito-En is Lite”

motto[12]
(ref to char)

You can also return a reference.

```
int main() {
  string motto = “Ito-En is Life”;

  motto[12] = ‘t’;
  cout << motto << endl;
}
```

Prints the string with the character changed.



You can also return a reference.

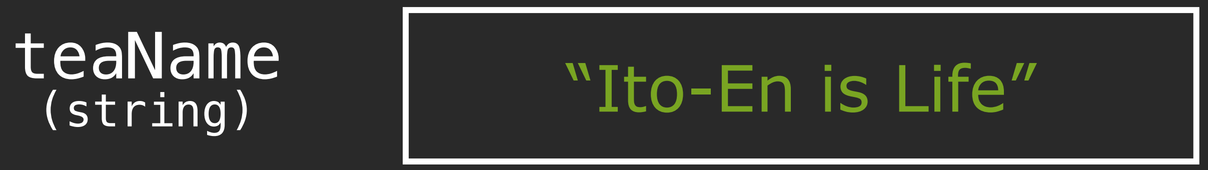
```
int main() {
  string motto = “Ito-En is Life”;
  auto letter = motto[12];
  letter = ‘t’;
  cout << motto << endl;
}
```

Here, we use a separate local variable, which creates a copy.

You can also return a reference.

```
int main() {
  string motto = “Ito–En is Life”;
  auto letter = motto[12];
  letter = ‘t’;
  cout << motto << endl;
}
```

Here, we use a separate local variable, which creates a copy.



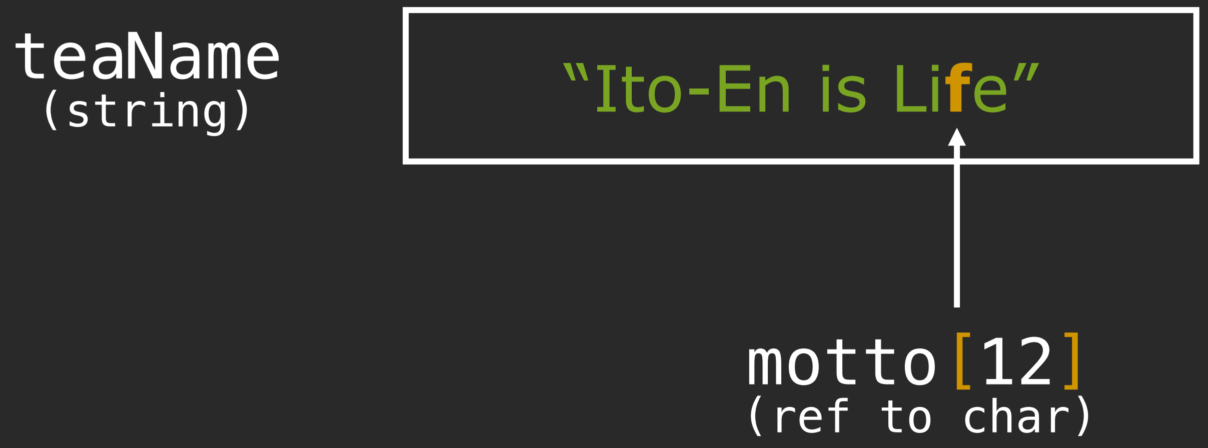
teaName
(string)

“Ito-En is Life”

You can also return a reference.

```
int main() {
  string motto = “Ito-En is Life”;
  auto letter = motto[12];
  letter = ‘t’;
  cout << motto << endl;
}
```

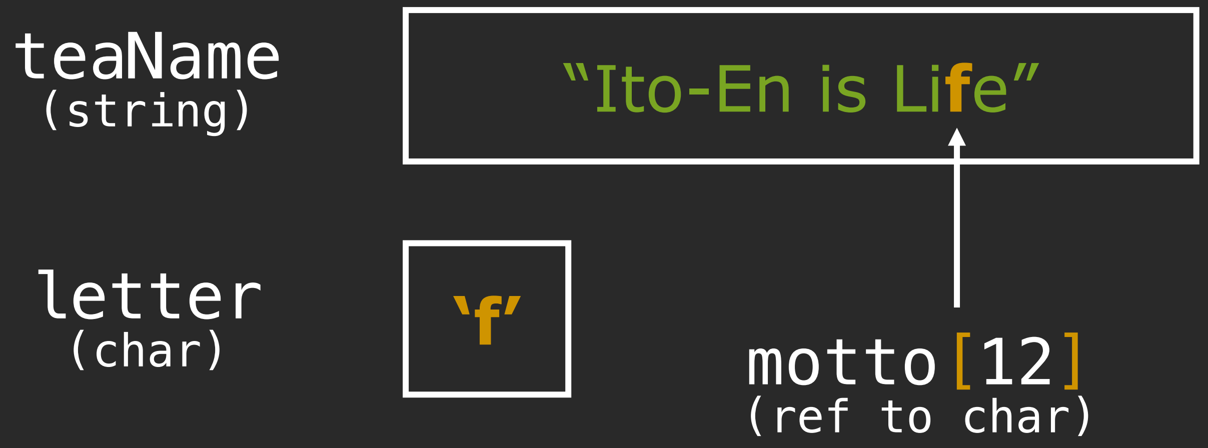
Here, we use a separate local variable, which creates a copy.



You can also return a reference.

```
int main() {
  string motto = “Ito-En is Life”;
  auto letter = motto[12];
  letter = ‘t’;
  cout << motto << endl;
}
```

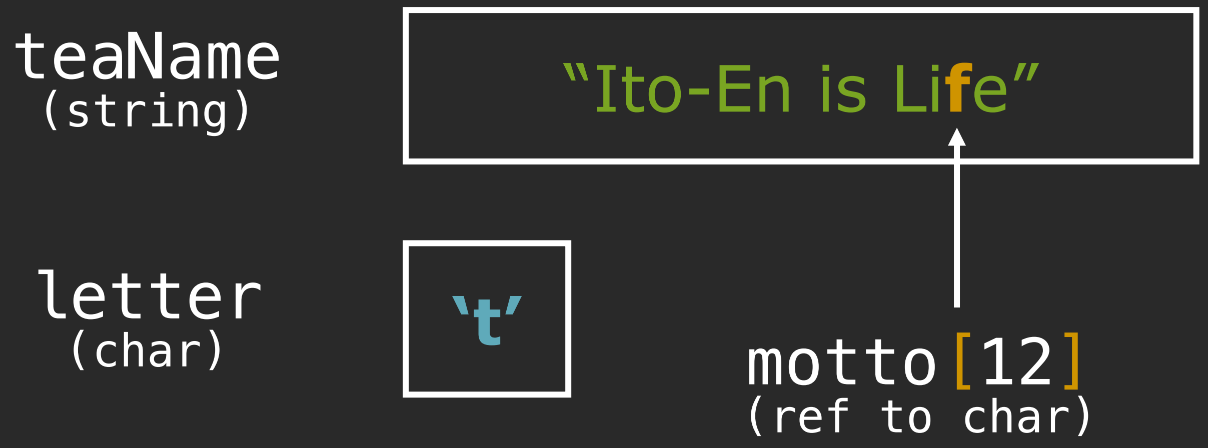
Reminder: using auto will not create a reference.



You can also return a reference.

```
int main() {
  string motto = “Ito-En is Life”;
  auto letter = motto[12];
  letter = ‘t’;
  cout << motto << endl;
}
```

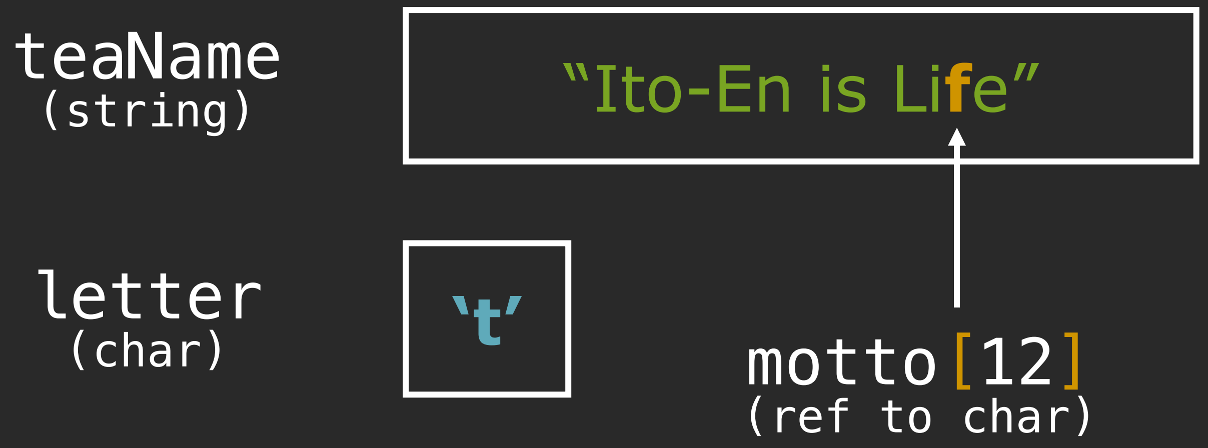
Here, we use a separate local variable, which creates a copy.



You can also return a reference.

```
int main() {
  string motto = “Ito-En is Life”;
  auto letter = motto[12];
  letter = ‘t’;
  cout << motto << endl;
}
```

Prints the string with the character unchanged.



Be careful about dangling references: references to variables out of scope.

```
string& getRandomWord() {
  string word = “COMPUTER”;
  return word;
}
```

```
int main() {
  string& randomWord = getRandomWord();
  cout << randomWord << endl;
}
```

Be careful about dangling references: references to variables out of scope.

```
  string& getRandomWord() {
    string word = “COMPUTER”;
    return word;
  }
----------------------------------------------------------------
  int main() {
    string& randomWord = getRandomWord();
    cout << randomWord << endl;
  }
```

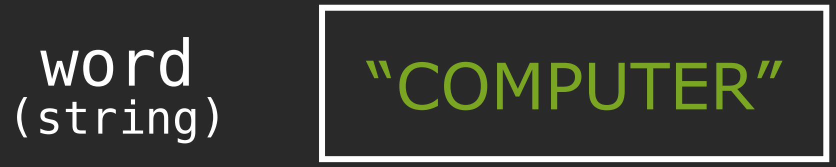
Be careful about dangling references: references to variables out of scope.

```
string& getRandomWord() {
  string word = “COMPUTER”;
  return word;
}
```

```
int main() {
  string& randomWord = getRandomWord();
  cout << randomWord << endl;
}
```

Be careful about dangling references: references to variables out of scope.

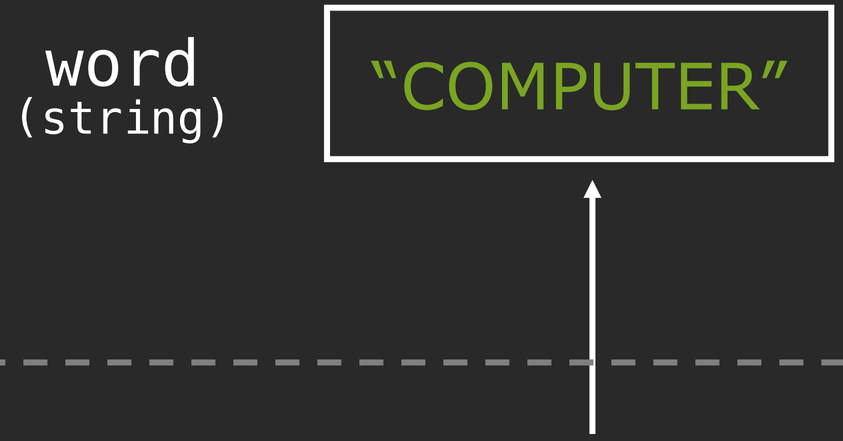
```
string& getRandomWord() {
  string word = “COMPUTER”;
  return word;
}
```



```
int main() {
  string& randomWord = getRandomWord();
  cout << randomWord << endl;
}
```

Be careful about dangling references: references to variables out of scope.

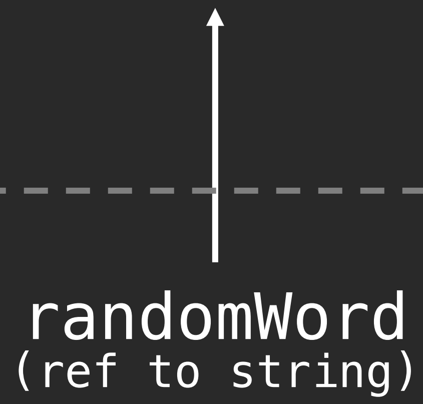
```
string& getRandomWord() {
  string word = “COMPUTER”;
  return word;
}
-------------------------------------------------------------
int main() {
  string& randomWord = getRandomWord();
  cout << randomWord << endl;
}
```



Be careful about dangling references: references to variables out of scope.

```
string& getRandomWord() {
  string word = “COMPUTER”;
  return word;
}
```

```
int main() {
  string& randomWord = getRandomWord();
  cout << randomWord << endl;
}
```

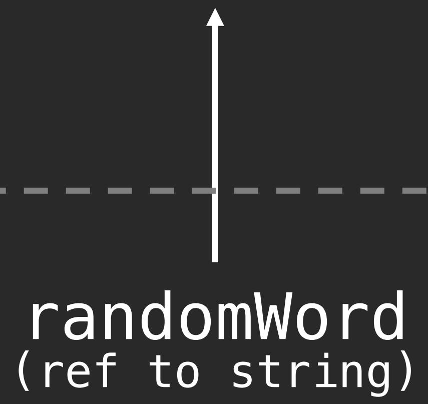


randomWord
(ref to string)

Be careful about dangling references: references to variables out of scope.

```
string& getRandomWord() {
  string word = “COMPUTER”;
  return word;
}
```

```
int main() {
  string& randomWord = getRandomWord();
  cout << randomWord << endl;
}
```



randomWord
(ref to string)

Prints garbage values!

There are times to return references, but don’t use it merely to “avoid copies”.

```
string getRandomWord() {
  string word = “COMPUTER”;
  return word;
}
```

```
int main() {
  string randomWord = getRandomWord();
  cout << randomWord << endl;
}
```

Makes copies but prints the correct string. We’ll discuss how to avoid copies later.

We want to go through all the Courses in the vector and add 1 to start and end time.

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

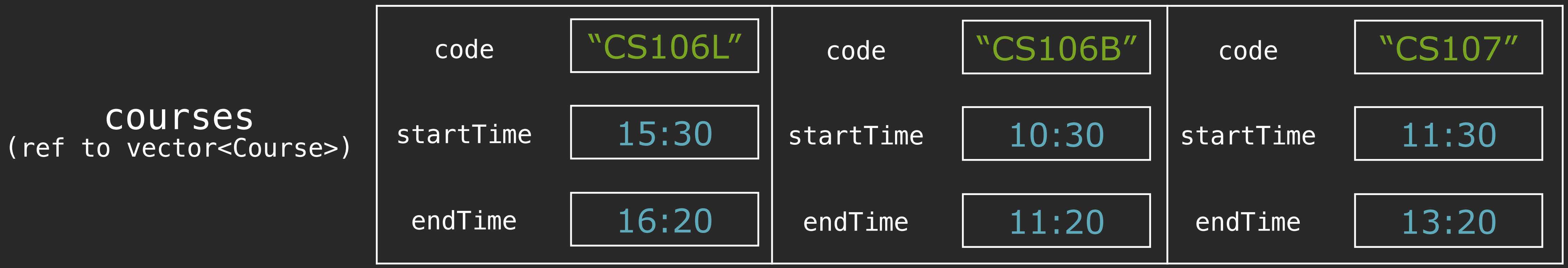
This code is buggy!

We want to go through all the Courses in the vector and add 1 to start and end time.

```
void transformToDST(vector<Course>& courses) {
  for (int i = 0; i < course.size(); ++i) {
    auto course = courses[i];
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

This equivalent code is buggy!

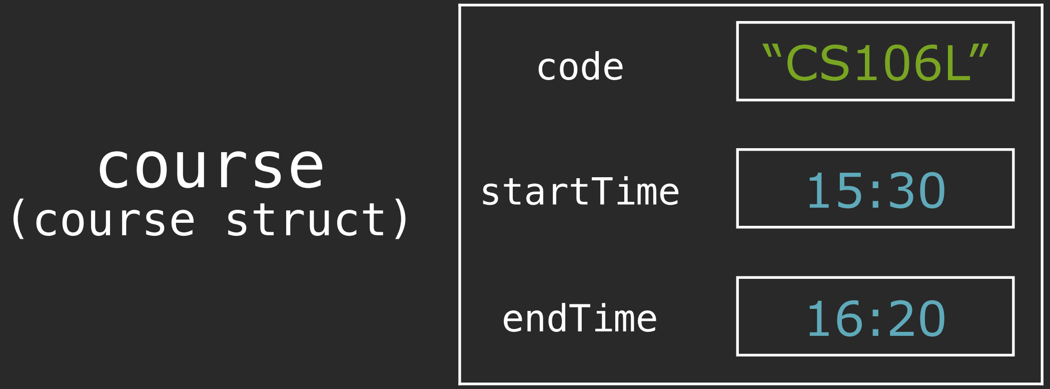
```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```



courses
(ref to vector<Course>)

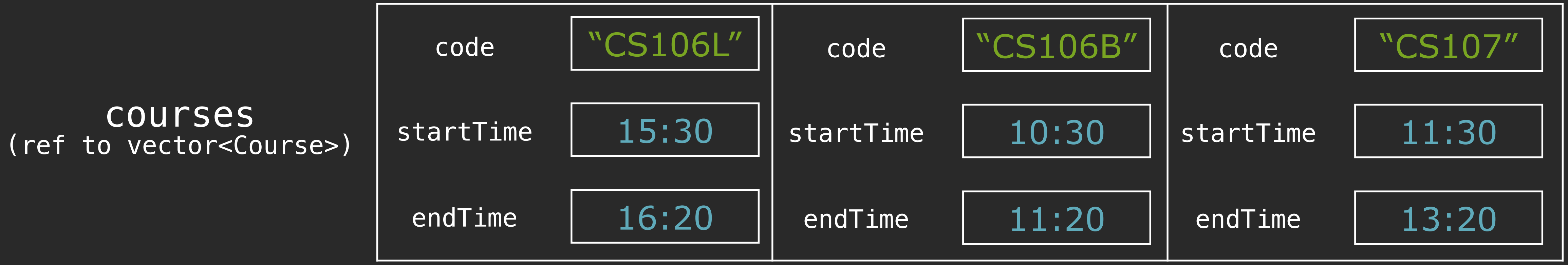
code	"CS106L"	code	"CS106B"	code	"CS107"
startTime	15:30	startTime	10:30	startTime	11:30
endTime	16:20	endTime	11:20	endTime	13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```



course
(course struct)

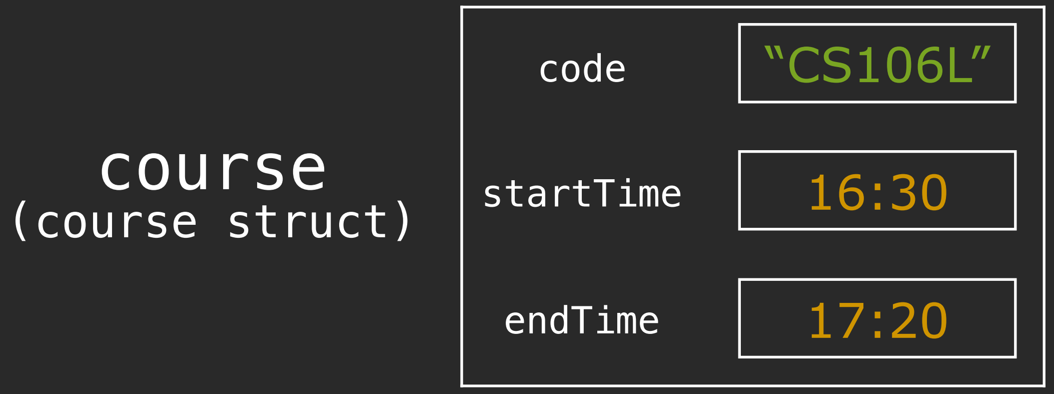
code	“CS106L”
startTime	15:30
endTime	16:20



courses
(ref to vector<Course>)

code	“CS106L”	code	“CS106B”	code	“CS107”
startTime	15:30	startTime	10:30	startTime	11:30
endTime	16:20	endTime	11:20	endTime	13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

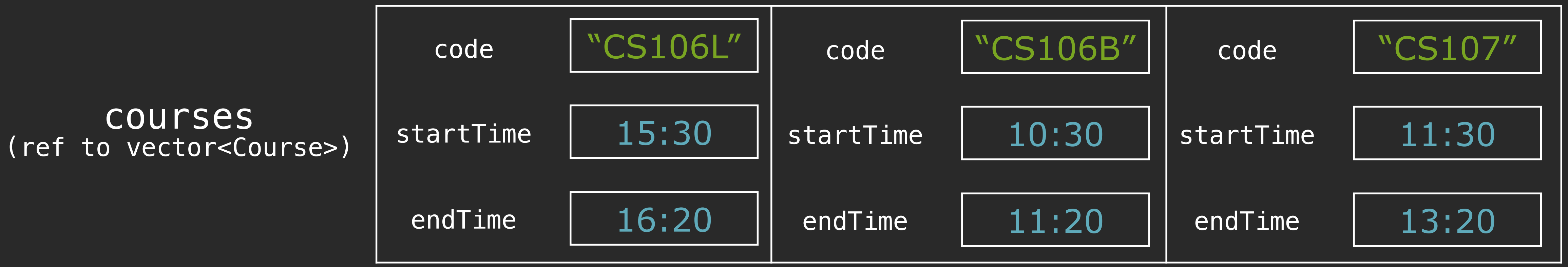


course
(course struct)

code “CS106L”

startTime 16:30

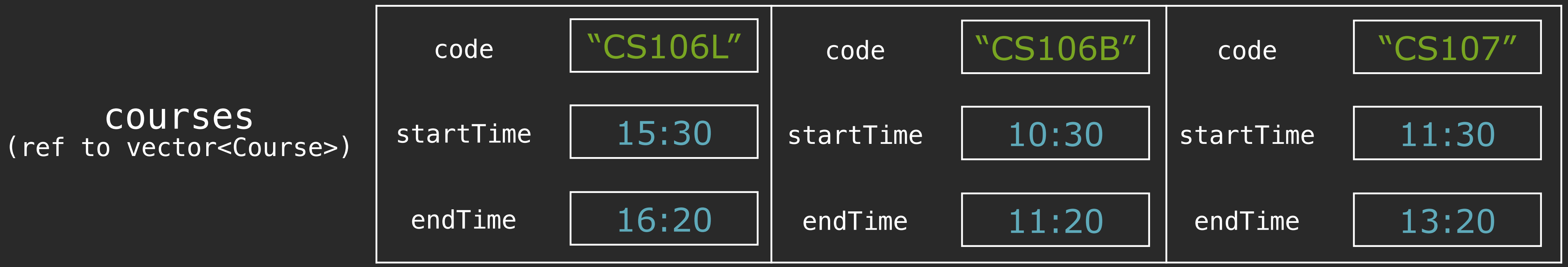
endTime 17:20



courses
(ref to vector<Course>)

code	“CS106L”	code	“CS106B”	code	“CS107”
startTime	15:30	startTime	10:30	startTime	11:30
endTime	16:20	endTime	11:20	endTime	13:20

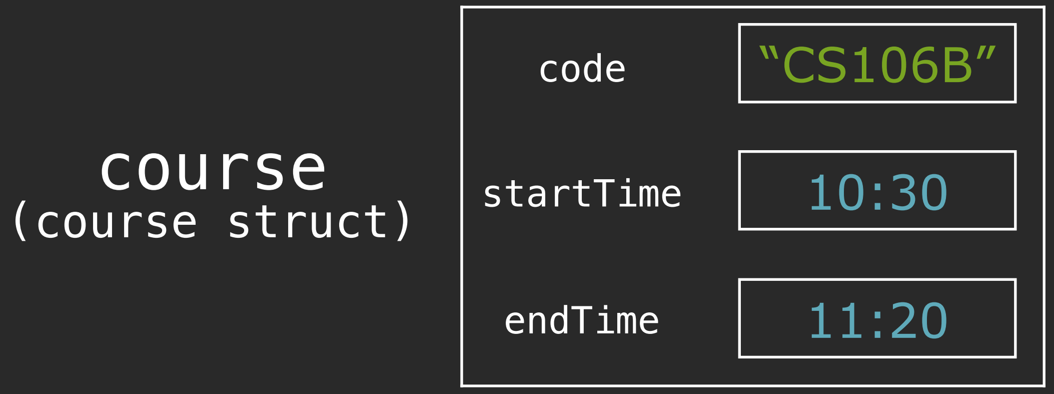
```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```



courses
(ref to vector<Course>)

code	"CS106L"	code	"CS106B"	code	"CS107"
startTime	15:30	startTime	10:30	startTime	11:30
endTime	16:20	endTime	11:20	endTime	13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

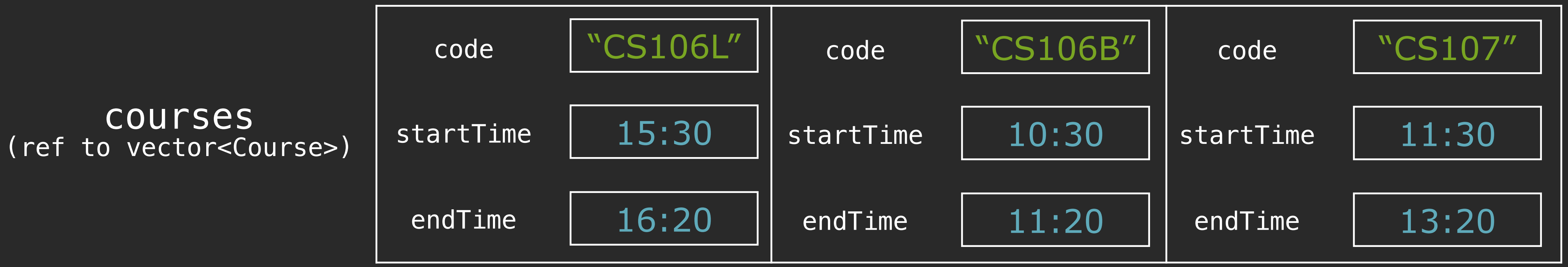


course
(course struct)

code "CS106B"

startTime 10:30

endTime 11:20



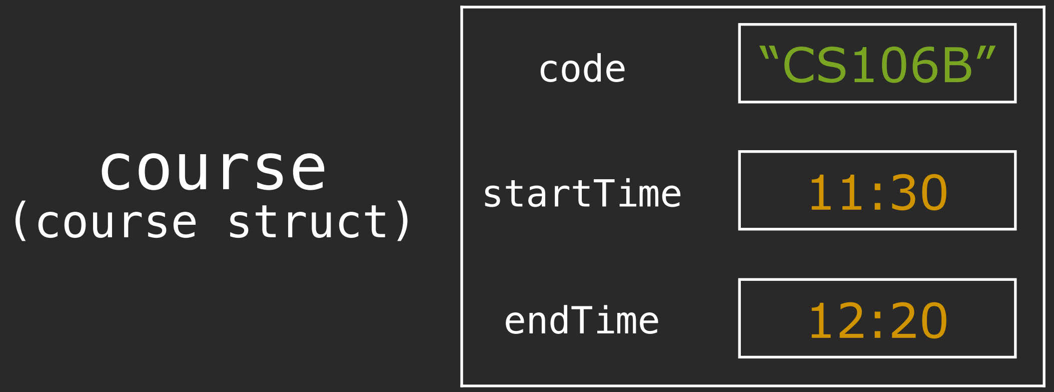
courses
(ref to vector<Course>)

code "CS106L" | code "CS106B" | code "CS107"

startTime 15:30 | startTime 10:30 | startTime 11:30

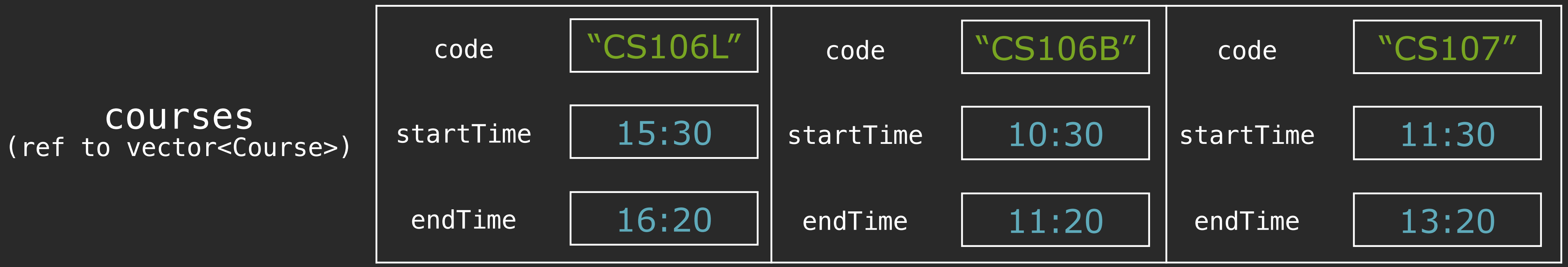
endTime 16:20 | endTime 11:20 | endTime 13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```



course
(course struct)

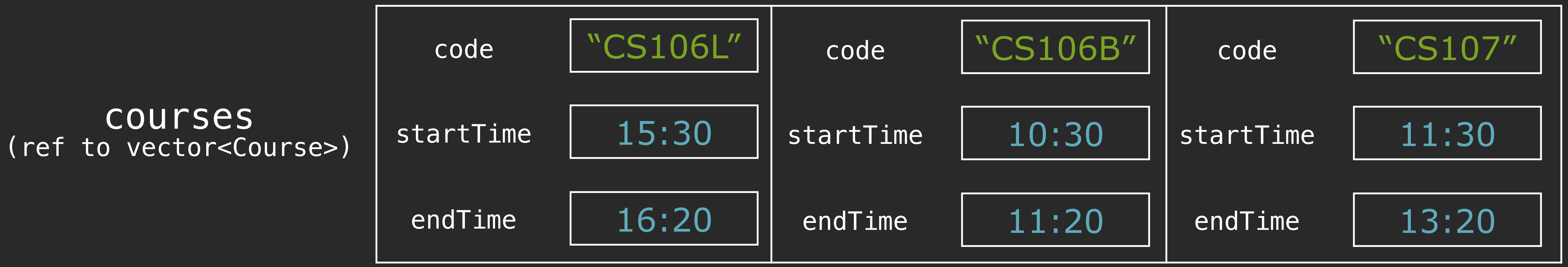
code	"CS106B"
startTime	11:30
endTime	12:20



courses
(ref to vector<Course>)

code	"CS106L"	code	"CS106B"	code	"CS107"
startTime	15:30	startTime	10:30	startTime	11:30
endTime	16:20	endTime	11:20	endTime	13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```



courses
(ref to vector<Course>)

code "CS106L"
startTime 15:30
endTime 16:20

code "CS106B"
startTime 10:30
endTime 11:20

code "CS107"
startTime 11:30
endTime 13:20

We want to go through all the Courses in the vector and add 1 to start and end time.

```
void transformToDST(vector<Course>& courses) {
  for (int i = 0; i < course.size(); ++i) {
    auto course = courses[i];
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

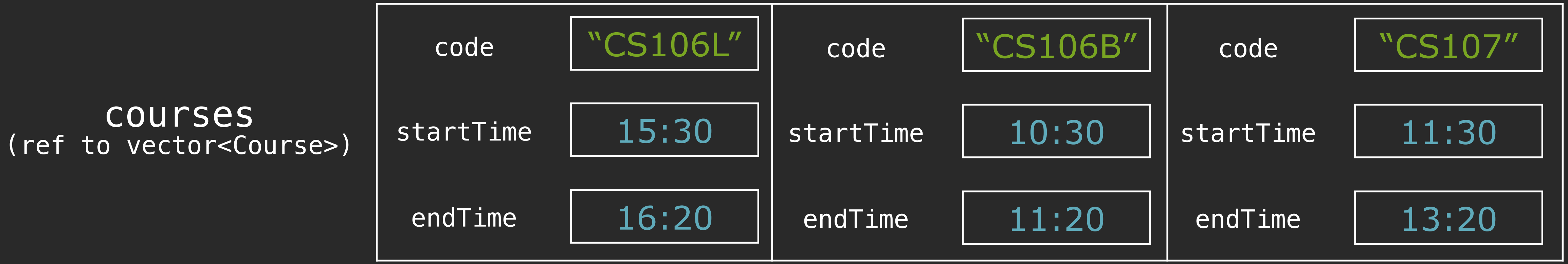
This line creates a copy.

We want to go through all the Courses in the vector and add 1 to start and end time.

```
void transformToDST(vector<Course>& courses) {
  for (auto& course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

This is equivalent to type inferring course as a reference.

```
void transformToDST(vector<Course>& courses) {
  for (auto& course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```



courses
(ref to vector<Course>)

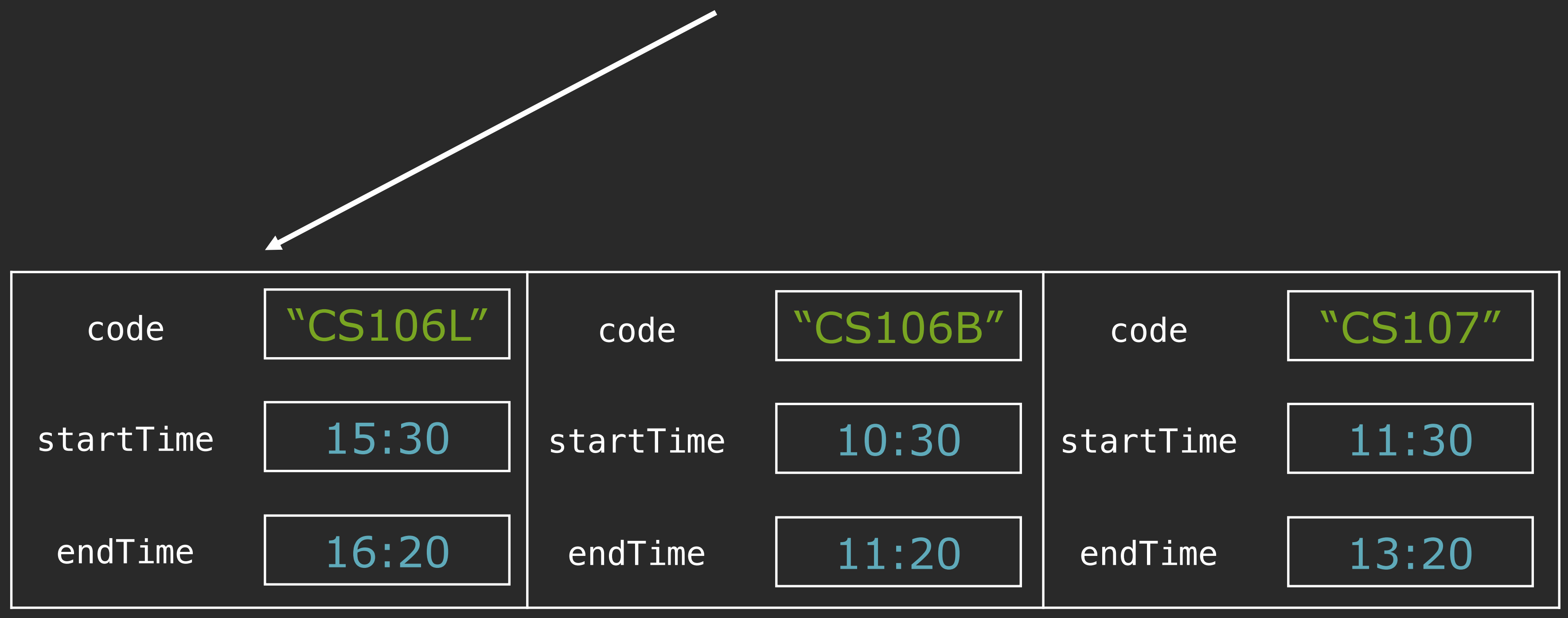
code "CS106L"
startTime 15:30
endTime 16:20

code "CS106B"
startTime 10:30
endTime 11:20

code "CS107"
startTime 11:30
endTime 13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto& course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

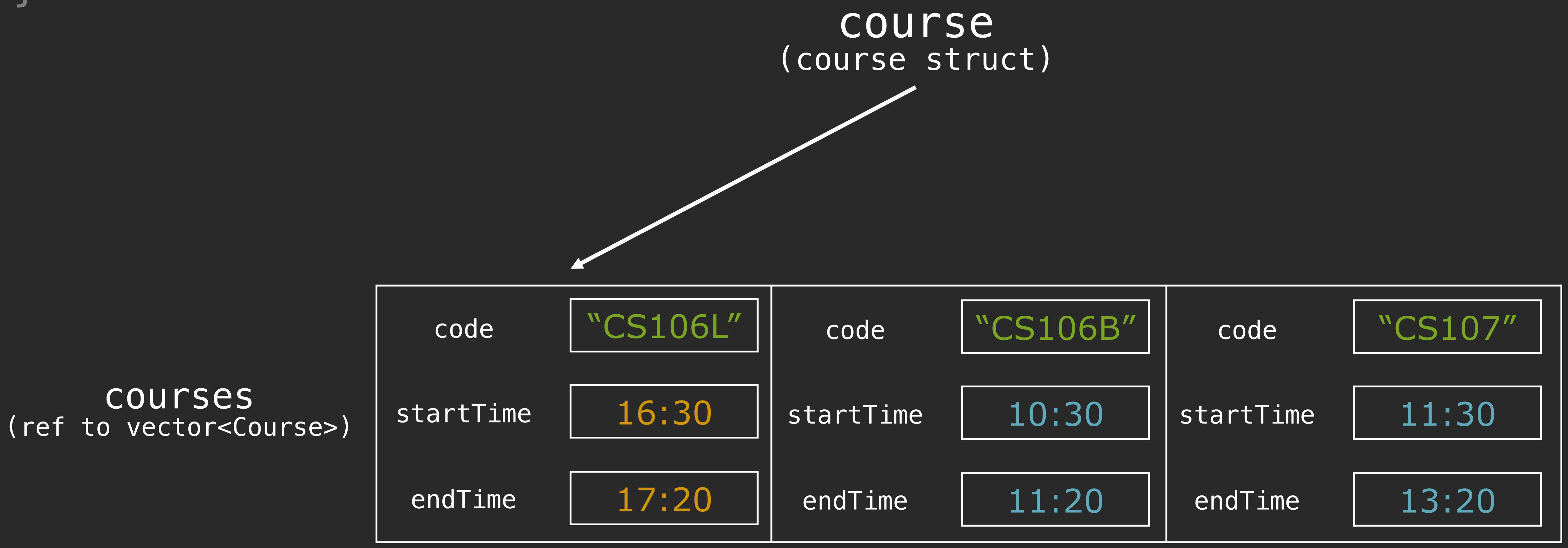
course
(course struct)



courses
(ref to vector<Course>)

code	"CS106L"	code	"CS106B"	code	"CS107"
startTime	15:30	startTime	10:30	startTime	11:30
endTime	16:20	endTime	11:20	endTime	13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

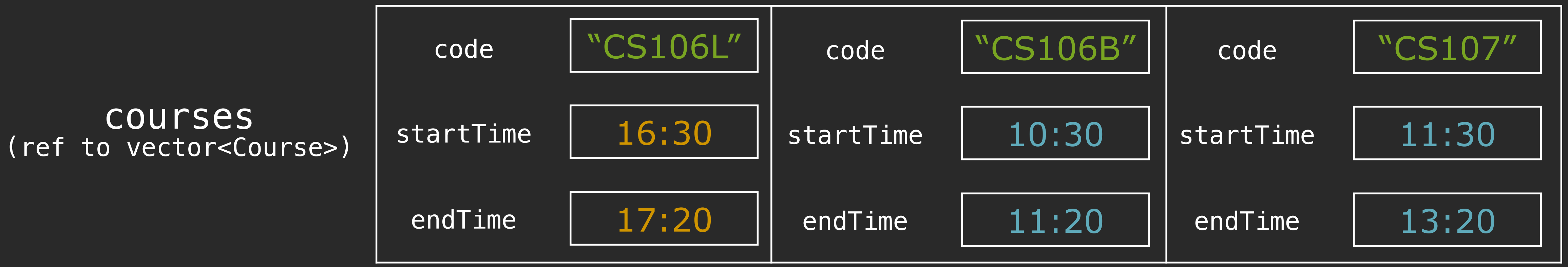


course
(course struct)

courses
(ref to vector<Course>)

code	“CS106L”	code	“CS106B”	code	“CS107”
startTime	16:30	startTime	10:30	startTime	11:30
endTime	17:20	endTime	11:20	endTime	13:20

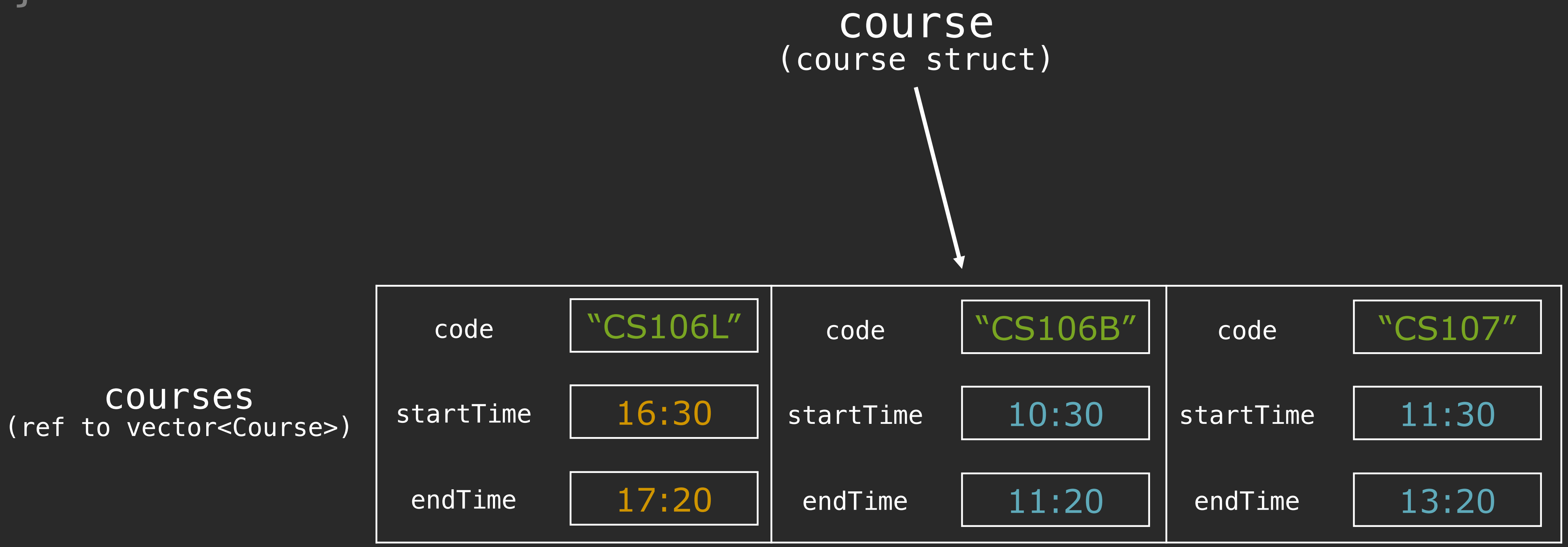
```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```



courses
(ref to vector<Course>)

code	"CS106L"	code	"CS106B"	code	"CS107"
startTime	16:30	startTime	10:30	startTime	11:30
endTime	17:20	endTime	11:20	endTime	13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

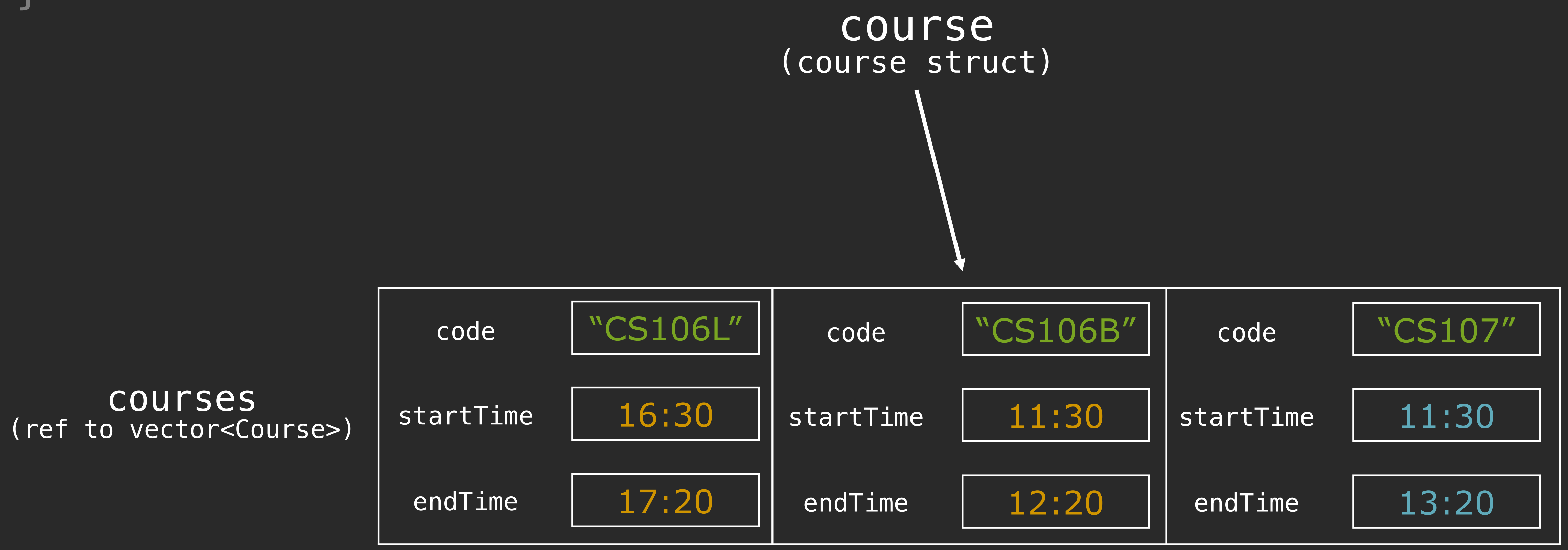


course
(course struct)

courses
(ref to vector<Course>)

code	"CS106L"	code	"CS106B"	code	"CS107"
startTime	16:30	startTime	10:30	startTime	11:30
endTime	17:20	endTime	11:20	endTime	13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

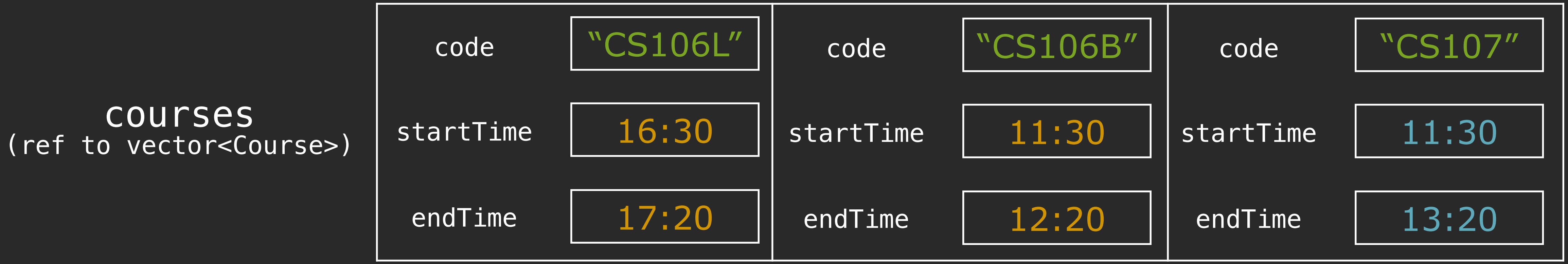


course
(course struct)

courses
(ref to vector<Course>)

code	“CS106L”	code	“CS106B”	code	“CS107”
startTime	16:30	startTime	11:30	startTime	11:30
endTime	17:20	endTime	12:20	endTime	13:20

```
void transformToDST(vector<Course>& courses) {
  for (auto course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```



courses
(ref to vector<Course>)

code	“CS106L”	code	“CS106B”	code	“CS107”
startTime	16:30	startTime	11:30	startTime	11:30
endTime	17:20	endTime	12:20	endTime	13:20

We want to go through all the Courses in the vector and add 1 to start and end time.

```
void transformToDST(vector<Course>& courses) {
  for (auto& course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

Remember: auto discards all qualifiers.

Can also use structured binding with references!

```
void transformToDST(vector<Course>& courses) {
  for (auto& [code, start, end, instructors] : courses) {
    start++;
    end++;
  }
}
```

In each iteration, unpacks each member as a reference.

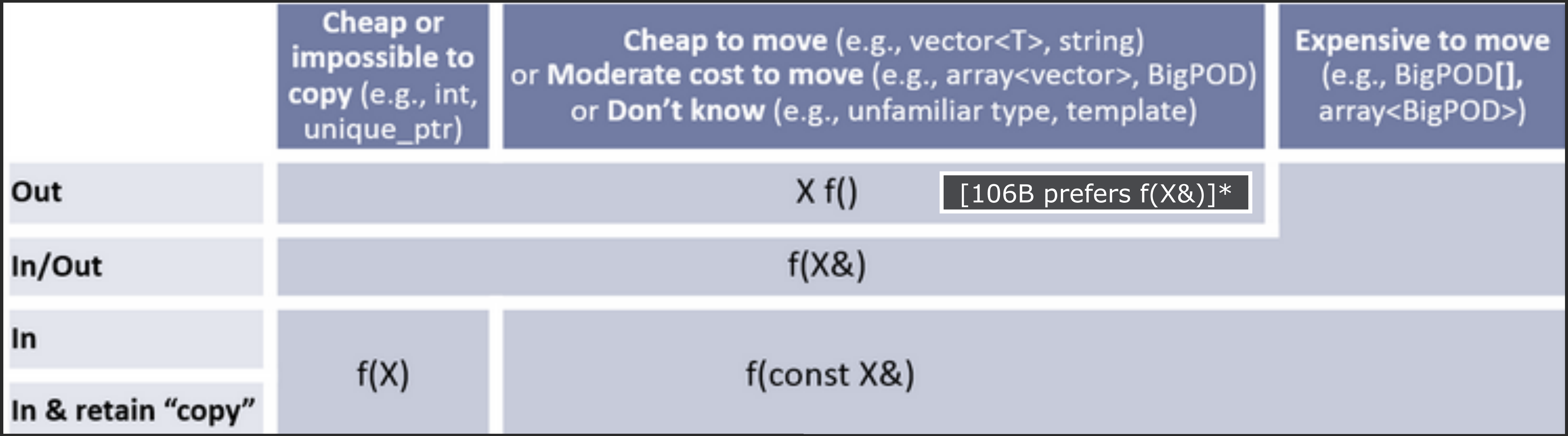
Review from 106B:
use const on large parameters.

```
void transformToDST(const vector<Course>& courses) {
  for (auto& course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

Use const on local variables that is not meant to change (especially references).

```cpp
void transformToDST(const vector<Course>& courses) {
  for (const auto& course : courses) {
    course.startTime.hour++;
    course.endTime.hour++;
  }
}
```

Parameters (in) and return values (out) guidelines for modern C++ code.

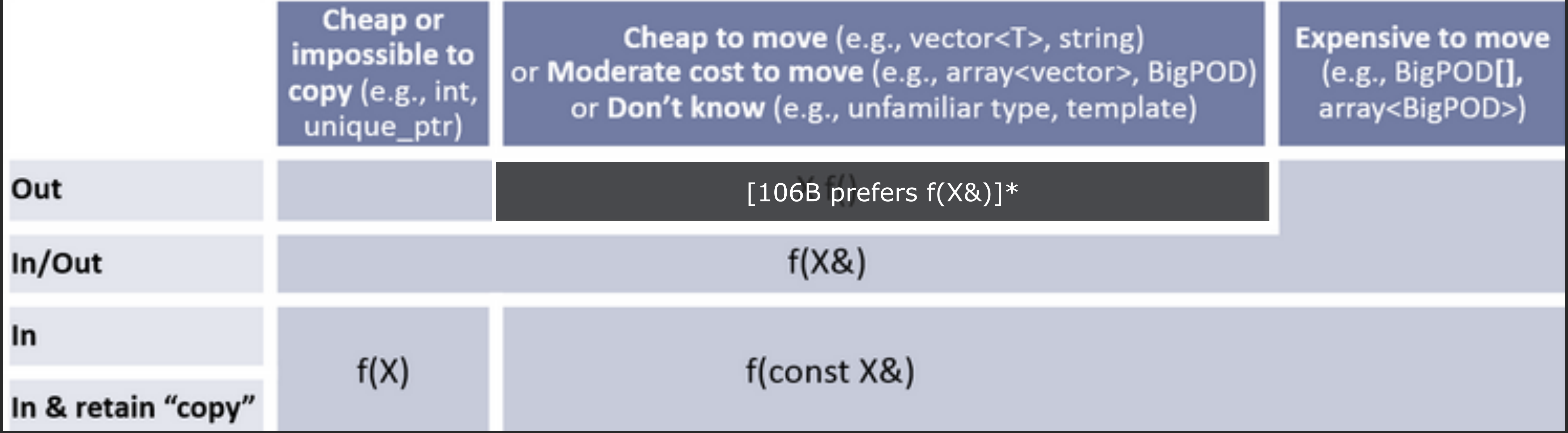


	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()	[106B prefers f(X&)]*	
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain "copy"			

Source: https://www.modernescpp.com/index.php/c-core-guidelines-how-to-pass-function-parameters

Reference parameter w/o const implies that it is an in/out parameter!

Parameters (in) and return values (out) guidelines for modern C++ code.

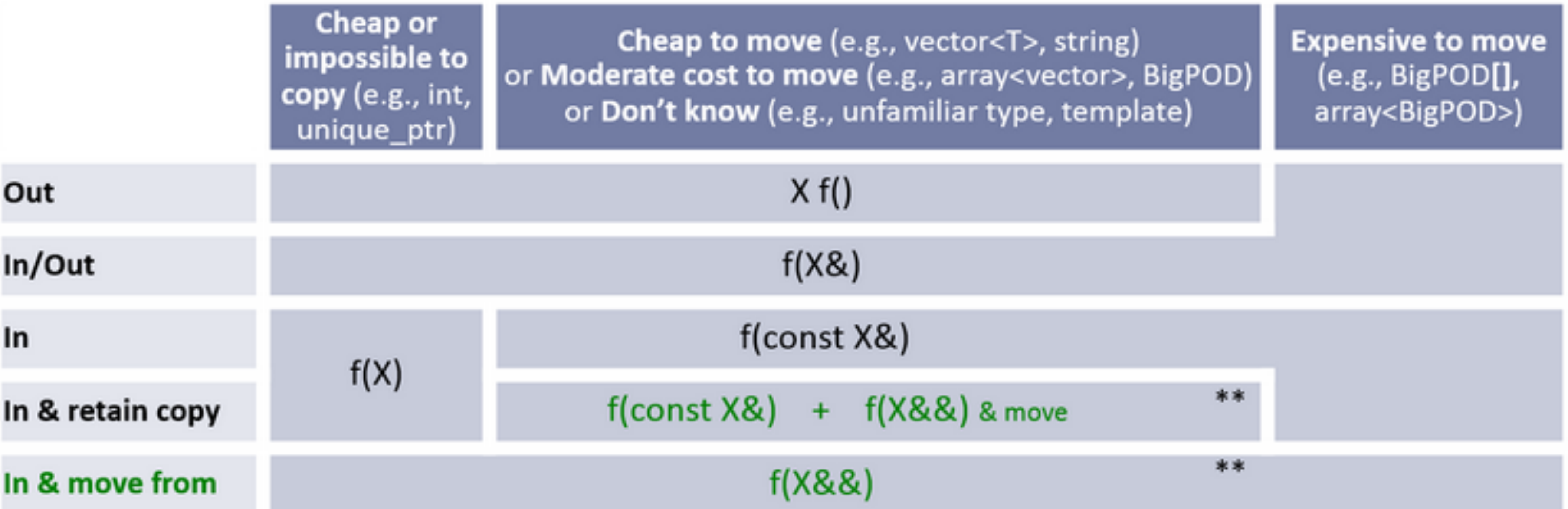


	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out		[106B prefers f(X&)]*	
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain "copy"			

Source: https://www.modernescpp.com/index.php/c-core-guidelines-how-to-pass-function-parameters

*Returning collections is fine with modern compilers (RVO), but not used in 106B.

Preview of week 8:
how to move instead of copy.

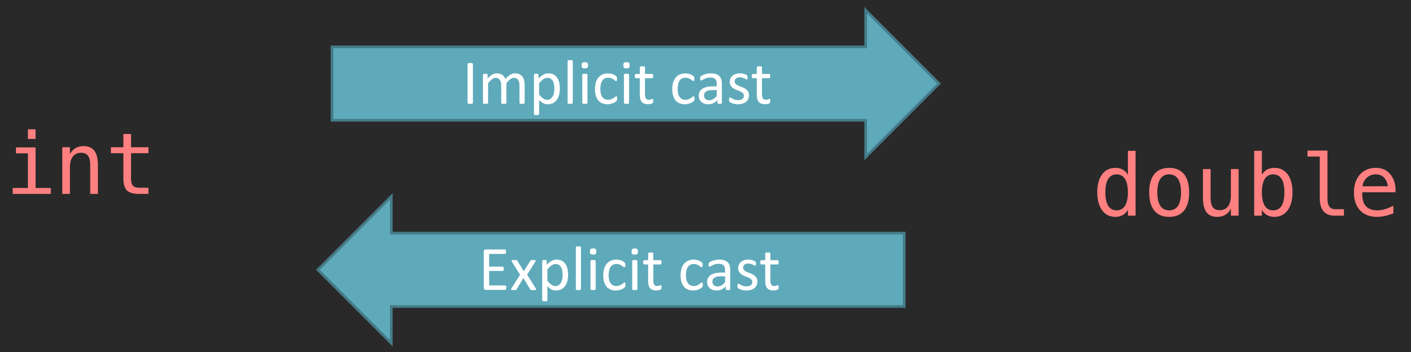


	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD**[]**, array<BigPOD>)
Out	X f()		f(X&)
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy		f(const X&) + f(X&&) & move **	
In & move from	f(X&&) **		

Source: https://www.modernescpp.com/index.php/c-core-guidelines-how-to-pass-function-parameters

Conversions have two directions.

```
int v1 = 3.4; // compiler warning
double v2 = 6; // OK
```



int

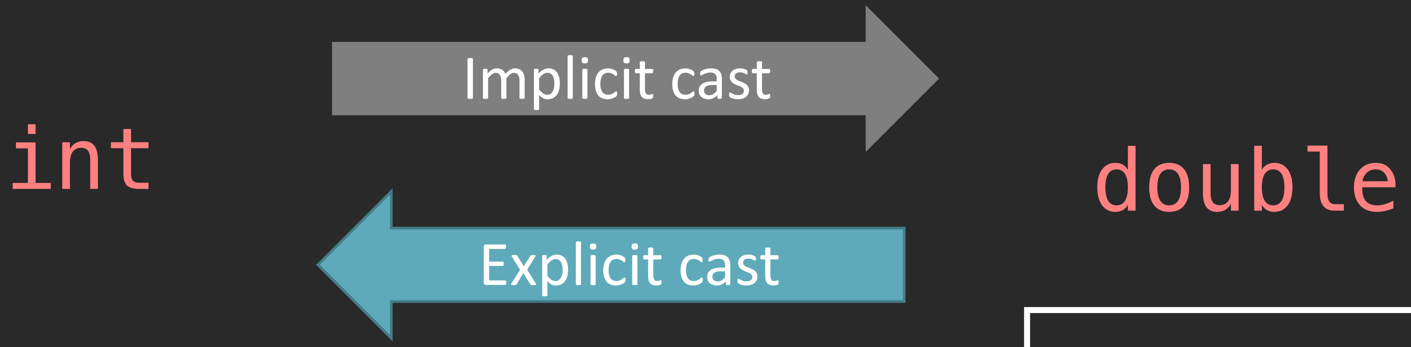
Implicit cast

Explicit cast

double

Conversions have two directions.

```
int v1 = 3.4; // compiler warning
double v2 = 6; // OK
```



int

Implicit cast

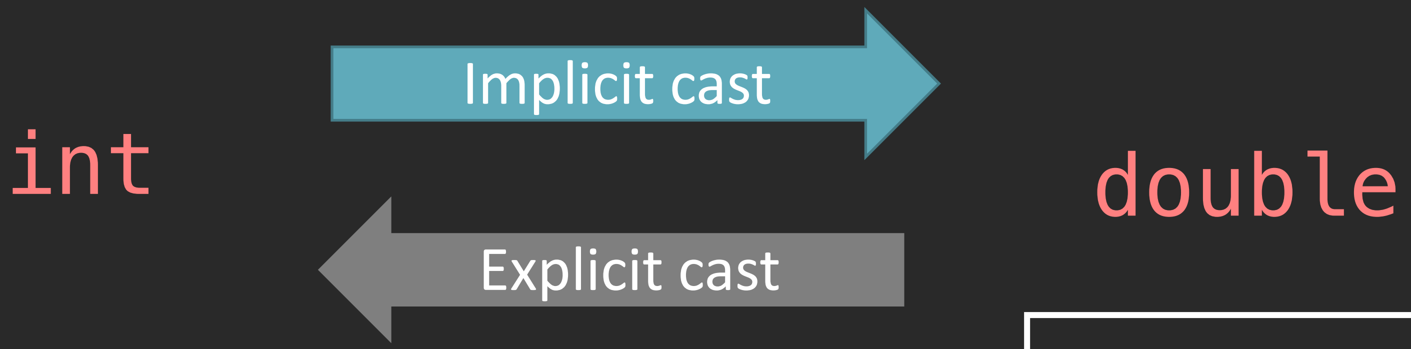
Explicit cast

double

Also known as promotion, will automatically be done for you.

Conversions have two directions.

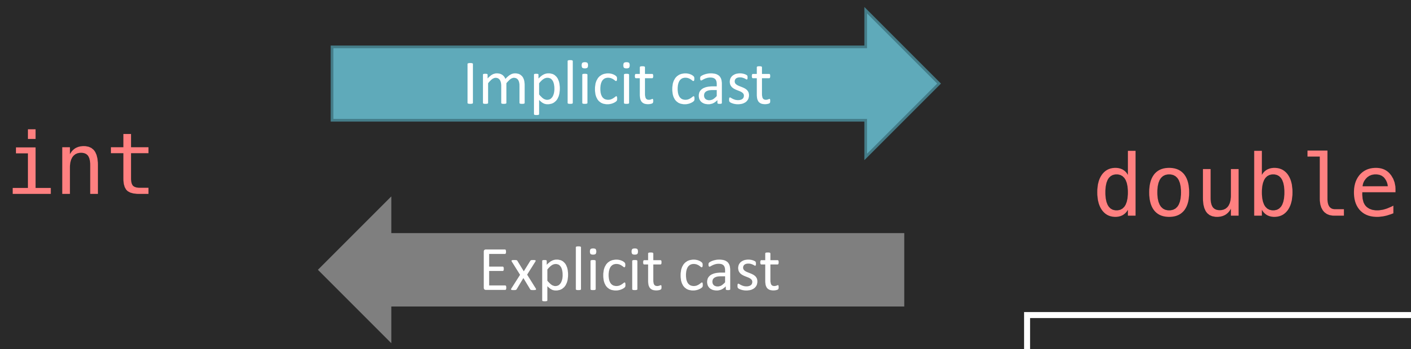
```
int v1 = 3.4; // compiler warning
double v2 = 6; // OK
```



Also known as coercion, requires a cast.

Conversions have two directions.

```
int v1 = static_cast<int> (3.4);
double v2 = 6; // OK
```



int

Implicit cast

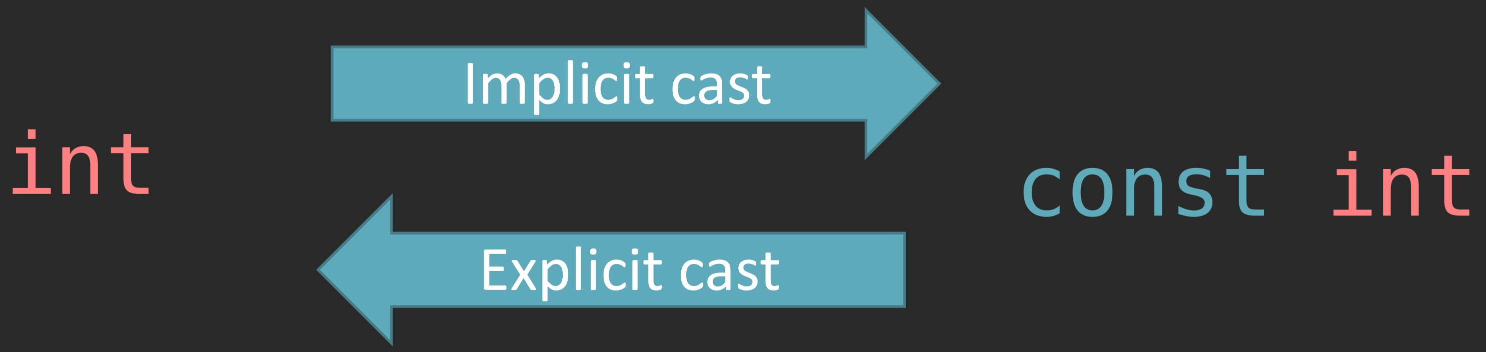
Explicit cast

double

Also known as coercion, requires a cast.

Conversions have two directions.

```
const int v1 = 3;
int v2 = v1; // compiler warning
```



int

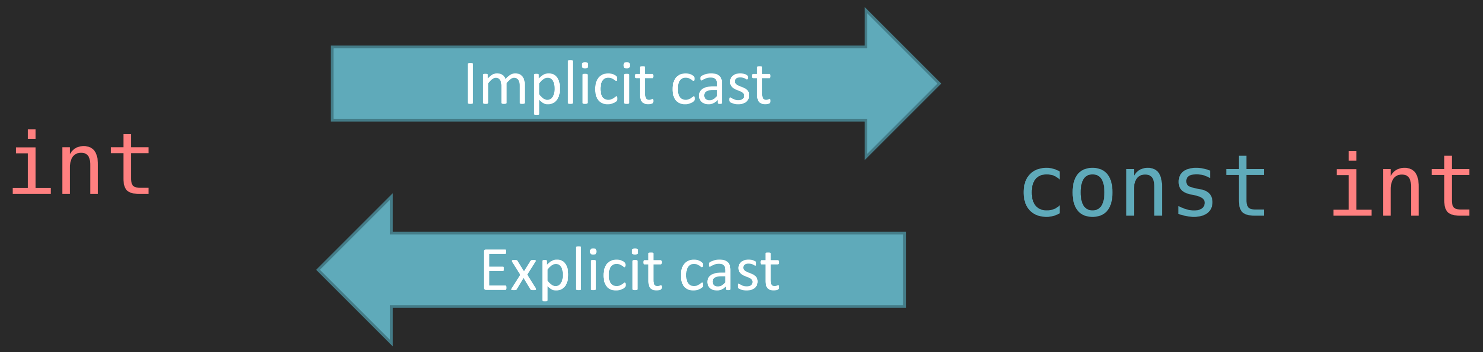
Implicit cast

Explicit cast

const int

Conversions have two directions.

```
const int v1 = 3;
int v2 = const_cast<int> (v1);
```



int

Implicit cast

Explicit cast

const int

initialization

In C++, depending on the type, there were too many ways to initialize a variable.

To solve this, C++11 (ironically) adds one more way: uniform initialization.

```
int main() {
  vector<int> vec{3, 1, 4, 1, 5, 9};
  Course now {“CS106L”, {15, 30}, {16, 30},
           {“Wang”, “Zeng”} };

}
```

We don’t have to specify the types – automatically deduced.

The return value can also be uniform initialized.

```
pair<int, int> findPriceRange(int dist) {
  int min = static_cast<int>(dist * 0.08 + 100);
  int max = static_cast<int>(dist * 0.36 + 750);
  return {min, max};
}

int main() {
  int dist = 6452;
  auto [min, max] = findPriceRange(dist);
  cout << “You can find prices between: ”
       << min << “ and “ << max;
}
```

A initializer list is a lightweight vector that can be used as a parameter.

```
vector::vector(initializer_list<T> init);
```

Constructor creates a vector with initial elements.

```
vector<int> vec{3, 1, 4, 1, 5, 9};
```

Using the uniform initialization syntax, the initializer list ctor is preferred over constructor.

```
int main() {
  vector<int> vec1{3}; // vector = {3}
  vector<int> vec2(3); // vector = {0, 0, 0}
}
```

First one calls ctor with initializer list, second calls constructor with one parameter.

enumerations

An enumeration is a type that is restricted to certain named constants.

```
enum class DayOfWeek {kMonday, kTuesday, [omitted], kSunday};

enum class TeaType {kGreen, kBlack, kOolong};
```

Enumerations are type safe and make your code more self-documenting.

```
DayOfWeek today = kThursday;

TeaType refreshment = kOolong;
```

You can use switch statements with enumerations!

```
DayOfWeek today = getDayToday();
switch(today) {
  case kTuesday:
  case kThursday:
    cout << "Yay 106L" << endl;
    break;
  case kFriday:
    cout << "Yay section!" << endl;
    break;
  default:
    cout << "So excited for 106L!" << endl;
}
```

file streams

Review: read file line by line

```
ifstream file(filename); // open
string line;

while (getline(file, line)) {
  // do something with line
}

file.close(); // technically don’t need this
```

Review: read file token by token

```
ifstream file(filename); // open
string token;

while (file >> token) {
  // do something with line
}

file.close(); // technically don’t need this
```

Example

implementing Axess, take 2

manipulators + overloading >> or <<

There are some keywords that will change the behavior of the stream when inserted.

`endl`	insert newline and flush stream
`ws`	skips all whitespace until it finds another char
`boolalpha`	prints “true” and “false for bools.

`hex`	prints numbers in hexadecimal
`setpercision`	adjusts the precision of printed numbers

We can use manipulators to pad the output.

```
cout << ”[“ << setw(10) << “Ito” << “]”;
```

```
Output: [        Hi]
```

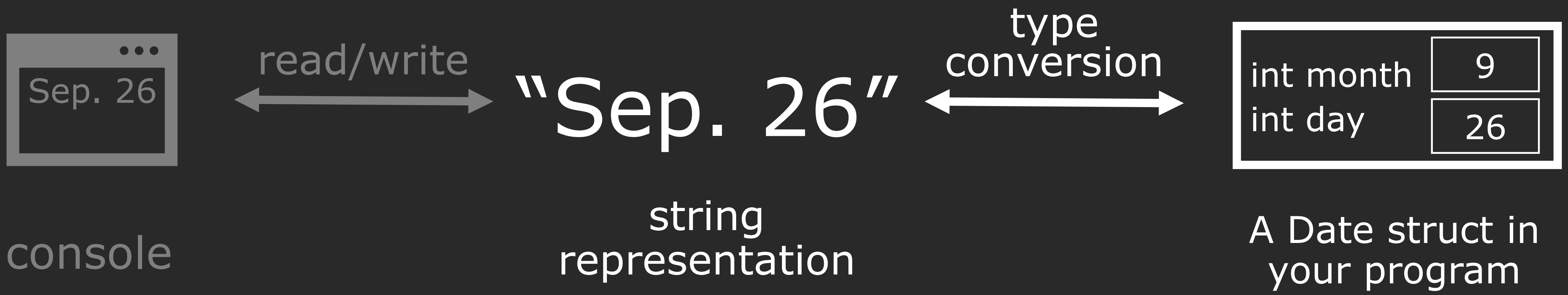
```
cout << ”[“ << left << setw(10) << “Ito” << “]”;
```

```
Output: [Hi        ]
```

```
cout << ”[“ << left << setfill(‘-’) << setw(10) << “Ito” << “]”;
```

```
Output: [Hi--------]
```

C++ does not know how to convert a custom structure to/from a string.



Sep. 26

console

read/write

“Sep. 26”

string representation

type conversion

int month 9

int day 26

A Date struct in your program

You have to specify how the conversion to/from a string occurs.

```
ostream& operator<<(ostream& os, const Time& time) {
  os << time.hour << ":"
     << setfill('0') << setw(2) << time.minute;
  return os;
}
```

Other programming languages also allow you to turn an object into a string.

```
// java
String toString() {
    // create string from this object
}


// python
def __str__(self):
    // create string from self
```

You have to specify how the conversion to/from a string occurs.

```
ostream& operator<<(ostream& os, const Time& time) {
  os << time.hour << ":"
     << setfill('0') << setw(2) << time.minute;
  return os;
}
```

```
[ostream object] << [Time struct];
```

The header literally means:
how should this be interpreted.

Make sure to respect the return value of the << operator.

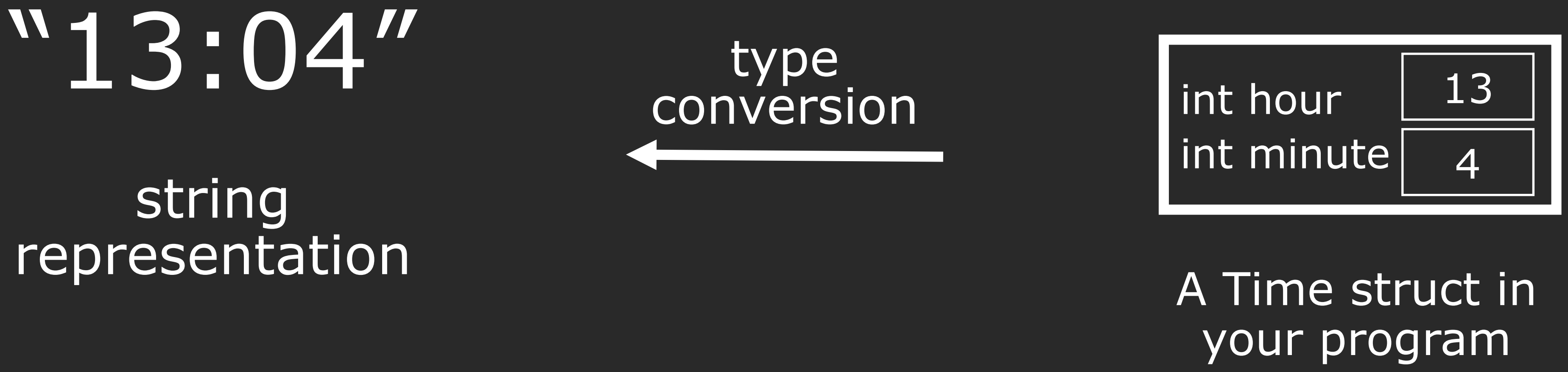
```
ostream& operator<<(ostream& os, const Time& time) {
  os << time.hour << ":"
     << setfill('0') << setw(2) << time.minute;
  return os;
}
```

```
[ostream] << [Time] << [Time];
```

The return value is what allows us to chain the <<‘s.

You have to specify how the conversion to/from a string occurs.

```
ostream& operator<<(ostream& os, const Time& time) {
  os << time.hour << ":"
     << setfill('0') << setw(2) << time.minute;
  return os;
}
```



“13:04”

string representation

type conversion

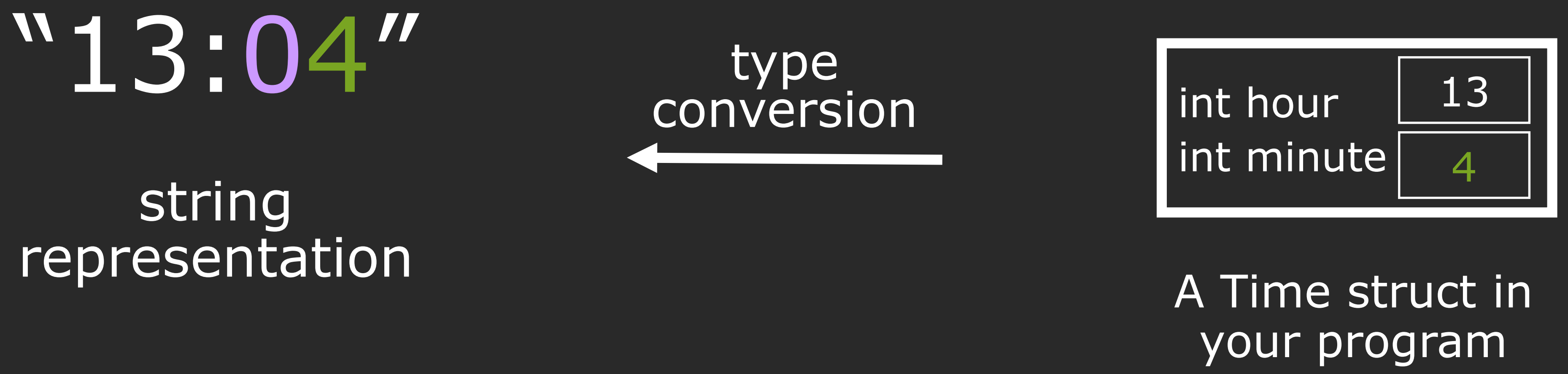
int hour 13

int minute 4

A Time struct in your program

Manipulators can help you format the string exactly as you want it.

```
ostream& operator<<(ostream& os, const Time& time) {
  os << time.hour << ":"
     << setfill('0') << setw(2) << time.minute;
  return os;
}
```



“13:04”

string representation

type conversion

int hour 13

int minute 4

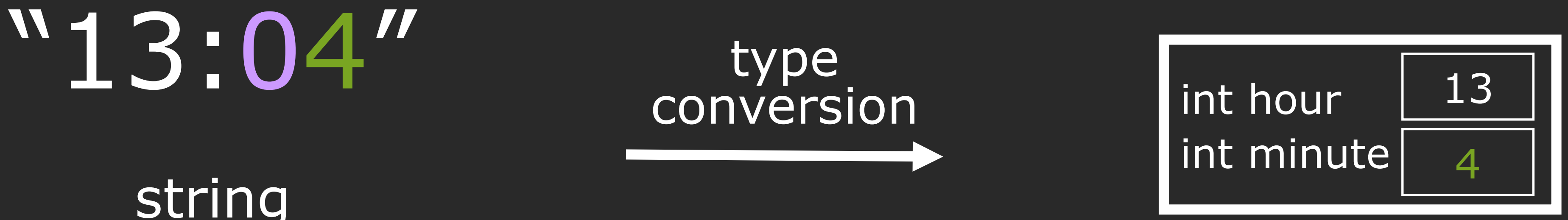
A Time struct in your program

Example

implementing Axess, stream >> overload

Reading input is harder because you have to worry about error-checking.

```
istream& operator>>(istream& is, Time& time) {
  ???
  return is;
}
```



string representation

A Time struct in your program

Be consistent with other input stream operations!

1. If the fail bit is on, do not do anything.

2. You can only read one token, nothing more, nothing less. (probably a good idea to copy the token into a stringstream).

3. If the operation failed, set the fail bit. Make sure the original stream and object are unchanged!

Be consistent with other input stream operations!

1. If the fail bit is on, do not do anything.

```
istream& operator>>(istream& is, Time& time) {
  if (!is) return is; // if fail do nothing

  // continued

}
```

Be consistent with other input stream operations!

2. You can only read one token, nothing more, nothing less. (probably a good idea to copy the token into a stringstream).

```
string timeString;
if (!(is >> timeString)) { // read exactly one token
    is.setstate(ios::failbit);
    return is;
}
istringstream ss(timeString);
// use ss;
```

Be consistent with other input stream operations!

3. If the operation failed, set the fail bit. Make sure the original stream and object are unchanged!

```
int hour, minute; char colon;
if (/* parse hour, minute, colon using >> */) {
   time = Time{hour, minute};
} else { // we didn’t change time
   is.setstate(ios::failbit);
}
```

Example

implementing Axess, take 2

Next time

STL Sequence Containers and Iterators