Types and Streams II

CS 106L Winter 2019 – Avery Wang and Anna Zeng

Game Plan

- input/output streams
- modern C++ types
- implementing simpio
- file streams

Recap

Recap: stringstreams

```
// buffer contains ”Ito En Green Tea “, pos at end
ostringstream oss("Ito En Green Tea “, stringstream::ate);

// str function returns characters in buffer as a string
cout << oss.str() << endl;

// Converts 16.9 to string and insert into buffer
oss << 16.9 << “ Ounce ”;

// prints ”Ito En Green Tea 16.9 Ounce “
cout << oss.str() << endl;
```

Our final solution to converting strings to integers.

```
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!

Recap: reading from the console

console

read →

"3.14"

string representation

type conversion →

3.14

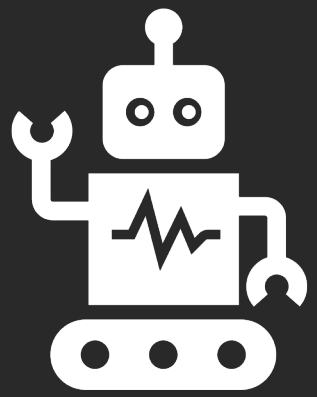
double in your program

Our first attempt at writing getInteger().

```
int getInteger(const string& prompt) {
    cout << prompt;

    int result;
    cin >> result; // what if this fails?

    return result;
}
```



Example

input streams, buffering, and waiting for user input.

cin

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

name
(string)

???

response
(string)

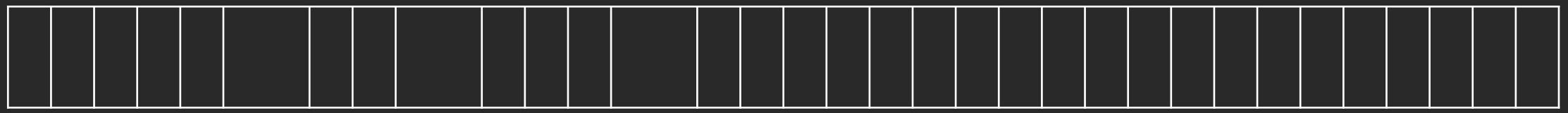
???

age
(int)

???

The lecture code included cout statements.

cin



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

name
(string)

???

response
(string)

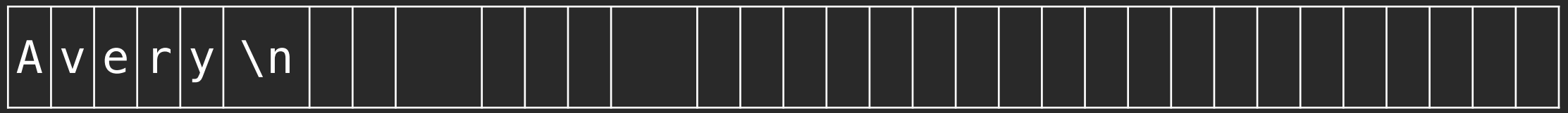
???

age
(int)

???

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

cin



Avery

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

name
(string)

???

response
(string)

???

age
(int)

???

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

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.


```
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.


```
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.


```
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.


```
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.


```
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!)


```
Avery
20
Avery20
```

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

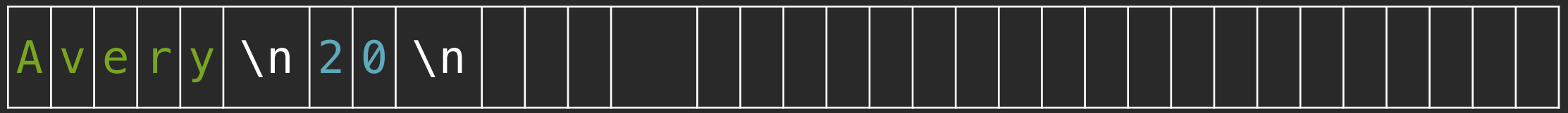
name (string): "Avery"

response (string): ???

age (int): 20

But attempting reading again will flush cout.

cin




```
Avery
20
Avery20|
```

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

name
(string)

"Avery"

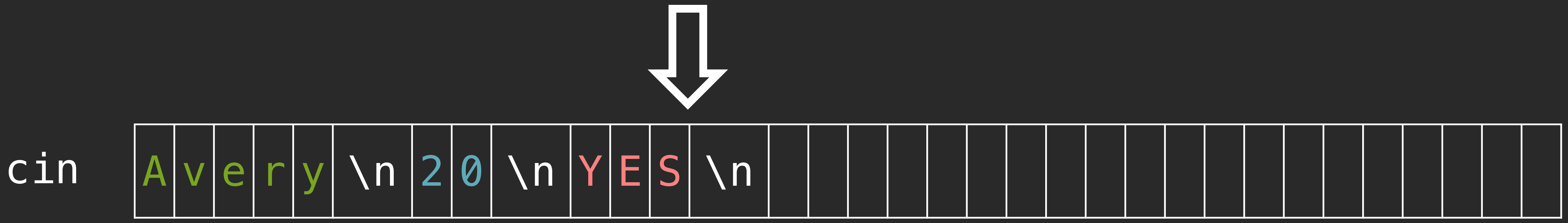
response
(string)

???

age
(int)

20

We prompt the user again.



cin

A	v	e	r	y	\n	2	0	\n	Y	E	S	\n																			

G F E B

```
Avery
20
Avery20YES
```

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

name (string): “Avery”

response (string): “YES”

age (int): 20

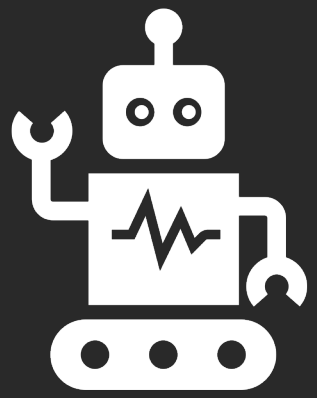
We type something, it’s transferred to the buffer, and read into the variable.

Key Takeways

- When does the program prompt the user for input?
- Why does the cout operation not immediately print the output onto the console? When is the output printed?
- Does the position pointer skip whitespace before the token or after the token with each >> operation? (this is important!)
- Does the position pointer always read up to a whitespace? If not, come up with a counterexample.

Key Takeways

- The program hangs and waits for user input when the position reaches EOF, past the last token in the buffer.
- All input operations will flush cout.
- The position pointer does the following:
  - consume all whitespaces (spaces, newlines, etc.)
  - reads as many characters until:
    - a whitespace is reached, or…
    - for primitives, the maximum number of bytes necessary to form a valid variable.
    - example: if we extract an int from “86.2”, we’ll get 86, with pos at the decimal point.



Example

when input streams go wrong

cin

```
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)

???

response
(string)

???

age
(int)

???

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

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.

Avery Wang

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

name (string): "Avery"

response (string): ???

age (int): ???

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

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.

Avery Wang

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

name (string): "Avery"

response (string): ???

age (int): ???

The fail bit is turned on.


```
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!)


```
Avery Wang
Avery-2736262
```

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

name (string): "Avery"

response (string): ???

age (int): ???

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

3 reason why >> with cin is a nightmare.

1. cin reads the entire line into the buffer but extracts 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.

implementing simpio

Our first attempt at writing getInteger().

```
int getInteger(const string& prompt) {
    cout << prompt;

    int result;
    cin >> result; // what if this fails?

    return result;
}
```

Drawing inspiration from our previous function?

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

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

    return result;
}
```

Our second attempt at writing getInteger().

```
int getInteger(const string& prompt) {
  cout << prompt;
  string token;
  cin >> token; // still a problem
  istringstream iss(token);

  int result; char trash;
  if (!(iss >> result) || iss >> trash)
    return getInteger(prompt); // bad recursion

  return result;
}
```

Our second attempt without recursion.

```
int getInteger(const string& prompt) {
  while (true) {
    cout << prompt;
    string token;
    cin >> token; // problem: only one token
    istringstream iss(token);

    int result; char trash;
    if (iss >> result && !(iss >> trash))
      return result;
  }
}
```

What is getline?

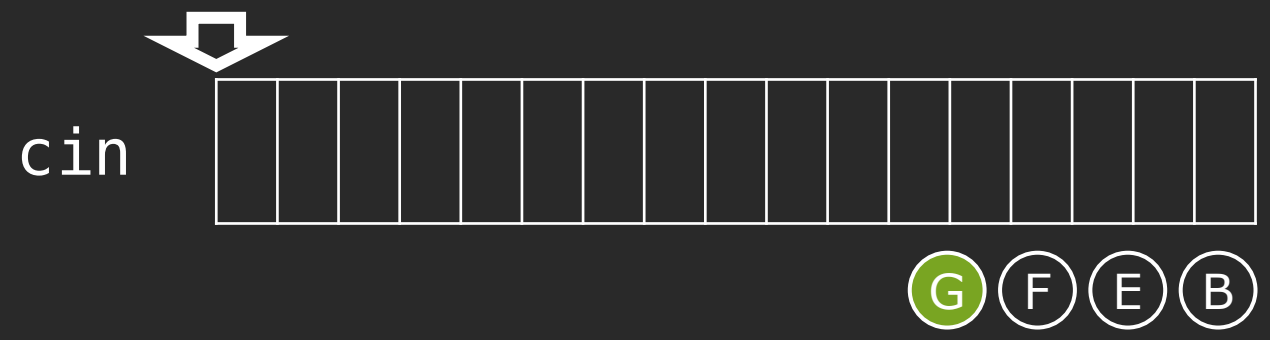
- Covered in CS 106B (probably)?
- Reads up to the next delimiter (by default '\n') and consumes it the delimiter. Position is now *past* the delimiter.
- Returns the "line" without the whitespace.

- Always use getline with cin instead of >> (why?)
- Always check the return value of getline (why?)

Third attempt: use getline to read from cin so there is nothing remaining in the buffer.

```
int getInteger(const string& prompt) {
  while (true) {
    cout << prompt;
    string line;
    if (!getline(cin, line)) // when does this fail?
      throw domain_error(…);
    istringstream iss(line);

    int result; char trash;
    if (iss >> result && !(iss >> trash))
      return result;
  }
}
```

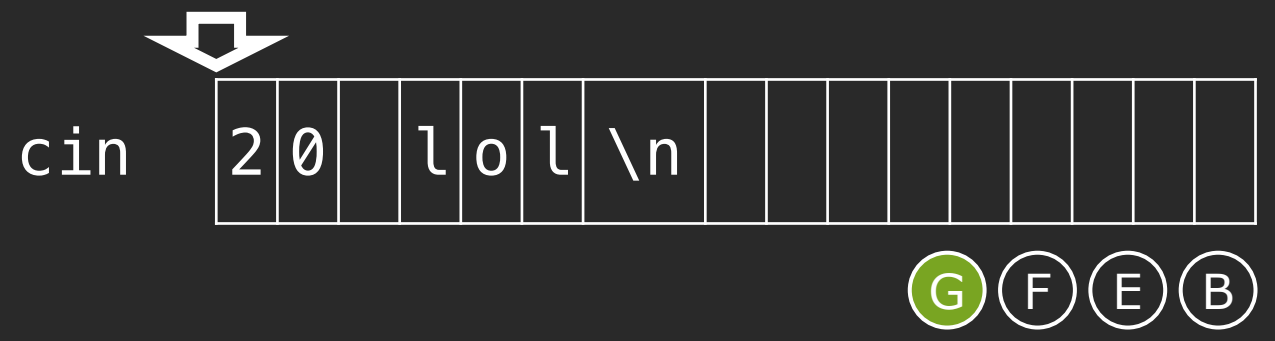



```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

result (int) ???

trash (char) ???

Let’s try running through the code!




```
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 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;
  }
}
```

line
(string)

"20 lol"

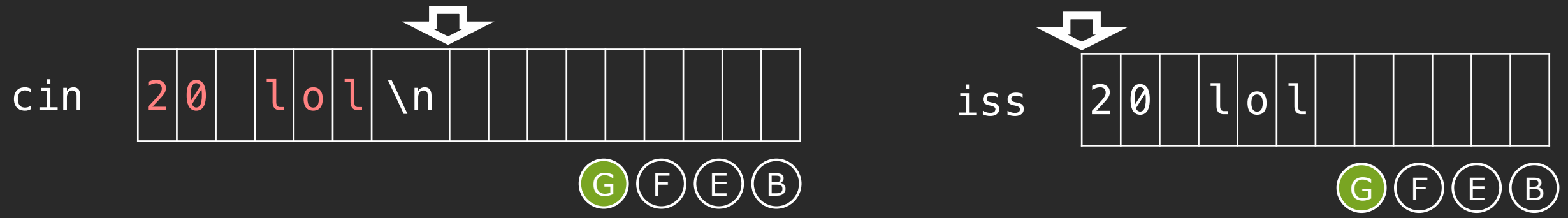
Try reading a line.
Since the buffer is empty, it prompts the user.


```
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;
  }
}
```

line
(string)

"20 lol"

Create an istringstream using that line.

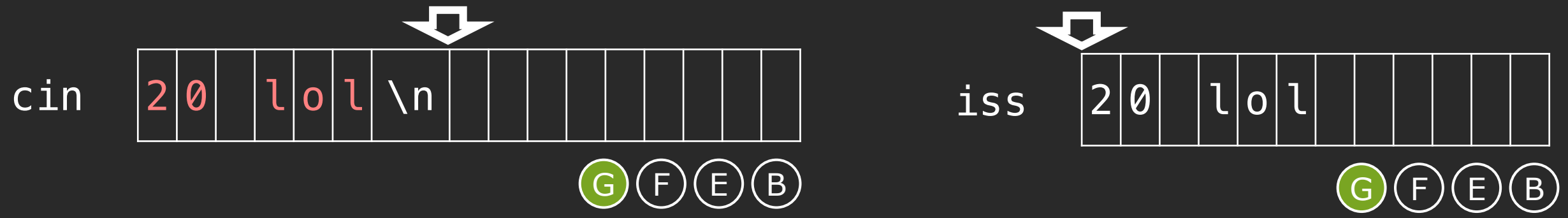



```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

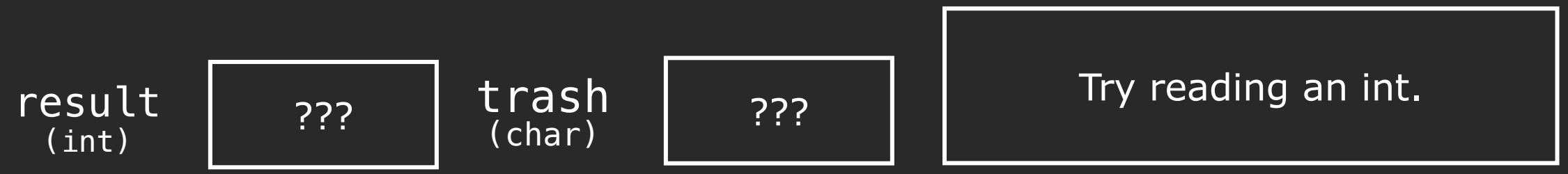
line
(string)

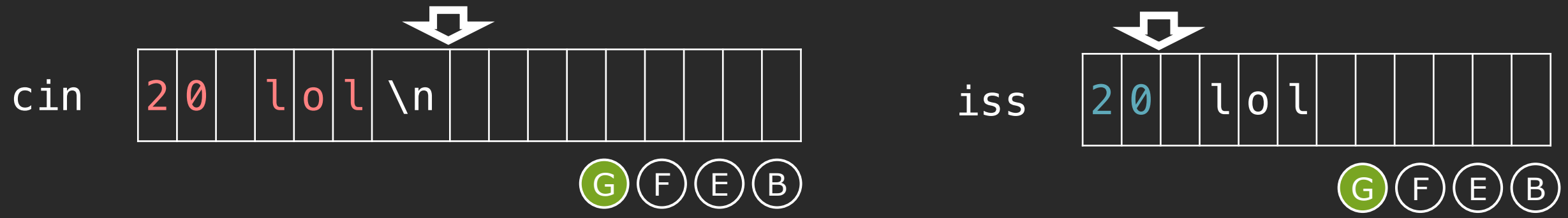
"20 lol"

Create an istringstream using that line.



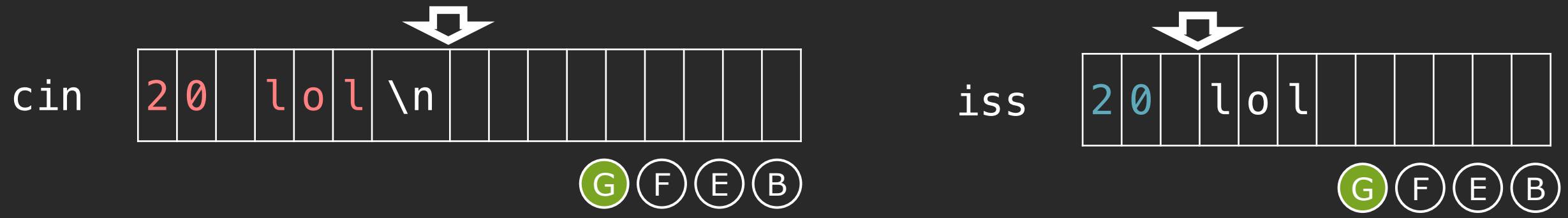

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```






```
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;
  }
}
```

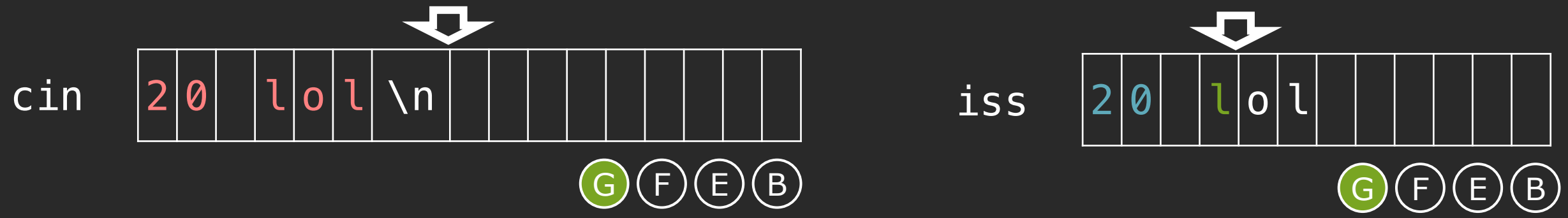
result (int)	20	trash (char)	???	Try reading an int.




```
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;
  }
}
```

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

Now try reading in any trash.

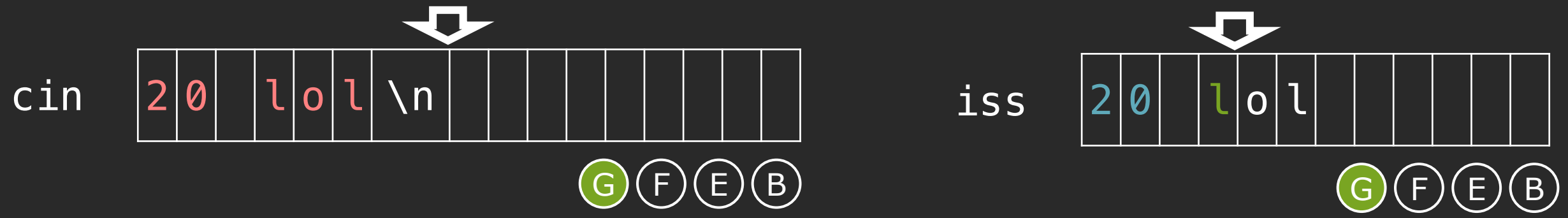



```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

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;
  }
}
```

result (int)	20	trash (char)	'l'

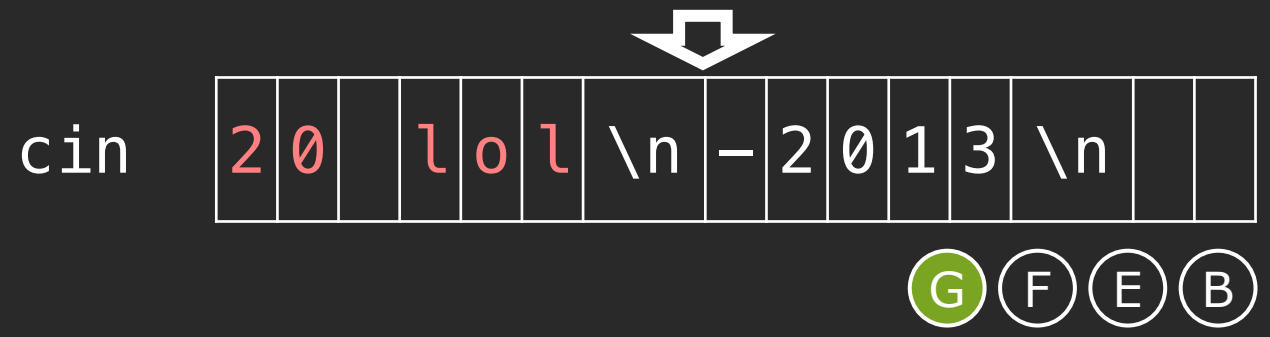
That succeeded, which is not what we wanted.


```
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;
  }
}
```

line
(string)

???

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

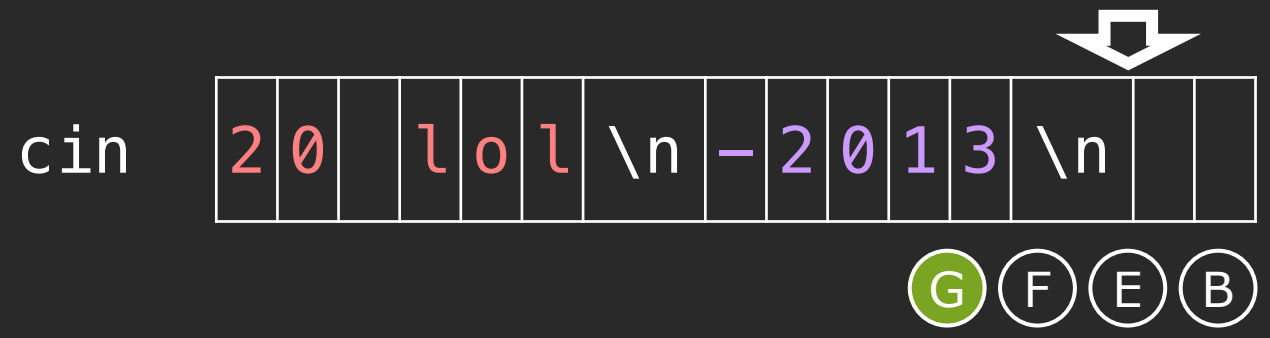



```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

line
(string)

???

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

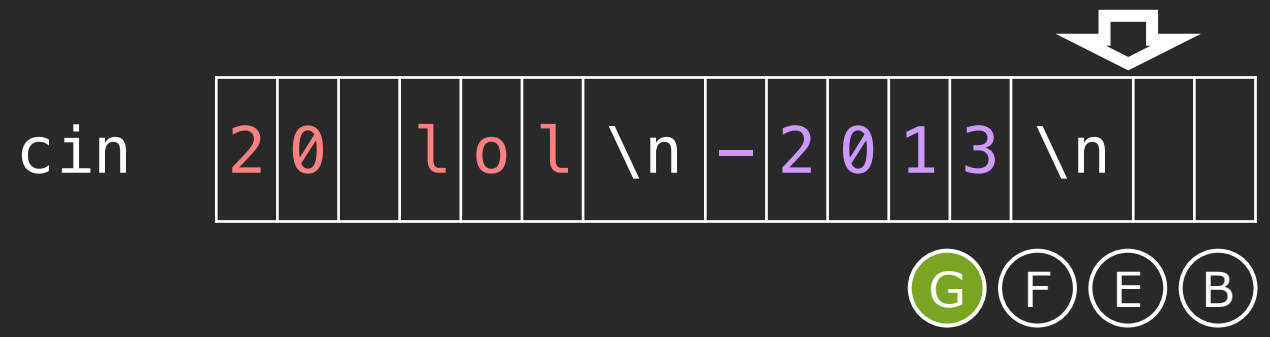



```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

line
(string)

"-2013"

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

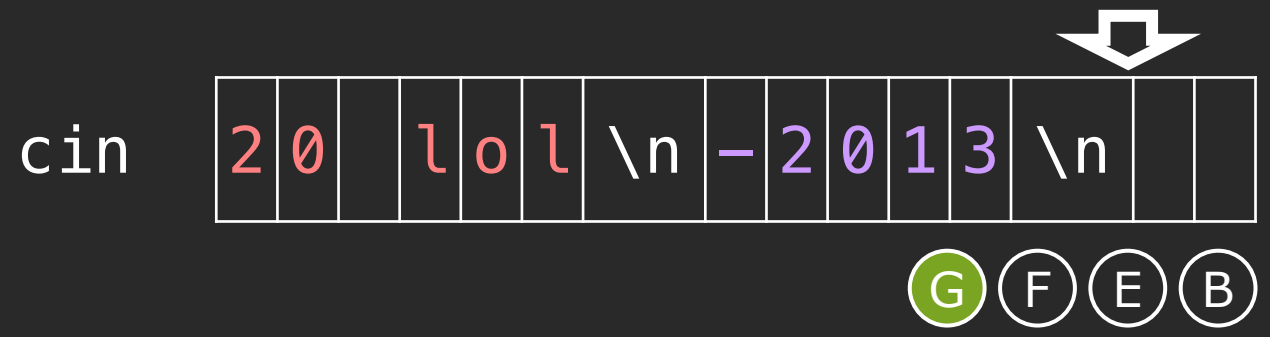


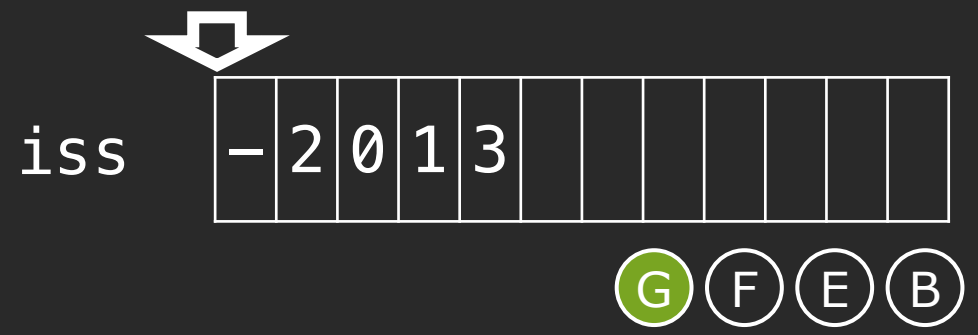

```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

line
(string)

"-2013"

Create a separate stringstream with the line we just read.



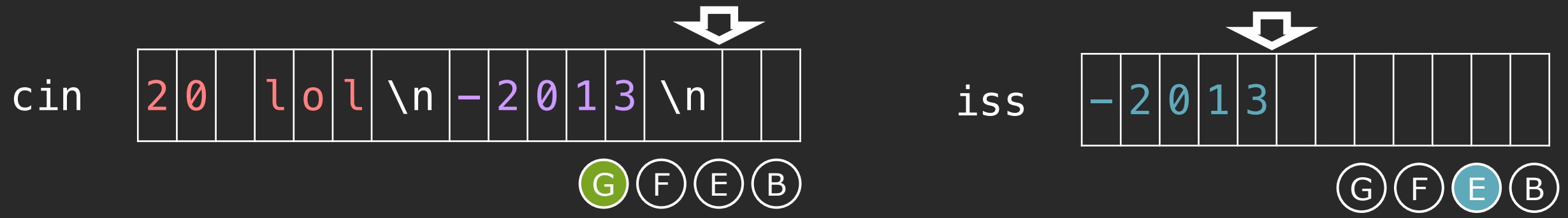



```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

result (int) ???

trash (char) ???

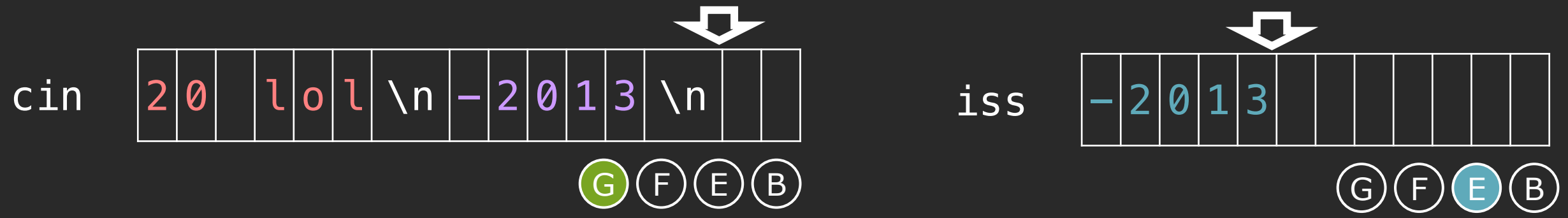
Try reading an int from the stringstream.




```
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)	???

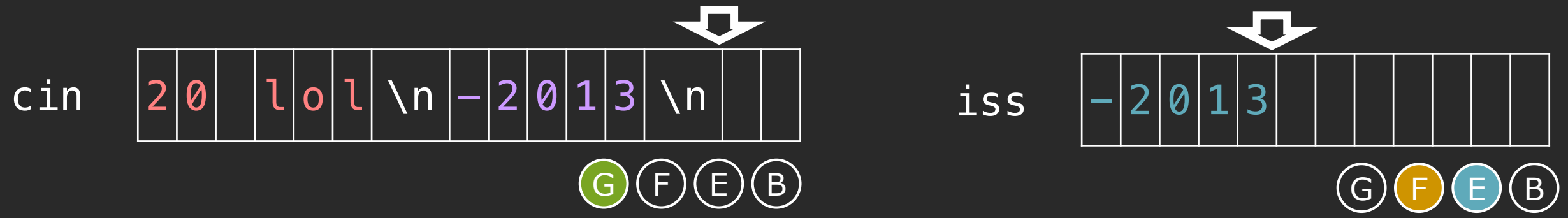
Try reading an int from the stringstream.




```
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 remainding characters from the buffer.

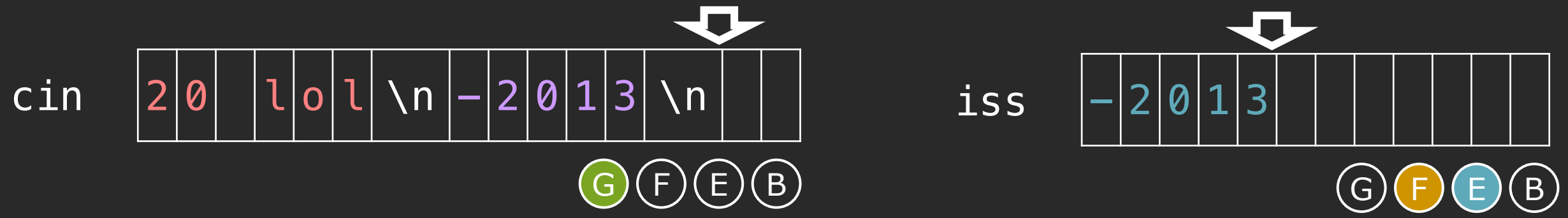



```
int getInteger() {
  while (true) {
    string line; int result; char trash;
    if (!getline(cin, line)) throw domain_error(…);
    istringstream iss(line);
    if (iss >> result && !(iss >> trash)) return result;
  }
}
```

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.

Full implementation with prompting!

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

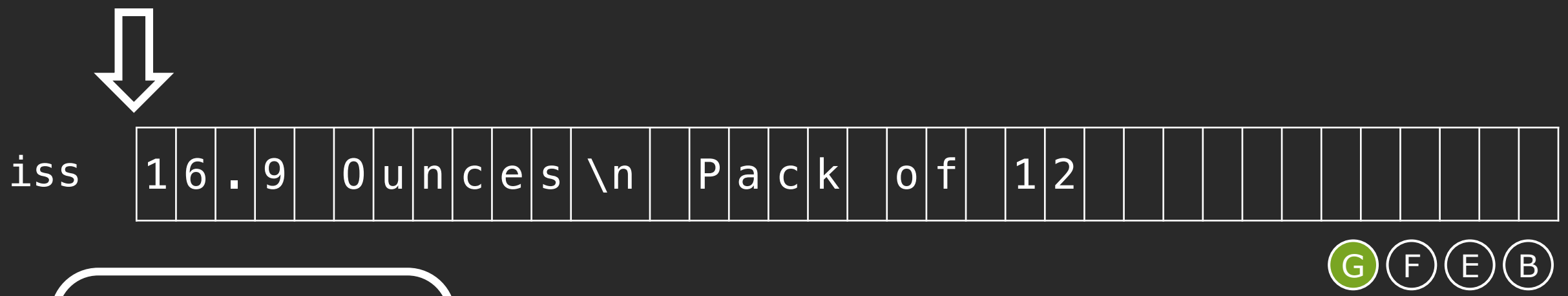
Final stream gotcha!

```
istringstream iss("16.9 Ounces\n Pack of 12");
double amount; string unit;

iss >> amount;

getline(iss, unit);

cout << "amount: " << amount << endl;
cout << "unit: " << unit << endl;
```




```
l
```

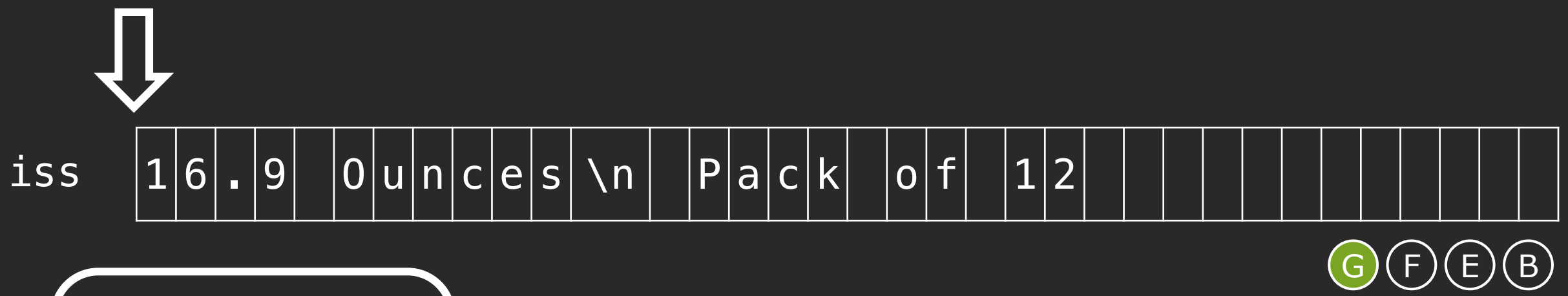
```
iss >> amount;

getline(iss, unit);
```

amount (double) ???

unit (string) ???

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




```
l
```

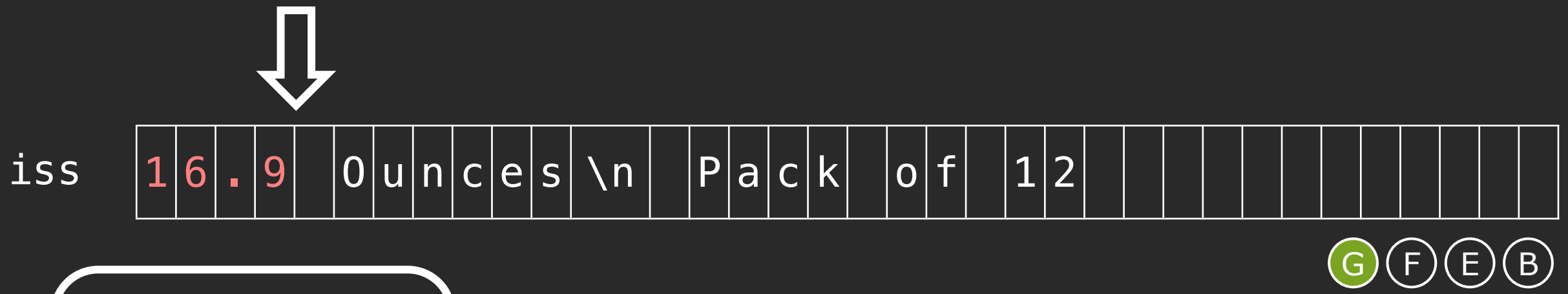
```
iss >> amount;

getline(iss, unit);
```

amount (double) ???

unit (string) ???

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



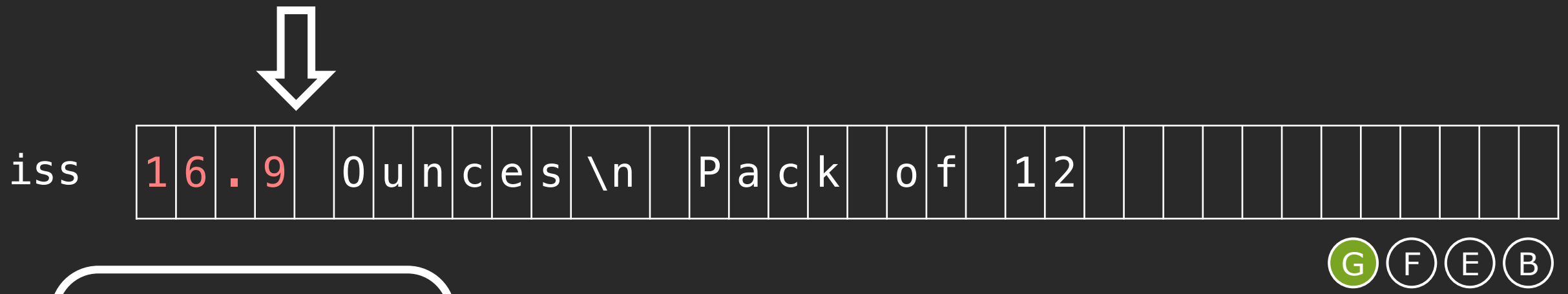

```
l
```

```
iss >> amount;

getline(iss, unit);
```

amount (double)	16.9
unit (string)	???

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

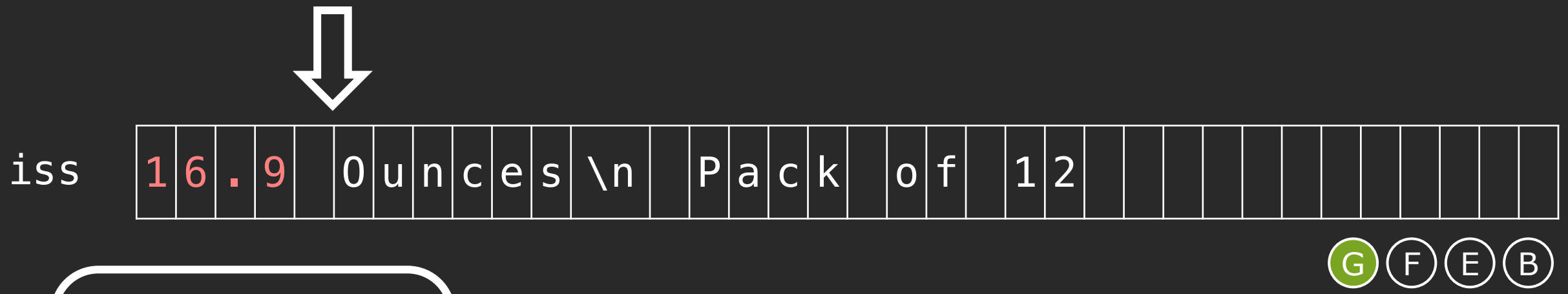



```
iss >> amount;

getline(iss, unit);
```

amount (double)	16.9
unit (string)	???

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

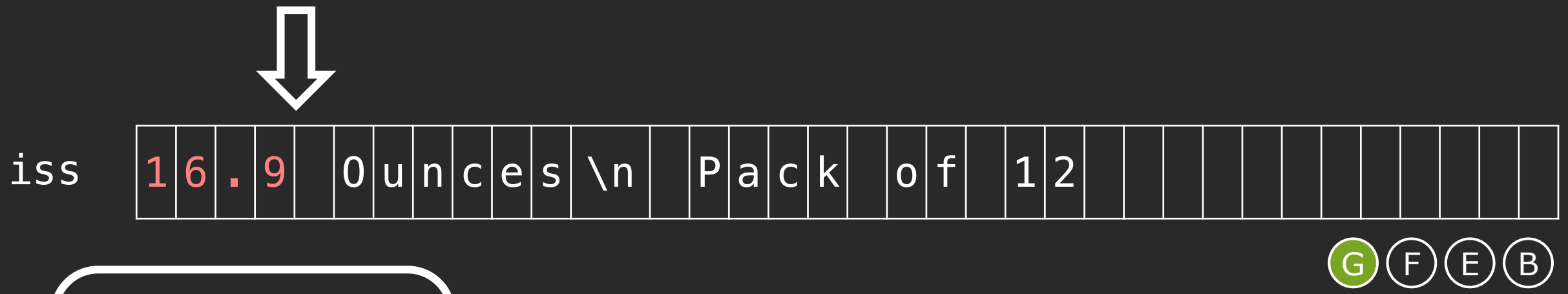



```
iss >> amount;

getline(iss, unit);
```

amount (double)	16.9
unit (string)	""

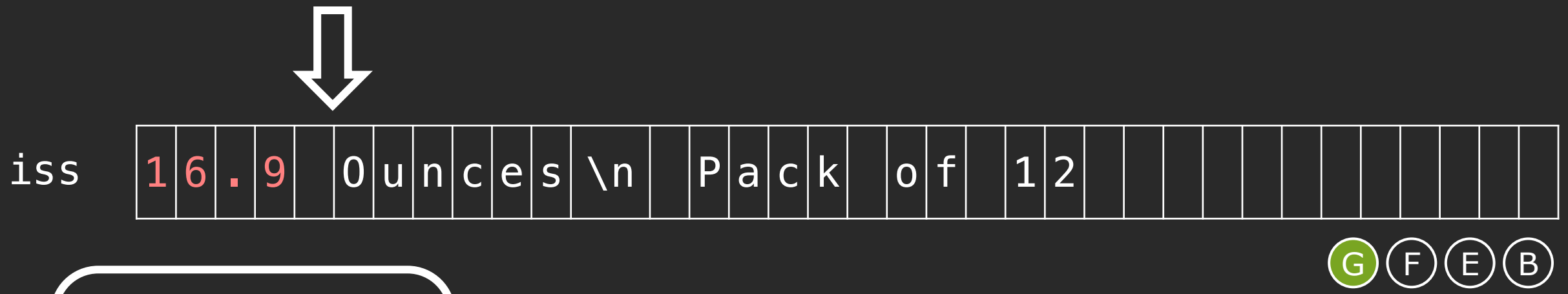
We want to read the entire line the user typed in into name.




```
iss >> amount;
iss.ignore();
getline(iss, unit);
```

amount (double)	16.9
unit (string)	???

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

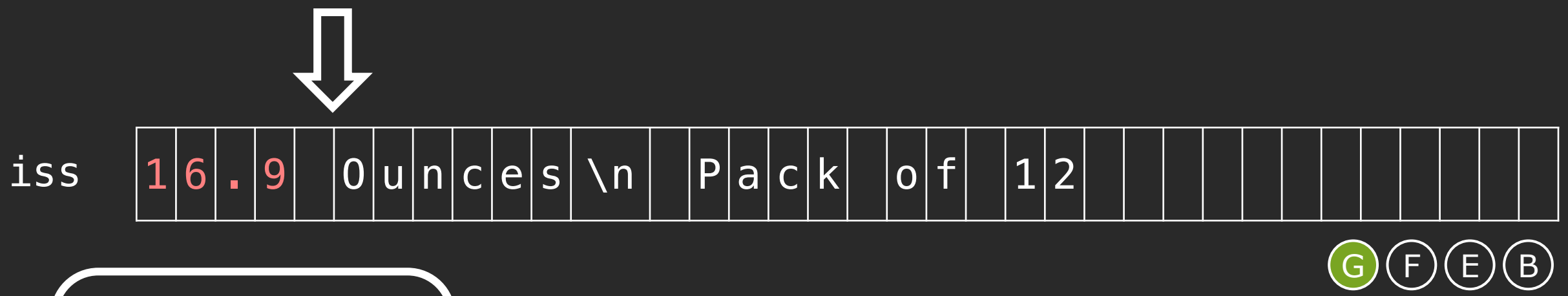



```
l
```

```
iss >> amount;
iss.ignore();
getline(iss, unit);
```

amount (double)	16.9
unit (string)	???

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

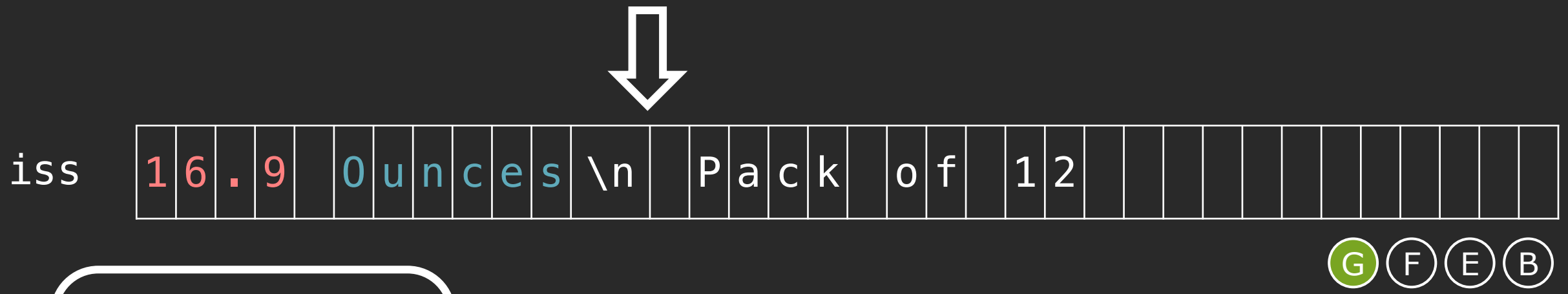



```
l
```

```
iss >> amount;
iss.ignore();
getline(iss, unit);
```

amount (double)	16.9
unit (string)	???

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




```
l
```

```
iss >> amount;
iss.ignore();
getline(iss, unit);
```

amount (double): 16.9

unit (string): "Ounces"

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

Be careful about mixing >> with getline!

- To solve the issue of getline retrieving a whitespace, use the ignore function.
- Generally try to avoid this problem by…
  - Using getline for cin, not >>.
  - Using >> when you are trying to parse space by space.
  - Using more advanced regex libraries if doing more advanced parsing.
- Do not use >> with the Stanford libraries which use getline.

Summary of Types and Streams II

- Use modern C++ constructs! (auto, uniform initialization, etc.)
- If you need error checking for user input, best practice is to:
  - use getline to retrieve a line from cin,
  - create a istringstream with the line,
  - parse the line using a stringstream, usually with >>.
- Use state bits to control streams and perform error-checking.
  - fail bit can check type mismatches
  - eof bit can check if you consumed all input

Your challenge for Thursday

```
// Write the following function which prompts
// the user for a filename, opens the ifstream to
// the file, reprompt if the filename is not valid
// and then return the filename.

string promptUserForFile(ifstream& stream,
                         string prompt = "",
                         string reprompt = "") {

    // your implementation here
}
```

Your challenge for Thursday

Read through the assignment 1 handout.

Understand the various structs provided in the starter code.

Try to complete all the file-reading and user i/o.

You'll learn more about STL vectors on Thursday (they are similar to the Stanford ones).

modern C++ types

note: the slides in this section is meant mostly as a reference, and doesn’t have a logical flow. During lecture we mostly focused on an example.

Ever get this annoying warning message about unsigned integers?

```
string str = “Hello World!”;
for (int i = 0; i < str.size(); ++i) {
     cout << str[i] << endl;
}
```

```
⚠ comparison of integers of different signs: 'int' and 'std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >::size_type' (aka 'unsigned long')   main.cpp   8
  /Users/averyw09521/code/cs106l/Lecture/StreamsII/main.cpp
⚠ comparison of integers of different signs: 'int' and 'std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >::size_type' (aka 'unsigned long') [-Wsign-compa   main.cpp   8
```

Integers which are never negative are often assigned type size_t.

Ever get this annoying warning message about unsigned integers?

```
string str = “Hello World!”;
for (int i = 0; i < str.size(); ++i) {
     cout << str[i] << endl;
}
```

```
⚠ comparison of integers of different signs: 'int' and 'std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >::size_type' (aka 'unsigned long')   main.cpp   8
  /Users/averyw09521/code/cs106l/Lecture/StreamsII/main.cpp
⚠ comparison of integers of different signs: 'int' and 'std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >::size_type' (aka 'unsigned long') [-Wsign-compa   main.cpp   8
```

This comparison is dangerous since it compares signed (i) with unsigned (str.size()).

Ever get this annoying warning message about unsigned integers?

```
string str = “Hello World!”;
for (size_t i = 0; i < str.size(); ++i) {
     cout << str[i] << endl;
}
```

Used mostly for functions dealing with indices.

What's the bug in this function?

```
string chopBothEnds(const string& str) {
    string result = "";
    for (size_t i = 1; i < str.size()-1; ++i) {
        result += str[i];
    }
    return result;
}
```

True story: I've failed coding challenges because of this bug.

Type aliases allow you to give another name for a type.

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

map_iterator begin = studentMap.cbegin();


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

When to use type aliases?

- When a type name is too long and a simpler alias makes the code more readable.
- In libraries there is a common name for a type within each class. Example:
  - vector::iterator, map::iterator, string::iterator
  - vector::reference, map::reference, string::reference

If the type is not important, then the compiler can figure it out for you.

```
// all iterators behave the same, and it doesn’t impact
// how I will use them, so let’s not worry about it!

auto begin = studentMap.cbegin();


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

Be careful about tricky auto gotchas!

```
auto calculateSum(const vector<string>& v) {
  auto multiplier = 2.4;
  auto name = "Avery";
  auto betterName1 = string{"Avery“};
  const auto& betterName2 = string{"Avery“};
  auto copy = v;
  auto& refMult = multiplier;
  auto func = [](auto i) {return i*2};

  return betterName;
}
```

Be careful about tricky auto gotchas!

```
// return type: string, notice can't use auto for parameter
auto calculateSum(const vector<string>& v) {
  auto multiplier = 2.4;                          // double
  auto name = "Avery";                            // char* (c-string)
  auto betterName1 = string{"Avery“};             // string
  const auto& betterName2 = string{"Avery“}; // const string&
  auto copy = v;                                  // vector<string>
  auto& refMult = multiplier;                     // double&
  auto func = [](auto i) {return i*2;};           // ???

  return betterName;
}
```

Careful: auto discards const and references!

When to use auto?

- When you don’t care what the type is (iterators)
- When its type is clear from context (templates)
- When you don’t know what the type is (lambdas)
- Don’t use it unnecessarily for return types.

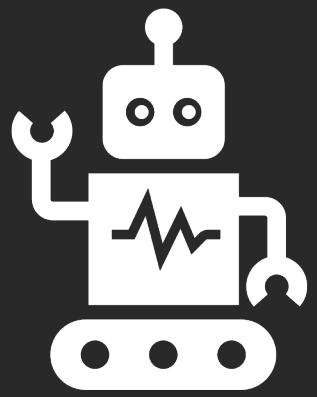
```
auto spliceString(const string& s);
```

Can you guess what this function returns? Not really.

Why use auto?

- Correctness: no implicit conversions, uninitialized variables.
- Flexibility: code easily modifiable if type changes need to be made.
- Powerful: very important when we get to templates!

- Modern IDE’s (eg. Qt Creator) can infer a type simply by hovering your cursor over any auto, so readability not an issue!



Example

auto, pair, tuple, structs, references, conversions, structured binding

When to use auto?

- When you don’t care what the type is (iterators)
- When its type is clear from context (templates)
- When you don’t know what the type is (lambdas)
- Don’t use it unnecessarily for return types.

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

pair/tuple functions

```
// make_pair/tuple (C++11) automatically deduces the type!
auto prices = make_pair(3.4, 5);          // pair<double, int>
auto values = make_tuple(3, 4, “hi”);     // tuple<int, int, char*>

// access via get/set
prices.first = prices.second;             // prices = {5.0, 5};
get<0>(values) = get<1>(values);          // values = {4, 4, “hi”};

// structured binding (C++17) – extract each component
auto [a, b] = prices;                     // a, b are copies of 5.0 and 5
const auto& [x, y, z] = values;           // x, y, z are const references
                                          // to the 4, 4, and “hi”.
```

struct functions

```
struct Discount {
  double discountFactor;
  int expirationDate;
  string nameOfDiscount;
}; // don't forget this semicolon :/

// Call to Discount's constructor or initializer list
auto coupon1 = Discount{0.9, 30, "New Years"};
Discount coupon2 = {0.75, 7, "Valentine's Day"};

coupon1.discountFactor = 0.8;
coupon2.expirationDate = coupon1.expirationDate;

// structured binding (C++17) – extract each component
auto [factor, date, name] = coupon1;
```

aceess by reference

references

```
string tea = “Ito-En”;
string copy = tea;
string& ref = tea;

// note: the string operator [] returns a reference to a char in string
tea[1] = ‘a’;            // tea = “Iao-En”;
copy[2] = ‘b’;           // tea = “Iao-En”; (no change)
ref[3] = ‘c’;            // tea = “IaocEn”;

char letterCopy = tea[4];
char& letterRef = tea[5];

letterCopy = ‘d’;        // tea = “IaocEn”; (no change)
letterRef = ‘e’;         // tea = “IaocEe”;
ref = copy;              // tea = “Iab-En”; cannot reassign reference
```

dangling references

never return references to local variables!

```
char& firstCharBad(string& s) {
  string local = s;
  return local[0];
}

char& firstCharGood(string& s) {
  return s[0];
}

int main() {
  string tea = “Ito-En”;
  char& bad = firstCharBad(tea);   // undefined, ref to local out of scope
  char& good = firstCharGood(tea); // good ref to tea[0]
}
```

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()		
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!

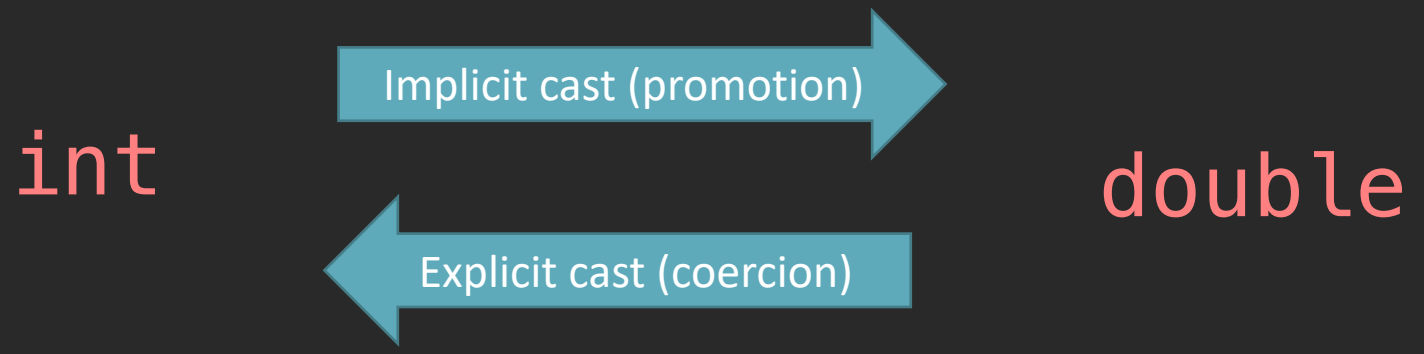
Preview of week 7:
moving rather than copying

	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()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy	f(X)	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

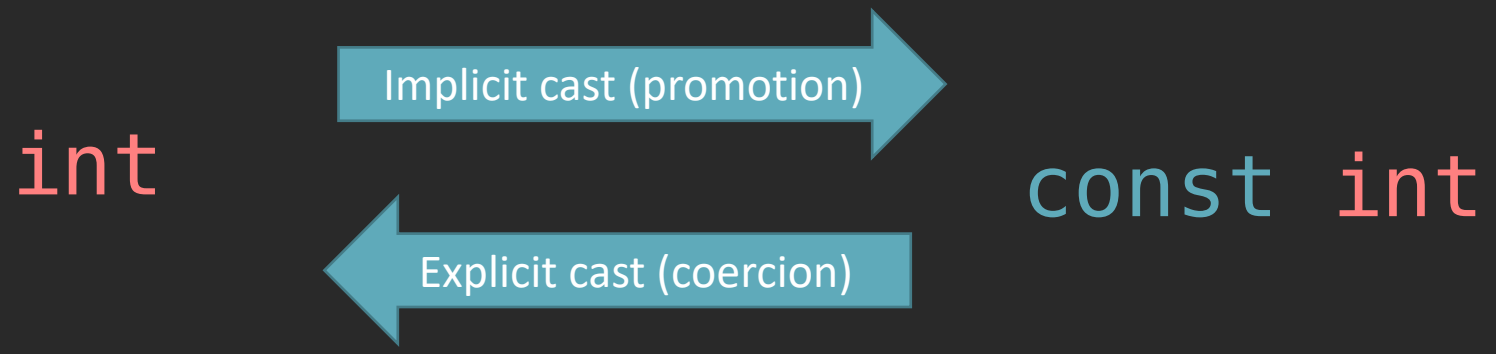
Conversions have two directions.

```
int v1 = static_cast<double>(3.4); // explicit cast
double v2 = 6;                     // promotion
```



Conversions have two directions.

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



Uniform initialization: a uniform way to initialize any variable.

```
struct Course {
  string code;
  Time start, end;
  vector<string> instructors;
};

struct Time {
  int hour, minute;
}
```

```
int main() {
  vector<int> vec{3, 1, 4, 1, 5, 9};
  Course now {“CS106L”,
              {13, 30}, {14, 30},
             {“Wang”, “Zeng”} };
}
```

Everything in one example!

```
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};  // uniform initialization
}

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

Next time

Intro the STL and Sequence Containers