四川大學
SICHUAN UNIVERSITY

Object Oriented Programming—C++

Lecture4 Streams & Containers

Qijun Zhao

College of Computer Science

Sichuan University

Spring 2023

Recap

- **Uniform Initialization**
  - A “uniform” way to initialize variables of different types!
- **References**
  - Allow us to alias variables
- **Const**
  - Allow us to specify that a variable can’t be modified

Streams

- **What are streams?**
- Output streams
- Input streams
- String streams!

Definition

stream: an abstraction for input/output. Streams convert between *data* and the *string representation of data.*

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
```

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
```

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s << std::endl;
```

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s << std::endl;
```

ERROR!

A stream you've used: cout

```
// use a stream to prin
std::cout << 5 << std::e
// and most from the STI
std::cout << "Sarah" <<
// Mix types!
std::cout << "Sarah is "
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s << std::endl;
```

Reminder: Our student struct

```
struct Student {
    string name;
    string state;
    int age;
};
```

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s.name << s.age << std::endl;
```

Works

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// Any primitive type + most from the STL work!
// For other types, you will have to write the
            << operator yourself!
```

We'll talk about how to write the << operator for custom types during lecture on Operators!

std::cout is an *output stream*. It has type **std::ostream**

Two ways to classify streams

By Direction:

- **Input streams:** Used for reading data (ex. ‘std::istream’, ‘std::cin’)
- **Output streams:** Used for writing data (ex. ‘std::ostream’, ‘std::cout’)
- **Input/Output streams:** Used for both reading and writing data (ex. ‘std::iostream’, ‘std::stringstream’)

By Source or Destination:

- **Console streams:** Read/write to console (ex. ‘std::cout’, ‘std::cin’)
- **File streams:** Read/write to files (ex. ‘std::fstream’, ‘std::ifstream’, ‘std::ofstream’)
- **String streams:** Read/write to strings (ex. ‘std::stringstream’, ‘std::istringstream’, ‘std::ostringstream’)

Streams

- What are streams?
- **Output streams**
- Input streams
- String streams!

Output Streams

- Have type std::ostream
- Can only ***send*** data to the stream
- Interact with the stream using the << operator
- Convert any type into string and ***send*** it to the stream
- std::cout is the output stream that goes to the console

```
std::cout << 5 << std::endl;
// converts int value 5 to string "5"
// sends "5" to the console output stream
```

Output File Streams

- Have type std::ofstream
- Only ***send*** data using the << operator
- Convert data of any type into a string and send it to the **file stream**
- Must initialize your own ofstream object linked to your file

```
std::ofstream out("out.txt");
// out is now an ofstream that outputs to out.txt
out << 5 << std::endl; // out.txt contains 5
```

std::cout is a *global constant object* that you get from

#include <iostream>

To use any other output stream,

you must first initialize it!

Streams

- What are streams?
- Output streams
- **Input streams**
- String streams!

What does this code do?

```
int x;
std::cin >> x;
// what happens if input is 5 ?
// how about 51375 ?
// how about 5 1 3 7 5?
```

Let's try it out!

A note about nomenclature

- "**>>**" is the **stream extraction operator** or simply extraction operator
  - Used to **extract data from a stream** and place it into a variable
- "**<<**" is the **stream insertion operator** or insertion operator
  - Used to **insert data into a stream** usually to output the data to a file, console, or string

std::cin is an input stream. It has type std::istream

Input Streams

- Have type std::istream
- Can only ***receive*** strings using the >> operator
- ***Receive*** a string from the stream and convert it to data
- std::cin is the input stream that gets input from the console

```
int x;
string str;
std::cin >> x >> str;
```

Nitty Gritty Details: std::cin

- First call to std::cin >> creates a command line prompt that allows the user to type until they hit enter

- Each >> ONLY reads until the next whitespace

    - Whitespace = tab, space, newline

- Everything after the first whitespace gets saved and used the next time std::cin >> is called

    - The place its saved is called a buffer!

- If there is nothing waiting in the buffer, std::cin >> creates a new command line prompt

- Whitespace is eaten; it won't show up in output

Think of a std::istream as a sequence of characters

4	2		a	b		4	\n

position

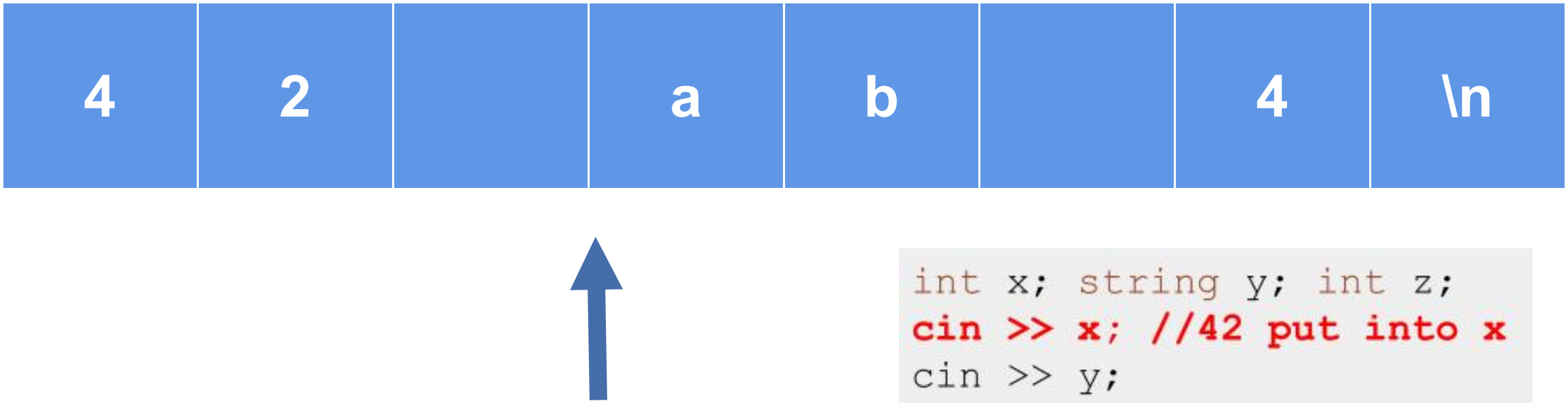
```
int x; string y; int z;
cin >> x;
cin >> y;
cin >> z;
```

Think of a std::istream as a sequence of characters

position

```
int x; string y; int z;
cin >> x; //42 put into x
cin >> y;
cin >> z;
```

Think of a std::istream as a sequence of characters




```
int x; string y; int z;
cin >> x; //42 put into x
cin >> y;
cin >> z;
```

Think of a std::istream as a sequence of characters


```
int x; string y; int z;
cin >> x;
cin >> y; //ab put into y
cin >> z;
```

position

Think of a std::istream as a sequence of characters


```
int x; string y; int z;
cin >> x;
cin >> y; //ab put into y
cin >> z;
```

position

Think of a std::istream as a sequence of characters


```
int x; string y; int z;
cin >> x;
cin >> y;
cin >> z; //4 put into z
```

position

Input Streams: When things go wrong

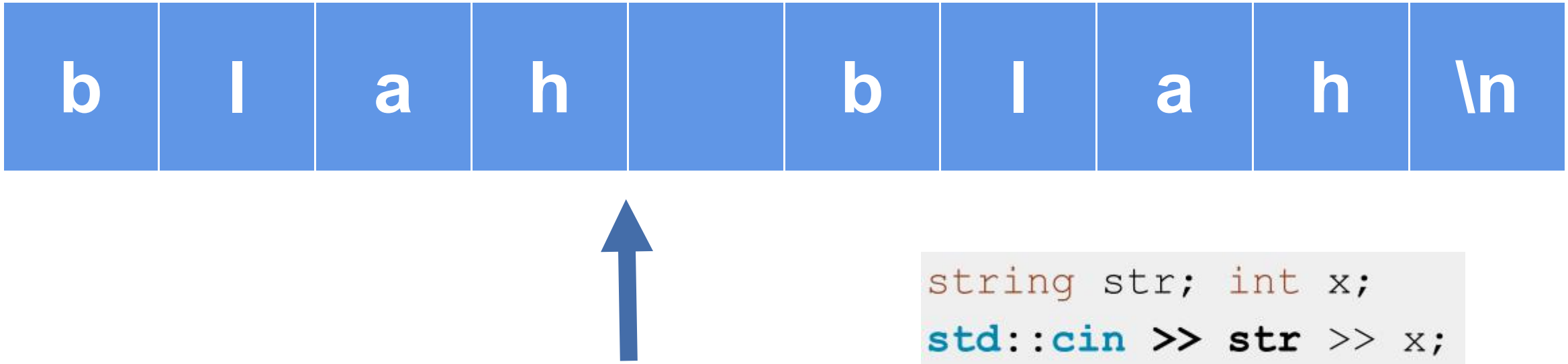
```
string str;
int x;
std::cin >> str >> x;
//what happens if input is blah blah?
std::cout << str << x;
```

Think of a std::istream as a sequence of characters

position

```
string str; int x;
std::cin >> str >> x;
```

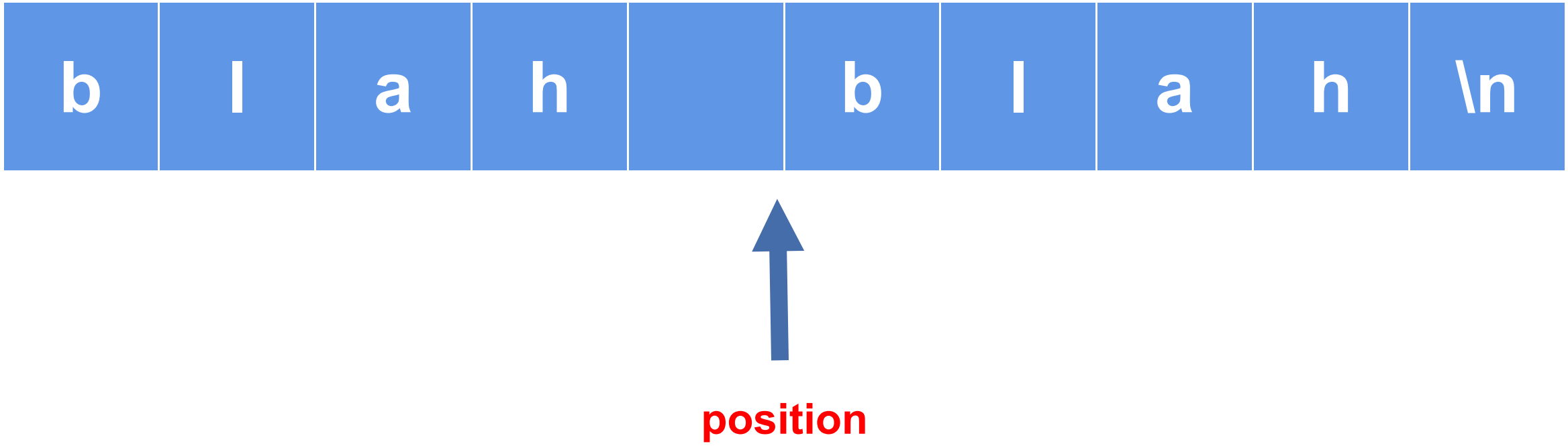
Think of a std::istream as a sequence of characters




```
string str; int x;
std::cin >> str >> x;
```

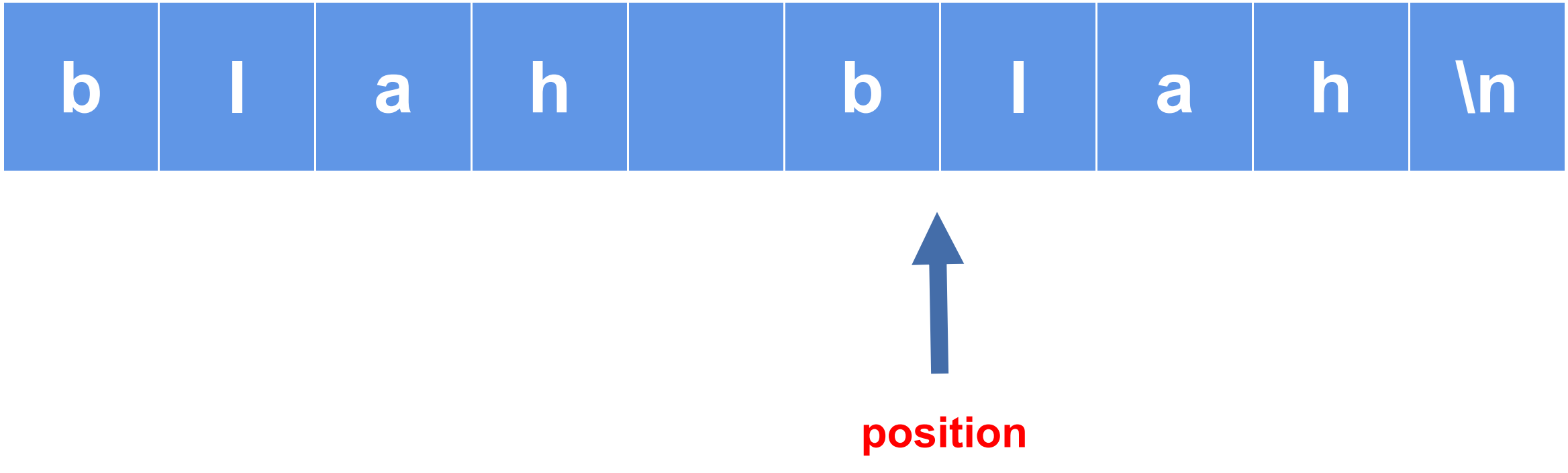
Think of a std::istream as a sequence of characters

```
string str; int x;
std::cin >> str >> x;
```



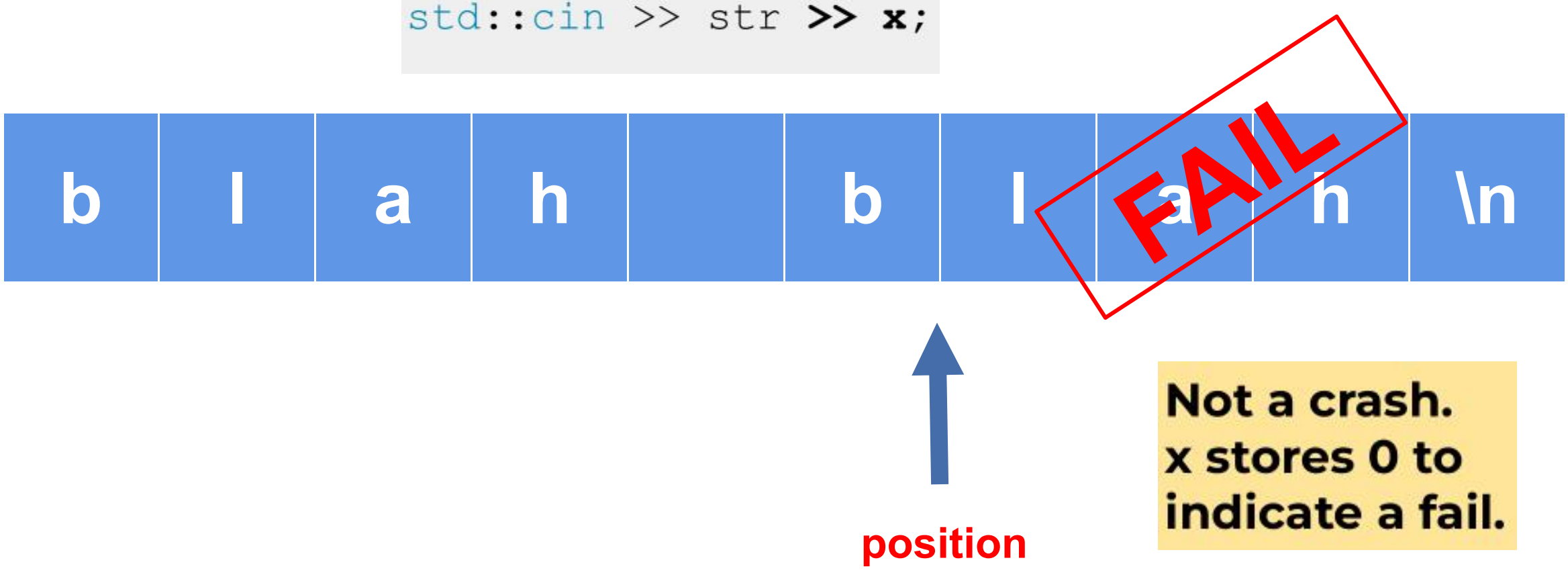
Think of a std::istream as a sequence of characters

```
string str; int x;
std::cin >> str >> x;
```



Think of a std::istream as a sequence of characters

```
string str; int x;
std::cin >> str >> x;
```



Not a crash.
x stores 0 to indicate a fail.

Input Streams: When things go wrong

```
string str;
int x;
string otherStr;
std::cin >> str >> x >> otherStr;
//what happens if input is blah blah blah?
std::cout << str << x << otherStr;
```

Let's try it out!

Input Streams: When things go wrong

```
string str;
int x;
string otherStr;
std::cin >> str >> x >> otherStr;
//what happens if input is blah blah blah?
std::cout << str << x << otherStr;
//once an error is detected, the input stream's
//fail bit is set, and it will no longer accept
//input
```

str → "blah"
x → 0
otherStr → NOTHING

Input Streams: When things go wrong

```
int age; double hourlyWage;
cout << "Please enter your age: ";
cin >> age;
cout << "Please enter your hourly wage: ";
cin >> hourlyWage;
//what happens if first input is 2.17?
```

Think of a std::istream as a sequence of characters

2	.	1	7	\n

position

```
cin >> age;
cout << "Wage: ";
cin >> hourlyWage;
```

Think of a std::istream as a sequence of characters


```
cin >> age;
cout << "Wage: ";
cin >> hourlyWage;
```

Think of a std::istream as a sequence of characters

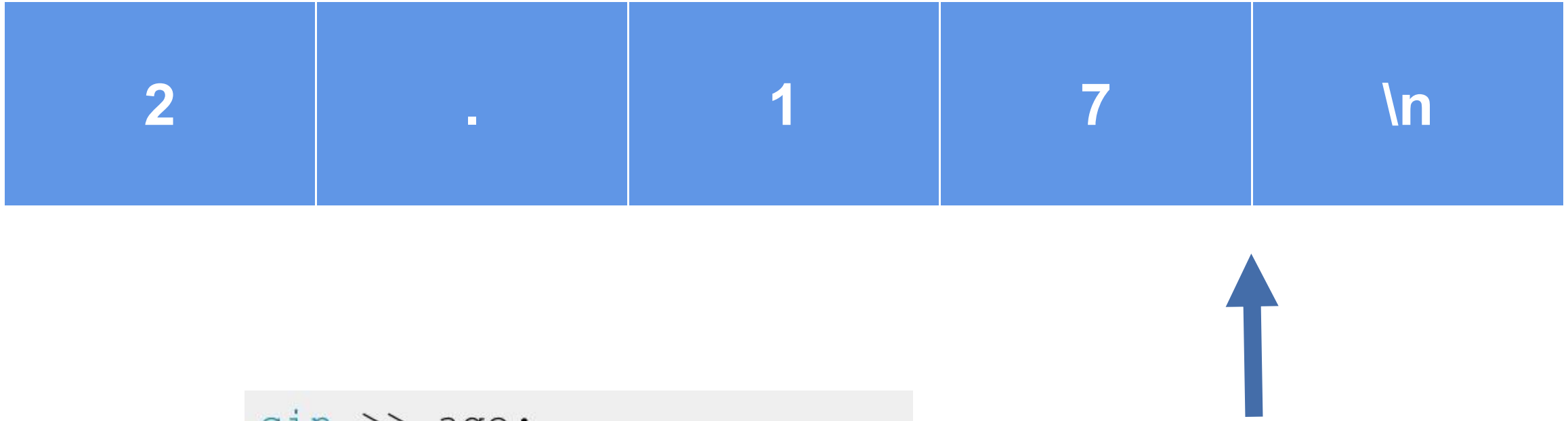
2	.	1	7	\n

position

Reads until it finds something that isn't an int!

```
cin >> age; // age = 2
cout << "Wage: ";
cin >> hourlyWage;
```

Think of a std::istream as a sequence of characters




```
cin >> age;
cout << "Wage: ";
cin >> hourlyWage;// =.17
```

std::getline()

```
// Used to read a line from an input stream
// Function Signature
istream& getline(istream& is, string& str, char delim);

// Designed to work with character sequences
```

getline reads from

& stores output in

Stops when read.
'\n' = default

std::getline(istream& is, string& str, char delim)

- **How it works:**
- Clears contents in str
- Extracts chars from is and stores them in str until:
    - End of file reached, sets EOF bit (checked using is.eof())
    - Next char in is is delim, extracts but does not store delim
    - str out of space, sets FAIL bit (checked using is.fail())
    - If no chars extracted for any reason, FAIL bit set

std::getline(istream& is, string& str, char delim)

- **How it works:**
- Clears contents in str
- Extracts chars from is and stor
    - End of file reached, sets E
    - Next char in is is delim, ex
delim
    - str out of space, sets FAIL
    - If no chars extracted for any reason, FAIL bit set

In contrast:

- **“>>”** only **reads until it hits whitespace** (so can’t read a sentence in one go)
- BUT **“>>”** can **convert data to built-in types** (like ints) while getline can only produce strings.
- AND **“>>”** only **stops reading at predefined whitespace** while getline can stop reading at any delimiter you define

Reading using >> extracts a single "word" or built-in type

including for strings

To read a whole line, use

std::getline(istream& stream, string& line);

How to use getline

- **Notice** getline(istream& stream, string&line)**takes in both parameters by reference!**

```
std::string line;
std::getline(cin, line); //line changed now!
//say the user entered "Hello World 42!"
std::cout << line << std::endl;
//should print out "Hello World 42!"
```

DEMO

IMPORTANT: Don’t mix >> with getline!

- >> reads up to the next whitespace character and does not go past that whitespace character.

- getline reads up to the next delimiter (by default, ‘\n’), and does go past that delimiter.

- TL;DR they don’t play nicely

Input File Streams

- Have type std::ifstream
- Only receives strings using the >> operator
  - Receives strings from a file and converts it to data of any type
- Must initialize your own ifstream object linked to your file

```
std::ifstream in("out.txt");
// in is now an ifstream that reads from out.txt
string str;
in >> str; // first word in out.txt goes into str
```

std::cin is a global constant *object* that you get from #include <iostream>

To use any other input stream, you must first initialize it!

Streams

- What are streams?
- Output streams
- Input streams
- **String streams!**

String streams

- **What**: A stream that can read from or write to a string object
- **Purpose**: Allows you to perform input/output operations on a string as if it were a stream

```
std::string input = "123";
std::stringstream stream(input);
int number;
stream >> number;
std::cout << number << std::endl;  // Outputs "123"
```

If you only want to read OR write data:

- Read only: std::istringstream
  - Give any data type to the istringstream, it'll store it as a string!
- Write only: std::ostringstream
  - Make an ostringstream out of a string, read from it word/type by word/type!
- Follows same patterns as the other i/ostreams!

Recap

- Streams convert between data of any type and the string representation of that data

- Streams have an endpoint: console for cin/cout, files for i/o fstreams, string variables for i/o stringstreams where they read in a string from or output a string to.

- To send data (in string form) to a stream, use stream_name << data

- To extract data from a stream, use stream_name >> data, and the stream will try to convert a string to whatever type data is

Containers

-Defining Containers

What is a container in C++?

- Containers in the STL

  Types of containers and how they work

- Container Adaptors

  Abstracting container implementation

Container: An object that allows us to collect other objects together and interact with them in some way.

Think of **vectors**, **stacks**, or **queues!**

Why containers?

What is the purpose of container types in programming languages?

Motivating containers

We've been using the idea of a Student struct for the past few lectures:

```
struct Student {
    string name; // these are called fields
    string state; // separate these by semicolons
    int age;
};

Student s;
s.name = "Sarah";
s.state = "CA";
s.age = 21; // use . to access fields
```

What if we had a whole class of students?

This is generalizable!

What if we need to store other types of data?

- Class grades
- Coordinates in a graph
- Mountains

What if we want to store it in a different way?

- FIFO vs LIFO
- Ascending order by some value

-Defining Containers

What is a container in C++?

- Containers in the STL

Types of containers and how they work

- Container Adaptors

Abstracting container implementation

The STL has many types of containers:

Both familiar:

- Vector
- Stack
- Queue
- Set
- Map

And unfamiliar:

- Array
- Deque
- List
- Unordered set
- Unordered map

New containers

- An **array** is the primitive form of a vector
  - Fixed size in a strict sequence
- A **deque** is a **d**ouble **e**nded **que**ue
- A **list** is a doubly linked list
  - Can loop through in either direction!

std::vector

What you want to do	std::vector<int>
Create a new, empty vector	`std::vector<int> vec;`
Create a vector with `n` copies of 0	`std::vector<int> vec(n);`
Create a vector with `n` copies of a value `k`	`std::vector<int> vec(n, k);`
Add a value `k` to the end of a vector	`vec.push_back(k);`
Remove all elements of a vector	`vec.clear();`
Get the element at index `i`	`int k = vec[i];` (does **not** bounds check)
Check size of vector	`vec.size();`
Loop through vector by index `i`	`for (std::size_t i = 0; i < vec.size(); ++i) ...`
Replace the element at index `i`	`vec[i] = k;` (does **not** bounds check)

std::vector

What you want to do	std::vector<int>
Create a new, empty vector	`std::vector<int> vec;`
Create a vector with `n` copies of 0	`std::vector<int> vec(n);`
Create a vector with `n` copies of a value `k`	`std::vector<int> vec(n, k);`
Add a value `k` to the end of a vector	`vec.push_back(k);`
Remove all elements of a vector	`vec.clear();`
Get the element at index `i`	`int k = vec[i];` (does **not** bounds check)
Check size of vector	`vec.size();`
Loop through vector by index `i`	`for (std::size_t i = 0; i < vec.size(); ++i) ...`
Replace the element at index `i`	`vec[i] = k;` (does **not** bounds check)

What does this mean?

Safety vs Speed

In choosing a programming language, there's always a tradeoff between **speed, power, and safety.**

C++ is really fast! Why is that?

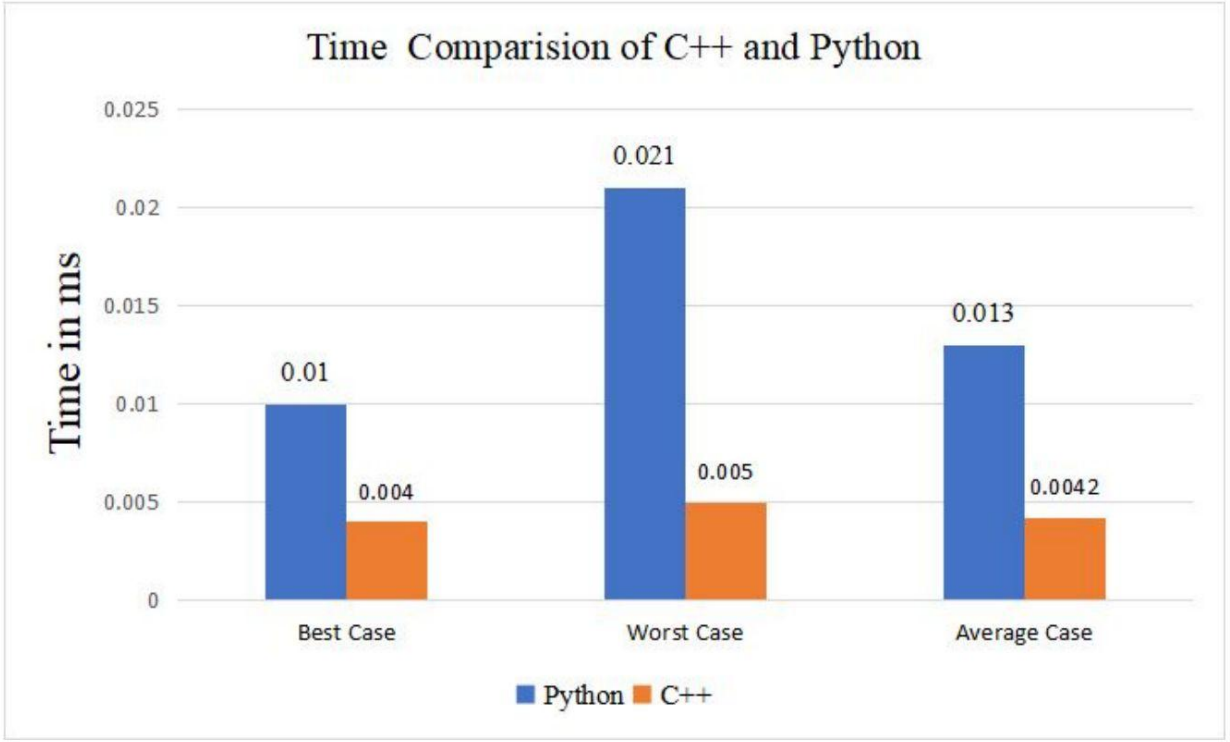


Fig. 13. Comparison of Time Utilization of Deletion Algorithm

C++ Design Philosophy

- Only provide the checks/safety nets that are necessary
- The programmer knows best!

Making sure what you're doing is allowed is your job!

std::set

What you want to do	std::set<int>
Create an empty set	`std::set<int> s;`
Add a value **k** to the set	`s.insert(k);`
Remove value **k** from the set	`s.erase(k);`
Check if a value **k** is in the set	`if (s.count(k)) ...`
Check if vector is empty	`if (vec.empty()) ...`

std::map

What you want to do	std::map<int, char>
Create an empty map	`std::map<int, char> m;`
Add key k with value v into the map	`m.insert({k, v});` `m[k] = v;`
Remove key k from the map	`m.erase(k);`
Check if key k is in the map	`if (m.count(k)) ...`
Check if the map is empty	`if (m.empty()) ...`
Retrieve or overwrite value associated with key k (***error*** *if key isn't in map)*	`char c = m.at(k);` `m.at(k) = v;`
Retrieve or overwrite value associated with key k (***auto-insert*** *if key isn't in map)*	`char c = m[k];` `m[k] = v;`

There are two types of containers:

Sequence:

- Containers that can be accessed sequentially
- Anything with an inherent order goes here!

Associative

- Containers that don't necessarily have a sequential order
- More easily searched
- Maps and sets go here!

All containers can hold all types of information! How do we choose which to use?

Vector implementation

How do vectors actually work?

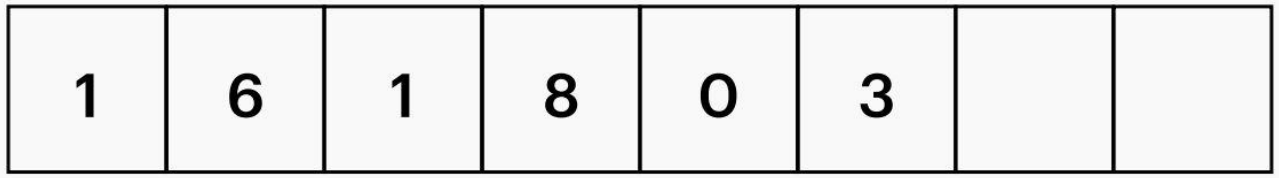
- At a high level, a vector is an **ordered** collection of elements of the **same type** that can grow and shrink in size.

Internally, vectors implement an array!

Vector implementation

We keep track of a few member variables:

- _size = number of elements in the vector
- _capacity = space allocated for elements

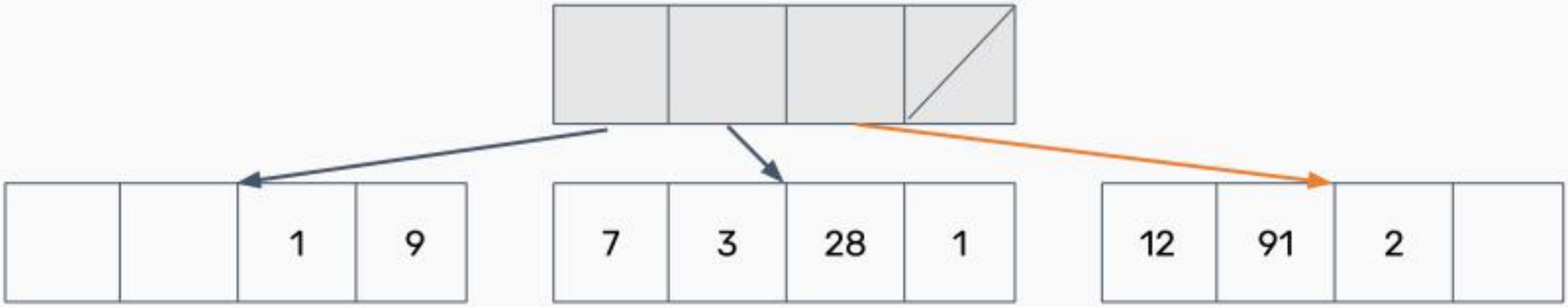


Don’t confuse these two!

What about a deque?

Deques can be implemented many different ways! Here's one:

- What if we had an array that stored other arrays?



Choosing sequence containers

What you want to do	std::vector	std::deque	std::list
Insert/remove in the front	Slow	Fast	Fast
Insert/remove in the back	Super Fast	Very Fast	Fast
Indexed Access	Super Fast	Fast	Impossible
Insert/remove in the middle	Slow	Fast	Very Fast
Memory usage	Low	High	High
Combining (splicing/joining)	Slow	Very Slow	Fast
Stability* (iterators/concurrency)	Bad	Very Bad	Good

Sequence Containers: Summary

- Sequence containers are for when you need to enforce some order on your information!
- Can usually use an **std::vector** for most anything
- If you need particularly fast inserts in the front, consider an **std::deque**
- For joining/working with multiple lists, consider an **std::list** (very rarely)

Map implementation

Maps are implemented with pairs! (**std::pair<const key, value>**)

- Note the const! Keys must be immutable.
- Why a pair and not a tuple?

Unordered maps/sets

Both maps and sets in the STL have an unordered version!

- Ordered maps/sets require a **comparison operator** to be defined.
- **Unordered** maps/sets require a **hash function** to be defined.

Simple types are already natively supported; anything else will need to be defined yourself.

Unordered maps/sets are usually faster than ordered ones!

Choosing associative containers

Lots of similarities between maps/sets! Broad tips:

- Unordered containers are **faster**, but can be difficult to get to work with nested containers/collections
- If using **complicated data types**/unfamiliar with hash functions, use an ordered container

So far:

- Sequence containers:
  - Arrays, vectors, deques, lists
- Associative containers:
  - Sets and maps
  - Unordered vs. ordered

-Defining Containers

What is a container in C++?

- Containers in the STL vs Stanford

Types of containers and how they work

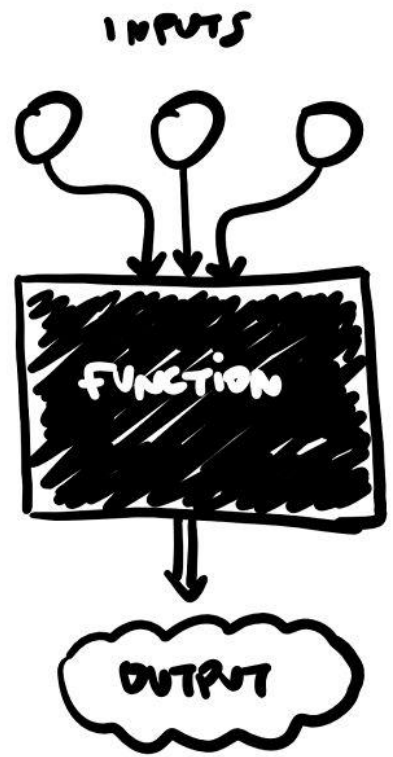
- Container Adaptors

Abstracting container implementation

Container Adaptors

Container adaptors are “wrappers” to existing containers!

- Wrappers modify the interface to sequence containers and change what the client is allowed to do/how they can interact with the container.
- How could we make a wrapper to implement a queue from a deque?



Let's ask the STL!

```
template <class T, class Container = deque<T> > class queue;
```

queues are implemented as ***containers adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***pushed*** into the ***"back"*** of the specific container and ***popped*** from its ***"front"***.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

empty
size
front
back
push_back
pop_front

```
std::queue<int> stack_deque;                    // Container = std::deque
std::queue<int, std::list<int>> stack_list;     // Container = std::list
```

Summary

- Containers are ways to collect related data together and work with it logically
- Two types of containers: sequence and associative
- Container adaptors wrap existing containers to permit new/restrict access to the interface for the clients.

四川大學
SICHUAN UNIVERSITY

Coding for love,
Coding for the world

Qijun Zhao

qjzhao@scu.edu.cn