ENGR 101 – Chapter 15

Strings, Streams, and I/O

Lecture Files – Google Drive

- This lecture involves *many* files!

- Go to the Lecture > Resources folder on the Google Drive
- Right click on *Lecture 18 Streams and IO* and click download.
- This should download a zip archive, which you can extract into your folder for lecture exercises.
  - Make sure you actually extract the files! Windows machines will often show you the contents of the files even though the files are still compressed.
- You might have to download them all individually if Google Drive is being slow. ☹

Libraries

- In MATLAB, a wealth of built-in functionality was available to use without doing anything extra.

- C++ also provides this through **libraries**, but you need to explicitly include them at the top of your program.

- For example, to use `cin` and `cout`:

```
#include <iostream>
```

Note: Don't use semicolons for lines starting with #.

A Few Common Libraries

`cmath`	Standard math library
`cstdlib`	C standard library (common functions e.g. `rand`)
`iostream`	Standard input/output (`cin` and `cout`)
`fstream`	File input/output through streams
`iomanip`	Set precision for printing floating point numbers
`string`	The `string` datatype
`vector`	The `vector` datatype (a container)

4 min

Exercise: `cstdlib` and `cmath`

- The C++ libraries have extensive online documentation.
  - Using this documentation is an important skill!
- Investigate the `cstdlib` and `cmath` libraries, and use functions from each to implement the following program:
  - Generate a random number between 0 and 9 (inclusive), take it to the 4^{th} power, and print the resulting number.

rand0-9.cpp

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
int main() {
  // YOUR CODE HERE
}
```

Start by googling for C++ and the library names!

Solution: cstdlib and cmath

- Generate a random number between 0 and 9 (inclusive), take it to the 4th power, and print the resulting number.

rand0-9.cpp

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
int main() {
  // The % 10 forces the int to 0-9 range
  int num = rand() % 10;

  cout << pow(num, 4) << endl;
}
```

- Did you try running the code multiple times?
  - Wait…it always yields the same "random" number!

Pseudorandom Numbers

- Random numbers in programming are not "truly random".
- Instead, they are generated from a *pseudorandom* sequence of numbers, which is deterministic, but looks random.

rand0-9.cpp

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
int main() {
  srand(101);

  // The % 10 forces the int to 0-9 range
  int num = rand() % 10;

  cout << pow(num, 4) << endl;
}
```

Set the random *seed* using the `srand()` function. You can pick any number as the seed – each will yield a different sequence of pseudorandom numbers.

Simulating Nondeterminism

- To get a different sequence of random numbers each time, we use a clever trick: **set the seed based on the current time**.

rand0-9.cpp

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>
using namespace std;
int main() {
  srand(time(0));

  // The % 10 forces the int to 0-9 range
  int num = rand() % 10;

  cout << pow(num, 4) << endl;
}
```

The time(0) call here returns the number of milliseconds since Jan 1, 1970. That gives a different seed each time you run the program!

Input and Output (I/O)

- We've covered basic C++ input/output at the terminal…
- Today, we'll take a look at several more patterns for I/O.
- We'll also cover reading and writing files in C++

An I/O Cookbook

- Many of the examples in this lecture showcase common patterns for input/output operations.

- An effective strategy would be to treat this set of examples as a "cookbook" you can refer back to for future cases.
  - (They might be good resources to bring to an exam. Just saying.)

Review: Printing Output

- To print output in C++, we need to send it to the "standard output stream".
- `cout` is a variable that represents this stream.
  - That's pronounced "c out" (two words).
- The `<<` operator sends output, and can be "chained" to send many different pieces on one line.

```
cout << "Hello World!" << endl; // print a greeting

// Print out the result
cout << "The result is" << z << "!" << endl;
```

`endl` represents a new line.

Review: User Input

- When the user types input at the terminal, it comes in via the "standard input stream", represented by `cin`.
- The `>>` operator reads input from a stream.
  - It can be chained, just like the << operator.
  - Input is interpreted according to the type of the target variable.

```
int x;
string s1;
string s2;

cin >> x; // read an int (e.g. 3, 72, -4)

cin >> s1 >> s2; // read two strings (e.g. "hi", "cat")
```

Note: Lobster doesn't support user input.

Common Pattern: Validating Input

- A `while` loop can be used to repeatedly ask the user for input until they satisfy some criteria.

```
#include <iostream>
using namespace std;
int main() {
  int x; // Declare x to hold user input

  cout << "Enter a positive number: "; // Initial request
  cin >> x; // First try

  // As long as they got it "wrong", keep asking!
  while ( !(x > 0) ) {
    // Ask again
    cout << "Please enter a POSITIVE number: ";
    cin >> x; // try again

  }
  cout << "You entered: " << x << endl;
}
```

4 min

Exercise: Annoying Echo Program

- Write a program that accepts input from the user and echoes it back, until the user types "stop". Fill in the code below:

echo.cpp

```
#include <iostream>
#include <string>
using namespace std;
// A very annoying program: It echoes until you say stop
int main() {
  string word;
  cin >> word;

  while (    /* TODO condition */    ) {
    // TODO echo the word
    // TODO read in another word
  }

  cout << "Ok fine I'll stop :(" << endl;
}
```

Common Pattern: Reading Up to a Sentinel

- A loop can be used to collect input until the user enters a special value, called a **sentinel**.

echo.cpp

```
#include <iostream>
#include <string>
using namespace std;
// A very annoying program: It echoes until you say stop
int main() {
  string word;
  cin >> word;

  while ( word != "stop" ) {
    cout << word << endl; // echo the word
    cin >> word; // wait for the next word
  }

  cout << "Ok fine I'll stop :(" << endl;
}
```

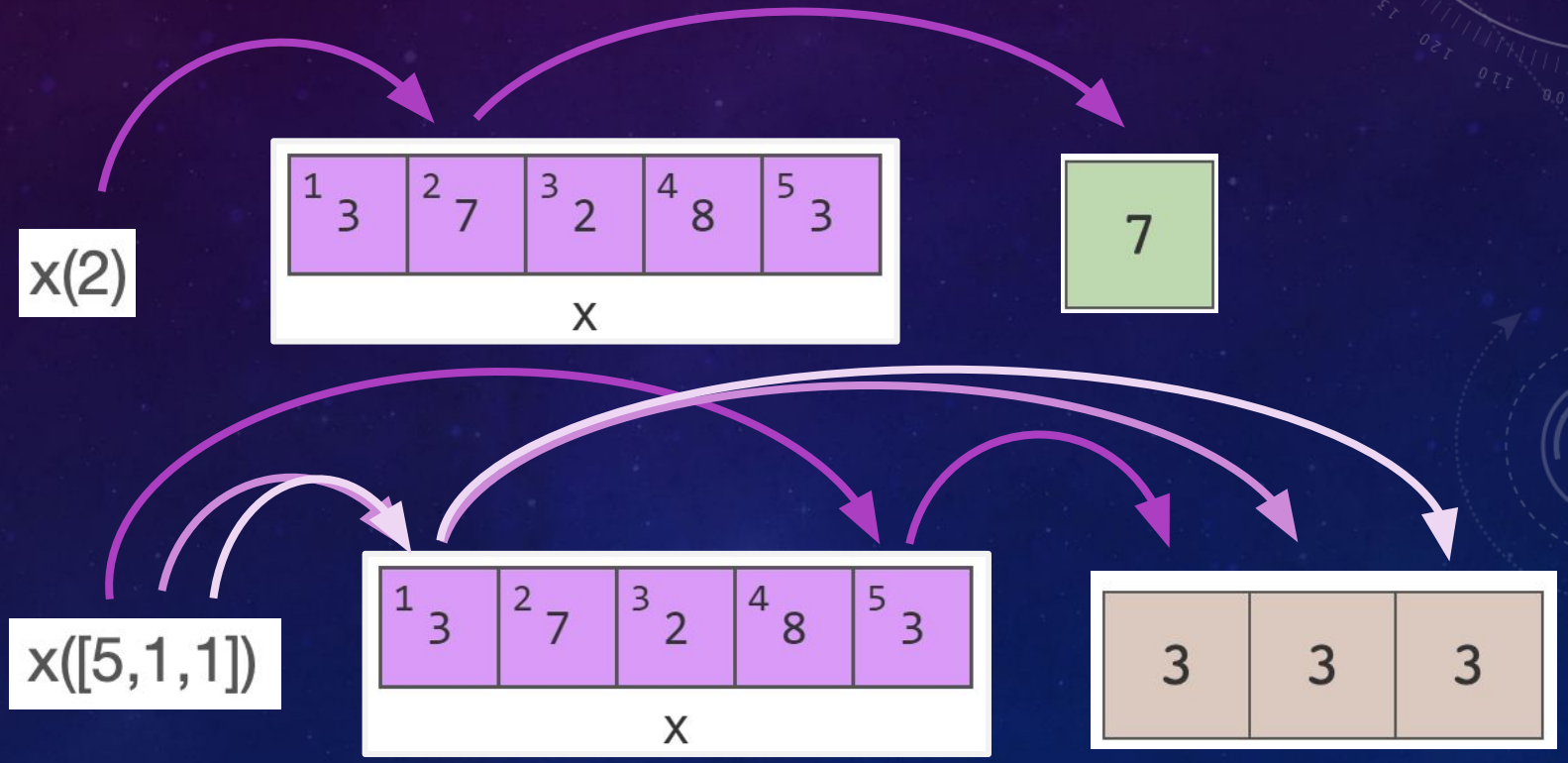
Initial read. Always happens.

Keep going until the sentinel is found.

Get more input from the user.

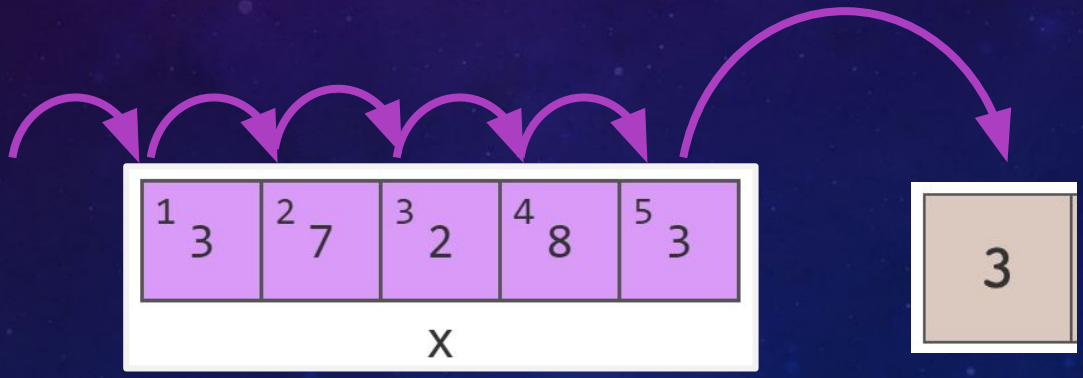
Recall: Using Indexing to Access Data

- Indexing (like in MATLAB and like we will see in C++ in the next lectures) uses **random access** to read and write data.



Using Streams to Access Data

- C++ uses input and output **streams** to access data.
- Streams (including `cin`, `cout`, and file streams) use **sequential access** to read and write data.



Using Streams to Access Data

- The << and >> operators write/read each piece of information *sequentially* – meaning one after another in the order they are arranged.

```
cout << "Hello World!" << endl; // print a greeting

// Print out the result
cout << "The result is" << z << "!" << endl;
```

```
int x;
string s1;
string s2;

cin >> x; // read an int (e.g. 3, 72, -4)

cin >> s1 >> s2; // read two strings (e.g. "hi", "cat")
```

Streams in General

- `cin` is an `istream` object
- `cout` is an `ostream` object

- There are many other kinds of streams…
  - …but they all work with << and >> in a similar fashion!

- Now, let's take a look at using **file streams** to read input from files and write output to files.

Files

- Generally, it is easiest to work with files in the same directory where you run the program.
- In this case, just provide the name of the file. Examples:

```
filename.txt
data.in
server_errors.log
```

The part after the file name is called the **file extension**. It's not required, but it usually provides some information about what the file is used for.

File Streams

- To use file streams, first we need the `fstream` library:

```
#include <fstream>
```

- For convenience, you'll also want the standard namespace:

```
using namespace std;
```

- To write output to a file, you use an `ofstream` object.

Note: Lobster doesn't support file streams.

File Output with ofstream

- First, create an ofstream:

```
ofstream fout("greet.out");
```

  - This creates an ofstream object named fout.
  - This ofstream will send output to a file named "greet.out".
    - If the file doesn't exist already, it will be created.
    - If the file already exists, it will be overwritten!

- Then, write to the file using the ofstream and <<

```
fout << "Hello! "  << endl;
```

greet.out

```
Hello!
```

File Output: Example

- A program to create a file containing the numbers 0 through 4 on separate lines:

numbers.out

```
0
1
2
3
4
```

numbers.cpp

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  ofstream fout("numbers.out");

  for(int x = 0; x < 5; ++x) {
    fout << x << endl;
  }

  fout.close();
}
```

It's a good practice to close the file once you're done using it.

Reprise: File Streams

- To use file streams, first we need the `fstream` library:

```
#include <fstream>
```

- For convenience, you'll also want the standard namespace:

```
using namespace std;
```

- To write output to a file, you use an `ofstream` object.
- To read input from a file, you use an `ifstream` object.

File Input with ifstream

words.in
```
apple
banana
pear
```

- First, create an ifstream:

```
ifstream fin("words.in");
```

  - This creates an ifstream object named fin.
  - This ifstream will read input from a file named "words.in".
    - If the file doesn't exist, there could be problems. You'll want to check to make sure the file is open…more on this later…

- Then, read from the file using the ifstream and >>

```
string word;
fin >> word;
```

File Input: Example

readWords.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
  ifstream fin("words.in");

  string word; // will hold input
  for(int x = 0; x < 3; ++x) {
    fin >> word; // read each word

    // Print each word
    cout << "Word " << x << ": " << word << endl;
  }

  fin.close();
}
```

words.in

```
apple
banana
pear
```

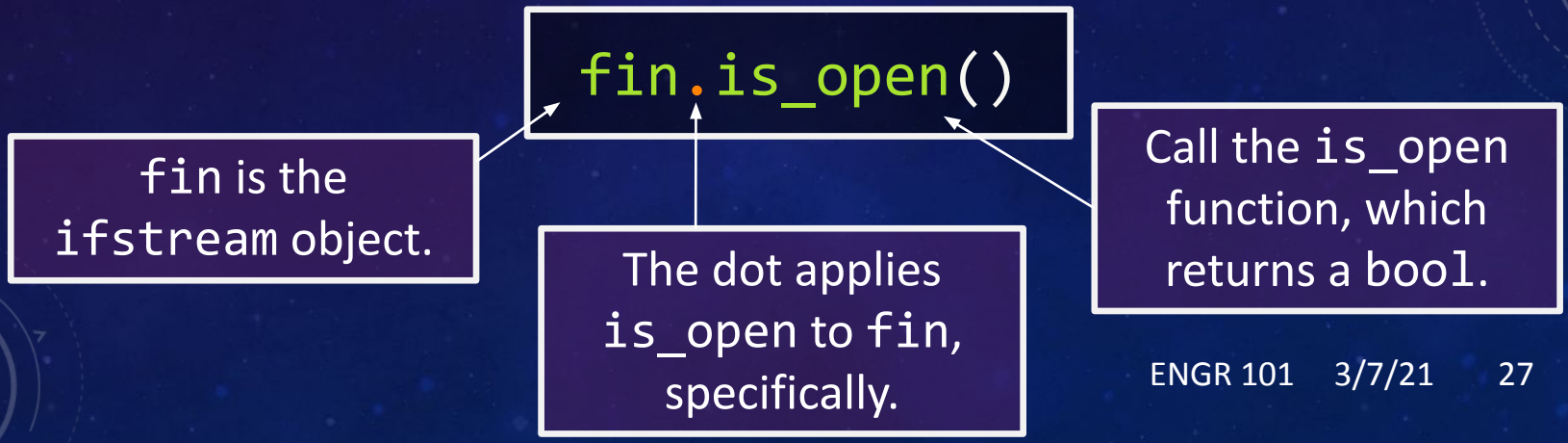
Printed to cout

```
Word 1: apple
Word 2: banana
Word 3: pear
```

Checking for Errors when Opening a File

- Sometimes, the program is unable to open a file for input.
  - e.g. It doesn't exist
  - e.g. A different program is using it
  - e.g. Your program doesn't have permission to use it

- Check whether the file was opened successfully using the `.is_open()` function, which is applied to the `ifstream`:



File Input: Example

readWordsV2.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
  ifstream fin("words.in");

  if( !fin.is_open() ) {
    cout << "Error opening file! ";
    return 1; // Leave main early
  }

  string word; // will hold input
  for(int x = 0; x < 3; ++x) {
    fin >> word;
    cout << "Word " << x << ": " << word << endl;
  }

  fin.close();
}
```

words.in

```
apple
banana
pear
```

The return value for `main` can be used as the **exit code** for the program. A nonzero value indicates an error.

Whitespace

- Each piece of input read by >> is delimited by **whitespace**.

- Whitespace includes:
  - Spaces
  - Tabs
  - Newlines

words1.in

```
apple
banana
cactus
pear
```

words2.in

```
apple   banana
   cactus
pear
```

These input files are the same to >>.

- Consecutive whitespace characters are considered as a single chunk of whitespace.

Common Pattern: Reading until the End

- In a previous example, we hardcoded the number of iterations for the input loop.
- What if we don't know this ahead of time?

words.in

```
apple
banana
cactus
dinosaur
elephant
frog
...
```

readWordsV3.cpp

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
  ifstream fin("words.in");

  string word; // will hold input
  while( fin >> word ) {
    cout << "Word: " << word << endl;
  }
  fin.close();
}
```

Strategy: Put the read operation in the condition of your loop.
It will yield `false` if you run out of input!

Break Time

We'll start again in 5 minutes.

7 min

Exercise: Word Replacement

- Write a program to open a file "dome.txt", replace each occurrence of the word "dome" with "DOME", and save the result to "dome_new.txt". (Assume words separated by spaces.)

wordReplace.cpp

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {

  // YOUR CODE HERE
  // HINT: Use both an ifstream and an ofstream
  // HINT: Use the "read until end" pattern
  // HINT: You can use == or != to compare strings

}
```

Feel free to look back through the last several slides for examples!

Lobster doesn't support file streams, so you'll have to write your code with an editor (e.g. VS Code) compile with g++.

Solution: Word Replacement

wordReplace.cpp

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
  string target = "dome";
  string replacement = "DOME";

  ifstream fin("dome.txt");
  if ( !fin.is_open() ) {
    cout << "Error opening dome.txt!" << endl;
    return 1;
  }

  ofstream fout("dome_new.txt");
  string word;
  while( fin >> word ) {
    if (word != target) { fout << word << " "; }
    else { fout << replacement << " "; }
  }
  fin.close();
  fout.close();
}
```

istreams and Types

- Streams work naturally with the built-in datatypes.
- Just use a variable of the desired type!
- The same rules for whitespace apply.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {

  ifstream fin("scores.in");
  int i;
  double d;
  string s;
  ...
}
```

```
// Expects 3, -2, 0, etc.
fin >> i;
```

```
// Expects 0.13, -2.5, .62, 8, etc.
fin >> d;
```

```
// Reads any sequence of characters
fin >> s;
```

Example: Averaging Numbers from a File

averageNumbers.cpp

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main() {

  ifstream fin("scores.in");
  if ( !fin.is_open() ) { cout << "Error!" << endl; return; }

  double total = 0; // Running total
  int count = 0; // How many numbers
  double num; // store each number as it's read

  while( fin >> num ) { // read each number until end
    total += num; // update total
    ++count; // update count of numbers
  }
  cout << (total / count) << endl; // compute/print result
  fin.close();
}
```

scores.in

```
78.3
56.41
67.3
89.7
95.2
100
...
```

`ostreams` and Types

- You can print out any of the basic datatypes using <<.
- The program will attempt to format the value in a reasonable way.
  - e.g. `strings` are printed just as they are.
  - e.g. `double` values are printed with a decimal point.

- The compiler doesn't automatically add spaces or newlines for you. You have to do this yourself!
  - Remember, `endl` represents a newline.

Floating Point Formatting with `ostreams`

- Individual streams can be configured to format `doubles` with a certain amount of decimal places.
  - e.g. `cout.precision(10);`

> `cout` will now print up to 10 decimal places.

- Other options exist for fixed precision and scientific notation, but we won't cover them in depth for 101.

1 min

A Common Mistake

- I added a bug to the code from earlier – can you find it?

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {

  // Some code omitted for brevity

  ofstream fout("dome_new.txt");
  string word;
  while( fin >> word ) {
    if (word != target) { cout << word << " "; }
    else { cout << replacement << " "; }
  }
  fin.close();
  fout.close();
}
```

Hint: You see output when you run it, but nothing goes into the output file…

Oops. We used cout instead of fout!

Naming streams

- Always using `fin` and `fout` can cause confusion, especially when working with multiple files!
- As with the other types of variables we've discussed, you should give your `ifstreams` and `ofstreams` meaningful names.

```
ifstream OGdomeText("dome.txt");
ofstream newDomeText("dome_new.txt");

ifstream scoresSource("scores.in");
ofstream scoresNorm("normalized_scores.txt");
ofstream scoresStats("statistics.txt");
ofstream summaryFile("summary.txt");
```

Disclaimer: We will continue to use `fin`/`fout` on slides because those names take up less space on the slides. Sorry! ☹

Common Pattern: Reading Multiple Pieces Together

- Let's write a program to read addresses from a file. They have three parts, separated by spaces:
  - House number, street name, district number

addresses.in

```
3825 Proxima_Lane 14
1943 Proxima_Lane 14
9304 Domey_Street 9
3321 1st_Avenue 12
```

readAddressParts.cpp

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  ifstream fin("addresses.in");
  int house; // house number
  string street; // street name
  int district; // district number
  while( fin >> house >> street >> district ) {
    ...
  }
}
```

Chaining three different read operations on each iteration.

Common Pattern: Reading a File Line-by-line

- If you want to read an entire line from a file, use the `getline` function with the istream and a string variable as arguments.
  - As usual, put the input operation (the call to `getline`) in the condition.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
  ifstream fin("input.in");

  string line; // a full line of input
  while( getline(fin, line) ) {
    ...
    ...
  }
}
```

addresses.in

```
3825 Proxima_Lane 14
1943 Proxima_Lane 14
9304 Domey_Street 9
```

readAddressByLine.cpp

On the first iteration, `line` would contain the full string `"3825 Proxima_Lane 14"`.

Download the files in
Lecture > Resources > Lecture 19 - Strings

ENGR 101 – Lecture 19

Strings

Laura Alford, Mark Guzdial, Rick Niciejewski 3/7/21

strings and the string Library

- A string is basically a sequence of characters.
  - In C++, use the `string` datatype.
- To use `string` variables or any of the built-in `string` functions, first include the `string` library.

```
#include <string>
using namespace std;

int main() {
  string s1 = "hello";
  string s2("hello");

  string s3 = "";

  string s4;
}
```

These are both valid syntax. They do the same thing.

A string with no characters is called an **empty string**.

If you don't initialize a string, it will be empty by default.

String Literals

- String literals are basically hardcoded `strings`, which are specified by a sequence of characters in double quotes ".

```
#include <string>
using namespace std;

string exclaim(string y) {
  return y + "!";
}

int main() {
  string s = "hello";
  s = s + " world";

  string s2 = exclaim('hello');

  "test" = s2;
}
```

Note: You technically don't need the `string` library to use string literals, but it would be needed for the rest of this example (e.g. to create variables with type `string`).

Error! Need double quotes.

Error! String literals can't be modified.

Concatenation

- **Concatenation** is the process of appending two strings together, one after the other.
- In C++, the + operator performs concatenation.

```
string s = "hello";
s = s + " world";
```

- The += operator updates a variable by concatenating an additional string onto it.

```
string s = "hello";
s += " world";
```

5 min

Exercise: repeat

- Write a function that repeats a string some number of times.

repeat.cpp

```
#include <iostream>
#include <string>
using namespace std;

string repeat(string s, int n) {

  // YOUR CODE HERE
}

int main() {
  string s = "abc";
  string s2 = repeat(s, 5);
  cout << s2 << endl; // "abcabcabcabcabc"

  cout << repeat("echo ", 3) << endl; // "echo echo echo "
}
```

Note: Passing the `string` parameter by value is wasteful – we'll come back to a better way to do this later in the slides.

Solution: repeat

- Write a function that repeats a string some number of times.

```
#include <iostream>
#include <string>
using namespace std;

string repeat(string s, int n) {
  string result = "";
  for(int x = 0; x < n; ++x) {
    result += s;
  }
  return result;
}

int main() {
  string s = "abc";
  string s2 = repeat(s, 5);
  cout << s2 << endl; // "abcabcabcabcabc"

  cout << repeat("echo ", 3) << endl; // "echo echo echo "
}
```

An alternate approach would be to use a while loop that counts down from n to 0.

Note: Passing the `string` parameter by value is wasteful – we'll come back to a better way to do this later in the slides.

`string` Comparison

- The `string` datatype supports the regular relational operators, based on a lexicographic (alphabetic) ordering.

  `==`, `!=`, `<`, `<=`, `>`, `>=`

- The first differing character determines the result.
- For example, the following comparisons all yield true:

```
apple < banana
apple < apps
apple < applesauce
apple > Apple
```

Same up until `l` vs. `s`

Extra letters make a string greater.

Comparison is case sensitive!

Note: We've intentionally left the quotes off this slide, so that it doesn't give the impression we are comparing string literals. See the next slide for details.

Warning!

- String literals are not what they seem.
  - It turns out string literals aren't actually of type `string`.
  - They are actually C-style strings (C being the predecessor to C++).
  - We won't get into the technical details of C-style strings in ENGR 101.

- What's the takeaway?
  - String literals can be used in most places actual `strings` can.
  - String literals can be stored into a string variable.
  - However, string literals can NOT be used in comparison (==, !=, <, …)
    - (Technically, it's fine if one side of the comparison is an actual `string` variable.)

`string` Representation

- Internally, the `string` datatype is fairly complex.
- However, for our purposes, it suffices to think of it as a sequence of values of the `char` datatype.

```
string s = "hello";
```

'h'	'e'	'l'	'l'	'o'

char Representation

- In memory, chars are actually stored as numbers.
- The number used to represent each character is determined by its ASCII code (a number 0-255):

032	sp	048	0	064	@	080	P	096	`	112	p
033	!	049	1	065	A	081	Q	097	a	113	q
034	"	050	2	066	B	082	R	098	b	114	r
035	#	051	3	067	C	083	S	099	c	115	s
036	$	052	4	068	D	084	T	100	d	116	t
037	%	053	5	069	E	085	U	101	e	117	u
038	&	054	6	070	F	086	V	102	f	118	v
039	'	055	7	071	G	087	W	103	g	119	w
040	(	056	8	072	H	088	X	104	h	120	x
041	)	057	9	073	I	089	Y	105	i	121	y
042	*	058	:	074	J	090	Z	106	j	122	z
043	+	059	;	075	K	091	[	107	k	123	{
044	,	060	<	076	L	092	\	108	l	124	|
045	-	061	=	077	M	093	]	109	m	125	}
046	.	062	>	078	N	094	^	110	n	126	~
047	/	063	?	079	O	095	_	111	o		

Implicit Conversions with char

- The numeric ASCII code is used whenever a `char` is implicitly converted to an `int`.

```
int x = 'a';
cout << x << endl; // prints 97
```

- Beware! It's easy to write code that looks like it does one thing, but actually does something else:

```
cout << ( '3' + '4' ) << endl; // prints 103
// ASCII code for '3' is 51
// ASCII code for '4' is 52
```

Special Characters

- Other ASCII codes are used to represent non-printable characters with special meanings:
  - e.g. A tab character has ASCII code 9
  - e.g. A newline character has ASCII code 10
- But how can we use these special characters? We can't exactly type them into our code as literals…

```
int main() {
  char aLetter = 'a';
  char aSpace = ' ';

  char aTab = '	';
  char aNewline = '
';
}
```

These both work fine.

These don't work.

Escape Sequences

- Some special characters (e.g. newline) would interfere with the syntax of our program.
- Instead, we can use an **escape sequence** to specify these special characters in string or character literals.

```
#include <string>
using namespace std;
int main() {
  char aLetter = 'a';
  char aSpace = ' ';

  char aTab = '\t';
  char aNewline = '\n';

  string s = "line one\nline two\nline three";
}
```

Generally, escape sequences start with the \ (backslash) character. The compiler considers the whole escape sequence as a single character.

The `length` and `size` Functions

- To get the number of characters in a `string`, use either the `length` function or the `size` function.
  - Use the `.` operator to apply these functions to a particular `string`.

```
#include <string>
using namespace std;
int main() {
  string str1 = "hello";
  cout << str1.size() << endl; // prints 5

  string str2 = "a b c ";
  cout << str2.size() << endl; // prints 6

  string str3 = "one\ntwo";
  cout << str3.length() << endl; // prints 7

  string str4 = "";
  cout << str4.length() << endl; // prints 0
}
```

Recall: `string` Representation

- Internally, the string datatype is fairly complex.
- However, for our purposes, it suffices to think of it as a sequence of values of the `char` datatype.

```
string s = "hello";
```

'h'	'e'	'l'	'l'	'o'

- Use indexing to work with the individual `chars` in a `string`…

`string` Indexing

- Just as in MATLAB, we can use indexing in C++ to access individual elements of a sequence like a `string`.
- Two key differences:
  - Indexing in C++ uses the square brackets [ ]. (rather than parentheses)
  - Indices start at 0! (In MATLAB, they started at 1.)

```
#include <string>
using namespace std;
int main() {
  string str = "hello";
  char c1 = str[1]; // c1 now has value 'e'
  char c2 = str[str.length() - 1]; // c2 now has value 'o'
  char c2 = str[str.length()]; // index 5 out of bounds!

  str[0] = 'j'; // change 1st char from 'h' to 'j'
  cout << str << endl; // now prints "jello"
}
```

Exercise: Predict the Output

4 min

```
#include <string>
using namespace std;

void func1(string str) {
  str[0] = str[str.length() - 1];
}

void func2(string &str) {
  str[0] = str[str.length() - 1];
}

int main() {
  string str1 = "hello";
  string str2 = "goodbye";

  func1(str1);
  func2(str2);

  cout << str1 << endl;
  cout << str2 << endl;
}
```

Solution: Predict the Output

```
#include <string>
using namespace std;

void func1(string str) {
  str[0] = str[str.length() - 1];
}

void func2(string &str) {
  str[0] = str[str.length() - 1];
}

int main() {
  string str1 = "hello";
  string str2 = "goodbye";

  func1(str1);
  func2(str2);

  cout << str1 << endl; // prints "hello"
  cout << str2 << endl; // prints "eoodbye"
}
```

Pass by value
Manipulating a value that is just a copy of the original value

Pass by reference
Referring to the original value, so changes are made to the original.

Break Time

We'll start again in 5 minutes.

7 min

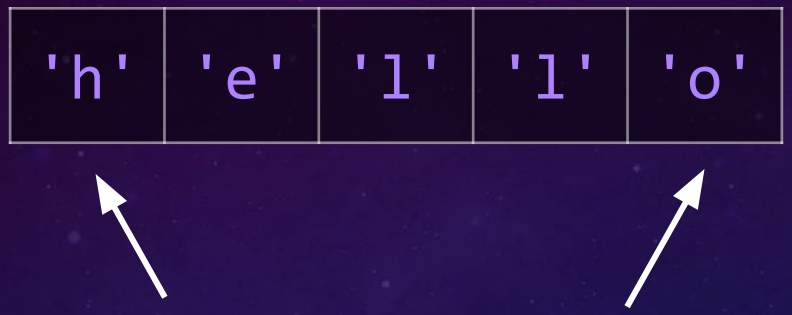
Exercise: isPalindrome

- A palindrome is a string that reads the same forward and backward.
  - For example: "racecar", "kayak", "tacocat", etc.

palindrome.cpp

```
#include <iostream>
#include <string>
using namespace std;

bool isPalindrome(string str) {

  return false; // REPLACE WITH YOUR CODE
}

int main() {
  string test1 = "racecar";
  cout << test1 << ": " << isPalindrome(test1) << endl;

  string test2 = "hello";
  cout << test2 << ": " << isPalindrome(test2) << endl;
}
```

Solution: One Strategy



start with first index
and last index

compare letters

if letters do not match, return `false`

move indices to the next letters “in” and repeat

continue to check letters while the indices don’t match

Solution: isPalindrome

```
bool isPalindrome(string str) {
  int left = 0;
  int right = str.length() - 1;
  while(left != right) {
    if (str[left] != str[right]) {
      return false;
    }
    ++left;
    --right;
  }
  return true;
}

int main() {
  string test1 = "racecar";
  cout << test1 << ": " << isPalindrome(test1) << endl;

  string test2 = "hello";
  cout << test2 << ": " << isPalindrome(test2) << endl;
}
```

This solution actually has a problem. We'll get to it shortly…

Unit Testing

- Let's upgrade our unit tests for `isPalindrome` with assertions.
- The `assert` function verifies the expected behavior of code.
- An assertion "fails" if its input is false, which indicates a bug.

```
#include <string>
#include <cassert>
using namespace std;

int main() {
  string input1 = "racecar";
  bool expected_output1 = true;
  bool actual_output1 = isPalindrome(input1);
  assert(actual_output1 == expected_output1);

  // A more concise way to write the same test
  assert(isPalindrome("racecar") == true);

  // An even more concise way to write the same test!
  assert(isPalindrome("racecar"));
}
```

To use `assert`, include the `cassert` library.

*Note – this doesn't compile quite yet, see the next slides.

Unit Testing

- We'll also move the tests to a separate file.

palindrome.cpp

```
bool isPalindrome(string str) {
  ...
}

int main() {
  string test1 = "racecar";
  cout << test1 << " " << isPalindrome(test1) << endl;
}
```

Only one main function is allowed!

palindrome_unit_tests.cpp

```
#include <string>
#include <cassert>
using namespace std;

bool isPalindrome(string str);

int main() {
  assert(isPalindrome("racecar") == true);
}
```

IMPORTANT! This function prototype is necessary to declare `isPalindrome` for this file.

- Compile and run with:

```
g++ palindrome.cpp palindrome_unit_tests.cpp -o palindrome_tests
./palindrome_tests
```

Unit Testing `isPalindrome`

- Compile and run the tests with:

```
g++ palindrome.cpp palindrome_unit_tests.cpp -o
palindrome_tests

./palindrome_tests
```

- We've added some more thorough tests:

palindrome_unit_tests.cpp

```
#include <string>
#include <cassert>
using namespace std;

bool isPalindrome(string str);

int main() {

  assert(isPalindrome("racecar")); // Positive case
  assert(!isPalindrome("hello")); // Negative case
  assert(isPalindrome("abccba")); // Special case: even length
  assert(isPalindrome("")); // empty string, technically true

  // If we get to this point, all the assertions must have passed!
  cout << "TESTS PASS" << endl;
}
```

Solution: isPalindrome

```
bool isPalindrome(string str) {
  int left = 0;
  int right = str.length() - 1;
  while(left != right) {
    if (str[left] != str[right]) {
      return false;
    }
    ++left;
    --right;
  }
  return true;
}

int main() {
  string test1 = "racecar";
  cout << test1 << ": " << isPalindrome(test1) << endl;

  string test2 = "hello";
  cout << test2 << ": " << isPalindrome(test2) << endl;
}
```

If the string has even length, `left` and `right` "skip" over each other.
Use `left < right` as the condition instead.

Indexing out of bounds

- C++ will gladly let you index out of bounds.
  - There's no immediate error like we would get in MATLAB!
- This can lead to very weird behavior…

```
#include <string>
using namespace std;
int main() {
  int x = 3;
  string str1 = "sad";
  string str2 = "frog";

  str2[str2.length()] = 'm'; // out of bounds = undefined behavior

  // At this point, all bets are off!
  cout << str1 << endl; // maybe prints "mad"... maybe not!
}
```

Note: In reality, memory won't be arranged quite like what's shown here. It's unlikely that precisely the behavior shown here would result.

Indexing with the at Function

- You can also use the `at` function to index into a string.
  - The advantage of `at` is that it checks whether the provided index is within bounds and if not, causes an error immediately.

```
#include <string>
using namespace std;

int main() {
  int x = 3;
  string str1 = "sad";
  string str2 = "frog";

  str2.at(str2.length()) = 'm'; // out of bounds = ERROR MESSAGE

  // Code will never even get to this point
  cout << str1 << endl;
}
```

- Of course, the extra check means `.at()` is slightly slower than `[]`

Other `string` functions you may find useful…

- `substr`
- `erase`
- `find`

- Let's see some examples

string substr

- Use the `substr` function to get a copy of a part of a `string`.

`str.substr(start, length)`

```
#include <iostream>
#include <string>
using namespace std;

int main() {

  string str = "hello world!";
  cout << str.substr(0, 5) << endl; // "hello"

  cout << str.substr(7, 2) << endl; // "or"

  cout << str.substr(1) << endl; // "ello world!"
}
```

If no length is specified, goes to the end of the `string`.

string find

- Use the `find` function to get the index at which a substring first occurs in an original string.

```
str.find(query)
str.find(query, offset)
```

```
int main() {
  string str = "red fish blue fish one fish two fish";

  int x = str.find("fish"); // 4
  int y = str.find("fish", 5); // 14
  int z = str.find("fish", y + 1); // 23

  if (str.find("banana") == string::npos) {
    cout << "substring not found!" << endl;
  }

}
```

If the substring is not found, `find` returns the special value `string::npos`.

string erase

- To remove part of a `string`, use the `erase` function.

```
str.erase(start, length)
```

```
int main() {
  string str = "this is a string";

  str.erase(10, 2); // "this is a ring"

  str.erase(7, string::npos); // "this is"

  str.erase(4); // "this"

}
```

If the second parameter is `string::npos`, it will erase all until the end of the `string`.