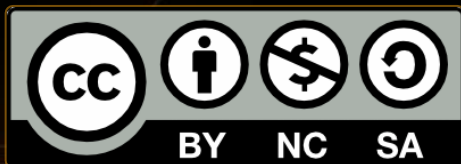


Strings and Streams

Representing Text, Working with Streams from Files and Strings



Georgi Georgiev

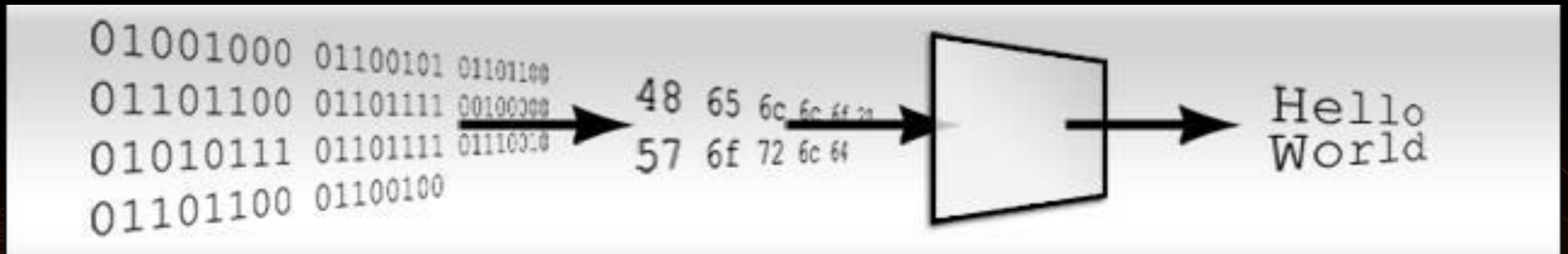
A guy that knows C++

000 :	013 :.	026 :→	039 :	052 :.	065 :H	078 :n	091 :L	104 :n	117 :u
001 :☒	014 :f	027 :←	040 :C	053 :5	066 :B	079 :O	092 :\\	105 :i	118 :v
002 :☒	015 :*	028 :L	041 :)	054 :6	067 :C	080 :P	093 :J	106 :j	119 :w
003 :♥	016 :▶	029 :♦	042 :*	055 :7	068 :D	081 :Q	094 :^	107 :k	120 :x
004 :♦	017 :◀	030 :▲	043 :+	056 :8	069 :E	082 :R	095 :_	108 :l	121 :y
005 :♣	018 :↑	031 :▼	044 :,	057 :9	070 :F	083 :S	096 :`	109 :m	122 :z
006 :♣	019 :!!	032 :	045 :-	058 ::	071 :G	084 :T	097 :a	110 :n	123 :{
007 :♦	020 :¶	033 :!	046 :.	059 :;	072 :H	085 :U	098 :b	111 :o	124 :
008 :☐	021 :§	034 :"	047 :/	060 :<	073 :I	086 :V	099 :c	112 :p	125 :}
009 :^	022 :	035 :#	048 :@	061 :-	074 :T	087 :h	100 :d	113 :e	126 :~

Table of Contents

1. Representing Text in Computers
2. C++ Text Representation
 1. C-Strings
 2. The **std::string** class
3. Streams
 - The **std::stringstream**
 - Streaming to/from Files





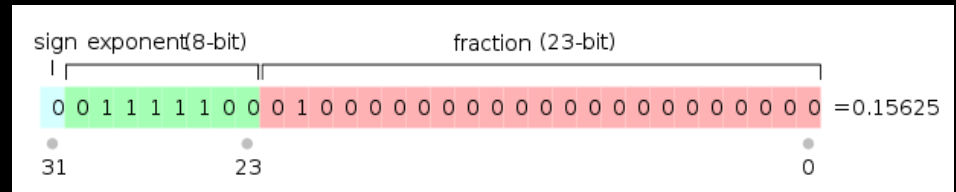
Representing Text in Computers

Bytes, Code Points, Encoding

Representing Text in Computers

- Data is bytes of 1s and 0s
 - Interpreted in different ways
 - Interpretation & size = data type
- Ways we interpret bytes:
 - Binary number -> integer types
 - IEEE754 -> floating-point types
 - Binary "code point" -> char types
- Characters (text) is just another interpretation of binary data

$$\begin{array}{cccccccc} 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ \hline 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ \hline 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1 \\ = 155 \end{array}$$


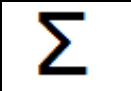

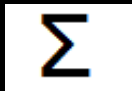


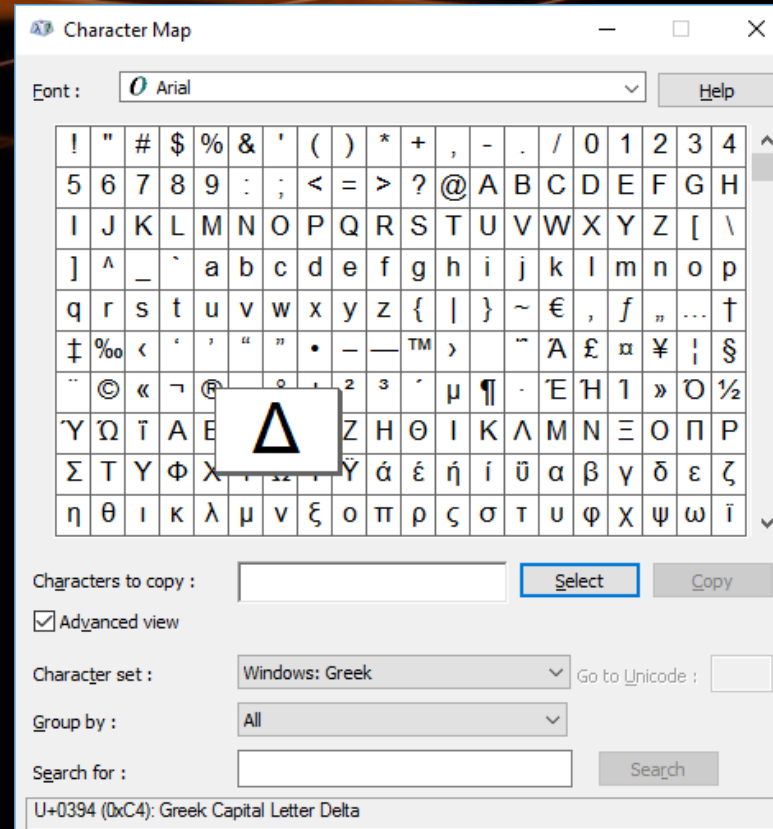
000:	013:ƒ	026:→	039:’	052:4	065:Ǽ	078:Ń	091:İ	104:h	117:u
001:☒	014:ſ	027:←	040:(053:5	066:B	079:0	092:↘	105:i	118:v
002:☼	015:⚡	028:↵	041:)	054:6	067:C	080:P	093:J	106:j	119:w
003:♥	016:►	029:⦿	042:⌘	055:7	068:D	081:Q	094:ˆ	107:k	120:x
004:♦	017:◀	030:▲	043:+	056:8	069:E	082:R	095:ˉ	108:l	121:y
005:♣	018:‡	031:▼	044:,	057:9	070:F	083:S	096:˘	109:m	122:z
006:♠	019:!!	032:⬆	045:-	058::	071:G	084:T	097:a	110:n	123:{
007:•	020:¶	033:†	046:.	059:;	072:H	085:U	098:b	111:o	124:
008:☐	021:§	034:”	047:✓	060:<	073:I	086:V	099:c	112:p	125:}
009:◦	022:⌂	035:#	048:0	061:=	074:J	087:W	100:d	113:q	126:~
010:☑	023:‡	036:\$	049:1	062:>	075:K	088:X	101:e	114:r	127:Δ
011:♂	024:↑	037:℥	050:2	063:?	076:L	089:Y	102:f	115:s	
012:♀	025:↓	038:&	051:3	064:e	077:M	090:Z	103:g	116:t	

Representing Text in Computers

- Text is just a sequence of characters
- A character (letter, symbol, etc.) is just one or more bytes
 - The binary representation of a **number**
 - Interpreted as a **code point** from a **character set (charset)**
- Character set – a group of characters (Latin, Cyrillic, etc.)
- Code point – unique number assigned to a character in a charset
 - E.g. ASCII code point **65 (0x41)** is '**A**' (English capital letter A)

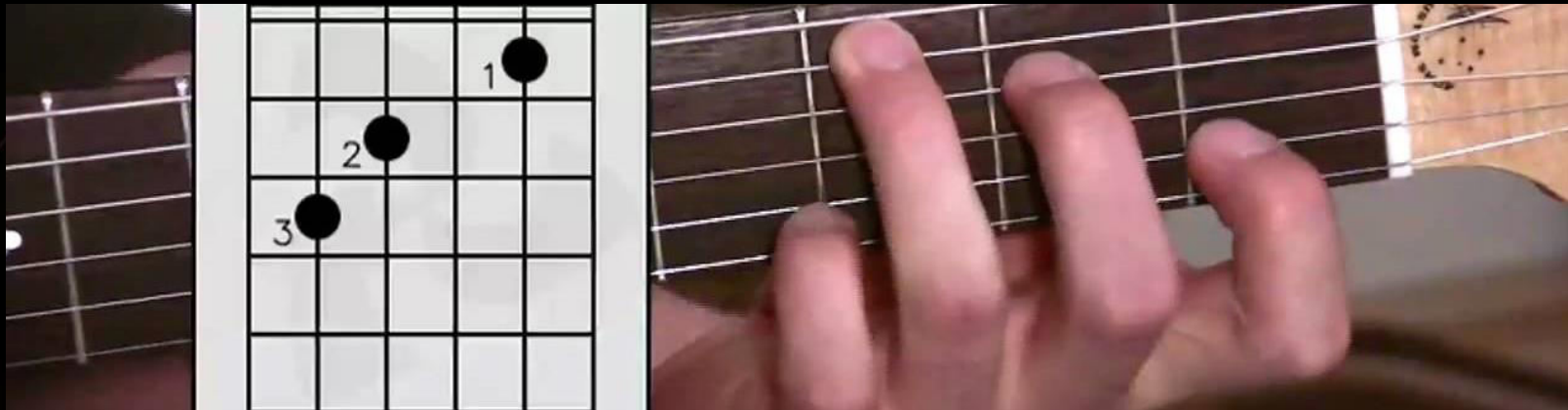
Character Sets & Unicode

- ASCII is the base charset – code points from **0** to **127**
 - English letters, digits, punctuation, control symbols (e.g. tab)
- "Extended ASCII" – code points from **128** to **255**
 - Different charsets use those codepoints for different characters
 - E.g. Windows: Cyrillic code point **211** (**0xD3**) is **у** 
 - But Windows: Greek code point **211** (**0xD3**) is **Σ** 
- Unicode unifies charsets to represent all the world's characters
 - E.g. **у**  is **1059** (**U+0423**) and **Σ**  is **931** (**U+03A3**)



Representing Text in Computers

LIVE DEMO



C++ Text Representation

C-Strings and `std::string` class

C++ Text Representation

- C++ has good native support for the ASCII charset
 - **char** data type (usually) covers code points 0 to 255
 - We'll discuss Unicode later
- Text types (sequences of characters) are called **strings**
- C++ has two standard ways of working with text
 - Character arrays, aka C-Strings (legacy from the C language)
 - The C++ **std::string** – a "smart" wrapper of a C-String

char Arrays, aka C-Strings

- An array of char, e.g. `char str[]`, with the following rules:
 - Should be null-terminated, i.e. end with `'\0'`, which is `char(0)`
 - `'\0'` counts as an element – it affects array size
- Null-terminator tells C++ where the string ends
 - C++ arrays don't know their size, remember?
- Char arrays with NO null-terminator are NOT used as strings
 - E.g. don't use `cin`, `cout`, or C-String functions
 - Can still use like an array of any other data type

Using C-Strings

- Initialization can happen with array initializer or literal
 - If using normal array initializer, don't forget the `'\0'` at the end

```
char text[16] = {'C', '+', '+', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g', '\0'};  
char sameText[] = {'C', '+', '+', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g', 0};  
char sameTextAgain[] = "C++ Programming";  
char sameTextYetAgain[16] = "C++ Programming";
```

- **cin** & **cout** can directly write to and read from C-Strings

- **cout** prints until it reaches `'\0'`
- **cin** works correctly only if array can fit entered data

```
char arr[100];  
cin >> arr;  
cout << arr << endl;
```

C-Strings Basics

LIVE DEMO

Quick Quiz

TIME:

- What will the following code print?

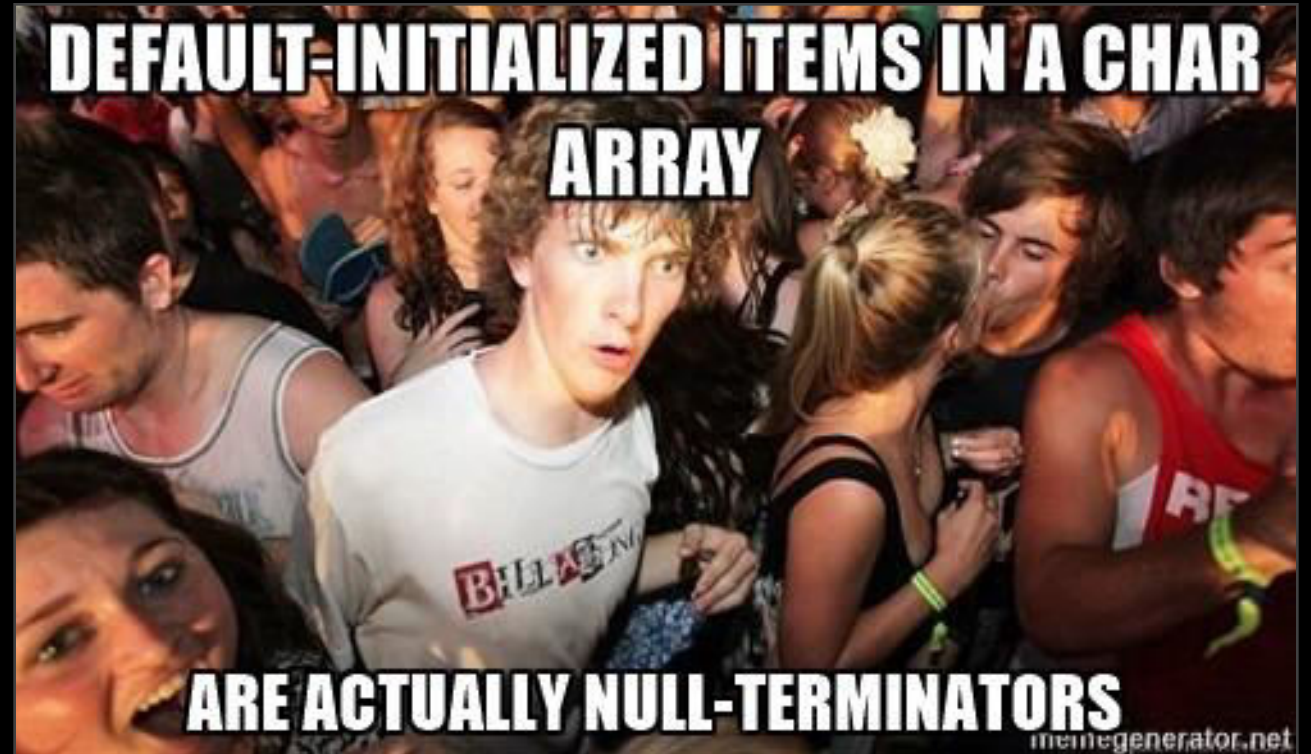
```
char line1[4] = {'a', 'b', 'c'};  
char line2[] = {'d', 'e', 'f'};  
cout << line1 << endl;  
cout << line2 << endl;
```

- a) It won't – there will be a compile-time error
- b) behavior is undefined
- c) First line **abc**
Second line **def**
- d) First line **abc**, second line is undefined

C++ PITFALL: NON-TERMINATED C-STRINGS

Most C-String compatible code will expect a null-terminator when using a C-String. If there is none, the code can't know where the string ends – it just continues on until it reaches null somewhere in memory

But, if you give less items to array initializer than array size is – the remaining items get default values, which for char is exactly the null-terminator



Some C-String Built-in Functions

- C-String functions are defined in the `<cstring>` header
- **strcat(destination, source)**
 - Appends (concatenates) **source** C-String into **destination** C-String
 - **destination** needs to be long enough for source + null-terminator
- **strlen(str)**
 - Returns length of C-String in **str** (based on the null-terminator)
- **strstr(str, search)**
 - Returns the address (pointer) of **search** in **str**, **NULL** if not found
 - **int index = strstr(str, search) - str;** gets you the index

C-String Built-in Functions

LIVE DEMO

The `std::string` Class

The C++ Way for Working with Text

The `std::string` Class

- The C++ **string** encapsulates a null-terminated C-String
 - **#include<string>**
- Declare like a normal variable, e.g. **string s;**
 - Empty ("", size 0) if only declared (it gets default-initialized)
 - Can be initialize with C-String or string literal

```
string theFoxPart = "the quick brown fox";  
string theActionPart("jumps over");  
char dogPartCString[] = "the lazy dog";  
string sentence = theFoxPart + string("---") +  
                  theActionPart + string(3, '-')  
                  + dogPartCString;
```

std::string Basics

- C++ Strings can be used with **cin/cout**
- **size() & length()** return the number of **chars**
- The **[]** operator is supported – similar to **[]** for a char array
- The **+** operator concatenates two strings

```
string name;  
cin >> name; cout << name;
```

```
string hello = "hello";  
for (int i = 0; i < hello.size(); i++)  
    cout << hello[i] << endl;
```

```
hello[1] = 'a';  
cout << hello << endl; //hallo
```

```
string helloName = hello + string(" ") + name;  
cout << helloName << endl; // e.g. "hello George"
```

`std::string` Basics

LIVE DEMO

std::string Comparisons and Search

- Two strings can be compared with any comparison operator
 - operators `<`, `<=`, `==`, `>=`, `>` compare the strings lexicographically

```
string s1 = "cat", s2 = "canary";  
if (s1 < s2) { cout << s1 << " is before " << s2 << endl; }  
else        { cout << s1 << " is after " << s2 << endl; }
```

- str.find(search)** returns the index of **search** in the **str**
 - If **search** is not found, returns the **string::npos** value (-1)

```
cout<<"nar"<<" at index "<<s1.find("nar")<<" in "<<s2;
```

`std::string`

Comparisons and Search

LIVE DEMO

std::string Find All Occurrences

- The `find(search, index)` overload takes a start index
 - The search starts from that index (ignores results before it)
 - `string s = "aha"; cout << s.find("a", 1);` prints 2
- We can use this to search all occurrences of a substring
 - Each time search from after the last index where we found it

```
string str = "canary";
int foundIndex = str.find("a");
while (foundIndex != string::npos) {
    cout << foundIndex << endl;
    foundIndex = str.find("a", foundIndex + 1);
}
```

std::string Substring, Erase, Replace

- `substr(index, length)` returns a new string
 - With **length** characters, starting from **index**
 - `string s = "abc"; cout << s.substr(1,2);` prints **bc**
- `erase(index, length)` changes a string by removing chars
 - Removes **length** characters, starting from **index**
 - `string s = "abc"; s.erase(1,2); cout << s;` prints **a**
- `replace(index, length, str)` changes a string by replacing
 - Characters in **[index, index + length)** replaced by **str**
 - `string s = "abc"; s.replace(1,2,"cme"); cout << s;` prints **acme**

`std::string`

Substring, Erase, Replace

LIVE DEMO

Supporting Unicode in C++

- **char** supports ASCII, **string** is a **char** array, no Unicode there
- **wchar_t** and **wstring**
 - Variants of **char** & **string** that support system's max code point
 - **wchar_t** on Unicode systems is 32-bit,
 - But on Windows **wchar_t** is 16-bit (UTF-16)
- C++11 adds **char16_t**, **char32_t**, **u16string** & **u32string**
- Built-in support is not very good – storing is ok, operations not
- Best approach: use external libraries – QT, ICU, UTF8-CPP, etc.

Streams, Reading by Line, File I/O

`std::stringstream`, File Streams

- Streams offer an abstraction over incoming/outgoing data
 - E.g. **cin** and **cout** are abstractions of the console input/output
- Practically speaking, streams are ways of reading/writing data
- A stream can be constructed for any type of data container
 - Arrays, strings, memory
 - Files, network connections, the keyboard buffer, etc.

The `std::stringstream`

- A stream that works on a string (instead of the console or a file)
- `#include<sstream>`
- Can read data from a string, can write data to a string
 - There are limited `istringstream/ostringstream` versions that only read/write respectively
- Useful for working on a string “word-by-word”
 - E.g. reading in numbers from a string
 - E.g. creating a string with text and numbers

Reading with `std::istream`

- `istream` is a limited `stringstream` than only reads
 - If you only want to read, use it instead of `stringstream`
- Initialize `istream` by giving it a `string` to read from

```
string str = "3 -2";  
istream numbersStream(str);
```

- From then on, use the stream just like `cin`

```
int num1, num2;  
numbersStream >> num1 >> num2;  
int sum = num1 + num2;
```

Writing with `std::ostringstream`

- `ostringstream` is a limited `stringstream` than only writes
- Initialize `ostringstream` like a normal variable

```
ostringstream stream;
```

- From then on, use the stream just like `cin`

```
stream << "The sum is " << num1 + num2 << endl;
```

- To get the string when you're done, call `str()`

```
cout << stream.str();
```

Using `std::stringstream`

LIVE DEMO

Reading with `getline()` and Streams

- `getline(stream, targetStr)` reads an entire line of text
 - Or until a delimiter **char** (additional parameter) is reached
 - From the provided **stream** and puts it into **targetStr**
 - Avoid mixing `cin>>` and `getline(cin,...)`
<http://stackoverflow.com/a/18786719>

```
istringstream in("a word");  
  
string line;  
getline(in, line);  
cout << line << endl; // a word
```

```
istringstream in("a.word");  
  
string line;  
getline(in, line, '.');  
cout << line << endl; // a
```


Parsing Numbers from a Line

- **getline()** already gives us the line as a string
- Streams allow us to read strings/numbers separated by spaces
- How do we know when to stop?
 - Streams can be used as a **bool** value
 - A stream is **true** if it still has something to read
 - A stream is **false** if the input ended, or if there was an error

Parsing Numbers from a Line

- Read the line from **cin** into a **string** with **getline()**
- Create an **istringstream** over that **string**
- Read numbers from the stream while the stream is **true**
 - Add numbers to a **vector** to use them later

```
string line; getline(cin, line);
istringstream lineStream(line);
vector<int> numbers;
int currentNumber;
while (lineStream >> currentNumber) {
    numbers.push_back(currentNumber);
}
```

Parsing Numbers from a Line

LIVE DEMO

Streams to and from Files

- We saw that streams work the same way
 - Regardless of whether they are over the console, or a string
 - Same goes for files – you just have to create a file stream
- **#include<fstream>**
- **ifstream** is for reading, **ofstream** is for writing
- Text reading/writing with same operators, functions, concepts
 - **<<** for writing, **>>** for reading, **getline()** reads line, etc.
 - Can be used as **bool** just like **cin**, **cout** and **stringstream**

Using Streams to/from Files

- Declare the stream and open the file

- Input streams expect the file to exist (enter error state otherwise)
- Output streams create or overwrite the file on opening. There are parameters to tell the stream to append instead

```
ifstream input;  
input.open("input.txt");  
int a, b;  
input >> a >> b;  
input.close();  
  
ofstream output;  
output.open("output.txt");  
  
output << a + b << endl;  
output.close();
```


Using Streams to/from Files

- Declaration and opening can be shortened

```
ifstream input("input.txt");  
int a, b;  
input >> a >> b;  
input.close();
```

```
ofstream output("output.txt");  
output << a + b << endl;  
output.close();
```

- **close()** is automatically called when stream goes out of scope
 - i.e. when the declaring block ends
 - Still, you should close if you're not using the stream
- To make an output stream append instead of overwrite:
 - **ofstream output("output.txt", fstream::app);**

Summary

- Text is a sequence of bytes interpreted by special rules
- C++ has two standard ways of working with text
 - **std::string** is the C++ way for working with text
 - Knows size, has special operators and utility functions
 - C-Strings (**char** arrays) are the legacy C approach
- Streams are abstractions for writing/reading data
- Many more **string** & stream functions we didn't cover
 - <http://www.cplusplus.com/reference/>, <http://en.cppreference.com>



Strings and Streams



Questions?