Lecture 3: Strings

CS 106B: Programming Abstractions

Spring 2020, Stanford University Computer Science Department

Lecturers: Chris Gregg and Julie Zelenski

Reading: Programming Abstractions in C++, Chapters 3 & 4

Slide 2

Announcements

- Section signups opened yesterday and will be open until Sunday, April 12 at 5pm PDT. Sign up on the CS198 Website (https://cs198.stanford.edu/cs198/auth/default.aspx). **All listed section times are in PDT**.
- Lecture Format
    - We are currently working on finding a way to export the live Q&A logs and post them on the course website.
    - For in-lecture questions, we want to start incorporating more live questions directly from students! If you want to ask a question, you have two options:
        - Raise your virtual hand and we will unmute you and call on you
        - Type your question in the Q&A window followed with "(ask live)" and we can give you a chance to ask your question live

- Assignment 0 is due today, Friday April 10 PDT
- Assignment 1 will be released on Saturday early-afternoon PDT. Keep your eyes on the course website for an announcement once it's released!

Slide 3

Today's Topics

- Strings
  - C++ strings vs C strings
  - Characters
  - Member Functions
  - Stanford Library extensions
  - `char`
  - The `<cctype>` library

Slide 4

Strings in C++

Not this type of string:

or this one:

```
#include<string>
...
string s = "hello";
```

- A string is a sequence of characters, and can be the empty string: ""
- In C++, a string has "double quotes", not single quotes:
  - `"this is a string"`
  - `'this is not a string'`
- Strings are similar to Python and Java strings, although the functions have different names and in some cases different behavior.
- The biggest difference between a Python or Java string and a C++ string is that C++ strings are mutable (changeable).
- The second biggest difference is that in C++, we actually have two types of strings (more on that in a bit)

Slide 5

Strings and Characters

Strings are made up of *characters* of type **char**, and the characters of a string can be accessed by the index in the string (this should be familiar):

```
string stanfordTree = "Fear the Tree";
```

index:	0	1	2	3	4	5	6	7	8	9	10	11	12
character:	'F'	'e'	'a'	'r'	' '	't'	'h'	'e'	' '	'T'	'r'	'e'	'e'

```
char c1 = stanfordTree[3];    // 'r'
char c2 = stanfordTree.at(2); // 'a'
```

- Notice that **char**s have single quotes and are limited to one ASCII character. A space **char** is `' '`, not `''` (in fact, `''` is not a valid char at all. It is hard to see on the slide, but there is an actual space character between the single quotes in a valid space char, and there is no space in the not-valid example)
- There are two ways to loop through the characters in a string:

```
for (int i = 0; i < stanfordTree.length(); i++) {
    cout << i << " : '" << stanfordTree[i] << "'" << endl;
}
cout << endl;

for (char c : stanfordTree) {
    cout << "'" << c << "'" << endl;
}
cout << endl;
```

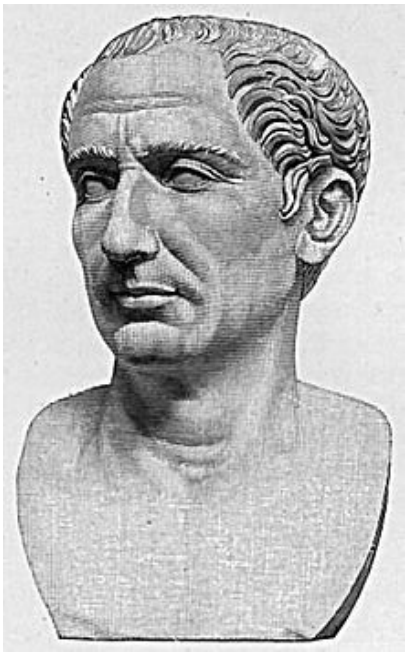
Output:

```
0 : 'F'
1 : 'e'
2 : 'a'
3 : 'r'
4 : ' '
5 : 't'
6 : 'h'
7 : 'e'
8 : ' '
9 : 'T'
10 : 'r'
11 : 'e'
12 : 'e'

'F'
'e'
'a'
'r'
' '
't'
'h'
'e'
' '
'T'
'r'
'e'
'e'
```

Slide 6

ASCII



Char	Value	Char	Value	Char	Value
(sp)	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
-	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(del)	127

The *Caesar Cipher* is an encryption formed by *shifting* letters to the left or right in an alphabet, and re-encoding a phrase with the new letters. For a Caesar Cipher of +3, all 'A's would become 'D's, all 'B's would become 'E's, etc. At the end of the alphabet, the letters wrap, so 'X's become 'A's, 'Y's become 'B's, and 'Z's become 'C's. Julius Caesar himself is said to have used this encryption scheme to discuss battle plans with his generals, and it was (in his time) never decrypted.

- Characters have a numerical representation, as shown in the **ASCII** table above.
  - **cout << (int) 'A' << endl; // 65**
  - This means you can perform math on characters, but you need to be careful:

```
string plainText = "ATTACK AT DAWN";
string cipherText = "";
int key = 5; // caesar shift by five

// only works for uppercase!
for (int i=0;i<(int)plainText.length();i++) {
    char plainChar = plainText[i];
    char cipherChar;
    if (plainChar >= 'A' && plainChar <= 'Z') {
        cipherChar = plainText[i] + key;
        if (cipherChar > 'Z') {
            cipherChar -= 26; // wrap back around
        }
    } else {
        cipherChar = plainChar;
    }
    cipherText += cipherChar;
}
cout << "Plain text:  " << plainText << endl;
cout << "Cipher text: " << cipherText << endl;
```

Output:

```
Plain text:  ATTACK AT DAWN
Cipher text: FYYFHP FY IFBS
```

Slide 7

The **<cctype>** library

- **#include<cctype>**
- This library provides functions that check a single **char** for a property (e..g, if it is a digit), or return a **char** converted in some way (e.g., to uppercase)
  - **isalnum** : checks if a character is alphanumeric
  - **isalpha** : checks if a character is alphabetic
  - **islower** : checks if a character is lowercase
  - **isupper** : checks if a character is an uppercase character
  - **isdigit** : checks if a character is a digit
  - **isxdigit** : checks if a character is a hexadecimal character
  - **iscntrl** : checks if a character is a control character
  - **isgraph** : checks if a character is a graphical character
  - **isspace** : checks if a character is a space character
  - **isblank** : checks if a character is a blank character
  - **isprint** : checks if a character is a printing character
  - **ispunct** : checks if a character is a punctuation character
  - **tolower** : converts a character to lowercase
  - **toupper** : converts a character to uppercase
- Examples:

```
string mixed = "ab80c3d27";
cout << "The digits in " << mixed << ": " << endl;
for (int i = 0; i < mixed.length(); i++) {
  if (isdigit(mixed[i])) {
      cout << mixed[i] << endl;
  }
}
string s = "my string";
for (int i = 0; i < s.length(); i++) {
  s[i] = toupper(s[i]);
}
cout << "Now " << s << " is all UPPERCASE." << endl;
```

Slide 8

String Operators

- As in Python and Java, you can concatenate strings using **+** or **+=**

```
string s1 = "Chris";
string s2 = s1 + "Gregg"; // s2 == ChrisGregg
```

- Like in Python (but *unlike*) in Java, you can compare strings using relational operators:

```
string s3 = "Zebra";
if ((s1 < s3) && (s3 != "Walrus")) {
    cout << s1 << " < " << s3 << endl;
    cout << "letters earlier in the alphabet are " << endl
         << "less than letters later in the alphabet."
         << endl;
}

cout << endl;

s1[0] = tolower(s1[0]); // s1 now == "chris"
if (s1 > s3) {
    cout << s1 << " > " << s3 << endl;
    cout << "UPPERCASE letters are less than LOWERCASE letters"
         << endl;
}
```

Output:

```
Chris < Zebra
letters earlier in the alphabet are
less than letters later in the alphabet.

chris > Zebra
UPPERCASE letters are less than LOWERCASE letters
```

- Unlike in Python and Java, strings are mutable and can be changed (!):

```
s3.append("Giraffe"); // s2 is now "ZebraGiraffe"
s3.erase(4,3); // s2 is now "Zebrraffe" (which would be a very cool animal)
s3[5] = 'i'; // s2 is now "Zebrriffe"
s3[9] = 'e'; // BAD!!!1! PROGRAM MAY CRASH! POSSIBLE BUFFER OVERFLOW! NO NO NO!
```

- Unlike in Python and Java, C++ does not bounds check for you! The compiler doesn't check for you, and Qt Creator won't warn you about this. We have entered the scary territory of *you must know what you are doing*. Buffer overflows are a critical way for viruses and hackers to do their dirty work, and they can also cause hard to track down bugs.

- The following functions are part of the string class, and can be useful:
  - **`s.append(str)`** : add text to the end of a string
  - **`s.compare(str)`** : return **`-1`** , **`0`** , or **`1`** depending on relative ordering
  - **`s.erase(index, length)`** : delete text from a string starting at given index

- `s.find(str)`
  `s.rfind(str)` : first or last index where the start of `str` appears in this string (returns `string::npos` if not found)
- `s.insert(index, str)` : add text into a string at a given index
- `s.length() or s.size()` : number of characters in this string
- `s.replace(index, len, str)` : replaces `len` chars at given index with new text
- `s.substr(start, length) or s.substr(start)` : the next length characters beginning at `start` (inclusive); if length omitted, grabs till end of string

---

Slide 9

C++ -vs- C strings

- C++ has (confusingly) two kinds of strings:
  - C strings (char arrays), inherited from the C language
  - C++ strings (string objects), which is part of the standard C++ library.
  - When possible, declare C++ strings for better usability (you will get plenty of C strings in CS 107!)
- Any string literal such as "hi there" is a C string.
- C strings don't have member functions, and you must manipulate them through regular functions. *You also must manage the memory properly – this is SUPER IMPORTANT and involves making sure you have allocated the correct memory – again, this will be covered in detail in CS 107.*
  - E.g., C strings do not have a `.length()` function (there are no member functions, as C strings are not part of a class.
- You can convert between string types:
  - `string("text");` converts C string literal into C++ string
  - `string s = someCStr;` converts a C string into a C++ string
  - `string.c_str()` returns a C string out of a C++ string

Slide 10

C string issues

- Does not compile! C strings can't be concatenated with `+`

```
string hiThere = "hi" + "there";
```

- These three all compile and work properly.

```
string hiThere = string("hi") + "there";
string hello = "hi";
hello += "there";
```

- Bug: sets `n` to the *memory address* of the C string "42" (ack!). Qt Creator will produce a warning.

```
int n = (int) "42";
```

- Works – this explicitly converts "42" to a C++ string, and then uses the *Stanford* library, **`stringToInteger()`** to convert to an **`int`**.

```
int n = stringToInteger("42");
```

Slide 11

More C string issues

- Both bugs. Produces garbage, not **`"hi?"`** or **`"hi42"`** (memory address stuff).

```
string hiQuestion = "hi" + '?'; // C-string + char
string hi41 = "hi" + 41;  // C-string + int
```

- Does work because of the empty C++ string at the beginning

```
string okayHighQuestion = string("") + "hi" + '?';
```

- Works, because of auto-conversion.

```
string howdy = "hi";
howdy += '?'; // "hi?", char '?' is converted to string
```

- Adds character with ASCII value 41, ')', doesn't produce "hi?41".

```
s += 41; // "hi?)"
```

- Works, because of conversion from **`int`** to **`string`** using a function.

```
s += integerToString(41); // "hi?41"
```

Slide 12

Mystery Function! What is the output?

```
void mystery(string a, string &b) {
    a.erase(0,1);
    b += a[0];
    b.insert(3, "FOO");
}

int main() {
    string a = "Stanford";
    string b = "Tree";
    mystery(a,b);
    cout << a << " " << b << endl;
    return 0;
}
```

Answer:

Solution

Stanford TreFOOet

Slide 13

The Stanford String library

```
#include "strlib.h"
```

These are *not* string class functions.

- **`endsWith(str, suffix)`**
  `startsWith(str, prefix)` : returns **`true`** if the given string begins or ends with the given prefix/suffix text
- **`integerToString(int)`**
  `realToString(double)`
  `stringToInteger(str)`
  `stringToReal(str)` : returns a conversion between numbers and strings
- **`equalsIgnoreCase(s1, s2)`** : **`true`** if **`s1`** and **`s2`** have same **`chars`** , ignoring casing
- **`toLowerCase(str)`**
  `toUpperCase(str)` : returns an upper/lowercase version of a string
- **`trim(str)`** : returns string with surrounding whitespace removed
- Use as follows (remember, not member functions of the **`string`** class!):

```
if (startsWith(nextString, "Age: ")) {
  name += integerToString(age) + " years old";
}
```

Slide 14

String recap

- C++ has both C strings and C++ strings. Both are, under the hood, simply arrays of characters. C++ strings handle details for you automatically, C-strings do not.
- C++ strings are much more functional and easier to use
- Many times (but not always), C-strings auto-convert to C++ strings when necessary
- Characters are single-quoted, single-character ASCII numerical values (be careful when applying arithmetic to them)
- C++ strings have many functions you can use, e.g., **`s.length()`** and **`s.compare()`**
- The Stanford library also has some extra string functions, which are not part of the string class, but are helpful

---

Slide 15

References and Advanced Reading

- References (in general, not the C++ references!):
  - Chapter 3 of Textbook (/class/cs106b/resources/textbook.html)
  - **`<cctype>`** functions (http://en.cppreference.com/w/cpp/header/cctype)
  - Caesar Cipher (https://en.wikipedia.org/wiki/Caesar_cipher)
- Advanced Reading:
  - C++ strings vs C strings (http://cs.stmarys.ca/~porter/csc/ref/c_cpp_strings.html)
  - String handling in C++ (https://en.wikipedia.org/wiki/C%2B%2B_string_handling)
  - Stackoverflow: Difference between **`string`** and **`char[]`** types in C++ (http://stackoverflow.com/questions/1287306/difference-between-string-and-char-types-in-c)