CS 106B
Lecture 4: C++ Strings

Thursday, June 29, 2017

Programming Abstractions
Summer 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 3 - 4

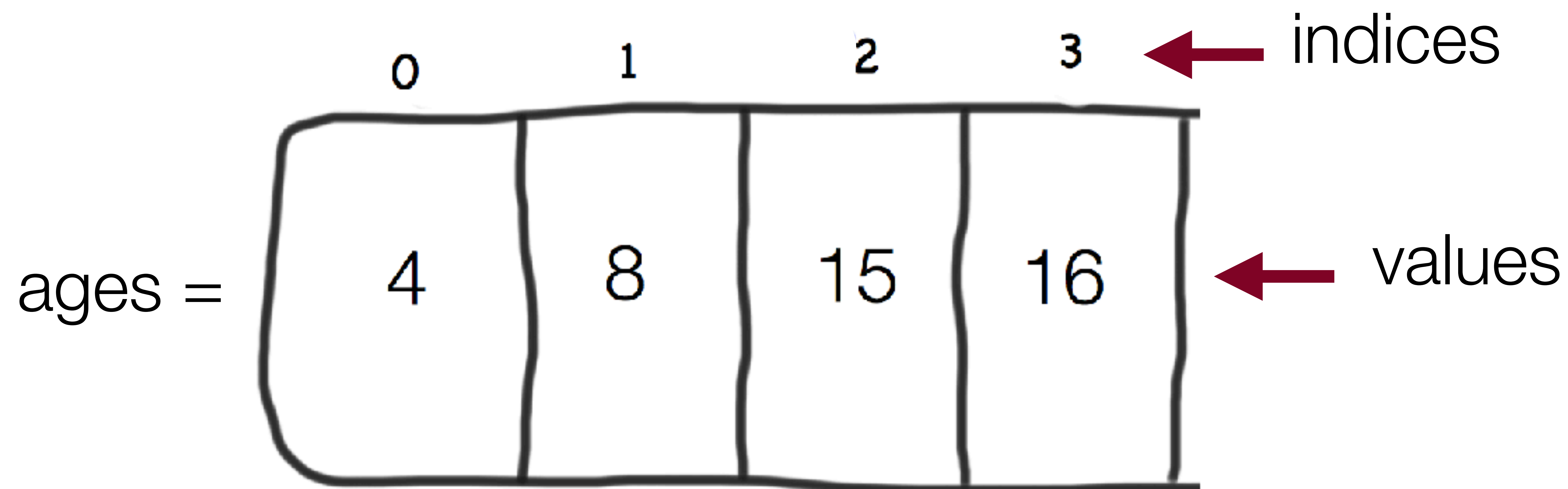
Today's Topics

- Logistics:
  - YEAH Hours video posted

- Vector review and Big-O with Vectors
- Strings
  - C++ strings vs C strings
  - Characters
  - Member Functions
  - Stanford Library extensions
  - Char and <cctype>

Vectors and Big O

As we discussed yesterday, a Vector is simply an array under the hood:



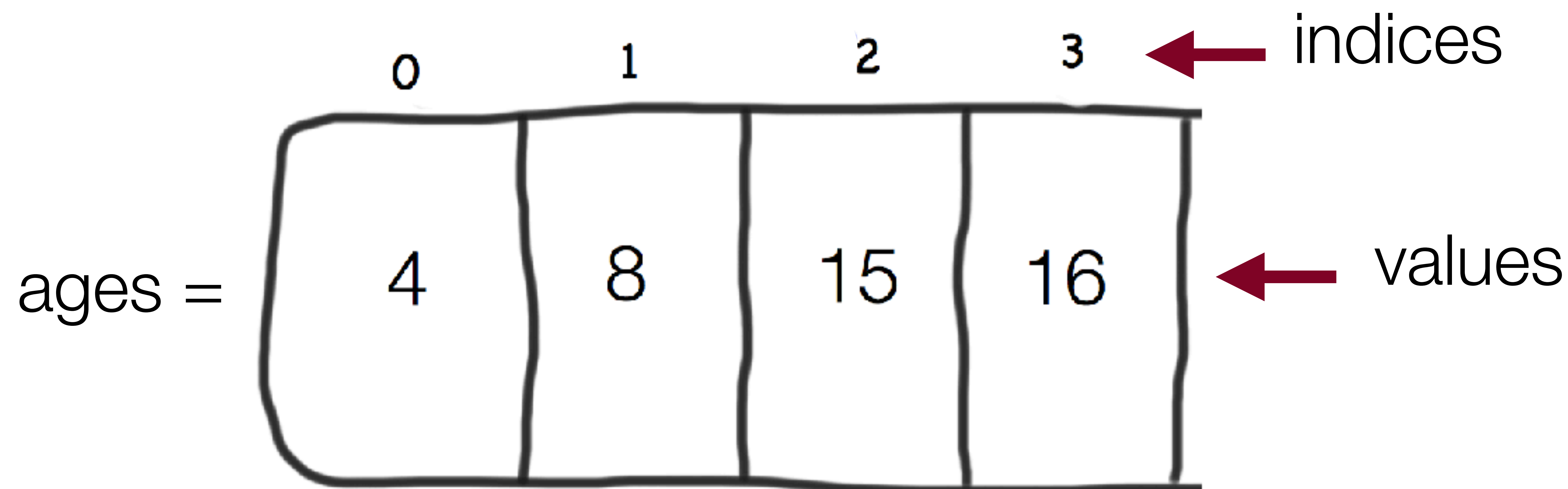
In your computer's memory, an array is just a series of contiguous locations where you can put one value after another, and the computer can access those values with *random access* by the *index* of the value:

what is the index for `ages[2]`? **2**
what is the value for `ages[2]`? **15**

By the way: `ages[2]` is an *overloaded* function, identical to `ages.get(2)`

Vectors and Big O

As we discussed yesterday, a Vector is simply an array under the hood:



What is the Big O for accessing an element in a Vector? E.g., what is the Big O for the following:

`ages[2]`? **Big O(1): Constant, because the computer can immediately access the value, and it doesn't matter if there are four values in the Vector, or four million.**

Vectors and Big O

Let's look at the Big O for the other Vector functions:

Function	Big O (worst case)
`vec.size()` Returns the number of elements in the vector.	O(1)
`vec.isEmpty()` Returns `true` if the vector is empty.	O(1)
`vec[i]` Selects the `i`th element of the vector.	O(1)
`vec.add(value)` Adds a new element to the end of the vector.	O(1)
`vec.insert(index, value)` Inserts the value before the specified index position.	O(n)
`vec.remove(index)` Removes the element at the specified index.	O(n)
`vec.clear()` Removes all elements from the vector.	O(1)

Vectors and Big O

Can we do better than O(n) (worst case) for `vec.remove(index)`?

It depends! Let's assume that the Vector is completely unordered:

0	1	2	3	4	5	6	7
25	8	3	13	49	82	27	19

How does `vec.remove(2)` work?

Vectors and Big O

The Vector's remove(int index) function might look like this, assuming the internal array is called vec, and the number of elements in the array is called count:

```
for (int i=index; i < count-1; i++) {
    vec[i] = vec[i+1];
}
count--;
```

count = 8

0	1	2	3	4	5	6	7
25	8	3	13	49	82	27	19

For `vec.remove(2)`, the loop moves 13 to index 2, then 49 to index three, etc., all the way up to the end of the array.

Vectors and Big O

The Vector's remove(int index) function might look like this, assuming the internal array is called vec, and the number of elements in the array is called count:

```
for (int i=index; i < count-1; i++) {
    vec[i] = vec[i+1];
}
count--;
```

count = 8

0	1	2	3	4	5	6	7
25	8	3	13	49	82	27	19

For `vec.remove(2)`, the loop moves 13 to index 2, then 49 to index three, etc., all the way up to the end of the array.

Vectors and Big O

The Vector's remove(int index) function might look like this, assuming the internal array is called vec, and the number of elements in the array is called count:

```
for (int i=index; i < count-1; i++) {
    vec[i] = vec[i+1];
}
count--;
```

count = 8

0	1	2	3	4	5	6	7
25	8	13	13	49	82	27	19

For `vec.remove(2)`, the loop moves 13 to index 2, then 49 to index three, etc., all the way up to the end of the array.

Vectors and Big O

The Vector's remove(int index) function might look like this, assuming the internal array is called vec, and the number of elements in the array is called count:

```
for (int i=index; i < count-1; i++) {
    vec[i] = vec[i+1];
}
count--;
```

count = 8

0	1	2	3	4	5	6	7
25	8	13	49	49	82	27	19

For `vec.remove(2)`, the loop moves 13 to index 2, then 49 to index three, etc., all the way up to the end of the array.

Vectors and Big O

The Vector's remove(int index) function might look like this, assuming the internal array is called vec, and the number of elements in the array is called count:

```
for (int i=index; i < count-1; i++) {
    vec[i] = vec[i+1];
}
count--;
```

count = 8

0	1	2	3	4	5	6	7
25	8	13	49	82	82	27	19

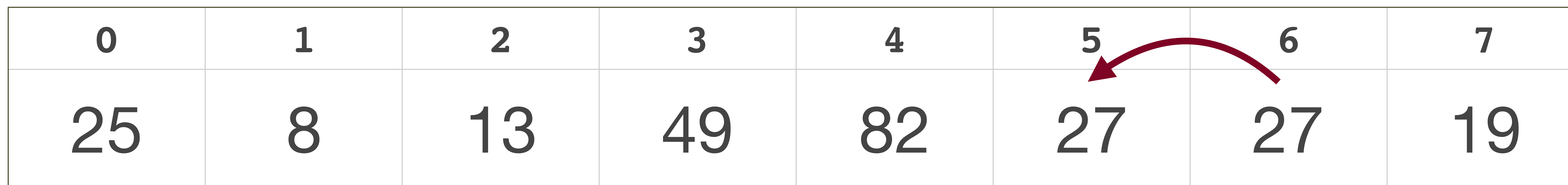
For `vec.remove(2)`, the loop moves 13 to index 2, then 49 to index three, etc., all the way up to the end of the array.

Vectors and Big O

The Vector's remove(int index) function might look like this, assuming the internal array is called vec, and the number of elements in the array is called count:

```
for (int i=index; i < count-1; i++) {
    vec[i] = vec[i+1];
}
count--;
```

count = 8



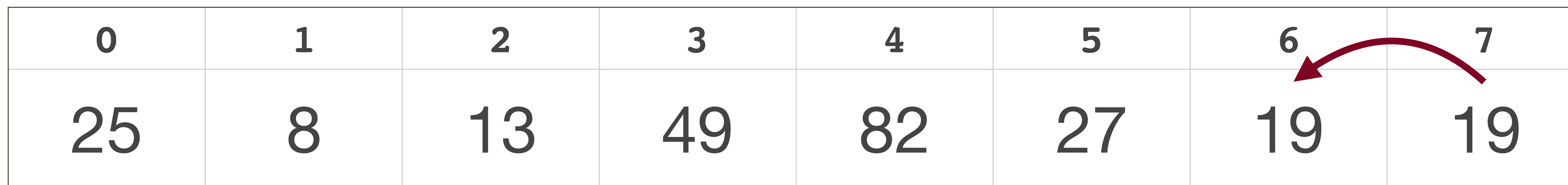
For `vec.remove(2)`, the loop moves 13 to index 2, then 49 to index three, etc., all the way up to the end of the array.

Vectors and Big O

The Vector's remove(int index) function might look like this, assuming the internal array is called vec, and the number of elements in the array is called count:

```
for (int i=index; i < count-1; i++) {
    vec[i] = vec[i+1];
}
count--;
```

count = 8



For `vec.remove(2)`, the loop moves 13 to index 2, then 49 to index three, etc., all the way up to the end of the array.

Vectors and Big O

The Vector's remove(int index) function might look like this, assuming the internal array is called vec, and the number of elements in the array is called count:

```
for (int i=index; i < count-1; i++) {
    vec[i] = vec[i+1];
}
count--;
```

count = 7

0	1	2	3	4	5	6	7
25	8	13	49	82	27	19	19

Finally, it decrements the count.

Vectors and Big O

Can we do better than O(n) for removing a value from a vector?
Remember, assume that the vector is *unordered*.
Talk to your neighbor!
We can do this in O(1)!

```
int sz = vec.size();
vec[i] = vec[sz-1];
vec.remove(sz-1);
```

count = 8

0	1	2	3	4	5	6	7
25	8	3	13	49	82	27	19

Vectors and Big O

Can we do better? Remember, assume that the vector is *unordered*.
Talk to your neighbor!

We can do this in O(1)!

```
int sz = vec.size();
vec[i] = vec[sz-1];
vec.remove(sz-1);
```

count = 8

0	1	2	3	4	5	6	7
25	8	3	19	49	82	27	19

Vectors and Big O

Can we do better? Remember, assume that the vector is *unordered*.
Talk to your neighbor!

We can do this in O(1)!

```
int sz = vec.size();
vec[i] = vec[sz-1];
vec.remove(sz-1);
```

count = 7

0	1	2	3	4	5	6	7
25	8	3	19	49	82	27	19

Vectors and Big O

Let's look at a program to test this!

Strings (3.1)

(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 Java strings, although the functions have different names and in some cases different behavior.
- The biggest difference between a 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)

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:

```
string s = "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 = s[3]     // 'r'
char c2 = s.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)

ASCII

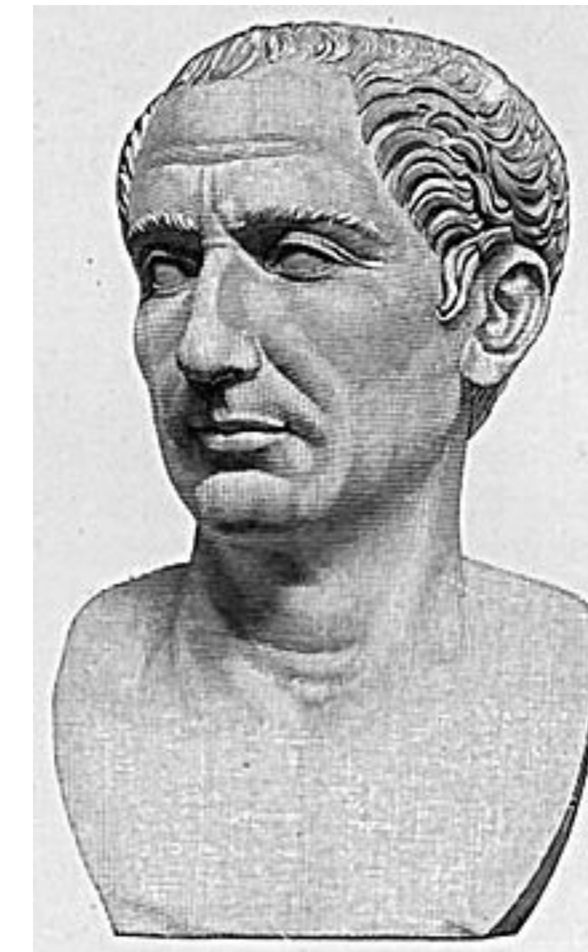
- Characters have a numerical representation,

```
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
```

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

String Operators (3.2)

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

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

- Unlike in Java, you can compare strings using relational operators:

```
string s3 = "Zebra";
if ((s1 > s3) && (s3 != "Walrus")) { // false
   ...
}
```

- Unlike in 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 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.

String Member Functions

Function	Description
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

```
string name = "Donald Knuth";
if (name.find("Knu") != string::npos) {
    name.erase(7, 5); // "Donald"
}
```

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 into C++ string
  - **`string.c_str()`** returns a C string out of a C++ string

C string issues

```
string s1 = "hi" + "there";
```

- Does not compile; C strings can't be concatenated with +.

```
string s2 = string("hi") + "there";
string s3 = "hi"; // "hi" is auto-converted to string
s += "there";
```

- These all compile and work properly.

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

- Bug; sets **`n`** to the memory address of the C string **`"42"`** (ack!). Qt Creator will produce an error, too

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

- Works, because of explicit conversion of "42" to a C++ string (and **`stringToInteger()`** is part of the Stanford C++ library)

C string issues

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

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

```
string s = string("") + "hi" + '?'
```

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

```
string s = "hi"; // char '?' is concatenated to string
s += '?'; // "hi?"
```

- Works, because of auto-conversion.

- `s += 41; // "hi?)"`
- Adds character with ASCII value 41, **')'**, doesn't produce **"hi?41"**.

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

- Works, because of conversion from int to string.

What's the Output? (Talk to your neighbor!)

```
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:

Stanford TreFOOet

Stanford String Library (3.7)

`#include "strlib.h"`

These are *not* string class functions.

Function	Description
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

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

Recap

- **Strings**
  - 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

References and Advanced Reading

- **References (in general, not the C++ references!):**
  - Textbook Chapter 3
  - <cctype> functions: http://en.cppreference.com/w/cpp/header/cctype
  - Code from class: see class website (https://cs106b.stanford.edu)
  - 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

Extra Slides

String Exercise (work with your neighbor)

Write a function called **`nameDiamond`** that accepts a string as a parameter and prints it in a "diamond" format as shown below.

- For example, nameDiamond("CHRIS") should print:

```
C
CH
CHR
CHRI
CHRIS
 HRIS
  RIS
   IS
    S
```

String Exercise Possible Solution

One possible solution (break into two parts!)

```
void nameDiamond(string s) {
    int len = (int)s.length(); // cast length to int to avoid warning
    // print top half of diamond
    for (int i = 1; i <= len; i++) {
        cout << s.substr(0, i) << endl;
    }

    // print bottom half of diamond
    for (int i = 1; i < len; i++) {
        for (int j = 0; j < i; j++) { // indent
            cout << " ";              // with spaces
        }
        cout << s.substr(i, len - i) << endl;
    }
}
```