

PIC 10A 1C Week 4a Notes. TA: Eric Kim.

Strings

Strings are fairly special, as they are the first example of **objects** in this course! We'll learn more about objects (and classes) later in the course, but at a high-level: objects have both internal variables and functions, aka "methods".

String concatenation: +

We can join strings together using the addition operator "+". For example:

```
string s1 = "Cannon";      string s2 = "ball";
string s3 = s1 + s2;      // s3 is: "Cannonball"
```

We can also concatenate strings and string-literals, and strings and characters. However, you can't concatenate two string literals:

```
string s4 = s1 + "fodder"; // s4 is: "Cannonfodder"
string s5 = s1 + 's'; // s5 is: "Cannons"
cout << "hi " + "there"; // Error! Can't use + on two string literals.
cout << string("hi") + "there"; // Works! Make "hi" a string type
first.
```

String indexing/modifying

You can retrieve individual characters of the string via the index operator. **Note that we index starting from 0**. The first character is at index 0, the second character at index 1, etc.

```
string s = "pony";
cout << s[2]; // prints: n
```

You can also change a string by assigning individual characters:

```
s[3] = 'g'; // Warning: Can't do "g"! string literal vs char
cout << s; // prints: pong
```

string::length(), string::size()

You can ask a string for its length/size by calling the length() and/or size() methods. Note that these two methods return the type `size_t`, ie "size type". The type `size_t` is a type specially designed to store length information. You can think of them as "unsigned int" if you like, but it's good practice to use `size_t` for size-related values.

```
string myname = "Corey";
size_t len = myname.length(); // or myname.size(). len is 5
cout << "length is: " << len; // cout knows how to display size_t
```

string::substr(size_t pos, size_t len)

Returns a new string object that starts from index "pos", and spans "len" characters. The output string won't go past the end of the string.

Also, if you omit the 2nd argument (len), then `substr()` will go until the end of the string.

Examples:

```
string mystr = "paunch";
cout << mystr.substr(3,2); // prints: nc
cout << mystr.substr(1); // prints: aunch
cout << mystr.substr(2,100000); // prints: unch
```

Indexing out of bounds

A common bug in programs is to index past the end of a string. For instance, observe the following:

```
string myname = "Bob";
cout << myname[3];
```

Here, we indexed *outside* of the string. The resulting value is unsafe to use, as the value may be compiler dependent. Accessing out of bounds can even crash the program (an example of a *runtime error*)! So, be careful to not index out of bounds!

Aside: On my laptop, "cout << myname[3]" displayed a space, but myname[4] crashed my program with a runtime error.

String comparisons

You can compare strings using the operators: `<`, `>`, `<=`, `>=`, and `==` (equals). In C++, strings are compared in lexicographic order in the following manner:

[EMPTY] < [SPACE] < 0 < 1 < ... < 8 < 9 < A < B < ... < Y < Z < a < b < ... < y < z

Or written another way:

[EMPTY] < [SPACE] < [DIGITS] < [UPPERCASE LETTERS] < [lowercase letters]

You can also think of [EMPTY] as the end of the string.

Examples:

```
bool a1 = string("a") < string("b"); // a1 is true
bool a2 = string("9") > string("1"); // a2 is true
bool a3 = string("z") < string("a"); // a3 is true
bool a4 = string("e") == string("e"); // a4 is true
```

One can also compare arbitrary-length strings. For instance, if we are evaluating `s1 < s2`:

Compare strings `s1`, `s2` character by character.

If a mis-match is found at index `i`, then output `s1[i] < s2[i]`

If `s1` is shorter than `s2`, and `s1` matches `s2` as much as possible, then output 1

Else, output 0.

Examples:

```
cout << ("Dog" < "dog") << endl; // 1, because "D" is less than "d"
cout << ("Dog" < "Doga") << endl; // 1. strings match, but str1 is shorter
cout << ("Dog" < "Dogaaaa") << endl; // 1. strings match, but str1 is
shorter
cout << ("Doga" < "Dog") << endl; // 0. strings match, but str2 is shorter
```

Warning: Don't confuse comparing strings and string literals. Comparing two string literals does not do the "right" thing - never do this!

```
bool a4 = "a" < "b"; // Don't do this!
```

Aside: If you really want to know, comparing "a" < "b" results in the computer comparing the memory addresses of the string literal "a" versus the string literal "b". So, there's almost never a reason why one would do this.

String objects vs String Literals

Anything in double-quotes (ie "hello") is considered a **string literal**. String literals, unlike string objects, do not have methods. You can't call length() on a string literal:

```
cout << "hello".length(); // Errors!
```

When we create a string object with a string literal, the string class has code that knows how to convert a string literal into a string object:

```
string myname1 = "Eric";
string myname2 = string("Eric");
```

The two above lines are equivalent: the string() function (also known as a *constructor*) knows how to, given a string *literal*, create a string *object*. We'll learn about constructors in a few weeks.

Constant values: const

To denote a variable that should never be modified, we can add the **const** keyword:

```
double myPI = 3.1415;
myPI = 3.0; // Hm, we allowed program to change pi...
double const PI = 3.1415;
PI = 3.0; // Compiler error! Can't modify const values.
```

Putting it all together

1. What does the following program display? If the code errors, explain why.

```
string s1 = "panic";
string s2 = s1.substr(2);
string s3 = "disco";
string s4 = s3.substr(2, 1) + s3[s3.length() - 1];
string s5 = s4 + s2;
cout << s5;
```