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'; // Note: Can't do "g"! string literal vs char
cout << s; // prints: pong
```

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

You can ask a string for it's 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/unsigned long long" 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 (true).
Else, output 0 (false).

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're curious, comparing "a" < "b" results in the computer comparing the memory addresses of the string literal "a" versus the string literal "b". 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(); // Error!
string s("hello");
cout << s.length(); // Ok!
```

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("Eric");
```

The two lines above achieve the same purpose: they both create a string object set to "Eric".

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.
```

References

To define a variable that points to a different variable, we can use references:

```
int x = 42;
int& xr = x; // xr is a reference to x
cout << xr << endl; // Displays: 42
x = -1;
cout << xr << endl; // Displays: -1
```

You can have const references, but there are several rules. Suppose x is a variable, and xr is a reference to x:

```
1. If x is const, then xr must be a const reference.
2. If x is not const, then xr can either be const or not const.
```

Finally, generally it is illegal to have a reference to a literal:

```
int& foo = 100; // Error!
```

However, C++ lets you set a const reference to a literal:

```
const int& foo = 100; // This is ok!
```

IO Manipulation

C++ offers several features to easily display text output in an organized (and even aesthetically pleasing) manner. This functionality is part of the iomanip standard library: #include <iomanip>.

Widths (setw) and Padding (setfill)

Code:	**Code:**

`cout << "Age vs Weight"`
`  << endl;`
`cout << 6 << " " << 40`
`  << endl;`
`cout << 18 << " " << 120`
`  << endl;`
`cout << 35 << " " << 130`
`  << endl;`
	`cout << "Age vs Weight" << endl;`
	`cout << setfill('0');`
	`cout << setw(3) << 6 << " " << setw(4) << 40`
	`  << endl;`
	`cout << setw(3) << 6 << " " << setw(4) << 40`
	`  << endl;`
	`cout << setw(3) << 18 << " " << setw(4)`
	`  << 120 << endl;`
	`cout << setw(3) << 35 << " " << setw(4)`
	`  << 130 << endl;`

Output:
`Age vs Weight`
`6 40`
`18 120`
`35 130`
	Output:
	`Age vs Weight`
	`006 0040`
	`018 0120`
	`035 0130`

Warning: Take care that you pass a **character** to setfill(), rather than a string literal. For instance, don't do `setfill("0")`, instead do `setfill('0')`.
Note: setw (set width) only modifies the next output, ie **short-term** change. This is why I had to repeat setw(3) and setw(4) for each output. On the other hand, setfill() modifies all future outputs, ie **long-term** changes.

setprecision, fixed/scientific

We can set the number of digits to display via setprecision(), ie doing "cout << setprecision(4)" will tell cout to only display (at most) four decimal digits, rounding where necessary.
You can explicitly tell cout to display numbers in scientific notation by using the `scientific` manipulator.
On the other hand, `fixed` is a manipulator to display numbers in decimal-point (ie fixed-point) notation. `fixed` will never display an exponent field.

Code:	**Output:**

`cout << fixed << setprecision(6);`
`cout << 5.12345678 << endl;`
`cout << setprecision(3);`
`cout << 51.9 << endl;`
`cout << scientific;`
`cout << 51.9 << endl;`
	`5.123457 // 7, because rounding`
	`51.900`
	`5.190e+001`

Aside: If you don't specify "fixed" or "scientific", then cout defaults to a "hybrid" mode halfway in between fixed and scientific. This hybrid mode is a little difficult to describe, but in this course we stick only to either "fixed" or "scientific".

left/right justification

By default, text is displayed in a left-justified manner. We can change the justification to the right:

`cout << setfill('x');`
`cout << left << setw(6) << 1234 << endl;`
`cout << right << setw(6) << 1234 << endl;`
	Output:
	`1234xx`
	`xx1234`

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;
```

2. A text "flower pot" looks like this:

```
==================
== Hello world! ==
==================
```

A string is contained within a bunch of equal signs '='. Note that the first and third lines have a number of '=' related to the length of the string, whereas the second line always has only 4 '='.
Write a **complete** C++ program that asks the user for a string, and outputs the string within a flower pot.
Example:

```
Enter some text: Billy Shears
==================
== Billy Shears ==
==================
```