COMSC-200 Lecture Topic 7
Problem Solving With OOP

Reference
Deitel Ch.16,18

The `string` Class
a "container" class: array of `char`s
works with `cout <<`
works with `cin >>`
works with `getline(cin, ...)` ...not cin.getline
requires:

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <string>
using std::getline;
```

Constructor Syntax

```
#include <string>
using std::string;
string s; // a blank string
string s("Hello");
string s = "Hello";
```
implicitly uses constructor

```
string s(8, ' '); // char or int equivalent
```
a string of 8 blanks

Assignment Syntax

```
s = "Hello";
```
...or `cin >> s;`
...or `getline(cin, s);`
delimiter is `\n`

```
string t(...); s = t;
```

Concatenation Syntax
`s =` *any number of plus-delimited strings and quoted literals (or chars, but be careful!)*
`s +=` ...works, too
`string("") + ...`
an unnamed object

Mutation Syntax

```
s[0] = 'h';
s[1] = 65; // 0-255
```

Member Functions
`s.length()` // #of characters
`s.c_str()` // const char*
`if (s.find(t) == string::npos)` // *not* found

A `toString()` Function

```
string Time::toString() const
{
  ostringstream sout;
  sout << setfill('0');
  sout << h << ":" << setw(2) << m << ":" << setw(2) << s;
  return sout.str();
}
```

Pre-loaded Input Stream
use `sin` in place of `cin`
for use with program testing

```
#include <sstream>
using std::istringstream;

cout << "Enter hrs, mins, secs on separate lines";
istringstream sin;
sin.str("12\n30\n15"); // stage input for 12:30:15

sin >> hours;
sin >> minutes;
sin >> seconds;
```

Simple Exceptions

```
throw "Invalid input";

try{...}
catch(const char* ex){...ex...}
```
no includes required for simple exceptions

...or use an integer code...

```
throw 5;
catch(int ex){cout << "Error #" << ex;}
```

...or use a formatted string...

```
throw sout.str();
catch(string ex){cout << "Error: " << ex;}
```

Exceptions In The Route Class
throw exception from constructors if route's end != leg's start
guard against memory leaks!

The Standard Template Library
`string` derives from the STL's **basic_string**
`basic_string<char>` is a `string` *(by typedef)*
`basic_string<wchar_t>` is the unicode version *(cannot cout)*

Pointer vs Reference Variables
reference variables *cannot* be unassigned

`s.swap(t)` // swaps two strings

Comparing Strings

`if (s == t)` also `!=`
lexicographical:`if (s.compare(t))`
also: `< > <= >=` operators

Building Formatted Strings

```
#include <sstream>
using std::ostringstream;
ostringstream sout; // a buffer
sout << ...
sout.str() -- returns a string
```

e.g., Rider's destination floor
pointer can be NULL
e.g., Elevator's destination floor

Object Accounting: mixing *dynamic memory allocation* with *exception throwing* in constructors
This topic applies to constructors that both (1) allocate dynamic memory *and* (2) can throw an exception. It's possible for constructors to throw an exception. In that case, the object they are to construct actually never gets constructed. Hence the destructor never gets called for that object.

If dynamic memory got allocated *before* an exception gets thrown, it's important to *deallocate* that memory *before* throwing the exception.

Try this test program to convince yourself that the destructor is not called:

```
#include <iostream>
using std::cout;
using std::endl;

class X
{
  public:
  X(){cout<<"X::X "<<(long)this<<endl; throw "Exiting now";}
  ~X(){cout<<"X::~X "<<(long)this<<endl;}
};

int main()
{
  cout << "Creating X x\n";
  try {X x;}
  catch (const char* ex) {cout << ex << endl;}
}
```

So, is the book correct in what it says about destructors being called when an exception's thrown by a constructor?