COMSC-200 Lecture Topic 6
Object Self-Referencing

Reference
Deitel Ch.10.5-10.6

`static` Functions
`static` vs. `friend`
  friend has object in parameter list
  static may not
example: (class Math)
```
static double sqrt(double)
double Math::sqrt(double)
Route ShortestRoute::getShortestRoute(string, string)
```
as alternative to non-static functions:
example: `static int diff(const Time&, const Time&);`
  vs. `int diff(const Time&) const;`
usage: `Time::diff(t1, t2);` vs. `t1.diff(t2);`

Private Constructors
having only a private default constructor
  prevents creation of objects outside class
used for classes of static functions
or have only a copy constructor -- public or otherwise
  `Time(const Time&){}` // inline example

The "Lost Parameter"
example: the `Time` class
  `output()` member function
  vs. a `output(Time&)` static function
  or a `output(Time&)` friend function
in the class member function, the parameter
  variable has *no name*

The `this` Keyword
a constant mutating pointer to the "host object"
```
Time t;
t.output(); // "t" is the host object
```
for explicit reference to members
  example: `this->hour` vs. `hour`
for calling other functions
  example: `output(*this);`
  ...may require dereferencing
to allow cascading calls
  example: `return this;` // a pointer
  ...or: `return *this;` // a reference
```
t.setHr(9).setMin(34).setSec(42);
```

Dynamic Memory Allocation
compile-time vs. run-time
when unnamed objects are created
  usually not known at run-time
  or need to persist when scope ends
    after loops
    after function calls
applies to single objects
applies to arrays of size unknown at compile-time

The `new` Keyword
creates an object; returns its memory address
expression, usually assigned to a pointer
example: `Time* t = new Time;`
  "t" is a variable mutating pointer
    null if failed
constructor matches `()` parameter list data type

The `delete` Keyword
use to deallocate memory allocated to object
prevents "memory leaks" in some OSs
example: `delete d;`

Variable-Sized Arrays
example: `Time* ta = new Time[n];` where "n" is an `int`
another: `const Time* ta = new Time[n];` // no const after "new"
creates "n" objects using the default constructor
  default constructor required
example: `delete [] ta;`
`new` *requires* `delete`
`new []` *requires* `delete []`

Variable-Sized Arrays Of Pointers
example: `Time** ta = new Time*[n];` where "n" is an `int`
another: `const Time** ta = new const Time*[n];`

Functions Without Host Objects
no need to access data members directly
no need for an object to exist
like C-functions and friend functions
also, data members not associated with individual objects
  instance counters, common constants, public constants

`static` Data Members
1. unique IDs for individual objects
example: `static int count;` // #of Time objects created
  in constructor(s) init list: `ID(count++)`
in **.cpp** file: `int Time::count = 0;`
2. variables not associated with individual objects
example: `static int count;` // #of Time objects that exist
  in constructor(s): `count++;`
  in destructor: `count--;`
3. global constants (can be public)
example: `static const double PI;` // class Math
in **.cpp** file: `double Math::PI = 3.14159;`
or `static const double PI = 3.14159;` in class definition
  compound statement allowed for `static const` only

Lab 6
do *not* use instance counters in lab 6

Requirements when using dynamic memory

```
class X
{
  // private data members
  ...

  public:
  // constructors
  ...

  // setters and getters
  ...

  // for dynamic memory management
  X(const X&); // copy constructor topic 3
  ~X(); // destructor topic 3
  X& operator=(const X&); // assignment operator topic 8

  // private functions
  ...
};
```

`const` Pointers to Pointers

In the following example:

```
const X* const * const array = new X*[n];
```

...a dynamic array of pointers to objects of the class X are created. Here is what each `const` keywords means:

first `const`: cannot change what `array[i]` points to, as in `array[i]->aDataMember = 0;` or `array[i]->aMutatorFunction();`

second `const`: cannot change `array[i]` to point to something else, as in `array[i] = 0;` , so it's rarely used!

third `const`: cannot change `array` to point to something else, as in `array = 0;`

Note that `new X*[n]` is an *expression*, and can be used inside parentheses of initializer lists.