COMSC-165 Lecture Topic 5
Advanced C++ Functions

Reference
Deitel, chapter 5.12-5.23

The Function Call Stack
how OSs track function calls
what happens to functions' local variables

Empty Parameter Lists
the optional `void` keyword

example: `int getMyAge();`
vs `int getMyAge(void);`

Inline Functions
the `inline` keyword
reduces function call overhead
(size vs speed tradeoff)

example prototype: `inline int getMyAge();`
example definition: `inline int getMyAge(){...}`

C++ Reference Parameters
the trailing `&` symbol (ampersand)
creates *aliases* for other variables
no good reason for this in same scope
...but useful across scopes: parameters

reference parameters: "pass by reference"
...vs "pass by value"
handy way to use parameters for *output*

ok to use for literal values as parameters?
...*no*
...unless "const" modifier is used

example: `void swapTwoValues(int, int);`
vs `void swapTwoValues(int&, int&);`

C++ Default Parameters
use assignment in prototype
`int avg(int = 10, int = 20);` // prototype
do *not* do this in function definition,
unless there is no prototype

The Scope Resolution Operator
the optional, leading `::` symbol
to distinguish global variables from local
to resolve name conficts

Function Overloading In C++
when multiple functions have the same name
...must have different parameter types
the "function signature": name+parameter type sequence
return type is *ignored*

C++ Function Templates
```
template <class T>
T max(T a, T b)
{
  T result = a;
  if (a < b) result = b;
  return result;
}
```

optional: can be in a separate H file
optional: a prototype

Recursive Loops
another syntax for loops, where functions call themselves
advantage: each cycle has its own set of local variables
e.g., Fibonacci sequence solution
disadvange: it's a resource hog!
e.g., Fibonacci sequence limitation

Review Of Floating Point Number Formatting

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

#include <iomanip>
using std::setw;
using std::setprecision;

  // right-justify in a 20-space column (the default)
  cout << setw(20); // applies only to NEXT cout'ed value
  cout << "Hello"; // printed with 15 spaces BEFORE the word Hello

  // left-justify in a 20-space column
  cout.setf(ios::left, ios::adjustfield); // ios::right works, too
  cout << setw(20); // applies only to NEXT cout'ed value
  cout << "World"; // printed with 15 spaces AFTER the word World

  // set #of significant digits
  cout << 1000000.0; // printed as 1E6
  cout.unsetf(ios::fixed|ios::showpoint); // THIS...
  cout << setprecision(7); // applies to ALL following cout'ed values
  cout << 1000000.0; // printed as 1000000

  // set #of places after decimal to 2
  cout.setf(ios::fixed|ios::showpoint); // THIS...
  cout << setprecision(2); // applies to ALL following cout'ed values
  cout << 3.14159265; // printed as 3.14

  // unset fixed #of places after decimal
  cout.unsetf(ios::fixed|ios::showpoint); // THIS...
  cout << setprecision(6); // applies to ALL following cout'ed values
  cout << 3.14159265; // printed as 3.141593

  // fill spaces with zeros
  cout.fill('0');
  cout.setf(ios::right, ios::adjustfield); // resets to default, right
  cout << setw(4); // applies only to NEXT cout'ed value
  cout << 12; // printed with 2 zeros BEFORE the number 12
```