Function practice

William Hendrix

Lecture 11

Outline

- Recursion
- Minilab

Reference parameters

- When function is called, values in function call are copied into variables before function starts
  - If you pass a variable as a parameter, the original value will not change
  - Exceptions: global variables and arrays
  - Considered bad style to modify parameter values in the function
- What if we want to change parameter values?
  - E.g., `void swap(int a, int b)`

- Reference parameters
  - Denoted by appending `&` to type
  - E.g., `void swap(int& a, int& b)`
    - When calling, a variable must be specified for this parameter
  - Useful when you need to return multiple pieces of information

The call stack

- Memory used by the operating system to maintain program state
  - IDEs/debuggers can print it out for you
- Each function has a “frame”, including
  - where the program needs to go when function is done
  - all local variables for function
- Every time you call a function, more memory is allocated on the call stack for your function
  - Memory is released when you return
  - Extra time for memory allocation/deallocation can be noticeable if you call functions very frequently
  - You can call functions within functions until you run out of memory

Recursion

- A function can be called within *any* function
- Recursion: calling a function within itself
- Recursive calls should always have different parameter values
  - Otherwise, program will loop until you run out of memory
    - "Stack overflow" error
- Also, some "base cases" must be solved directly
  - All "recursive cases" must reduce to base cases eventually
- Example: factorial

```
int fact(int n)
{
  if (n < 0)
    return -1;
  else if (n == 0)
    return 1;
  return n * fact(n - 1);
}
```

Minilab

- Write a program that reads in a number *n* between 1 and 8 (inclusive) and prints out all permutations of the numbers 1 through *n*.
  - *Strategy:* try a recursive strategy where you find all permutations with 1 in the first position, then swap 2 into the first position, etc. Once you've "fixed" the first position, find all permutations with the next value in the second position, then swap the third value into the second position, etc.

- E.g., *n = 3*

1 2 3	2 1 3	3 1 2
1 3 2	2 3 1	3 2 1

- Recommended functions:

```
void printAllPermutations(int array[], int len);
void recursivePerm(int array[], int len, int fixed);
void printArray(int array[], int len);
```

Tonight

Lab 3 is due next Monday at 11:59pm

Recommended reading: Advanced topic 2.4, Sections 7.1, 7.2, 7.4, 7.5