Week 4 - Wednesday

COMP 2400

Last time

- What did we talk about last time?
- Functions

Questions?

Project 2

Quotes

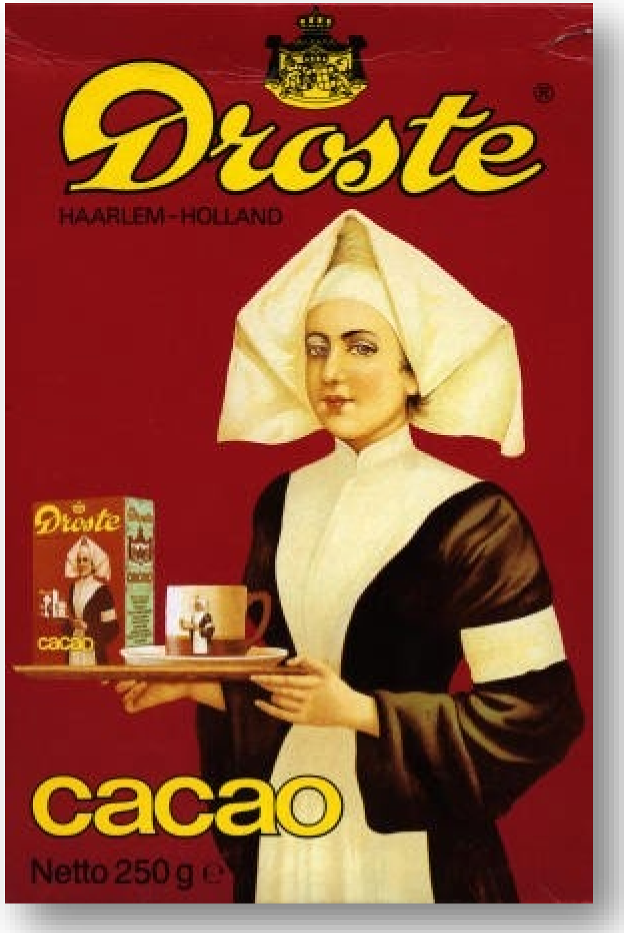
> *Unix never says "please."*
>
> Rob Pike

- It also never says:
    - "Thank you"
    - "You're welcome"
    - "I'm sorry"
    - "Are you sure you want to do that?"

Recursion

What is recursion?

- Defining something in terms of itself
- To be useful, the definition must be based on progressively simpler definitions of the thing being defined
- If a function calls itself (directly or indirectly), it's recursive



Top down

Explicitly:

- $n! = (n)(n-1)(n-2) \ldots (2)(1)$

Recursively:

- $n! = (n)(n-1)!$
- $1! = 1$

- $6! = 6 \cdot 5!$
  - $5! = 5 \cdot 4!$
    - $4! = 4 \cdot 3!$
      - $3! = 3 \cdot 2!$
        - $2! = 2 \cdot 1!$
          - $1! = 1$
- $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$

Useful recursion

Two parts:

- Base case(s)
  - Tells recursion when to stop
  - For factorial, $n = 1$ or $n = 0$ are examples of base cases
- Recursive case(s)
  - Allows recursion to progress
  - "Leap of faith"
  - For factorial, $n > 1$ is the recursive case

Approach for problems

- Top down approach
- Don't try to solve the whole problem
- Deal with the next step in the problem
- Then make the "leap of faith"
- Assume that you can solve any smaller part of the problem

Walking to the door

- Problem: You want to walk to the door
- Base case (if you reach the door):
  - You're done!
- Recursive case (if you aren't there yet):
  - Take a step toward the door

Problem

Implementing factorial

- Base case ($n \leq 1$):
  - $1! = 0! = 1$
- Recursive case ($n > 1$):
  - $n! = n(n-1)!$

Code for factorial

```
long long factorial (int n)
{
  if (n <= 1)
     return 1;
  else
     return n*factorial (n – 1);
}
```

Base Case

Recursive Case

Count the zeroes

- Given an integer, count the number of zeroes in its representation
- Example:
  - 13007804
  - 3 zeroes

Recursion for zeroes

- Base cases (number less than 10):
  - 1 zero if it is 0
  - No zeroes otherwise
- Recursive cases (number greater than or equal to 10):
  - One more zero than the rest of the number if the last digit is 0
  - The same number of zeroes as the rest of the number if the last digit is not 0

Code for zeroes

```
int zeroes (int n)
{
  if (n == 0)
     return 1;
  else if (n < 10)
     return 0;
  else if (n % 10 == 0)
     return 1 + zeroes (n / 10);
  else
     return zeroes (n / 10);
}
```

Base Cases

Recursive Cases

Searching in a sorted array

- Given an array of integers in (ascending) sorted order, find the index of the one you are looking for
- Useful problem with practical applications
- Recursion makes an efficient solution obvious
- Play the **High-Low** game

Recursion for binary search

- Base cases:
  - The number isn't in the range you are looking at. Return -1.
  - The number in the middle of the range is the one you are looking for. Return its index.
- Recursion cases:
  - The number in the middle of the range is too low. Look in the range above it.
  - The number in middle of the range is too high. Look in the range below it.

Code for binary search

```
int search (int array[],
int n, int start, int end)
{
  int midpoint = (start + end)/2;
  if (start >= end)
    return -1;
  else if (array[midpoint] == n )
    return midpoint;
  else if (array[midpoint] < n)
    return search (array, n,
            midpoint + 1, end);
  else
    return search (array, n, start,
             midpoint);
}
```

Base Cases

Recursive Cases

Programming practice

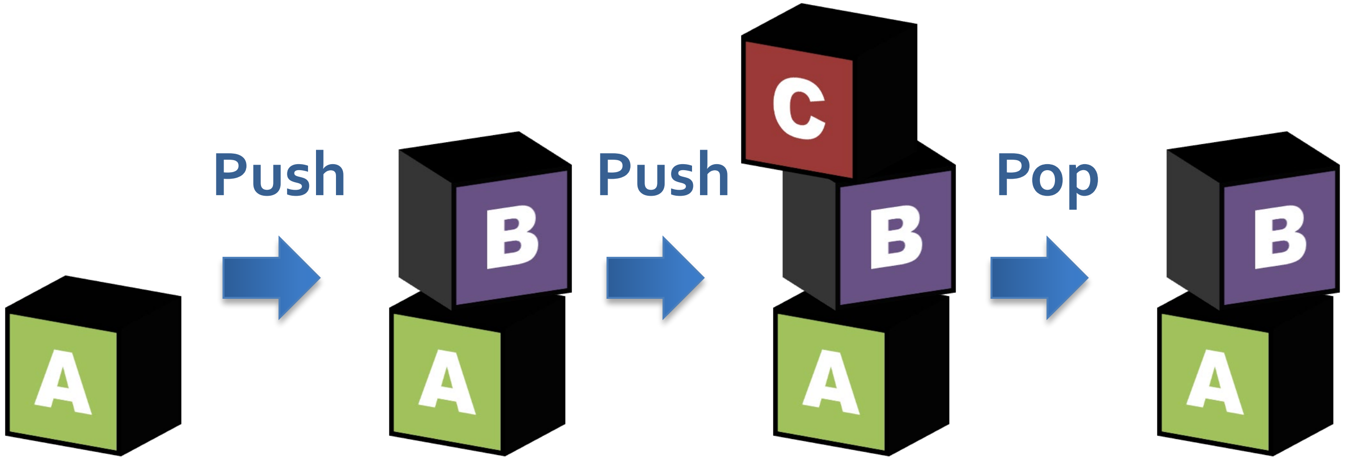
- Write a recursive function to determine the number of digits in a number

How does recursion work in the computer?

- Is there a problem with calling a function from the same function?
- How does the computer keep track of which function is which?

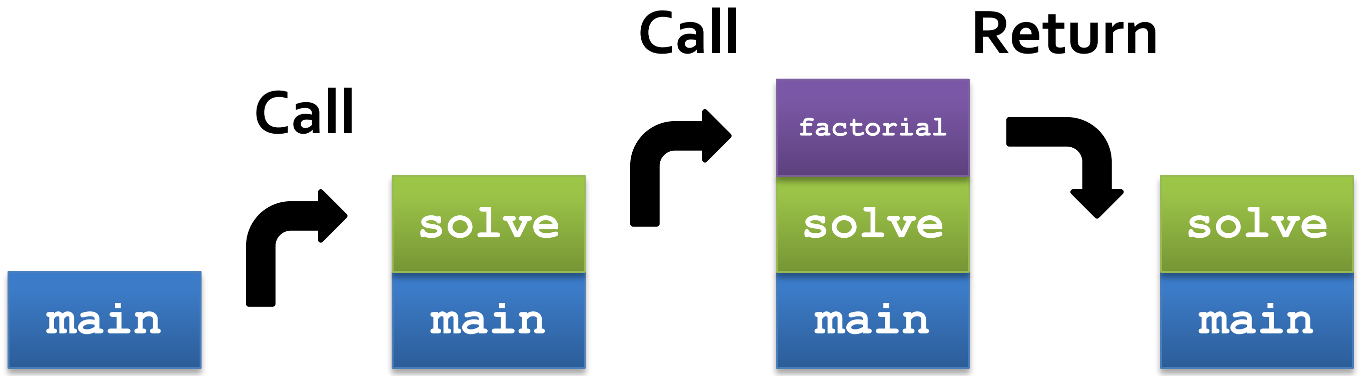
Stacks

- A stack is a FILO data structure used to store and retrieve items in a particular order
- Just like a stack of blocks:



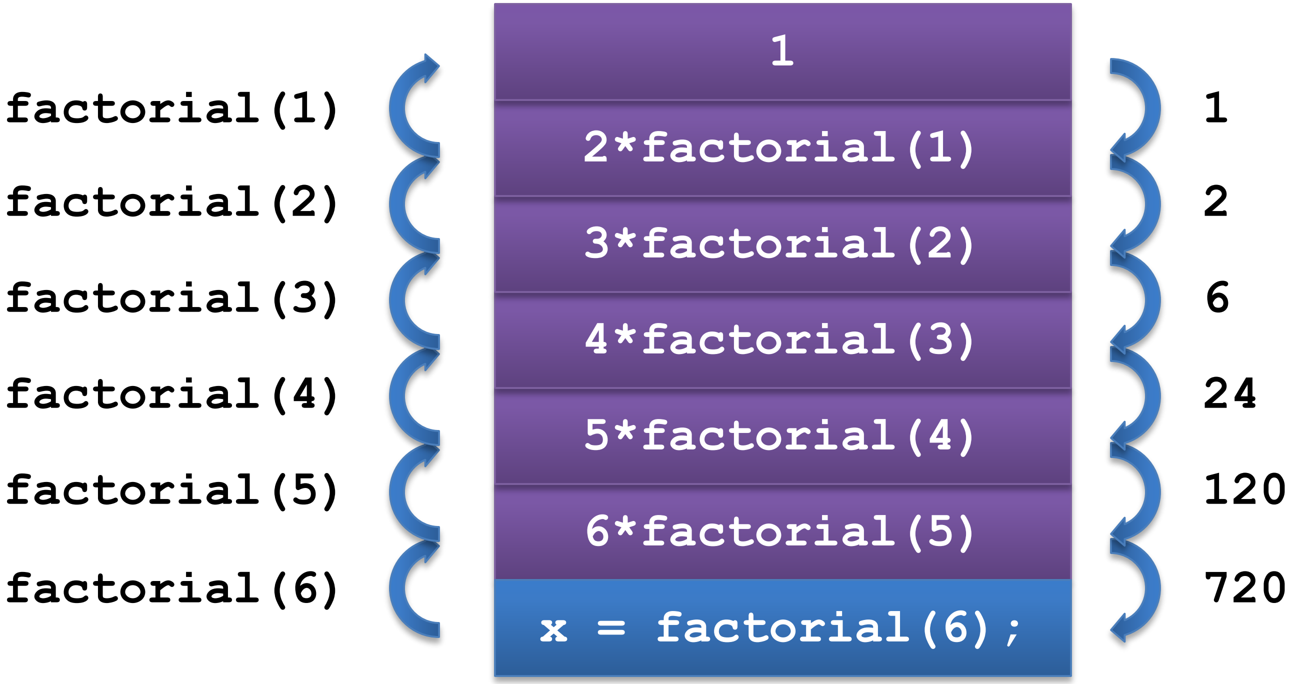
Call stack

- In the same way, the local variables for each function are stored on the call stack
- When a function is called, a copy of that function is **pushed** onto the stack
- When a function returns, that copy of the function **pops** off the stack



Example with Factorial

- Each copy of factorial has a value of ***n*** stored as a local variable
- For 6! :



factorial(1)
factorial(2)
factorial(3)
factorial(4)
factorial(5)
factorial(6)

1
2*factorial(1)
3*factorial(2)
4*factorial(3)
5*factorial(4)
6*factorial(5)
x = factorial(6);

1
2
6
24
120
720

Efficiency

- Calling functions has overhead, so calling a function 1,000 times is usually much slower than running equivalent code in a loop 1,000 times
- Modern compilers, however, are relatively good at optimizing recursive calls
- Some of the most commonly used recursive algorithms (binary search and binary search tree manipulation) run in O(log ***n***)
  - The overhead is less noticeable since the function isn't called many times
  - People looking for serious performance tuning will usually convert those algorithms to iterative implementations

Stack overflow

- The segment of memory dedicated to the stack is limited in size
- Too many recursive calls will overflow the stack
- Even if your program would get the right answer with an unlimited stack, it will crash after what's usually tens of thousands of calls
- Be careful when writing recursion that might go thousands deep
    - Another reason to stick to O(log *n*) algorithms

Stack overflow example

- The following recursive function adds the number from 1 up to **`n`**
- It follows almost the same shape as **`factorial()`**

```
long sumUpTo(int n)
{
      if (n == 1)
             return 1;
      else
             return n + sumUpTo(n - 1);
}
```

- The **`sumUpTo()`** function works just fine for values like 100
- It will get a stack overflow on values like 500000

Ticket Out the Door

Upcoming

Next time…

- Scope
- Processes

Reminders

- Read LPI chapter 6
- Finish Project 2
    - Due Friday by midnight!