CIT237

Chapter 20: Recursion

November 27, 2019

Reminders / Announcement

- Quiz 7 will be held at the start of class on

  Wednesday, December 11.

  The material covered on Quiz 7 will be announced as the date gets closer.
- Project 3:

  EXTENDED due date is December 9.
- Our last day of class is Monday, December 16.
  - In addition to a lecture, we will be demonstrating Lab solutions during that class.

Introduction to Recursion

- A recursive function contains a call to itself:

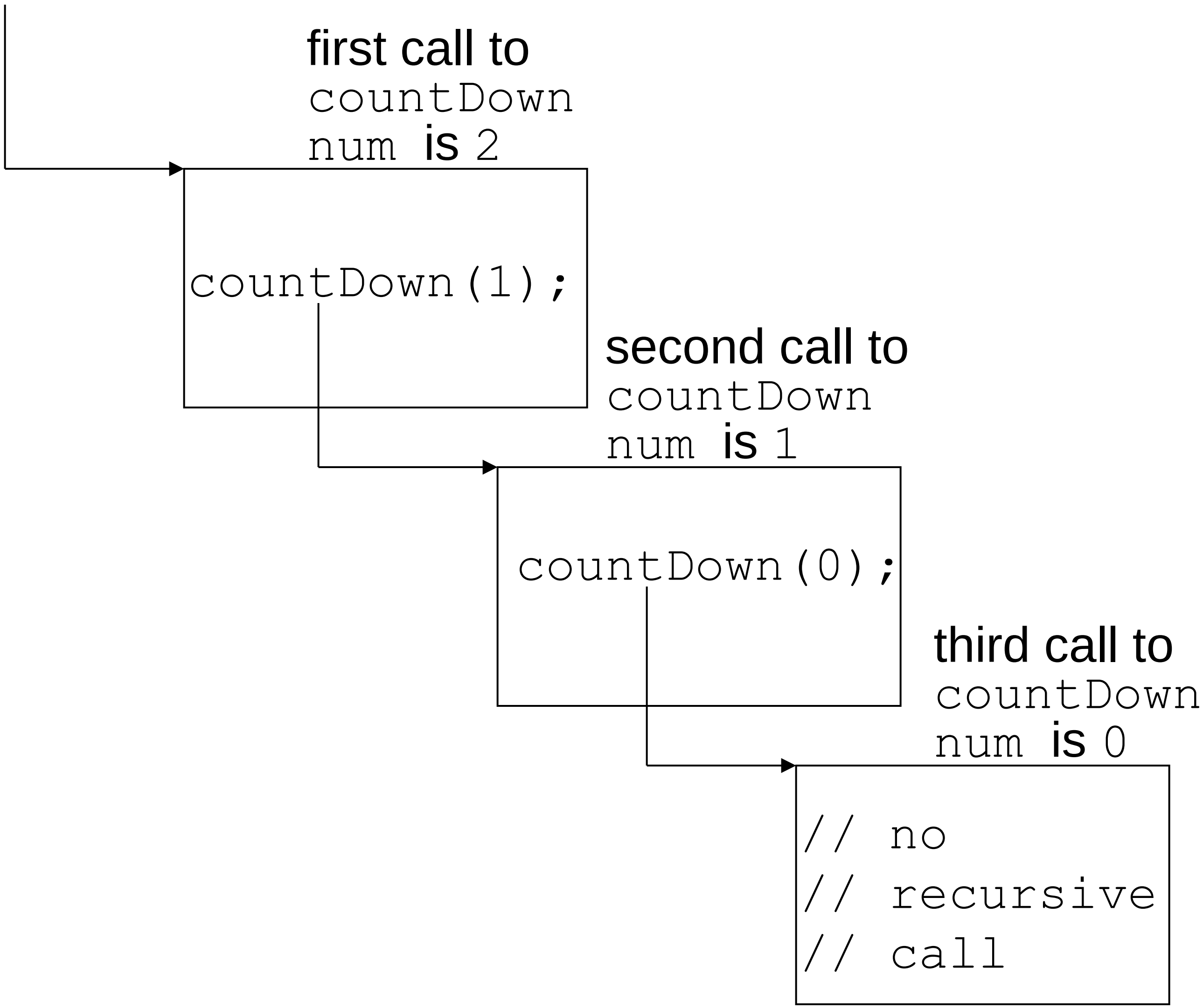
```
void countDown(int num)
{
    if (num == 0)
        cout << "Blastoff!";
    else
    {
        cout << num << "...\n";
        countDown(num-1); // recursive
    }                     // call
}
```

What Happens When Called? (1)

If a program contains a line like `countDown(2);`

1. `countDown(2)` generates the output `2...`, then it calls `countDown(1)`
2. `countDown(1)` generates the output `1...`, then it calls `countDown(0)`
3. `countDown(0)` generates the output `Blastoff!`, then returns to `countDown(1)`
4. `countDown(1)` returns to `countDown(2)`
5. `countDown(2)` returns to the original calling function

What Happens When Called? (2)



output:

2...

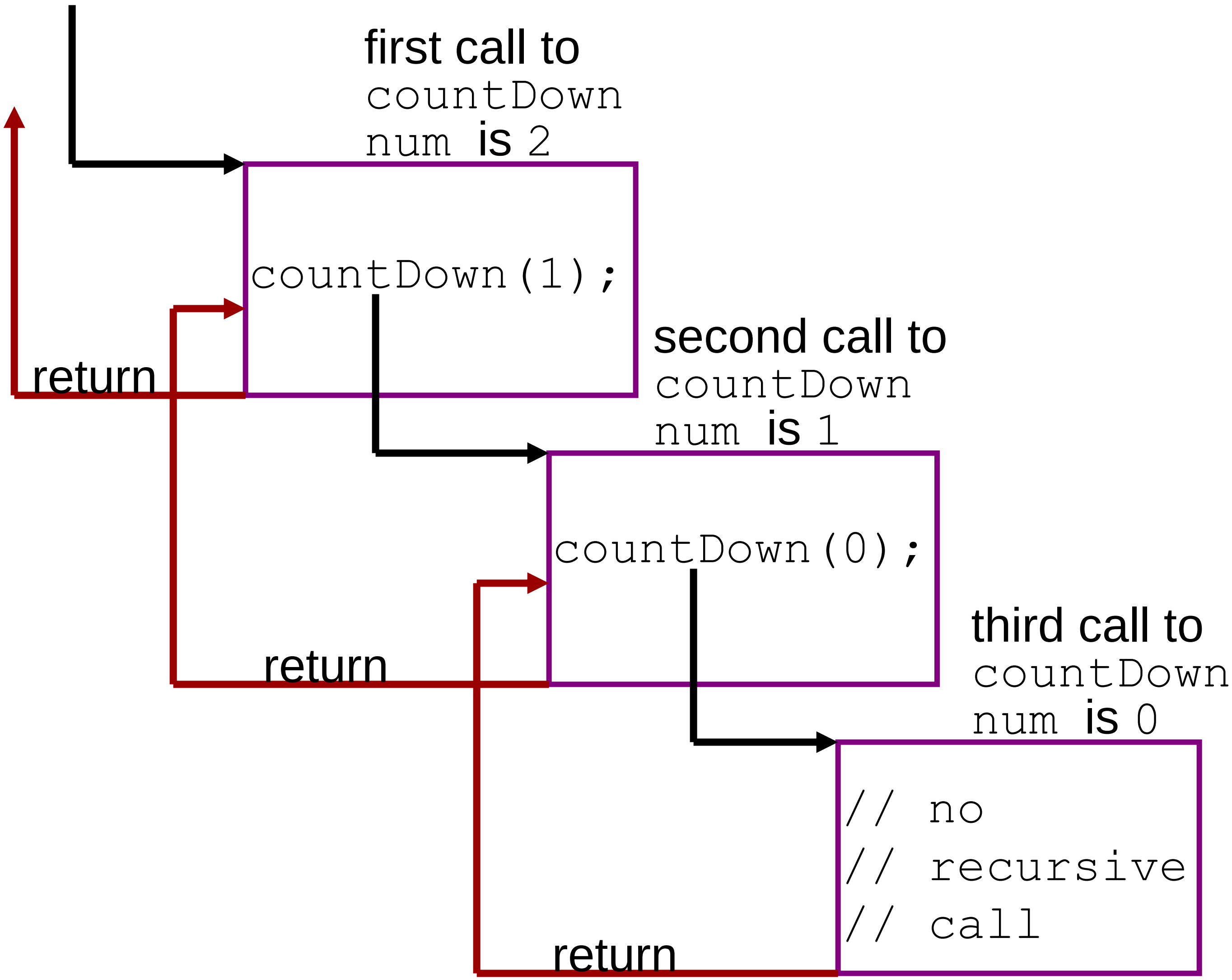
1...

Blastoff!

What Happens When Called? (3)

- Each time a recursive function is called, a new copy of the function runs, and new instances of the calling parameters and local variables are created.
- As each copy finishes executing, it returns to the copy of the function that called it.
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function.

What Happens When Called?   (4)



output:

2...

1...

Blastoff!

Recursive Functions - Purpose

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
- The simpler-to-solve problem is known as the <u>base case</u>
- Recursive calls stop when the base case is reached

Stopping the Recursion (1)

- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call
- In the sample program, the test is:

```
if (num == 0)
```

Stopping the Recursion (2)

```
void countDown(int num)
{
   if (num == 0) // test
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); // recursive
   }                    // call
}
```

Stopping the Recursion (3)

- Recursion uses a process of breaking a problem down into smaller problems until the problem can be solved
- In the `countDown` function, a different value is passed to the function each time it is called
- Eventually, the parameter reaches the value in the test, and the recursion stops

Stopping the Recursion (4)

```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1);// note that the value
   }                  // passed to recursive
}                     // calls decreases by
                      // one for each call
```

Can you see the bug in this function?

```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1);
   }
}
```

- What input value would cause trouble?
  - What would the symptom(s) of the problem be?
  - Can you suggest a code change to guard against this occurrence?

Types of Recursion

- Direct
  - a function calls itself
- Indirect
  - function A calls function B, and function B calls function A
  - function A calls function B, which calls …, which calls function A

The Recursive "Factorial" Function

Remember from Math class:

- The factorial function:

  `n! = n*(n-1)*(n-2)*...*3*2*1` if `n>0`

  `n! = 1` if `n=0`

- We can compute factorial of **`n`** if the factorial of `(n-1)` is known:

  `n! = n * (n-1)!`

- `n = 0` is the base case

The Recursive Factorial Function

```
int factorial (int num)
{
  if (num > 0) {
      return num * factorial(num-1);
  }
  else  {
      return 1;
  }
}
```

Example: VerboseRecursion

- Prints intermediate results of the factorial calculation.
- Also prints the address of the local variable that holds each intermediate result.
- See also the sample program on Moodle.

Recursive Linked List Operations

- Recursive functions can be members of a linked list class
- Example applications:
  - Compute the size of (number of nodes in) a list.
  - Compute the total of all data values in the list.
  - Traverse the list in <u>reverse</u> order.

Counting the Nodes in a Linked List

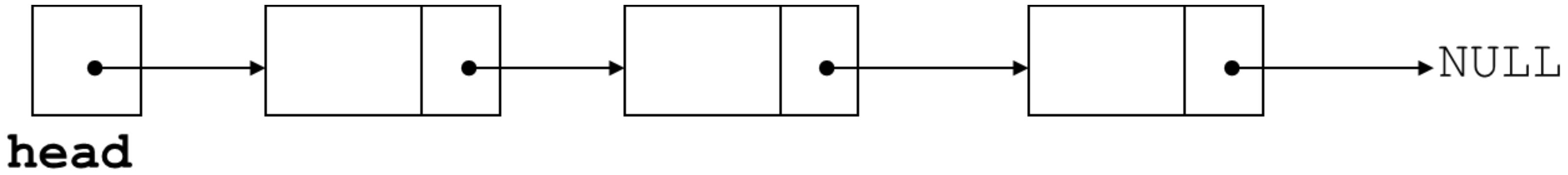
- Uses a pointer to visit each node
- Algorithm:

  Initially, the pointer starts at head of the list.
  - If the pointer is NULL, return 0 (base case).
  - If the pointer is not NULL, use a recursive call to advance to next node.
  - Upon returning from the recursive call, return

    1+(whatever value the recursive call returned).

- That is, each return value is:

  (the length of the *remaining* list) + 1.



Contents of a List in Reverse Order

- If you have a *singly-linked* list, then displaying the list contents in <u>reverse order</u> could be very messy *without* using recursion.
- <u>Algorithm</u>:

  Initially, the pointer starts at head of the list.
  - If the pointer is NULL, return (base case)
  - If the pointer is not NULL, use a recursive call to advance to next node.
  - <u>Upon returning from the recursive call</u>, display contents of the current node.

Recursion vs. Iteration

- Benefits (+), disadvantages(-) of recursion:
  - \+ Models some algorithms most accurately
  - \+ Results in shorter, simpler functions
  - – May execute very slowly, because of function-call overhead.
  - – Often uses more memory, because of local variables that are present for *each* level of recursion.
- Benefits (+), disadvantages(-) for iteration:
  - \+ Usually executes more efficiently than recursion.
  - – Sometimes harder to code or understand: if an algorithm is inherently recursive in nature, then a non-recursive implementation of the algorithm may be more complicated.

Recursion: Summary

- Any recursive function or program must have a "base case" to avoid an effective "endless loop".
- The recursive logic must eventually reach the "base case". That is, each recursive step should bring the program closer to the "base case".
- Recursive algorithms are often *less* efficient than their iterative counterparts.
- The choice to use a recursive algorithm is usually based on simplicity.