Topic 16 – Recursion

Problem Solving

- So far we have learned how to solve problems using repetition, where we repeatedly execute a body of code to reach a solution
  - Each iteration advances the solution towards completion
- In repetition we use the following control structures to manage the flow of our code
  - Looping
    - For-Loop
    - While Loop
    - Do-While Loop

Recursion for Problem Solving

- There some specific problems that can be solved by creating a function or procedure that calls itself
  - The function will continue calling itself until it reaches the solution, stopping the process of calling itself
  - solving the problem by reducing it to smaller versions of itself
- This concept is called recursion
  - Process in which the result of each repetition is dependent upon the result of the next repetition
  - It could simplify coding but may not be efficient
- Recursion uses function call (calling itself) instead of loops to solve the problem

Recursion Example

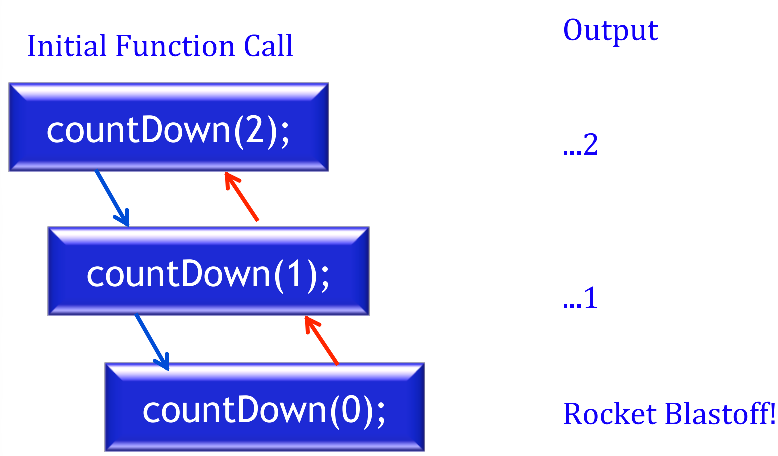
- Let's assume we want to create a function that implements a countdown to launch a rocket
  - Starting from an initial number, it will decrement that number until reach zero (0) and the rocket will be launched
  - The function prototype would be:

```
void countDown(int initialCount);
// decrement by one initialCount until reaches
// zero to launch a rocket
```

countDown Function (recursive)

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

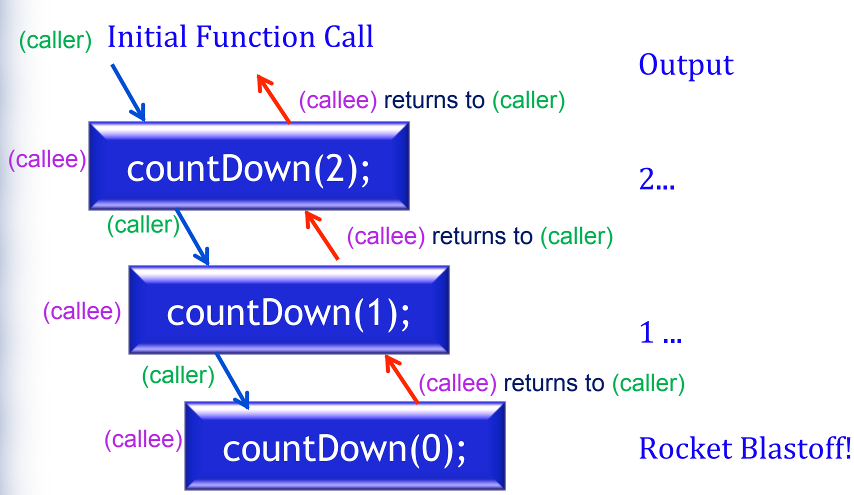
How Does Recursion Work?



Recursive Calls

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

How Does Recursion Work?



How the Stop Recursion

- Any recursive function must always include a test to determine if another recursive call should be executed, or if the recursion should stop with this call
  - In our countDown function, the test is:

  if (initialCount == 0)
- A different parameter value is passed to the function each time it is recursive called
- Eventually, the parameter reaches the value in the test, and the recursion stops
  - The test is commonly referred to as the base case

Stopping Recursion

Test to stop recursion

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

Value decreases for each call

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
- Direct recursion is mostly common used as Indirect recursion might be complex and not easy to design
- Tail recursive function
  - recursive function in whose last statement executed is the recursive call

Recursion vs. Iteration

- Every Loop can be implemented Recursively
- Loops alone cannot implement every Recursive algorithm
- Recursion
  - **Advantages:** Models certain algorithms most accurately; Results in shorter, simpler functions
  - **Disadvantages:** May not execute very efficiently
- Iteration
  - **Advantages:** Executes more efficiently than recursion
  - **Disadvantages:** Often is harder to code or understand

Exercise

- Re-write the countDown function using an iteration (looping) based implementation

```
void countDown(int initialCount)
{
        while (initialCount != 0)
        {
                cout << initialCount << "...\n\n";
                initialCount --;
        }
        cout << "Rocket Blastoff!";
}
```

Loop & Recursion: Comparison

```
int main()
{
  int countDownCnt;

  countDownCnt = 5;


  while(countDownCnt != 0)
  {

     cout << countDownCnt
          << "...\n\n";


     countDownCnt--;
  }

  cout << "Rocket Blastoff!";
  return 0;
```

Initialize

Check

Change

```
void countDown(int currentCount);
int main()
{
  int initialCount = 5;

  countDown(initialCount);
  return 0;
}
void countDown(int currentCount)
{
        // Base Case
        if (currentCount == 0)
            cout << "Rocket Blastoff!";
        else
        {
            cout << currentCount
                 << "...\n\n";
            // Recursive Call
            countDown(currentCount - 1);
        }
}
```

Designing Recursive Functions

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem
  - You need to define a problem in terms of a smaller version of itself
- The simpler-to-solve problem is known as the base case
  - Recursive calls stop when the base case is reached
    - If there is no base case the will be an infinite recursion
  - You have to establish a base case for the solution of the problem

Designing Recursive Functions

- Steps to design a recursive function:
  - Understand problem requirements
  - Determine limiting conditions
  - Identify base case(s) and provide a direct solution to each base case
  - Identify general cases and provide a solution to each general case in terms of smaller versions of itself (recursive call)

Recursive Factorial Function

- Design a recursive function to solve number factorial
- Recall: The factorial function for n (n!)

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

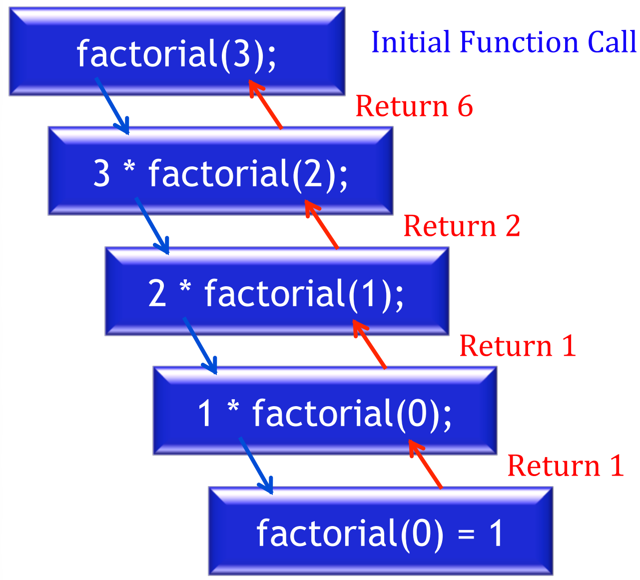
  n! = 1 if n = 0
- Can compute factorial of n if the factorial of (n-1) is known:

  n! = n * (n-1)!
- n = 0 is the base case

Recursive Factorial Function

```
int factorial (int num)
{
	if (num == 0)  //Base Case
		return 1;
		// factorial for n = 0
	else		//Recursive Case
		return num * factorial(num - 1);
		// use recursion to determine
		// factorial of (num - 1)
}
```

Calculating the Factorial



Exercise

- Re-write the factorial function using an iteration (looping) based implementation

```
int factorial (int num)
{
        int fact;
        int i;
        fact = 1;
        // loop through the sequence of intergers
        for (i = 1; i <= num; i++)
            fact *= i;
        return fact;
}
```

Other Application for Recursion

- Recursive Linked List Operations
  - Compute the size of (number of nodes in) a list
  - Traverse the list in reverse order
- Recursive Binary Function

Size of a Linked List

- Uses a pointer to visit each node
- Algorithm:
  - pointer starts at head of list
  - if pointer is NULL,
    - return 0 (base case)
  - else, return 1 + number of nodes in the list pointed to by current node (recursive call)

Contents of a List in Reverse Order

- Algorithm:
  - pointer starts at head of list
  - if the pointer is NULL
    - return (base case)
  - else, advance to next node (recursive call)
  - Upon returning from recursive call, display contents of current node

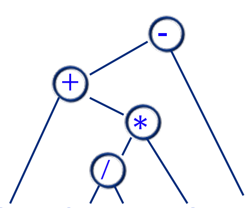
Recursive Binary Search Function

- Binary search algorithm can easily be written to use recursion
- Base case(s): desired value is found, or no more array elements to search
- Algorithm (assume array in ascending order):
  - if middle element of array segment is desired value, then done (base case)
  - else, if the middle element is too large, repeat binary search (recursive call) in first half of array segment
  - else, if the middle element is too small, repeat binary search (recursive call) on the second half of array segment

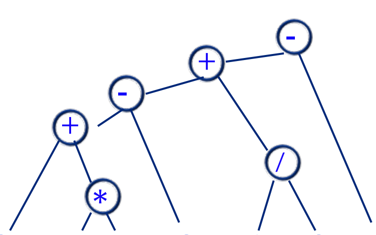
Algorithms that Require Recursion

- Maze Traversal
  - Given a maze to traverse, at each intersection
    - 1) Choose a direction to travel
    - 2) If a choice leads to a dead-end
      - **return to the previous choice made**
      - **choose an alternate direction to travel**
- Mathematical Order of Operations
  - Reading single digit mathematical expressions from a string
  - Calculate the value of the expressions
    - 2 + 8 / 4 * 2 - 1
    - 3 + 4 * 5 – 2 + 6 / 2 – 7

Mathematical Order of Operations



- 2 + 8 / 4 * 2 – 1



- 3 + 4 * 5 – 2 + 6 / 2 – 7

Mathematical Order of Operations

- Problem Description
  - Reading single digit mathematical expressions from a c-string
  - Calculate the value of the expressions
    - 2 + 8 / 4 * 2 - 1
    - 3 + 4 * 5 - 2 + 6 / 2 - 7
  - The algorithm to solve these expressions requires the use of a stack
  - Options:
    - 1) Use a loop & separate stack
    - 2) Use recursion which automatically uses the machine's call stack