Slides by Andrew DeOrio
and James Juett

# EECS 280

## Programming and Introductory Data Structures

## Tail Recursion

# Recursion review

- A function that calls itself is *recursive*

```
int factorial (int n) {
  // REQUIRES: n >= 0
  // EFFECTS:  computes n!

  if (n == 0) return 1;    // base case
  return n*factorial(n-1); // recursive step
}
```

# Recursion review

- Two features of problems make recursion a good solution
- **Subproblems** are similar and “smaller” (closer to a base case)
- A **Base Case** that can be solved without recursion

```
int factorial (int n) {
  if (n == 0) return 1;
  return n*factorial(n-1);
}
```

# Writing recursive functions

- Don’t try to do it all in your head
- Instead, treat it like an inductive proof

- Identify the “trivial” base case and write it explicitly
- *Assume there is a function that can solve smaller versions of the same problem*
  - Figure out how to get from the smaller solution to the bigger one

# Recursion review

- Two features of problems make recursion a good solution
- **Subproblems** are similar and “smaller” (closer to a base case)
- A **Base Case** that can be solved without recursion

```
int factorial (int n) {
  if (n == 0) return 1;
  return n*factorial(n-1);
}
```

# Recursion and the stack

```
int factorial (int n) {
  if (n == 0) return 1;
  return n * factorial(n-1);
}
```

- How many multiplications?
- How many stack frames (max)?
- This gives us a rough measure of how much memory is used by `factorial`

# Recursion and the stack

```
int factorial (int n) {
  if (n == 0) return 1;
  return n * factorial(n-1);
}
```

- How many multiplications?
  - x *(linear time)*
- How many stack frames (max)?
  - x+1 (*linear space*)
- This gives us a rough measure of how much memory is used by `factorial`

# An iterative version

```
int factorial(int n) {
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

- ]How many multiplications?

- How many stack frames (max)?

# An iterative version

```
int factorial(int n) {
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

- How many multiplications?
  - x (linear time)
- How many stack frames (max)?
  - 1 (*constant space*)

- Constant space means that even when n gets bigger, we still need only 1 stack frame

# Recursion vs. Iteration

Recursive

```
int factorial(int n){
  if (n == 0) return 1;
  return n *
    factorial(n-1);
}
```

- Computation is done **after** the “repetition”
- Multiplication happens during **passive flow**
- We need to keep track of each stack frame with each value of `n`

Iterative

```
int fact(int n){
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

- Computation is done **before** the “repetition”
- Multiplication happens in **active flow**. There is no passive flow in iteration.
- Once a value of `n` is multiplied into `result`, we can just forget it!

# Is all recursion like this?

- Does recursion always do work in the passive flow?
- No

- Example:

```
void countToTen(int x) {
  cout << start << endl;
  if (x == 10) return;
  countToTen(x + 1);
}
```

# Tail calls

- A function call is a **tail call** if it is the very last thing in its containing function
- The calling function has **no pending work** to do after a tail call (in the passive flow)
- Avoids saving the stack frame!

```
void countToTen(int x) {
  cout << start << endl;
  if (x == 10) return;
  countToTen(x + 1); //no more work to do
}
```

# Tail call optimization

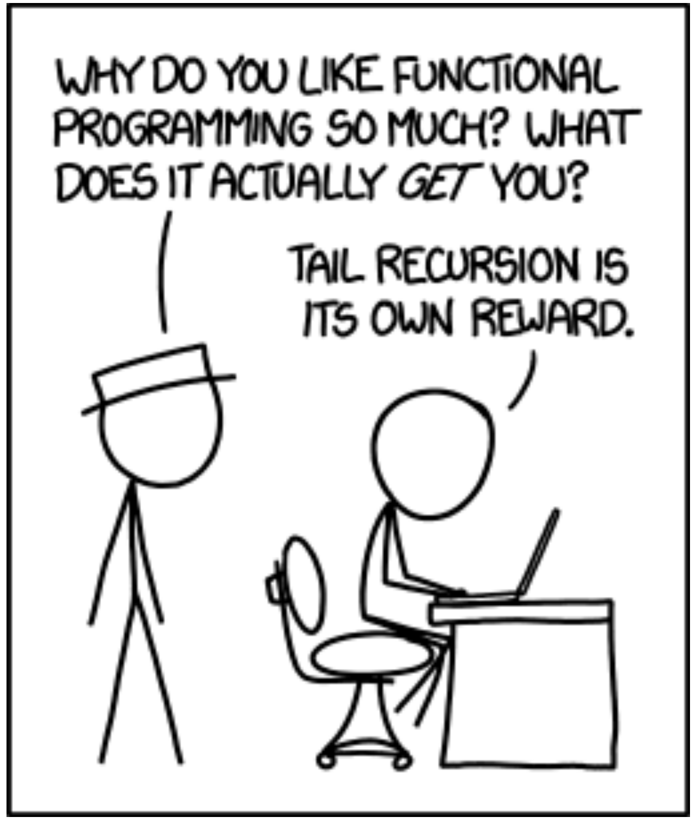
- Most compilers are able to recognize tail calls and optimize them
- "Reuse" the old stack frame for new recursive call
- `g++ -O2` includes tail call optimization
- Tail call optimization can take a recursive algorithm from linear to constant space complexity as long as it only makes tail calls

# Tail recursion

- We say a function is "**tail recursive**"[1] if ALL the recursive calls it makes are tail calls

```
//REQUIRES: x >= 1
//EFFECTS: returns floor of base 2 logarithm of x
int log2(int x){
 if (x == 1) return 0; // BASE CASE
 return 1 + log2(x/2); // RECURSIVE CASE
             // not a tail call
}
```

# Tail Recursion






Cartoon credit: xkcd.com

# Tail-recursive or not? Why?

```
void countdown1(int n) {
  if (n <= 0) {
    return;
  } else {
    cout << n << endl;
    countdown1(n-1);
    return;
  }
}
```

```
$ ./a.out
3
2
1
```

# Tail-recursive or not? Why?

```
void countdown1(int n) {
  if (n <= 0) {
    return;
  } else {
    cout << n << endl;
    countdown1(n-1);
    return;
  }
}
```

```
$ ./a.out
3
2
1
```

Tail-recursive. No work to do after recursive call returns.

# Tail-recursive or not? Why?

```
void countdown1(int n) {
  if (n <= 0) {
    return;
  } else {
    cout << n << endl;
    countdown1(n-1);
    return;
  }
} // for comparison
```

```
void countdown2(int n) {
  if (n <= 0) return;
  cout << n << endl;
  countdown2(n-1);
}
```

```
void countdown3(int n) {
  if (n > 0) {
    cout << n << endl;
    countdown3(n-1);
  }
}
```

Bonus question: identify the base case and recursive steps.

# Tail-recursive or not? Why?

```
void countdown1(int n) {
  if (n <= 0) {
    return;
  } else {
    cout << n << endl;
    countdown1(n-1);
    return;
  }
} // for comparison
```

Tail-recursive. No work to do after recursive call returns.

```
void countdown2(int n) {
  if (n <= 0) return;
  cout << n << endl;
  countdown2(n-1);
}
```

```
void countdown3(int n) {
  if (n > 0) {
    cout << n << endl;
    countdown3(n-1);
  }
}
```

# Tail-recursive or not? Why?

```
void countdown4_help
(int n, int i) {

  if (i > n) return;
  countdown4_help(n, i+1);
  cout << i << endl;
}

void countdown4(int n) {
  countdown4_help(n, 1);
}
```

# Tail-recursive or not? Why?

```
void countdown4_help
(int n, int i) {

  if (i > n) return;
  countdown4_help(n, i+1);
  cout << i << endl;
}

void countdown4(int n) {
  countdown4_help(n, 1);
}
```

Not tail-recursive. A helper function does not always make a tail-recursive function!

# Example: countdown4(3)

```
void countdown4_help
(int n, int i) {

  if (i > n) return;
  countdown4_help(n, i+1);
  cout << i << endl;
}

void countdown4(int n) {
  countdown4_help(n, 1);
}
```

# Tail-recursive or not? Why?

```
void countdown5(int n) {
  if (n <= 0) return;
  cout << n << endl;
  if (n % 2) { // n is odd
    countdown5(n-1);
    return;
  } else {     // n is even
    cout << n-1 << endl;
    countdown5(n-2);
    return;
  }
}
```

# Tail-recursive or not? Why?

```
void countdown5(int n) {
  if (n <= 0) return;
  cout << n << endl;
  if (n % 2) { // n is odd
    countdown5(n-1);
    return;
  } else {     // n is even
    cout << n-1 << endl;
    countdown5(n-2);
    return;
  }
}
```

Tail-recursive. It's OK to have two recursive calls, as long as there is no pending computation after the call.

# A tail recursive factorial

Recursive

```
int factorial(int n){
  if (n == 0) return 1;
  return n *
    factorial(n-1);
}
```

Iterative

```
int fact(int n){
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

## Tail Recursive

```
int factorial(int n){
  int result = 1;
  if (n == 0) return result;
  result *= n;
  return factorial(n-1);
}
```

Tail Call

Note: this function doesn't get the right answer.
We'll fix it soon.

# A tail recursive factorial

**Iterative**

```
int fact(int n){
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

**Tail recursive**

```
int factorial(int n){
  int result = 1;
  if (n == 0) return result;
  result *= n;
  return factorial(n-1);
}
```

- What's wrong with this tail recursive version?

# A tail recursive factorial

**Iterative**

```
int fact(int n){
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

**Tail recursive**

```
int factorial(int n){
  int result = 1;
  if (n == 0) return result;
  result *= n;
  return factorial(n-1);
}
```

- What's wrong with this tail recursive version?
- We get a new version of `result` whose value is 1 in every new stack frame, so `result` never grows
- This isn't a problem in the iterative version because there's only one stack frame and only one `result` variable

# A tail recursive factorial

**Iterative**

```
int fact(int n){
  int result = 1;
  while (n > 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

**Tail recursive**

```
int factorial(int n, int result){
  int result = 1;
  if (n == 0) return result;
  result *= n;
  return factorial(n-1, result);
}
```

- Make `result` a parameter
- There is still a separate `result` object for each call (stack frame), but we can pass the updated value along to the next

# A tail recursive factorial

- A few tweaks to the style

```
int factorial(int n, int result){
  int result = 1;
  if (n == 0) return result;
  result *= n;
  return factorial(n-1, n * result);
}
```

# A tail recursive factorial

- But wait ... we broke the procedural abstraction
- We changed the function signature

```
int factorial(int n, int result){
  if (n == 0) return result;
  return factorial(n-1, n * result);
}
```

# A tail recursive factorial

- Use a helper function to fix the procedural abstraction

```
int factorial_helper(int n, int result){
  if (n == 0) return result;
  return factorial_helper(n-1, n * result);
}

int factorial(int n) {
  return factorial_helper(n ,1);
}
```

# A tail recursive factorial

- Draw the stack frames

```
int factorial_helper(int n, int result){
  if (n == 0) return result;
  return factorial_helper(n-1, n * result);
}

int factorial(int n) {
  return factorial_helper(n ,1);
}

int main() {
  factorial(3);
  return 0;
}
```