CSE 143

Recursion

Chapter 2

Advanced Reading: Chapter 5

3/25/2001 I-1

Insist without Iterating

```
char InsistOnYorN (void) {
	char answer;
	cout << "Please enter y or n: " << endl;
	cin >> answer;
	switch (answer) {
		case 'y': return 'y';
		case 'n': return 'n';
		default:
			return InsistOnYorN( );
	}
}
```

3/25/2001 I-2

Recursion

- A **recursive** definition is one which is defined in terms of itself
- Examples:
  - Compound interest: "The **value after 10 years** is equal to the interest rate times the **value after 9 years**."
  - A phrase is a "palindrome" if the 1st and last letters are the same, and what's inside is itself a palindrome (or is empty).

3/25/2001 I-3

Computer Science Examples

- Recursive procedure: a procedure that invokes itself
- Recursive data structures: a data structure may contain a pointer to an instance of the same type

```
struct Node {
  int data;
  Node *next;
};
```

- Recursive (inductive) definitions: if `A` and `B` are arithmetic expressions, then `(A) + (B)` is a valid expression

3/25/2001 I-4

Factorial

`n!` ("`n` factorial") can be defined in two ways:

- Non-recursive definition

  `n! = n * (n-1) (n-2) … * 2 * 1`

- Recursive definition

$$n! = \begin{cases} 1 & \text{, if } n = 1 \\ n * (n-1)! & \text{, if } n > 1 \end{cases}$$

0! is usually defined to be 1

Undefined for negative numbers

3/25/2001 I-5

Factorial (2)

- How do we write a function that reflects the recursive definition?

```
int factorial(int n) {
  assert(n >= 1);
  if ( n == 1 )
    return 1;
  else
    return n * factorial(n-1);
}
```

- The `factorial` function invokes itself.
- How can this work?

3/25/2001 I-6

What Makes Recursion Work?

- Review: local variables and formal params are
  - allocated when { } block is entered,
  - deleted when block is exited.
- Here's how:
  - Whenever a function is called (or { } block is entered), a new "activation record" is created, containing:

    ```
    -- a separate copy of all local variables and parameters
    -- control info, such as where to return to
    ```
  - Activation record is alive until the function returns

    ```
    Then it is destroyed
    ```
  - This applies *whether or not* function is recursive!

Simplified Model

- ***Every time you call a function, you get a fresh copy of it.***
  - **If you call recursively, you end up with more than one copy of the function active**
- ***When you exit a function, only that copy of it goes away.***
- In reality...
  - there's only one copy of the code (instructions), but separate copies of the data (variables and parameters)

Tracing the Process

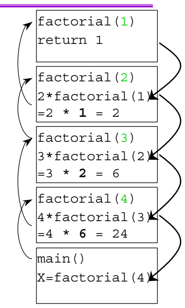
- To trace function calls
  - draw a box each time a function is called.
  - draw an arrow from caller to called function
  - label data (local vars, params) inside the box
  - indicate the returned value (if any)
  - cross out the box after return

    ```
    and don't reuse it!
    ```
- Question: how is this different from a "static call graph"?
- Note that *no* special handing is needed just because a function happens to be recursive!

Trace Example

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

...
int main (void) {
  int x = factorial(4);
  cout << "4! = " << x << endl;
...
```



What is Recursion?

- A programming technique
  - a function calling itself
- An approach to problem-solving
  - Look for smaller problems similar to the larger problem
- A way of thinking about algorithms
  - Turns out to lead to good mathematical analyses
- The natural algorithmic technique when recursive data structures are involved

Recursion takes practice

- Eventually it becomes a natural habit of thought

What About Efficiency??

- Is recursion faster/slower/smarter/more powerful etc. than iteration? We'll talk about that, too -- later
- Learning *how* to drive a car, vs learning *when and where* to drive a car.
  - Different kinds of knowledge
  - The first especially requires focused practice

Infinite Recursion

- **Mathematically:**
  - **n! = n * (n-1)! = (n-1)! * n**
  - **Why not program it in that order?**

```
int BadFactorial(n) {
  int x = BadFactorial(n-1);
  if ( n == 1 )
     return 1;
  else
     return n * x;
}
```

- What is the value of `BadFactorial(2)`?
- The rule: Must always have some way to make recursion stop, otherwise it runs forever:

Using Recursion Properly

- For correct recursion (recursion that does something useful and eventually stops), need two parts:
  1. One or more ***base cases*** that are not recursive
     `if ( n == 1 ) return 1; // no recursion in this case`
  2. One or more ***recursive cases*** that operate on *smaller* problems that get *closer* to a base case
     return n * **factorial(n-1)**;
     //factorial(n-1) is a smaller problem than factorial (n)
- The base case(s) should **always** be checked before the recursive calls

Linear Search

- Problem statement: Given an array `A` of `N` `int`s, search for an element with value `x`
- First, an iterative solution:

```
// Return index of x if found, or -1 if not

int Find (int A[], int N, int x)
{
  for ( int i = 0; i < N; i++ )
    if ( A[i] == x )
      return i;
  return -1;
}
```

- How efficient is this?
  - Might find x on first step, or you might have to check all N values
  - On average, it takes about N/2 times through the loop

Binary Search

- If array is *sorted*, we can search faster
  - Start search in middle of array
    `if x is right there in the middle, you're done`
  - If `x` is less than middle element, need to search only in lower half
  - If `x` is greater than middle element, need to search only in upper half
  - continue the seach within the half chosen
- Why is this faster than linear search?
  - At each step, linear search throws out *one* element
  - Binary search throws out *half* of remaining elements
- Why is recursion natural here?

Example

Find 26 in the following sorted array:

```
1 3 4 7 9 11 15 19 22 24 26 31 35 50 61
                 ↑
                      22 24 26 31 35 50 61
                               ↑
                      22 24 26
                          ↑
                            26
                             ↑
```


Binary Search (Recursive)

```
int find(int A[], int size, int x) {
  return findInRange(A, x, 0, size-1);
}

int findInRange(int A[], int x, int lo, int hi) {
   if (lo > hi) return -1;
   int mid = (lo+hi) / 2;
   if (x == A[mid])
     return mid;
   else if (x < A[mid])
     return findInRange(A, x, lo, mid-1);
  else
     return findInRange(A, x, mid+1, hi);
}
```

Kick-off and Helper Functions

- Previous example illustrates a common pattern:
  - Top-level "kick-off" function
    - Not itself recursive
    - Starts the recursion going
    - Returns the ultimate answer
  - Helper function
    - Contains the actual recursion
    - May require additional parameters to keep track of the recursion
- Client programs only need call the kick-off function

3/25/2001 I-19

Recursion with Array Params

```
double sum (double iArray [ ], int from, int to) {
  //find the sum of all elements in the array between index "from" and index "to"
        if (from > to)
                return 0.0;
        return iArray[from] + sum (iArray, from+1, to);
}
//Client code:
double CashValues[200];
...
double total = sum (CashValues, 0, 199);
```

- Implemented without kick-off/helper structure
  - but might benefit from having it

3/25/2001 I-20

Recursion vs. Iteration

- When to use recursion?
  - Processing recursive data structures
  - "Divide & Conquer" algorithms:
    1. Divide problem into subproblems
    2. Solve each subproblem recursively
    3. Combine subproblem solutions
- When to use iteration instead?
  - Nonrecursive data structures
  - Problems without obvious recursive structure
  - Problems with obvious iterative solution
  - Functions with a large "footprint"
    - especially when many iterations are needed

3/25/2001 I-21

In Theory...

- Any iteration can be rewritten using recursion, and vice-versa (at least in theory)
  - but the rewrite is not always simple!
- Iteration is generally more efficient
  - somewhat faster
  - takes less memory
- A compromise:
  - If the problem is naturally recursive, design the algorithm recursively first
  - Later convert to iteration if needed for efficiency
  - General principle: "Make it right, then make it efficient"

3/25/2001 I-22

So Should You Avoid the R-word?

- If a single recursive call is at the very end of the function:
  - Known as *tail recursion*
  - Easy for a smart compiler to automatically rewrite using iteration (but not commonly done by C/C++ compilers)
- Recursive problems that are not tail recursive are harder to automatically rewrite nonrecursively
  - Usually have to simulate recursion with a stack

3/25/2001 I-23

Summary

- Recursion is something defined in terms of itself
- Activation records make it work
- Elements of recursive functions
  - Base case(s)
  - Recursive case(s)
    - Base case always checked first
- When to use/when to avoid

As the course unfolds, we'll see more and more cases where recursion is natural to use

3/25/2001 I-24