CSE 143

Recursion

[Chapter 5]

Recursion

- A **recursive** definition is one which is defined in terms of itself
- Example:
  - 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).

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 definitions: if **A** and **B** are postfix expressions, then **A B +** is a postfix expression

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

Factorial (2)

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

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

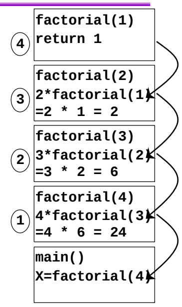
- Note that the `factorial` function calls itself.
- How can this work?

Activation Records

- Remember that local variables and parameters are allocated when a function is entered, deleted when the function exits (automatic storage).
- Here's how:
  - Whenever a function is called, a new activation record is pushed on the runtime stack, containing:
    `-- a separate copy of all local variables and parameters`
    `-- control info (e.g. return address)`
  - Activation record is popped at end of function (or block)
- A recursive function call is <u>no different</u> in this respect
  - Each recursive call has its own copy of locals

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;
...
```



Infinite Recursion

- Must always have some way to make recursion stop, otherwise it runs forever:

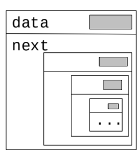
```
int BadFactorial(n) {
  int x = BadFactorial(n-1);
  if ( n == 1 )
    return 1;
  else
    return n * x;
}
```

- What is the value of **`BadFactorial(2)`**?

Infinite Recursion (2)

Similar problems with incorrect recursive definitions of data types:

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



How does **`Node *next;`** solve this problem?

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;`
  2. One or more ***recursive cases*** that operate on *smaller* problems that get *closer* to the base case(s)
     `return n * factorial(n-1);`
- The base case(s) should **always** be checked before the recursive calls

Recursive Data Structures

- Many data structures are defined in terms of (pointers to) other instances of themselves
  - Linked Lists and Trees (coming soon) are examples:

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

struct TreeNode {
  int data;
  TreeNode *left;
  TreeNode *right;
};
```

  - Recursive data structures suggest recursive algorithms

Printing a Linked List

```
void print(Node* first) {
  if (first == NULL)
    return;
  else {
    cout << first->data << " ";
    print(first->next);
  }
}
```

- How many recursive calls do we make when printing the list <1,2,3,4>?

Printing in Reverse Order

At first, seems difficult

All the pointers point only forward.

Recursion to the rescue!

```
void print(Node* first) {
   if (first == NULL)
       return;
   else {
       print(first->next);
       cout << first->data << " ";
     }
}
```

3/10/99 437

Summing a List

```
int listSum(Node* list) {
  if (list == NULL)
    return 0;        // empty list has sum == 0
  else
    return list->data + listSum(list->next);
}
```

- How would you modify this to count the length of a list? Add N to each element of a list?
- What is the cost in time compared to a loop? How much space does it take?

3/10/99 438

List Remove

- Remove nodes with a given data value from list

```
Node* ListRemove(Node *first, int v){
  if (first == NULL)
    return NULL;
  else if (first->data != v){
    Node* newNode = new Node;
    newNode->data = first->data;
    newNode->next = ListRemove(first->next, v);
    return newNode;
  }
  else
    return ListRemove(first->next, v);
}
```

3/10/99 439

Recursion: Not Just for Lists

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

3/10/99 440

Insist without Iterating

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

3/10/99 441

What does this function do?

```
int mystery (int x) {
  assert (x > 0);
  if (x == 1)
     return 0;
  int temp = mystery (x / 2);
  return 1 + temp;
}
```

3/10/99 442

How about this one

```
int g(int n)
{
    if (n <= 1)
        return 1;
    else if (n % 2 == 0)        // n even
        return 1 + g(n / 2);
    else                        // n odd
        return 1 + g(3* n + 1);
}

g(7) = 1 + g(22) = 2 + g(11) = 3 + g(34) =
4 + g(17) = 5 + g(52) = 6 + g(26) = 7 + g(13) =
8 + g(40) = 9 + g(20) = 10 + g(10) = 9 + g(5) =
10 + g(16) = 11 + g(8) = 12 + g(4) =
13 + g(2) = 14 + g(1) = 15
```

3/10/99 443

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 obvious iterative structure
- Any iteration can be rewritten using recursion, and vice-versa (at least in theory)
- Iteration generally uses less (stack) space and is somewhat faster

3/10/99 444

Which is Better?

- If a single recursive call is at the very end of the function:
  - Known as *tail recursion*
  - Easy to rewrite iteratively
- Recursive problems that are not tail recursive are harder to write nonrecursively
  - Usually have to simulate recursion with a stack
- Some programming languages provide no loops (loops are implemented through **if** and recursion)

3/10/99 445

Summary

- Recursion is something defined in terms of itself
  - Recursive procedures
  - Recursive data structures
  - Recursive definitions
- Activation records make it work
- Two parts of all recursive functions
  - Base case(s)
  - Recursive case(s)
  - Base case always checked first
- Examples
  - Factorial (n!)
  - Recursive linked list manipulation

3/10/99 446