Programming Abstraction in C++

Eric S. Roberts and Julie Zelenski

Stanford University
2010

Chapter 6. Recursive Procedures

Outline

Introduction

So far the problems are simple. Most of them, like factorial and Fibonacci sequence, can be solved by iterative method. Why recurssion?

Introduction

So far the problems are simple. Most of them, like factorial and Fibonacci sequence, can be solved by iterative method. Why recurssion?

We will solve some complex problems, very difficult to solve using iterative technique, whereas recursion provides elegant solutions.

Introduction

So far the problems are simple. Most of them, like factorial and Fibonacci sequence, can be solved by iterative method. Why recurssion?

We will solve some complex problems, very difficult to solve using iterative technique, whereas recursion provides elegant solutions.

Key. Decompose a problem to a smaller one of the same form, then apply the recursive leap of faith.

Outline

Tower of Hanoi

Function prototype:

```
MoveTower(int n, char start, char finish, char temp);
```

n: Number of disks

`start`, `finish`, `temp`: Three spires.

Rules

1. You can only move one disk at a time.
2. You are not allowed to move a larger disk on top of a smaller disk.

Applying the recursive leap of faith

The case of n.

Suppose that we have a solution for the problem of smaller size $n-1$ of the same form.

1. Move the top $n-1$ disks from `start` to `temp` using the solution for $n-1$;
2. Move the disk (largest) from `start` to `finish`;
3. Move the $n-1$ disks from `temp` to `finish` using the solution for $n-1$;

Applying the recursive leap of faith

The case of n.

Suppose that we have a solution for the problem of smaller size $n-1$ of the same form.

1. Move the top $n-1$ disks from `start` to `temp` using the solution for $n-1$;
2. Move the disk (largest) from `start` to `finish`;
3. Move the $n-1$ disks from `temp` to `finish` using the solution for $n-1$;

Stopping point (simple problem).
Each time the problem size is reduced by one, eventually the size is reduced to one. The solution is simple.

```
void MoveSingleDisk(char start, char finish);
```

A recursive algorithm

```
void MoveTower(int n, char start, char finish, char temp) {
    if (n == 1) {
        MoveSingleDisk(start, finish);
    } else {
        MoveTower(n-1, start, temp, finish);
        MoveSingleDisk(start, finish);
        MoveTower(n-1, temp, finish, start);
    }
}
```

A recursive algorithm

```
void MoveTower(int n, char start, char finish, char temp) {
    if (n == 1) {
        MoveSingleDisk(start, finish);
    } else {
        MoveTower(n-1, start, temp, finish);
        MoveSingleDisk(start, finish);
        MoveTower(n-1, temp, finish, start);
    }
}
```

Validating the algorithm.

Rule 1 (move one at a time). Only `MoveSingleDisk` moves disk.

Rule 2. When moving the top $n-1$ disks, we leave the bottom (largest) disks behind. Any disk put on top of them is smaller.

A recursive algorithm

```
void MoveTower(int n, char start, char finish, char temp) {
    if (n == 1) {
        MoveSingleDisk(start, finish);
    } else {
        MoveTower(n-1, start, temp, finish);
        MoveSingleDisk(start, finish);
        MoveTower(n-1, temp, finish, start);
    }
}
```

Validating the algorithm.

Rule 1 (move one at a time). Only `MoveSingleDisk` moves disk.

Rule 2. When moving the top $n - 1$ disks, we leave the bottom (largest) disks behind. Any disk put on top of them is smaller.

Still not convinced?

Tracing the process, $n = 3$

```
void MoveTower(int n, char start, char finish, char temp) {
    if (n == 1) {
        MoveSingleDisk(start, finish);
    } else {
        MoveTower(n-1, start, temp, finish);
        ...
    }
}
```

```
MoveTower        n    start finish temp
                 3    A     B      C
                   MoveTwer(2, start, temp, finish)
MoveTower        n    start finish temp
                 2    A     C      B
                   MoveTower(1, start, temp, finish)
MoveTower        n    start finish temp
                 1    A     B      C
                   MoveSingleDisk(start, finish)
MoveSingleDisk   start  finish
                 A      B
                   return
```

Tracing the process (cont.)

```
void MoveTower(int n, char start, char finish, char temp) {
    if (n == 1) {
        MoveSingleDisk(start, finish);
    } else {
        MoveTower(n-1, start, temp, finish);
        MoveSingleDisk(start, finish);
        ...
    }
}
```

MoveTower	**n** **start** **finish** **temp** $\boxed{3}$ $\boxed{A}$ $\boxed{B}$ $\boxed{C}$ **MoveTwer(2, start, temp, finish)**
MoveTower	**n** **start** **finish** **temp** $\boxed{2}$ $\boxed{A}$ $\boxed{C}$ $\boxed{B}$ **MoveSingleDisk(start, finish)**
MoveSingleDisk	**start** **finish** $\boxed{A}$ $\boxed{C}$ **return**

Tracing the process (cont.)

```
void MoveTower(int n, char start, char finish, char temp) {
    if (n == 1) {
        MoveSingleDisk(start, finish);
    } else {
        MoveTower(n-1, start, temp, finish);
        MoveSingleDisk(start, finish);
        MoveTower(n-1, temp, finish, start);
    }
}
```

```
MoveTower       n   start finish temp
                3   A     B      C
                MoveTwer(2, start, temp, finish)
MoveTower       n   start finish temp
                2   A     C      B
                MoveTower(1, temp, finish, start)
MoveTower       n   start finish temp
                1   B     C      A
                MoveSingleDisk(start, finish)
MoveSingleDisk  start finish
                B     C
                return
```

Outline

Generating permutations

Function prototype

```
void ListPermutations(string str);
```

Example

```
ABC
ACB
BAC
BCA
CAB
CBA
```

Generating permutations (cont.)

For each character in the string, move it to the prefix, then permute the rest, a shorter string.

It is convenient to set up two strings: `prefix` and `rest`, initially, an empty `prefix`.

Generating permutations (cont.)

For each character in the string, move it to the prefix, then permute the rest, a shorter string.

It is convenient to set up two strings: `prefix` and `rest`, initially, an empty `prefix`.

Applying the recursive leap of faith: For a string of length n, suppose we have a solution for a string of length $n - 1$, then

for each character in the string
    remove the character from string and
    put it in `prefix`;
    permute `rest` (length $n - 1$);

Stopping point: `rest` is empty.

Generating permutations (cont.)

A wrapper:

```
void ListPermutations(string str) {
    RecursivePermute("", str);
}

void RecursivePermute(string prefix, string rest) {
    if (rest == "") {
        cout << prefix << endl;
    } else {
        for (int i = 0; i < rest.length(), i++) {
            string newPrefix = prefix + rest[i];
            string newRest = rest.substr(0, i)
                            + rest.substr(i+1);
            RecursivePermute(newPrefix, newRest);
        }
    }
}
```

Generating permutations (cont.)

A wrapper:

```
void ListPermutations(string str) {
    RecursivePermute("", str);
}

void RecursivePermute(string prefix, string rest) {
    if (rest == "") {
        cout << prefix << endl;
    } else {
        for (int i = 0; i < rest.length(), i++) {
            string newPrefix = prefix + rest[i];
            string newRest = rest.substr(0, i)
                            + rest.substr(i+1);
            RecursivePermute(newPrefix, newRest);
        }
    }
}
```

Question: Why the wrapper?