Programming Abstraction in C++

Eric S. Roberts and Julie Zelenski

Stanford University
2010
Chapter 6. Recursive Procedures
Outline

1. Tower of Hanoi

2. Generating Permutations
Introduction

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

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

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 recursion?

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

1. Tower of Hanoi

2. Generating Permutations
Function prototype:

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

```c
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, $n = 3$

```c
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-1, temp, start, finish);
        MoveSingleDisk(start, finish);
    }
}
```

<table>
<thead>
<tr>
<th>MoveTower</th>
<th>n</th>
<th>start</th>
<th>finish</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MoveTower</th>
<th>n</th>
<th>start</th>
<th>finish</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MoveTower</th>
<th>n</th>
<th>start</th>
<th>finish</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MoveSingleDisk</th>
<th>start</th>
<th>finish</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

```c
return
```

The process starts with moving the largest disk from the source tower to the target tower using an auxiliary tower. Then, the previous disks are moved from the source tower to the auxiliary tower, which is followed by moving the largest disk back. Finally, the smallest disk is moved from the auxiliary tower to the target tower.
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);
        ...
    }
}

<table>
<thead>
<tr>
<th>MoveTower (n=3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n: 3</td>
</tr>
<tr>
<td>start: A, finish: B, temp: C</td>
</tr>
<tr>
<td>MoveTower (n=2, start, temp, finish)</td>
</tr>
<tr>
<td>n: 2</td>
</tr>
<tr>
<td>start: A, finish: C, temp: B</td>
</tr>
<tr>
<td>MoveSingleDisk (start, finish)</td>
</tr>
<tr>
<td>start: A, finish: C</td>
</tr>
<tr>
<td>return</td>
</tr>
</tbody>
</table>
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);
    }
}
Outline

1. Tower of Hanoi

2. Generating Permutations
Generating permutations

Function prototype

```c
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:

```cpp
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:

```cpp
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?