Introduction to “C” Programming

Gabriel Hugh Elkaim
Winter 2015

Recursion
Recursion

- Solving problems by breaking them into smaller parts
- "divide and conquer"
- Relies on the problem having self-similarity

Example

```c
int Factorial(int n)
{
    if (n <= 1)
    {
        return 1;
    }
    return n * Factorial(n - 1);
}
```

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Partial results pushed on stack
Factorial term replaced with result

5! = 5 * 4!
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Factorial term replaced with result

[3] \(4! = 4 \cdot 3!\)

[4] \(5! = 5 \cdot 4!\)

Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
</tr>
<tr>
<td>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Factorial term replaced with result

[2] \(3! = 3 \cdot 2!\)

[3] \(4! = 4 \cdot 3!\)

[4] \(5! = 5 \cdot 4!\)
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Partial results pushed on stack
Factorial term replaced with result

[1] 2! = 2 * 1!
[2] 3! = 3 * 2!
[3] 4! = 4 * 3!
[4] 5! = 5 * 4!

Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Partial results pushed on stack
Factorial term replaced with result

[0] 1! = 1
[1] 2! = 2 * 1!
[2] 3! = 3 * 2!
[3] 4! = 4 * 3!
[4] 5! = 5 * 4!
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0] 1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td>Factorial(1)</td>
<td>[1] 2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>Factorial(2)</td>
<td>[2] 3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
<td>[3] 4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td>[4] 5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>
**Recursion**

*Evaluation of Recursive Functions*

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
</tr>
<tr>
<td>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] 1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td>[1] 2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>[2] 3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td>[4] 5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0] (1! = 1)</td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td>[1] (2! = 2 \times 1!)</td>
<td>= 2 \times 1 = 2</td>
</tr>
<tr>
<td></td>
<td>[2] (3! = 3 \times 2!)</td>
<td>= 3 \times 2 = 6</td>
</tr>
<tr>
<td></td>
<td>[3] (4! = 4 \times 3!)</td>
<td>= 4 \times 6 = 24</td>
</tr>
<tr>
<td></td>
<td>[4] (5! = 5 \times 4!)</td>
<td>= 5 \times 24 = 120</td>
</tr>
</tbody>
</table>

Recursion
Summary

• Usable for solving problems that are divided into subproblems
  – Divide and conquer
• Initial conditions must be similar to conditions for any of the subproblems
  – No difference between solving the smaller computation stand-alone versus as part of a larger computation
• Requires well-defined termination condition
Recursion

Caveats

- Problem must have a well-defined termination condition/base case
- Must have enough memory
  - Memory use high from filling the function stack

Recursion

Limitations

- Limited stack space

Stack (top)

| Factorial(3) |
| Factorial(4) |
| Factorial(5) |
| Factorial(6) |
| Factorial(7) |
| Factorial(8) |
| Factorial(9) |
| Factorial(10) |
Recursion
Multiple recursion

- Recursion is not limited to a single function call

Example

```c
int Fibonacci(int n)
{
    if (n <= 1) {
        return 1;
    }
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

```
Stack (top)

Partial results pushed on stack
Function call replaced with result
```

```
F_4 = F_3 + F_2
```

\[ F_4 = F_3 + F_2 \]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

\[
\begin{align*}
\text{Stack (top)} & \\
\text{Fibonacci(3)} & \\
\text{Fibonacci(4)} & \\
\end{align*}
\]

\[F_3 = F_2 + F_1\]
\[F_4 = F_3 + F_2\]

Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

\[
\begin{align*}
\text{Stack (top)} & \\
\text{Fibonacci(2)} & \\
\text{Fibonacci(3)} & \\
\text{Fibonacci(4)} & \\
\end{align*}
\]

\[F_2 = F_1 + F_0\]
\[F_3 = F_2 + F_1\]
\[F_4 = F_3 + F_2\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

\[
\begin{align*}
F_1 &= 1 \\
F_2 &= F_1 + F_0 \\
F_3 &= F_2 + F_1 \\
F_4 &= F_3 + F_2
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

  **Stack** (top)
  
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

  Fibonacci(2)
  Fibonacci(3)
  Fibonacci(4)

  Partial results pushed on stack
  Function call replaced with result

  \[
  F_2 = 1 + F_0 \\
  F_3 = F_2 + F_1 \\
  F_4 = F_3 + F_2
  \]

Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

  **Stack** (top)
  
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

  Fibonacci(0)
  Fibonacci(2)
  Fibonacci(3)
  Fibonacci(4)

  Partial results pushed on stack
  Function call replaced with result

  \[
  F_0 = 1 \\
  F_2 = 1 + F_0 \\
  F_3 = F_2 + F_1 \\
  F_4 = F_3 + F_2
  \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

- Fibonacci(0)
- Fibonacci(2)
- Fibonacci(3)
- Fibonacci(4)

Partial results pushed on stack

Function call replaced with result

\[
F_0 = 1
\]
\[
F_2 = 1 + F_0
\]
\[
F_3 = F_2 + F_1
\]
\[
F_4 = F_3 + F_2
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_3 = 2 + F_1 \]
\[ F_4 = F_3 + F_2 \]

Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Fibonacci(1)
Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_1 = 1 \]
\[ F_3 = 2 + F_1 \]
\[ F_4 = F_3 + F_2 \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

F_3 = \[2 + 1\]
F_4 = \[F_3 + F_2\]

Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

F_4 = \[3 + F_2\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

\[ F_2 = F_1 + F_0 \]
\[ F_4 = 3 + F_2 \]

Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(1)</td>
</tr>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

\[ F_1 = 1 \]
\[ F_2 = F_1 + F_0 \]
\[ F_4 = 3 + F_2 \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

\[
\begin{align*}
\text{Stack (top)} & \quad \text{Partial results pushed on stack} \quad \text{Function call replaced with result} \\
\text{Fibonacci(2)} & \\
\text{Fibonacci(4)} & \\
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)
- Fibonacci(2)
- Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_2 = 1 + 1 \]
\[ F_4 = 3 + F_2 \]

Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)
- Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_4 = 3 + 2 \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Partial results pushed on stack
Function call replaced with result

\[ F_4 = 5 \]

Recursion
Self-similarity

• A structure that is similar to part of itself
  – Example: fractals

• Computation and data must be self-similar for recursion

• Previous examples only dealt with single integers

• But what about more complicated data?
Recursion

- For example, operating on a string
  - How to do that in C?
- Passing a single string through functions is trivial.
- But what about splitting the string up?

Recursion

- Recursive word count
  - Count 1 word per function call

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p)
    {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```
```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```
Recursion
Complex data example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

```
1 1
```

Recursion
Complex data example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

```
1 1 1
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

1 1 2

1 1 2

Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

1 3

1 3
Recursion
Complex data example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

Binary trees

\[
\begin{align*}
\text{Morse UB} & \quad (\cdot) \\
\text{ } & \quad \text{T} \quad \text{S} \\
\text{ } & \quad \text{S} \quad \text{0} \\
\text{ } & \quad \text{S} \quad \text{0} \\
\text{ } & \quad \text{S} \quad \text{0} \\
\text{ } & \quad \text{S} \quad \text{0} \\
\text{ } & \quad \text{S} \quad \text{0} \\
\end{align*}
\]
Binary trees

- ADT where nodes:
  - Have 0, 1, or 2 children
  - Have a single parent
- Generally nodes only know their children
- Generally traversal is top-down
  - From parents to children

Binary trees

- We will only discuss full binary trees
- Size is $2^n - 1$
Binary trees

Traversal

- Binary trees store data at each node
- So the tree must be traversed to access the node that has the data we want

Binary trees

Self-similarity
Binary trees
Self-similarity
Binary trees
Self-similarity

Binary trees
Serialization

- A linear representation of a tree
Binary trees

Serialization

- Root node at the 1\textsuperscript{st} element
- Left tree at the 2\textsuperscript{nd} element
- Right tree at 2^{n-1} element
Binary trees
Serialization

A B C D

Binary trees
Serialization

A B C

A

B

C
Binary trees
Serialization

A B C D

A
B C D

Binary trees
Serialization

A B C D

A
B C D
Binary trees
Serialization

LEFT FIRST DESCENT

A B C D E F G

A
B
C D

E
F G

A
B
C D

E
F G
Binary trees
Serialization

A B C D E F G

Binary trees
Serialization

8 2 4 7 2 5

8 2 4 7 2 5