CMPE-013/L

Introduction to “C” Programming

Bryant Wenborg Mairs

Spring 2014
typedef struct {
    uint8_t id;
    float tXPeriod;
    Message *msg;
} Message;  // attribute(1) packed

Message *m;

m = malloc(sizeof(Message));

if (m) {
    m->id = 22;
    m->tXPeriod = 0.33;
    m[0].id = 22;
    m[0].tXPeriod = 0.33;

    Segment s = line(1, 5, 3, 3);
    s.p1.x = -1;
}
Recursion
Recursion

- Solving problems by breaking them into smaller parts
- Start with smaller subproblems
- Relies on the problem having self-similarity

Example

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

$5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = \ldots$

- Divide and conquer
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

\[ 5! = 5 \times 4! \]
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. Factorial(5)
2. Factorial(4)

3. 4! = 4 * 3!
4. 5! = 5 * 4!
Recursion
Evaluation of Recursive Functions

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

Partial results pushed on stack
Factorial term replaced with result

Stack (top)

Factorial(3)
Factorial(4)
Factorial(5)

[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)

| Factorial(2) |
| Factorial(3) |
| Factorial(4) |
| Factorial(5) |

Partial results pushed on stack
Factorial term replaced with result

[1] \( 2! = 2 \times 1! \)
[2] \( 3! = 3 \times 2! \)
[3] \( 4! = 4 \times 3! \)
[4] \( 5! = 5 \times 4! \)
Recursion
Evaluation of Recursive Functions

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

Stack (top)

Factorial(1)
Factorial(2)
Factorial(3)
Factorial(4)
Factorial(5)

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></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>
<tr>
<td>Factorial(5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursion
Evaluation of Recursive Functions

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

Stack (top)

Partial results pushed on stack

Factorial(2)
Factorial(3)
Factorial(4)
Factorial(5)

Factorial term replaced with result

[0] 1! = 1 = 1
[1] 2! = 2 * 1! = 2 * 1 = 2
[2] 3! = 3 * 2! = 3 * 2 = 6
[4] 5! = 5 * 4! = 5 * 24 = 120

CMPE-013/L: “C” Programming
Recursion
Evaluation of Recursive Functions

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Partial results</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 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>Factorial(3)</td>
<td>[2] 3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td>Factorial(4)</td>
<td>[3] 4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>Factorial(5)</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)

Stack (top)

Partial results pushed on stack

Factorial term replaced with result

[0] 1! = 1
= 1

[1] 2! = 2 * 1!
= 2 * 1 = 2

[2] 3! = 3 * 2!
= 3 * 2 = 6

[3] 4! = 4 * 3!
= 4 * 6 = 24

[4] 5! = 5 * 4!

Stack: Factorial(5)

Factorial(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

Factorial(5)

[0] 1! = 1
= 1

[1] 2! = 2 * 1!
= 2 * 1 = 2

[2] 3! = 3 * 2!
= 3 * 2 = 6

[3] 4! = 4 * 3!
= 4 * 6 = 24

[4] 5! = 5 * 4!
= 5 * 24 = 120
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
Recusion

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)

Fibonacci(4)

Partial results pushed on stack

Function call replaced with result

F_4 = F_3 + F_2
Recursion
Evaluation of Recursive Functions

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

Stack (top)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Stack</th>
<th>Stack</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>F6</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

Fibonacci(3)
Fibonacci(4)

F3 = F2 + F1
F4 = F3 + F2
Recursion
Evaluation of Recursive Functions

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

Stack (top)

<table>
<thead>
<tr>
<th>Partial results pushed on stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function call replaced with result</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fibonacci(2)</th>
<th>F_2 = F_1 + F_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(3)</td>
<td>F_3 = F_2 + F_1</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
<td>F_4 = F_3 + F_2</td>
</tr>
</tbody>
</table>
Recursion
Evaluation of Recursive Functions

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

**Stack** (top)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fibonacci(1)</td>
</tr>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(3)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack

Function call replaced with result

- \( F_1 = 1 \)
- \( F_2 = F_1 + F_0 \)
- \( F_3 = F_2 + F_1 \)
- \( F_4 = F_3 + F_2 \)
Recursion

Evaluation of Recursive Functions

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

Stack (top)

<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(1)
Fibonacci(2)
Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

F_1 = 1
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)

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

Stack (top)

Partial results pushed on stack

F_2 = 1 + F_0
F_3 = F_2 + F_1
F_4 = F_3 + F_2

Function call replaced with result
Recursion
Evaluation of Recursive Functions

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

Stack (top)

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)

<table>
<thead>
<tr>
<th>Stack Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(0)</td>
</tr>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(3)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

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)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
<tr>
<td>Fibonacci(3)</td>
</tr>
<tr>
<td>Fibonacci(2)</td>
</tr>
</tbody>
</table>

Stacked results pushed on stack

Function call replaced with result

F<sub>2</sub> = 1 + 1
F<sub>3</sub> = F<sub>2</sub> + F<sub>1</sub>
F<sub>4</sub> = F<sub>3</sub> + F<sub>2</sub>
Recursion
Evaluation of Recursive Functions

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

**Stack** (top)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(3)</td>
<td></td>
</tr>
<tr>
<td>Fibonacci(4)</td>
<td></td>
</tr>
</tbody>
</table>

**Fibonacci**

- $F_3 = 2 + F_1$
- $F_4 = F_3 + F_2$

Partial results pushed on stack
Function call replaced with result
Recursion
Evaluation of Recursive Functions

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

Stack (top)

<table>
<thead>
<tr>
<th>Partial results pushed on stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function call replaced with result</td>
</tr>
</tbody>
</table>

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

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)

Fibonacci(3)
Fibonacci(4)

F_3 = 2 + 1
F_4 = F_3 + F_2

Partial results pushed on stack
Function call replaced with result
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 + F_2
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

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

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

Stack (top)

Fibonacci(1)
Fibonacci(2)
Fibonacci(4)

F_1 = 1
F_2 = F_1 + F_0
F_4 = 3 + F_2

Partial results pushed on stack
Function call replaced with result
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(4)

Partial results pushed on stack
Function call replaced with result

\[
\begin{align*}
F_2 &= 1 + F_0 \\
F_4 &= 3 + F_2
\end{align*}
\]
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_0 = 1
F_2 = 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></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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)

\[ F_4 = 3 + 2 \]
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

Complex data

• 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
Complex data example

- Recursive word count
  - Count 1 word per function call

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

```
This is the end.

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);
}
```

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);
}
```

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);
}
```

This is the end.
Recursion
Complex data example

Example

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

This is the end.

```
This is the end.
1 1 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);
}
```

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);
}
```

This is the end.

```
1 3
```
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.

4
Binary trees
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
Self-similarity
Binary trees

Serialization

• A linear representation of a tree
Binary trees

Serialization

<table>
<thead>
<tr>
<th>Root</th>
<th>Left subtree</th>
<th>Right subtree</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{1}</td>
<td>\text{2^{n-1}}</td>
<td>\text{2^{n-1}}</td>
</tr>
</tbody>
</table>

\[ 1 + 2^{n-1} - 1 = 2^n - 1 \]
Binary trees

Serialization

<table>
<thead>
<tr>
<th>Root</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
</table>

[Diagram of binary tree structure]
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 E F G
Binary trees

Serialization

A  B  C  D

A  B

A

B

C  D

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
Binary trees

Serialization

A | B | C | D | E | F | G

A
B
C
D
E
F
G
Binary trees

.Serialization

A  B  C  D  E  F

A

B

C  D

E

F  G
Binary trees

Serialization

A B C D E F G

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

A

B

C

D

E

F

G
Binary trees

Serialization

A     B     C     D     E     F     G

A

B     E

C     D     F     G
Binary trees
Serialization

2 0 4 7 2 5

0 2 4 7 2 5