CMPE-013/L

Introduction to “C” Programming

Maxwell J Dunne
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)

Factorial(5)
```

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

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

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

Partial results pushed on stack
Factorial term replaced with result
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] \(2! = 2 \times 1!\)
[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)

<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></td>
</tr>
<tr>
<td></td>
<td>[1]  2! = 2 * 1!</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[2]  3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[3]  4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td></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>
<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 * 1!</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[2] 3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[3] 4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td></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>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td>[0]</td>
<td>2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>[1]</td>
<td>3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>5! = 5 * 4!</td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td></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>1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td>2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>Factorial(3)</td>
<td>3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td>Factorial(4)</td>
<td>4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>Factorial(5)</td>
<td>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)

| [0] 1! = 1
| [1] 2! = 2 * 1!
| [2] 3! = 3 * 2!
| [3] 4! = 4 * 3!
| [4] 5! = 5 * 4!

Partial results pushed on stack

Factorial term replaced with result

= 1
= 2 * 1 = 2
= 3 * 2 = 6
= 4 * 6 = 24
Recursion
Evaluation of Recursive Functions

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

Stack (top)

<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] 1</td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td>[1] 2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td></td>
<td>[2] 3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td></td>
<td>[4] 5! = 5 * 4!</td>
<td>= 5 * 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

80%
Recursion
Limitations

- Limited stack space

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial(3)</td>
</tr>
<tr>
<td>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
<tr>
<td>Factorial(6)</td>
</tr>
<tr>
<td>Factorial(7)</td>
</tr>
<tr>
<td>Factorial(8)</td>
</tr>
<tr>
<td>Factorial(9)</td>
</tr>
<tr>
<td>Factorial(10)</td>
</tr>
</tbody>
</table>
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)

\[
F_4 = F_3 + F_2
\]

Stack (top)

Partial results pushed on stack
Function call replaced with result

Fibonacci(4)
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

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

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

Stack (top)

Partial results pushed on stack
Function call replaced with result

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

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

**Stack** (top)

<table>
<thead>
<tr>
<th>F_1</th>
<th>F_2</th>
<th>F_3</th>
<th>F_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F_1</td>
<td>F_2</td>
<td></td>
</tr>
<tr>
<td>F_1</td>
<td>F_1 + F_0</td>
<td>F_2 + F_1</td>
<td>F_3 + F_2</td>
</tr>
</tbody>
</table>

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

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)

Partial results pushed on stack

Function call replaced with result

F₀ = 1
F₂ = 1 + F₀
F₃ = F₂ + F₁
F₄ = F₃ + F₂

Maxwell J Dunne
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>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

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

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

Stack (top)

<table>
<thead>
<tr>
<th>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_2 = 1 + 1 )</td>
<td></td>
</tr>
<tr>
<td>( F_3 = F_2 + F_1 )</td>
<td></td>
</tr>
<tr>
<td>( F_4 = F_3 + F_2 )</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
Function call replaced with result

\[
\begin{align*}
F_3 &= 2 + F_1 \\
F_4 &= F_3 + F_2
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

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

```
Stack (top)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Partial results pushed on stack
Function call replaced with result

\[
\begin{align*}
F_1 &= 1 \\
F_3 &= 2 + F_1 \\
F_4 &= F_3 + F_2
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

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

Stack (top)
- Fibonacci(3)
- Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_3 = \begin{array}{c} 2 \end{array} + \begin{array}{c} 1 \end{array} \]
\[ F_4 = \begin{array}{c} F_3 \end{array} + \begin{array}{c} F_2 \end{array} \]
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)

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)

Partial results pushed on stack
Function call replaced with result

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

Maxwell J Dunne
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_2 = \[1 + F_0\]
F_4 = \[3 + F_2\]

Maxwell J Dunne
Recursion
Evaluation of Recursive Functions

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

Stack (top)

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

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)

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

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

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);
}
```

str

This is the end.
Recursion
Complex data example

Example

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

str    p
      ↓   ↓
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.
```

```
1
```
Recursion
Complex data example

Example

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

str

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 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.
  ^   ^   ^
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);
}
```

str
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.
```

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.

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

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

This is the end.

1 1 2
Recursion
Complex data example

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

<table>
<thead>
<tr>
<th>This</th>
<th>is</th>
<th>the</th>
<th>end.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

Traversals

- 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

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

Diagram of a binary tree with arrows pointing down to its children.
Binary trees

Serialization

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

[Diagram of a binary tree with arrows indicating the serialization process.]
Binary trees
Serialization

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

A

B

C

D

E

F

G

A


A
Binary trees

Serialization

A   B   C   D

A   B

A   B

A   B
Binary trees

Serialization

A
B
C

A
B
C
Binary trees

Serialization

A  B  C  D

A
  / \  
 /   \ 
B     D
  / \ 
C   D

CMPE-013/L: “C” Programming
Binary trees
Serialization

A  B  C  D

A → B → C → D
Binary trees
Serialization

A B C D E F G
Binary trees
Serialization

A  B  C  D  E  F

Diagram of a binary tree with nodes labeled A, B, C, D, E, F, and G.
Binary trees
Serialization

A  B  C  D  E  F  G
Binary trees
Serialization

A   B   C   D   E   F   G
Binary trees
Serialization

A       B       C       D       E       F       G
Binary trees
Serialization
6 pm - 8 pm tonight

CMPE-013/L

Toaster Oven Lab

Maxwell James Dunne
.9 file  set of .0 files  .c -> .0 -> .hex

Buttons, AD

LE Drs. h
Toaster Oven

START

- Button event 4 DOWN
  - Reset 2Hz Timer
  - Save initial start time
  - Turn oven on
    - Clear button event
    - Update the display

- Countdown
  - 2Hz timer triggered
    - Cooking time left > 0
      - Update the display/LEDs
      - Clear 2Hz timer flag

- Pending selector change
  - 2Hz timer triggered
    - Cooking time left <= 0
      - Update the display

- Pending reset
  - 2Hz timer triggered
    - Cooking time left <= 0
      - Clear timer flag
Integer Timing

Free running counters and precision

0.5 second intervals: \( \frac{2}{Hz} \)

10s \( \rightarrow \) 20 ticks

\[ \rightarrow 32768 \]

1 second

\[ \uparrow \]

RTC
$$\begin{align*}
\text{event} & \quad 0 \\
\text{event} & \quad 1 \\
\text{event} & \quad 2 \\
\text{event} & \quad 3 \\
\text{event} & \quad 4 \\
\text{event} & \quad 5 \\
\text{event} & \quad 6 \\
\text{event} & \quad 7
\end{align*}$$
Unsigned int

250
255

15 - 250 = 20

30 - 10 = 20

16-bit

65 seconds

32-bit

~49 days
OLED Display
Formatting and Update Cycles

Struct 0

```
mode: %s
Time: %s
Temp: %s
```

$n = 10$

$xA = 10$

Oven
EVENTS

EVENT ≠ SWITCH NOW

EVENT: DETECTABLE CHANGE
FL Down

FL REVERSED
Testing your event checker?

```c
while (1) {
    ThisEvent = CheckEvents();
    if (ThisEvent != NO_EVENT) {
        printf("Got event \#%d", ThisEvent);
    }
}
```
Hysteresis (DARK)

INFO (LIGHT)
static event_t prevEvent = INTO_LIGHT;

if (((prevEvent == INTO_DARK) &
    (Roach_LightLevel() < LOW_THRESH))) {
    prevEvent = INTO_LIGHT;
    return INTO_LIGHT;
}

if (((prevEvent == INTO_LIGHT) && (RLL() > HIGH_THRESH))) {
    prevEvent = INTO_DARK;
    return INTO_DARK;
}