CMPE-013/L

Introduction to “C” Programming

Maxwell J Dunne
Recursion
Recursion

$4! = 4 \cdot 3 \cdot 2 \cdot 1$

- 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!

Maxwell J Dunne
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>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
</tbody>
</table>

Factorial term replaced with result

\[ 4! = 4 \times 3! \]

\[ 5! = 5 \times 4! \]
Recursion
Evaluation of Recursive Functions

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

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

- Partial results pushed on stack
-Factorial term replaced with result

[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)

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)

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></td>
<td>[1] 2! = 2 * 1!</td>
<td></td>
</tr>
<tr>
<td>Factorial(1)</td>
<td>[2] 3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td>Factorial(2)</td>
<td>[3] 4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
<td>[4] 5! = 5 * 4!</td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td></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)

<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>3 * 2 = 6</td>
</tr>
<tr>
<td>[2]</td>
<td>4! = 4 * 3!</td>
<td>4 * 3 = 12</td>
</tr>
<tr>
<td>[3]</td>
<td>5! = 5 * 4!</td>
<td>5 * 4 = 20</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)

Stack (top)

Partial results pushed on stack
[0] 1! = 1
[1] 2! = 2 * 1!
[2] 3! = 3 * 2!
[3] 4! = 4 * 3!
[4] 5! = 5 * 4!

Factorial term replaced with result
= 1
= 2 * 1 = 2
= 3 * 2 = 6

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

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

**Stack (top)**

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

**Partial results pushed on stack**

<table>
<thead>
<tr>
<th>Index</th>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td>[1]</td>
<td>2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>[2]</td>
<td>3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td>[4]</td>
<td>5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>

**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)

Factorial(5)

[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
= 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
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

<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 $F_4$
  (based on code from previous slide)

Stack (top)

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)

Partial results pushed on stack

Function call replaced with result

Stack (top)

Fibonacci(3)

Fibonacci(4)

\[ F_3 = F_2 + F_1 \]

\[ F_4 = F_3 + F_2 \]
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>
<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

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

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

Stack (top)

Partial results pushed on stack
Function call replaced with result

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

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)

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

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

Partial results pushed on stack
Function call replaced with result

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

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(2)</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>

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

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

Stack (top)

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

Partial results pushed on stack

F_0 = 1
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)

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

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

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

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

Partial results pushed on stack

Function call replaced with result

\[
F_2 = 1 + 1
\]

\[
F_3 = F_2 + F_1
\]

\[
F_4 = F_3 + F_2
\]
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>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

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

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)

<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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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)

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

Partial results pushed on stack
Function call replaced with result

\[ \begin{align*}
F_3 &= 2 + 1 \\
F_4 &= F_3 + F_2
\end{align*} \]
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></td>
</tr>
</tbody>
</table>

Fibonacci(4)

**F₄ = 3 + F₂**
Recursion
Evaluation of Recursive Functions

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

Stack (top)

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

F₂ = F₁ + F₀
F₄ = 3 + F₂

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>Fibonacci(1)</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

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

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

Stack (top)

Fibonacci(2)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

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

$F_0 = 1$
$F_2 = 1 + F_0$
$F_4 = 3 + F_2$

Fibonacci(0)
Fibonacci(2)
Fibonacci(4)
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>Fibonacci(2)</td>
<td></td>
</tr>
<tr>
<td>Fibonacci(4)</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Stack (top)

Partial results pushed on stack
Function call replaced with result

Fibonacci(4)

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, ' '); // Line 1
    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

```c
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

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

str  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);
}
```

![Diagram showing recursion example](image)

```
This is the end.
```
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

Example

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

<table>
<thead>
<tr>
<th>str</th>
<th>p+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>1</td>
</tr>
<tr>
<td>is</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td></td>
</tr>
<tr>
<td>end.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</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);
}
```

str

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

1 1
Recursion
Complex data example

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

Example:
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

str 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 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  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
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

morse decoder
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

Traversing

- 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>
</table>

[Diagram of a binary tree with arrows showing serialization process]
Binary trees

Serialization
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
Binary trees

Serialization

A  B  C  D
Binary trees

Serialization

\[ 2^{n-1} \]

\[ 2^{n-1} \]

\[ 2 \]

\[ 2 \]
Binary trees
Serialization

A  B  C  D
Binary trees

Serialization

A  B  C  D
Binary trees
Serialization

A B C D E 4 F G
Binary trees
Serialization

A B C D E F
Binary trees
Serialization

A  B  C  D  E  F  G

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

Serialization

A  B  C  D  E  F  G
Binary trees
Serialization

A  B  C  D  E  F  G
Binary trees

Serialization

Diagram of a binary tree with nodes labeled 0, 2, 4, 5, 7, 8.
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Software Engineering

Design

Build
Software Engineering

Design process
Software Engineering

Principles

• Use consistent styling

• Summary:
  – Utilize whitespace
  – Good variable/function names
  – Comments that describe non-obvious code behavior
    • "How?" and "why?" are good questions to answer in comments
Software Engineering

Formatting code

- Ugly code
- Beautiful code
Software Engineering
Formatting non-code

• Comments that describe non-obvious code behavior
  - "How?" and "why?" are good questions to answer in comments

With OS example

```c
// First, determine the length of both items' data,
// given NULL data a -1 length so that it sorts to
// the head of the list.
int len1 = -1;
if (item1->data) {
    len1 = strlen(item1->data);
}
...
```
Software Engineering

Principles

• Modularity is important

• Why?
  – Supports code reuse
  – Simplifies changes
  – Allows for testing

• How?
  – Keep functions small
  – Minimize side effects
  – Information hiding/encapsulation

No Magic Numbers
\texttt{Speed\_slow}

\texttt{set\_speed(\text{\textasciitilde}n\text{\textasciitilde}seed)}

\texttt{set\_speed(\text{\textasciitilde}void)}

\texttt{set\_value(\text{\textasciitilde})}

\texttt{set\_speed}

\texttt{set\_value("seed", \text{\textasciitilde}new\text{\textasciitilde}value)}
Software Engineering

Principles

- Information hiding/encapsulation
- Summary:
  - Hide unimportant details from the user
  - Protects the user from breaking things
  - Separates backend from frontend
Software Engineering

Mantras

• Keep it simple, stupid
  – KISS
• Summary:
  – Don't solve problems you don't need to
  – Don't introduce unnecessary complexity
  – Prioritize for readability and modularity
  – Don't be clever and/or cute
  – Applies to code architecture and specific code constructs
Software Engineering

KISS example

Example

```c
ListItem *LinkedListGetFirst(ListItem *list)
{
    ListItem *tempPointer = NULL;
    if (list == NULL) {
        return NULL;
    }
    if (list->previousItem == NULL && list->nextItem != NULL) {
        return list;
    } else if (list->previousItem != NULL) {
        tempPointer = list;
        while (tempPointer->previousItem != NULL) {
            tempPointer = tempPointer->previousItem;
        }
    }
    return tempPointer;
}
```
Example

ListItem *LinkedListGetFirst(ListItem *list) {
    while (list && list->previousItem) {
        list = list->previousItem;
    }
    return list;
}
Software Engineering

Mantras

• Don't repeat yourself
  – DRY

• Summary:
  – Write code only once
  – Simplifies refactoring/incremental development
  – Avoids copy/paste errors
Software Engineering

Mantras

• You aren't gonna need it
  – YAGNI

• Summary:
  – Don't introduce features that are unnecessary
  – Don't write more code than you have to
  – Start small and build from there

  feature creep
Software Engineering

Principles

• Principle of Least Astonishment

• Summary:
  – Be consistent with user's expectations
  – Build on user's intuition
  – Applies to users and developers
    • so both the code and library/program functionality
  – Lowers learning curve
Software Engineering

Principle of Least Astonishment

- Functions/variables should have clear names
  - That should match their functionality!
  - Same for comments
- Functions should not do more than you would think
  - Minimize side effects
- Code should be grouped logically
- Functionality should follow precedence if any exists
Software Engineering

Principles

• Garbage in, garbage out

• Summary:
  – "A system's output quality usually cannot be better than the input quality"
  – So bad input results in garbage output
    • Instead of an error condition
  – Can propagate through the system
  – Can be mitigated by checking the input data

xKCD Johnny drop tables
Software Engineering

Principles

• Fault tolerant design

• Summary:
  – Plan for operating failures
    • Running out of memory ✓
    • Data being corrupted
  – Provide fallback modes
  – Important for complex software where minor errors can be common
  – Part of defensive programming

is NULL
Software Engineering

Principles

- Error tolerant design
- Summary:
  - Plan for user errors
    - "Fault tolerant design" applied to the human component
  - Primarily invalid user input
  - Important for complex software where minor errors can be common
  - Part of defensive programming
Software Engineering
Writing fault/error tolerant code

• Check return values for errors!
  – Many functions have special return values when there are errors, these should usually be checked
    – File accesses
    – scanf()
    – malloc() ✓ ✓

• Your code should have special error values
  – LinkedList library Return Last Error()

• Program should also return error if failure
Software Engineering

Writing fault/error tolerant code

- Errors should be exposed by libraries

**Good library**

```c
int  LinkedListSwapData(ListItem *firstItem,
                        ListItem *secondItem);

int  LinkedListSort(ListItem *list);

int  LinkedListPrint(ListItem *list);
```

**Bad library**

```c
void LinkedListSwapData(ListItem *firstItem,
                        ListItem *secondItem);

void LinkedListSort(ListItem *list);

void LinkedListPrint(ListItem *list);
```
Software Engineering
Writing fault/error tolerant code

- Errors should be exposed by libraries
- And handled by the program
- Not all errors can be recovered from
  - Fatal errors

Embedded example:

```c
int main(void) {
    if (!DataStoreInit()) {
        FATAL_ERROR();
    }
}
```

With OS example:

```c
int main(void) {
    if (!DataStoreInit()) {
        return DATASTORE_ERROR;
    }
}
```
Software Engineering

Principles

• Eating your own dogfood

• Summary:
  – When engineers use their own creations, they're generally better
  – More likely that bugs are fixed, features are added because they directly impact the developers
  – In use by all of industry
  – I do it
  – Better bug reports
Software Engineering
Pitfalls

• Premature Optimization
  – "root of all evil"

• Summary:
  – Optimizing code before performance is a critical factor
  – Optimizing reduces readability & modularity
  – Optimization not required for a lot of code
    • See Amdahl's Law
  – See KISS
Software Engineering

Teamwork

- Working as a group is **the** most challenging engineering practice
- Requires:
  - Good communication
- That's it!
Software Engineering
Teamwork

- Pair programming
- Summary:
  - Two developers work side by side: one driving, the other navigating
  - Just like driving:
    - Driver writes code
    - Navigator plans ahead, thinks of edge cases, double-checks driver
  - Requires frequent role switching to be effective!
Software Engineering

Teamwork

• Division of labor

• Summary:
  – Divide work into tasks that can be split between team members
  – Requires coordination to not step on each other's toes
  – Documentation is very important!
  – Can be useful to split testing and development between different people
5 P's

Prior planning prevents poor performance.

3AM  2AM

8 hours  24-36 hours

3 hours planning
CMPE-013/L

Toaster Oven Lab

Maxwell James Dunne
Toaster Oven

- Time
- Temp

**Reset**
- Set input selection for time
- Reset temp to default
- Turn oven off
- Update the display

**Start**
- Button counter >= LONG_PRESS
  - Switch input selection
  - Update time/temp from pot
  - Update the display
  - Clear button event

**Countdown**
- Button event 4_DOWN
  - 2Hz timer triggered &&
  - Cooking time left == 0
  - Update the display
  - Clear 2Hz timer flag
- Button event 3_DOWN
  - Store Free Running Time
  - Clear button event

**Pending Selector Change**
- Button event 4_UP
  - 2Hz timer triggered &&
  - Cooking time left > 0
  - Cooking time left -= 1
  - Update the display/LEDs
  - Clear 2Hz timer flag
- Button event 3_UP
  - 2Hz timer triggered &&
  - Cooking time left > 0
  - Cooking time left -= 1
  - Update the display
  - Clear timer flag

**Pending Reset**
- Button event 4_UP
  - Clear button event
Integer Timing
Free running counters and precision

\[ \text{time} : \ 20 \text{ seconds} \rightarrow 40 \text{ ticks} \]

\[ \text{event 1} \]
\[ \text{Start time} = \text{FRT} \]

\[ \text{event 2} \]
\[ \text{End time} = \text{FRT} \]
\[ \text{duration} = \text{End time} - \text{Start time} \]

Input capture

FP

\[ \rightarrow 2\text{ Hz} \]
OLED Display

Formatting and Update Cycles

store

print update

store

print

mode:

mode:

mode:

0% 5

(onevents,

(update
bake Broil

10%<%<%<%<%<))...
//

state
mode
time
oven on/off
Update display

Start mode

Time

Temp

3 sprint s

Toast

Bake

Broil
30 seconds 60 ticks

```
\( \text{int time} = \text{tick} > 71 ? 60, \text{tick} > 71 \) \%
```

```
0% < "_" < "_" < "_"
```

struct variables
"foo"
"b"
"foo"
"In bar"
"foo"
"foo"
"foo bar"