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)

\[ 5! = 5 \times 4! \]
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)

Partial results pushed on stack
Factorial term replaced with result

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

[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)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Factorial(1)</td>
<td></td>
</tr>
<tr>
<td>Factorial(2)</td>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td></td>
</tr>
<tr>
<td>Factorial(5)</td>
<td></td>
</tr>
</tbody>
</table>

Partial results pushed on stack

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] 1! = 1</td>
<td></td>
</tr>
<tr>
<td>[1] 2! = 2 * 1!</td>
<td></td>
</tr>
<tr>
<td>[2] 3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td>[3] 4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>[4] 5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>

Factorial term replaced with result
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>Factorial(1)</td>
<td>2! = 2 * 1!</td>
<td></td>
</tr>
<tr>
<td>Factorial(2)</td>
<td>3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
<td>4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td>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)
- Factorial(2)
- Factorial(3)
- Factorial(4)
- Factorial(5)

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 = 1
- 2 * 1 = 2
- 3 * 2 = 6
- 4 * 3 = 12
- 5 * 4 = 20
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
Recursion
Evaluation of Recursive Functions

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

Stack (top)

[0] \[1! = 1\]
[1] \[2! = 2 \times 1!\]
[2] \[3! = 3 \times 2!\]
[3] \[4! = 4 \times 3!\]
[4] \[5! = 5 \times 4!\]

Partial results pushed on stack
Factorial term replaced with result

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

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

**Stack** (top)

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></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>= 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
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

\[ f_n = f_{n-1} + f_{n-2} \]

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

\[ F_4 = F_3 + F_2 \]
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(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

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

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

Stack (top)

<table>
<thead>
<tr>
<th>Stack</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

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

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

Stack (top)

<table>
<thead>
<tr>
<th>Partial results pushed on stack</th>
</tr>
</thead>
<tbody>
<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>

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)

<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>

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)

Stack (top)

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

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

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

Stack (top)

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

\[
\begin{align*}
F_0 &= 1 \\
F_2 &= 1 + F_0 \\
F_3 &= F_2 + F_1 \\
F_4 &= F_3 + F_2 \\
F_5 &= F_4 + F_3 \\
F_6 &= F_5 + F_4 \\
\end{align*}
\]
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(0)  Fibonacci(2)  Fibonacci(3)  Fibonacci(4)

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>Stack</th>
<th>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 + 1</td>
<td>F2</td>
</tr>
<tr>
<td></td>
<td>F2 + F1</td>
<td>F3</td>
</tr>
<tr>
<td></td>
<td>F3 + F2</td>
<td>F4</td>
</tr>
</tbody>
</table>

F2 = 1 + 1
F3 = F2 + F1
F4 = F3 + F2
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(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)

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

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

Stack (top)

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

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

| 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)

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

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

\[
\begin{align*}
F_1 &= 1 \\
F_2 &= F_1 + F_0 \\
F_4 &= 3 + F_2
\end{align*}
\]
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 |
|--------------------------------|
| \( F_2 = 1 + F_0 \)           |
| \( F_4 = 3 + F_2 \)           |
Recursion
Evaluation of Recursive Functions

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

Partial results pushed on stack
Function call replaced with result

Stack (top)

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

\[ F_0 = 1 \]
\[ F_2 = 1 + F_0 \]
\[ F_4 = 3 + F_2 \]
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>
<tr>
<td></td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

F_2 = 1 + 1
F_4 = 3 + F_2

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>Function call replaced with result</td>
</tr>
</tbody>
</table>

F_4 = 3 + 2

Maxwell J Dunne
Recursion

Evaluation of Recursive Functions

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

\[ F_4 = 5 \]
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
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);
}
```

```plaintext
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

```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

```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
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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></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.

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

\[ \text{str} \quad \text{p+1} \]
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
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

- 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 indicating the hierarchy and the table above showing the concepts of root, left subtree, and right subtree.
Binary trees
Serialization

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

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

A  B  C  

A

B

C

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

A  B  C  D
Binary trees

Serialization

A   B   C   D
Binary trees
Serialization

A B C D E F G
Binary trees

Serialization

A B C D E F
Binary trees
Serialization
Binary trees
Serialization

A B C D E F G
Binary trees
Serialization

A  B  C  D  E  F  G
Binary trees
Serialization

Tree:
- 8
  - 2
    - 0
    - 4
  - 7
    - 2
    - 5
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Software Engineering
Design
Build
Software Engineering

Design process

- Initial Planning
- Requirements
- Analysis/Design
- Implementation
- Testing
- Evaluation
- Deployment
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

setter

set foo(l)

return 14?

code complete

5e + (foo();)

foo = ;
Software Engineering

Principles

- Information hiding/encapsulation

Summary:
- Hide unimportant details from the user
- Protects the user from breaking things
- Separates backend from frontend

int

INT
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

for ever
```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;
}
```
Software Engineering

KISS example

Example

```c
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
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

getchar

UART_getchar
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
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
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

• 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

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

```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
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
CMPE-013/L
Toaster Oven Lab
Maxwell James Dunne
Integer Timing
Free running counters and precision

time += t;
<= time;

if (time <= n)
24.5
25

25 \quad \rightarrow \quad 50
48
49
OLED Display
Formatting and Update Cycles

Update OLED1()
Switch

; nonupdate = TRUE

if (nonupdate) Update OLED()
C = 143 226;

static constexpr = 14 3226

#define FOO 143226
const Chor +

2:30

150s 150 ° 60 ⇒ 30
150/60 ⇒ 2
Countdown
if(2h2)

print
if(2 Hz)

ISR
2 Hz = TRIG
FSM
CMPE-167/L
Calibration

Maxwell James Dunne
Magnetometer Calibration

- Use Magnetometer to measure $B_x$ and $B_y$
- Above equation only works when level
  - In some areas $B$ has a significant downward component.

\[ \psi = \tan^{-1}\left(\frac{B_y}{B_x}\right) \]
Measurement Errors

- Hard Iron Errors
  - "Constant" biases
- Soft Iron Errors
  - Attitude dependent errors
- Scale Factor Errors
- Misalignment & Cross Axis Sensitivity Errors
  - Magnetometer triad is not aligned with the body axes of the airplane
  - The magnetometer triad may not be orthogonal
Graphical Description of Error Sources

- Soft Iron
- Hard Iron
- Magnetometer
Graphical Description of Error Sources

- Magnetometer
- Soft Iron
- Hard Iron
Old Calibration Method

Heading error is modeled by a truncated Fourier series. The Fourier coefficients are function of the sensor errors (c.f. LITEF AHRS Manual).

$$\delta\psi = A + B \sin(\psi) + C \cos(\psi) + D \sin(2\psi) + E \cos(2\psi)$$

“Swing” the airplane to estimate the Fourier Coefficients
New Calibration Method

Fixed to the Earth’s Locally Level Tangent Plane (Red)

Frame of Reference Fixed to the Aircraft’s Body (Yellow)

Locus of $H_{13}$ in the Aircraft Body Fixed Frame (Green)

NOTE: Calibration occurs in the magnetic field domain and not the heading domain!
Locus of Points (2-D)

- Misalignment & Soft Iron Effects (Blue)
- Hard Iron Effects (Green)

- $H_{13} + \delta H_{13}$
- $\delta H_{13}$
- Scale Factor (Red)
Locus of Points (3-D)
Estimation Principle

\[ R^2 = \frac{(x-x_0)^2}{a^2} + \frac{(y-y_0)^2}{b^2} + \frac{(z-z_0)^2}{c^2} + \frac{(x-x_0)(y-y_0)}{d^2} + \frac{(x-x_0)(z-z_0)}{e^2} + \frac{(y-y_0)(z-z_0)}{f^2} \]
Heading Estimate After Magnetometer Calibration

Heading Comparison

$\delta \psi$ (degree)

Time (sec)

Magnetometer

Truth

$\alpha = 3.5^\circ$
(Wide Band Noise)
Magnetometer Calibration

- Outputs of a perfect magnetometer triad will lie on sphere.
- Radius of the perfect sphere will be equal to the magnitude of the magnetic field vector.
- Magnetometer errors will alter this perfect sphere into an ellipsoid that is displaced from the center.
- The calibration process involves “reshaping” the magnetometer readings to fit on a sphere of with the prescribed radius.
Calibrated Magnetometer

- Magnitude of the earth’s magnetic field vector in the San Francisco Bay area is 0.5 Gauss
- Most of the pre-calibration errors are due to “hard iron” (bias)
more equations than unknowns

\[ y = mx + b \]

\[
\begin{bmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n \\
\end{bmatrix}
  =
  \begin{bmatrix}
    x_1 & 1 \\
    x_2 & 1 \\
    \vdots & \vdots \\
    x_n & 1 \\
  \end{bmatrix}
  \begin{bmatrix}
    m \\
    b \\
  \end{bmatrix}
\]

\[ y = \mathbf{A} \mathbf{p} \]
\[ y = Ax \]
\[ y \approx Ax \]
\[ r = Ax - y \]

Find \( x = x_{1s} \)

**LEAST SQUARES**
\( y = kx \)
\[ \| r \|_2^2 = (y - Ax)^\top(y - Ax) = x^\top A^\top A x - 2y^\top A x + y^\top y \]

\[ \frac{\partial \| r \|_2^2}{\partial x} = 0 = 2x^\top A^\top A - 2y^\top A \Rightarrow x^\top A^\top A - y^\top A = 0 \]

\[ \left[ x^\top A^\top A = y^\top A \right]^\top \]

\[ A^\top A x = A^\top y \]

\[ x = (A^\top A)^{-1} A^\top y \]

\[ x = A \backslash y \]
\hat{\mathbf{x}} = \left[ A^\mathbf{T} \mathbf{A} \right]^{-1} A^\mathbf{T} \mathbf{y} \leftarrow \text{LEAST SQUARES E\textsc{stimates of } \mathbf{x}}.

\min \sum \text{error}^2
\[ y = mx + b \]

\[ y = [x \ 1] \begin{bmatrix} m \\ b \end{bmatrix} \]

\[ \hat{p} = (X^TX)^{-1}X^TY \]

\[ x, y \text{ - pairs} \]

\[ \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1, 1 \\ \vdots \\ x_n, 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} \]
\[ y = \hat{a}_0 + \hat{a}_1 x + \hat{a}_2 x^2 + \hat{a}_3 x^3 \]

\[ y_i = \begin{bmatrix} 1 & x_i & x_i^2 & x_i^3 \end{bmatrix} \begin{bmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \hat{a}_2 \\ \hat{a}_3 \end{bmatrix} \]

\[ y = [A] \hat{\beta} \]

\[ \hat{\beta} = (A^T A)^{-1} A^T y \]
\[ \frac{(x-x_0)^2}{a^2} + \frac{(y-y_0)^2}{b^2} = R^2 \]

\[ \frac{x^2 - 2xx_0 + x_0^2}{a^2} + \frac{y^2 - 2yy_0 + y_0^2}{b^2} = R^2 \]

\[ x^2 - 2x x_0 + x_0^2 + \frac{a^2}{b^2} y^2 - \frac{2a^2}{b^2} y y_0 + \frac{a^2}{b^2} y_0^2 = \frac{a^2}{b^2} x_0^2 \]

\[ x^2 = 2x x_0 + 2y \frac{a^2}{b^2} y_0 - y^2 \frac{a^2}{b^2} + a^2 R^2 - x_0^2 - \frac{a^2}{b^2} y_0^2 \]

\[ x_i^2 = \begin{bmatrix} 2x_i & 2y_i & -y_i^2 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ \frac{a^2}{b^2} y_0 \\ a^2/b^2 \\ a^2/b^2 \end{bmatrix} + a^2 R^2 - x_0^2 - \frac{a^2}{b^2} y_0^2 \]
\[
\begin{align*}
\begin{bmatrix}
  x_1^2 \\
  \vdots \\
  x_n^2
\end{bmatrix} &=
\begin{bmatrix}
  2x_1 & 2y_1 & -y_1^2 & 1 \\
  \vdots & \vdots & \vdots & \vdots \\
  2x_n & 2y_n & -y_n^2 & 1
\end{bmatrix} \mathbf{P} \\
\mathbf{P} &= (X^T A)^{-1} X^T X \\
x_0 &= \hat{P}(1) \\
y_0 &= \frac{\hat{P}(2)}{\hat{P}(3)}
\end{align*}
\]
\[ y = Ax + B \]

3 x 1

\[ \left( \frac{x-x_0}{a} \right)^2 + \left( \frac{y-y_0}{b} \right)^2 = \frac{(x-x_0)(y-y_0)}{c^2} = k^2 \]