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));
}
```

5! = 5 * 4 * 3 * 2 * 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

Factorial(5)

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

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

Stack (top)

Partial results pushed on stack
Factorial term replaced with result

[3] 4! = 4 * 3!

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

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

Stack (top)

Partial results pushed on stack
Factorial term replaced with result

[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)

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

Partial results pushed on stack

Factorial term replaced with result

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

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

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

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

Partial results pushed on stack

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

Factorial term replaced with result

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 * 1 = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 * 2 = 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 * 6 = 24</td>
<td></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(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
= 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

Stack (top)

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

CMPE-013/L: “C” Programming
Recursion

Multiple recursion

- Recursion is not limited to a single function call

Example

```c
int Fibonacci(int n)
{
    if (n <= 1) {
        return 1;
    }
    return (Fibonacci(n - 1) + Fibonacci(n - 2));
}
```
Recursion
Evaluation of Recursive Functions

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

Stack (top)

Partial results pushed on stack
Function call replaced with result

F_4 = F_3 + F_2

Fibonacci(4)
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

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

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

Stack \((\text{top})\)

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

Partial results pushed on stack
Function call replaced with result

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

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

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

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

| F_1 = 1 |
| F_2 = F_1 + F_0 |
| F_3 = F_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)

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)

<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><strong>Fibonacci(0)</strong></td>
</tr>
<tr>
<td><strong>Fibonacci(2)</strong></td>
</tr>
<tr>
<td><strong>Fibonacci(3)</strong></td>
</tr>
<tr>
<td><strong>Fibonacci(4)</strong></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)

<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>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_0 = 1$</td>
<td></td>
</tr>
<tr>
<td>$F_2 = 1 + F_0$</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>

$F_0 = 1$
$F_2 = 1 + F_0$
$F_3 = F_2 + F_1$
$F_4 = F_3 + F_2$
Recursion
Evaluation of Recursive Functions

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

**Stack**

<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

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

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

Stack (top)

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

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

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

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

Fibonacci(3)
Fibonacci(4)

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)

<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(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)

Partial results pushed on stack
Function call replaced with result

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

Fibonacci(1)
Fibonacci(2)
Fibonacci(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></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

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)

Fibonacci(2)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

Partial results pushed on stack
Function call replaced with result

F_4 = 3 + 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

$F_4 = 5$
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;
    }
    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 |

CMPE-013/L: “C” Programming
Maxwell J Dunne
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
```
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);
}
```

Example:

```
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

---

Maxwell J Dunne
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);
}
```

Input: This is the end.

Output: 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);
}
```

Example:

```
This is the end.
```

Counts:

```
1 1 1 1
```
Recursion

Complex data example

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

This is the end.

1  1  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

Traversal

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

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

Serialization

• A linear representation of a tree
Binary trees
Serialization

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

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

Serialization
Binary trees

Serialization

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

Serialization

A  B  C  D  E  F  G
Binary trees

Serialization
Binary trees
Serialization

A B C
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

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

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

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

Fail Early, Fail Often

Iterative

Initial Planning
Requirements
Analysis/Design
Implementation
Testing
Evaluation
Deployment

Lab manual

git + push
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

// counter++
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

```
// 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);
}
...```

Maxwell James Dunne
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

CMPE-013/L: “C” Programming

Maxwell James Dunne
Software Engineering

Principles

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

ADL.4

Toaster oven

ADL.4
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;
}
```
Software Engineering

KISS example

Example

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

```c
for () ;
```
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

Proebox V2

foo(x)
bar(x)
Query

set
get
pull
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
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 LinkedListListPrint(ListItem *list);
```

**Bad library**

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

60k x
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
  – QA
6 pm - 8 pm tonight

CMPE-013/L

Toaster Oven Lab

Maxwell James Dunne
.9 file

set of .0 files

Buttons, AD

LEDs.h
Integers Timing
Free running counters and precision

\[ \frac{0.5 \text{ second intervals}}{10 \text{s}} \rightarrow 20 \text{ ticks} \]

\[ \rightarrow 32768 \]

\[ \frac{1 \text{ second}}{2Hz} \]

\[ \rightarrow 32768 \text{ RTCC} \]
\[
\begin{bmatrix}
\text{event } 0 \\
\text{event } 1 \\
\text{event } 2 \\
\text{event } 3 \\
\text{event } 4 \\
\text{event } 5
\end{bmatrix}
\]

\[
\begin{align*}
\text{event } 42 & : 0 \\
\text{event } 43 & : 1 \\
\text{event } 44 & : 2 \\
\text{event } 45 & : 3 \\
\text{event } 46 & : 4 \\
46 - 42 & = 4
\end{align*}
\]
Unsigned int

15 - 250 = 20

30 - 10 = 20

16-bit

64 = \approx 64 \text{ seconds}

32-bit

\approx 49 \text{ days}
OLED Display

Formatting and Update Cycles

```
struct {
  int mode;  // %s
  int time;  // %d
  int temp;  // %d
}

n = 10
xA = 10
oven
```

```
printf("%s", OLED);
```
Timers
1 Button Events
2 2 Hz 2 Hz Event = TRUE
3 5 Hz FRT++;

5 OLED updates
Output

$\Rightarrow 254 \times 2$

$(768 \gg 2) \ll 1 + 1$

$$\frac{27}{2} = \frac{13}{2} = 6.2 = 12$$

768 \gg 2 + 300

59271

60/2 = 30
$\frac{1}{6} \times 255 \text{ seconds} \times 2$

$183 \text{ seconds}$

$183 / 60 = 3$

$183 \times \frac{1}{60} = 3$
updatedOLED(struct Oven Info)

+ toast
+ bake
+ broil
# define OVENTOPON
"\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\n" 0 1

→

\%.*s,
Bake

Time: 0:33

Temp: 400