Random atmospheric noise
if (BEG == BE_1_UP)

UP

| down

0000 or 1111

Button_states(); 4-bits

static int Hist = 0;

Hist = Hist << 4;

Hist 1 = Button_states();
\[ 1 \quad 2 \quad 3 \quad 4 \]
\[ \text{down} \]
\[ 5 \quad 6 \quad 7 \quad 8 \quad 9 \]

\[ H_{is+} = \frac{1}{1} \]
\[ H_{is+} = 12 \]
\[ H_{is+} = 123 \]
\[ = 1234 \]

\[ = 2345 \]

\[ \text{if}(\text{Hist} \lor \text{BTN1M}) \]
\[ \text{down} \]
\[ \text{if}(\lnot \text{Hist} \lor \text{BTN1M}) \]
\[ \text{up} \]
if (Down pattern exists)
  if (Last event \neq \textit{down event})
    Down event has occurred
Part 2  ADC

OLED Printing

986  99%

1 interrupt
int x = &ADC1.BufX
Sum / 8;
Sum > 23;

1023 → 1986

OLED_Clear();

0-7

2.2.2.2 = 8
XXX XXXX
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

Factorial(4)

4! = 4 * 3!

Factorial(5)

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

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

Stack (top)

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

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)

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

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0] 1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td>Factorial(1)</td>
<td>[1] 2! = 2 * 1!</td>
<td></td>
</tr>
<tr>
<td>Factorial(2)</td>
<td>[2] 3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
<td>[3] 4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td>[4] 5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>
Recursion
Evaluation of Recursive Functions

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

Stack (top)

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

| [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
= 2 * 1 = 2

Maxwell J Dunne – Spring 2016
Recursion
Evaluation of Recursive Functions

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

Stack (top)

Partial results pushed on stack
Factorial term replaced with result

[0] 1! = 1
= 1

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

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

[3] 4! = 4 * 3!

[4] 5! = 5 * 4!
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>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
</tbody>
</table>

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!

= 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) |

<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

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)

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)

<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(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)
```

```
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)

<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

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

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

Partial results pushed on stack
Function call replaced with result

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

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

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

Stack (top)

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

<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></td>
<td></td>
<td></td>
<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)

Stack (top)

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

\[
F_3 = 2 + F_1 \\
F_4 = F_3 + F_2
\]
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

Maxwell J Dunne – Spring 2016
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(3)
Fibonacci(4)

\[ F_3 = 2 + 1 \]
\[ F_4 = F_3 + F_2 \]
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)

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>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(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)

\[
\begin{align*}
F_2 &= 1 + F_0 \\
F_4 &= 3 + F_2
\end{align*}
\]

Stack (top)

- Fibonacci(2)
- Fibonacci(4)

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

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

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

Stack (top)

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

\[
F_2 = 1 + 1 \\
F_4 = 3 + F_2
\]
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

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

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     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);
}
```

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);
}
```

```
str    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);
}
```

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   p
This   is  the  end.
```

Maxwell J Dunne – Spring 2016
Recursion
Complex data example

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

- `str`: The string to count words in.
- `p`: The pointer to the current character being checked.

Example:
```
This is the end.
```

Word count:
```
This: 1
is: 1
the: 1
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);
}
```

```c
This  is  the  end.
```

```c
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
const #define

- no floats

\[ 2 \text{Hz} = 0.5 \text{seconds} \]

- \[ 20 \text{seconds} = 40 \text{ticks} \]
Free running timer

\[ a = 3 \]

\[ 6 - a = 3 \]

\[ a = \text{freeTime} \]

```
if (freeTime - a > LONG_press)
    timeExpired
```
Vin + 8 - t 9 = 250

250 - 250 = 0

251 - 250 = 1

255 - 250 = 5

0 - 250 = 6
Binary trees

- ADT where nodes:
  - Have 0, 1, or 2 children
  - Have a single parent
- Generally nodes only know their children
- Generally traversal is top-down
  - From parents to children
Binary trees

- We will only discuss full binary trees
- Size is $2^n - 1$
Binary trees

Traversal

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

• A linear representation of a tree
Binary trees

Serialization

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

Diagram showing a binary tree with red and green circles highlighting different parts.
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 the 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

2^n-1
2^n-2
2^n-3
2^n-4

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

A
  B   E
    C   D

CMPE-013/L: “C” Programming
Maxwell J Dunne – Spring 2016
Binary trees
Serialization

A  B  C  D  E  F

Diagram of a binary tree:
- A is the root.
- B and C are children of A.
- D and E are children of B.
- F and G are children of E.
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

```
[ ] [ ] [ ] [ ] [ ] [ ]
```

```
8
  2   7
  0  4  2  5
```
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne

Spring 2016
Software Engineering

- Design
- Build
Software Engineering

Design process

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

Turnin
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

```
i++  / / i = i + 1
```
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

Leds.h
Buttons.c
3 files
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
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;
}
```
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 then 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
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

```
Embedded example

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

With OS example

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