CMPE-013/L

Introduction to “C” Programming

Maxwell J Dunne
Recursion
Recursion

- Solving problems by breaking them into smaller parts
- "divide and conquer"
- Relies on the problem having self-similarity

Example

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

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

Stack (top)

Partial results pushed on stack
Factorial term replaced with result

5! = 5 * 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>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factorial(5)</td>
<td>[3] 4! = 4 * 3!</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[4] 5! = 5 * 4!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Factorial term replaced with result
Recursion
Evaluation of Recursive Functions

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

Stack (top)

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

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

<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>

<table>
<thead>
<tr>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td>2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td>4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<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)

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

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

**Partial results pushed on stack**

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

**Factorial term replaced with result**

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

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

```
Stack (top)

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

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

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)

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

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(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

\[
F_3 = F_2 + F_1
\]

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

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

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>F_2 = F_1 + F_0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F_3 = F_2 + F_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F_4 = F_3 + F_2</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>

Fibonacci sequence:
- F_0 = 0
- F_1 = 1
- F_n = F_{n-1} + F_{n-2}
Recursion
Evaluation of Recursive Functions

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

Stack (top)

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)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fibonacci(1)</td>
</tr>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(3)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

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

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

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

Partial results pushed on stack
Function call replaced with result

F₀ = 1
F₂ = 1 + F₀
F₃ = F₂ + F₁
F₄ = F₃ + F₂
Recursion
Evaluation of Recursive Functions

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

Stack (top)

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

Maxwell J Dunne
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>Fibonacci(2)</td>
<td></td>
</tr>
<tr>
<td>Fibonacci(3)</td>
<td></td>
</tr>
<tr>
<td>Fibonacci(4)</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)

Partial results pushed on stack
Function call replaced with result

F_3 = 2 + F_1
F_4 = F_3 + F_2

Maxwell J Dunne

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

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)

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

\[
F_3 = 2 + 1
\]

\[
F_4 = F_3 + F_2
\]
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

\[ F_4 = 3 + F_2 \]
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)
  - Fibonacci(1)
  - Fibonacci(2)
  - Fibonacci(4)

- 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)

<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
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)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<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(4)</strong></td>
</tr>
</tbody>
</table>

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

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

```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);
}
```

```
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

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: string input
- p+1: string pointer incremented

```
This is the end.
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>is</td>
<td>the</td>
<td>end</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Recursion
Complex data example

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

```
This is the end.
```

1 1
Recursion
Complex data example

Example

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

The function `WordCount` counts the number of words in a string `str`. It uses recursion by calling itself with the substring starting from the next character after the space found by `strchr`. The base case is when `str` is null or reaches a space, in which case it returns 0. Otherwise, it returns 1 (for the current word) plus the result of the recursive call on the next substring.
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

CMPE-013/L: “C” Programming
Recursion
Complex data example

Example

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

This is the end.

1 1 1 1
Recursion
Complex data example

Example

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

This | is | the | end.
--- | --- | --- | ---
1 | 1 | 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

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

This is the end.

4
Binary trees

malloc()
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 serialization process]
Binary trees

Serialization

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

A  B  C  D
Binary trees

Serialization

A  B  C  D

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

A  B  C  D
Binary trees

Serialization

A  B  C  D  E  F  G

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

Serialization

A   B   C   D   E   F

Diagram:
Binary trees
Serialization

A   B   C   D   E   F   G
Binary trees

Serialization

A B C D E F G
Binary trees

Serialization
Binary trees
Serialization

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

Introduction to “C” Programming

Maxwell James Dunne
Hashing
Hashing

- Mapping data of arbitrary size into a fixed-size hash value
- Utilizes a hash function
- Effectively mapping values from a higher-dimensional space into a lower one
- Produces aliasing
Hashing

Example

<table>
<thead>
<tr>
<th>keys</th>
<th>hash function</th>
<th>hashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td></td>
<td>00, 01</td>
</tr>
<tr>
<td>Lisa Smith</td>
<td></td>
<td>02, 03</td>
</tr>
<tr>
<td>Sam Doe</td>
<td></td>
<td>04, 05</td>
</tr>
<tr>
<td>Sandra Dee</td>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>
Hashing

Uses

- CPU caches
- Datatypes: hashmap/dictionary
- Data verification: fingerprinting
- Data compression: vector quantization

\[ a \left[ \right. \]
Hashing
8-bit XOR

16-bit data → XOR() → 8-bit hash

16-bit data → XOR() → 8-bit hash
Checksums

Definition

- A small piece of data computed from an original source of data for the purposes of verifying it
- Can utilize **hashing**
- Relies on a **checksum algorithm**

![Diagram of checksum process]

Data → Checksum function → Checksum
Checksums

Uses

- Verify data transmit over radio
  - Such as in a telemetry stream for a robot
- Verify the integrity of a data burned to a CD
- Verify correctness of a file downloaded off the internet

.zip archive: apache-ant-1.9.4-bin.zip [PGP] [SHA1] [SHA512] [MD5]

File zilla
Checksums

Checksum functions

- SHA512
  - 512-bits
- MD5
  - 128-bits
- XOR
  - Usually wordsize to simplify computation, between 8- and 64-bits

CRC32 (DEADBEF)
Priority

10110101

ISBN

Priority

|   |   |   |   |   |   |

10%
Britney Spears
Checksums

Using checksums

- When used in message transmission, transmit both the data **and** the checksum
Checksums

Using checksums

- On message reception, recalculate the checksum and verify that it matches the one transmit
Checksums
XOR Checksum in C

Syntax

```c
uint8_t CalcStringChecksum(const char *data);
```

Example

```c
char *str = "Mary had a little lamb."

uint8_t strChecksum = CalculateStringChecksum(str);

printf("XOR(%s) = %02X\n", str, strChecksum);

Check\w= 64\text{to};
```
Random number generation
Random number generation

Usage

- Pretty much all games
  - Described with "randomness" and "variation"
- Security and cryptography
- Problem solving algorithms
- Music/video playback
- Recommendation systems
- User interfaces
Random number generation

Categories

- "True" random
  - Result of noisy physical phenomena
  - No initial input (besides, possibly, power)
  - No repeatable sequence
  - Not in the C standard

Pseudo-random
- Result of algorithm
- Relies on initial (seed) value
- Produces cycles of numbers
- In the C standard
Random number generation

Functions

Syntax

```c
void srand(unsigned int seed);
```

- **seed** is the initial value to iterate on
  - Remembered until next call to `srand()`
Random number generation

Functions

Syntax

```
int rand(void);
```

- Returns pseudo-random number based on seed
  - Values between \texttt{INT\_MIN} and \texttt{INT\_MAX}
  - See set by \texttt{srand()} otherwise defaults to 1
- All \texttt{rand()} calls with the same seed produce the same sequence.
void main()
{
    srand(67);
    int truth = rand(), guess;
    do {
        printf("Guess the number:");
        scanf("%f", &guess);
        if (guess == truth) {
            printf("You win!\nTry again.");
            truth = rand();
        }
    } while (1);
}
Random number generation

Initial seed

• But how do we choose a good initial seed?
• **Hardcode it**
  – The PS3 problem
• **Fake it**
  – Use compile-time information like ___DATE___ and ___TIME___
  – Use data that changes
    • Current date/time
    • User input
    • Physical sensors
Random number generation

Real-world example

```c
// The first part of our seed is a hash of the compilation time string.
char seed1[] = __TIME__;
int seed1Len = strlen(seed1);
int firstHalf = seed1Len / 2;
uint16_t seed2 = 0;
int i;
for (i = 0; i < seed1Len; i++) {
    seed2 ^= seed1[i] << ((i < firstHalf) ? 0 : 8);
}

// Now we hash in the time since first user input (which, as a 32-bit number, is split and each half is hashed in separately).
srand(seed2 ^ (counter >> 16) ^ counter);
```
Random number generation

Hardware crypto on the PIC32MZ

- The PIC32MZ series has hardware RNG

![Diagram of random number generation](image-url)
Random number generation

Difference between random and pseudo-random (taken from Random.org)

Random

PHP .rand()
Encryption
Encryption

- Encoding data such that only agents with a key can access it
- Used everywhere
  - Especially now with the NSA's shenanigans
- Relies on computational complexity and secret knowledge
Encryption
Types

• Multiple types of encryption:
  – Public key – Separate keys for encryption and decryption
  – Private/Symmetric key – Same key used for encryption and decryption
Encryption

Public key

- Separate keys for encryption and decryption
- Encryption key is public
  - Anyone can encode
- Decryption key is private
  - Only authorized parties can decode
Encryption
Public key

Bob
Hello Alice! → Encrypt → Alice's public key
   
   6EB69570
   08E03CE4

Alice
Hello Alice! → Decrypt → Alice's private key
Encryption

Symmetric key

- Single key for encryption and decryption
- Key needs to be kept private by all parties

![Diagram of encryption process]

Alice
Hello Alice!

Bob
Encrypt
6EB6957008E03CE4

Secret key

Decrypt
Hello Alice!
Encryption

Encryption function

- The operation for encrypting from a key must be known for encryption and decrypting
- Simplest bidirectional function is xor()
Encryption
Symmetric key example

- If Alice and Bob want to communicate, both need to agree on the private key.
Encryption
Symmetric key example

- If Alice and Bob want to communicate, both need to agree on the private key.
Encryption
Symmetric key example

- If Alice and Bob want to communicate, both need to agree on the private key.

Diagram:
- Alice
  - Encryption key
  - Encrypted data
  - Time

- Bob
  - xor()
  - Data
Encryption
Real-world example

- Problem: Two agents need to determine which goes first. Don't allow cheating
- Emulate flipping a coin
  - Agents each guess a number, depending on those numbers either the higher or lowest number wins
- Problem is time:
  - In real world systems, no event occurs simultaneously
  - If an agent sends their guess first, the other agent can cheat by choosing their guess appropriately
Encryption
Real-world example

- Solution: Split the guessing into 2 stages
  - Send an encrypted guess
  - After receiving the other agent's guess, send your decryption key.

- New problem:
  - If agent receives other agent's guess & key, they could cheat by generating a new guess and key that still has the same encrypted value (which they've already sent)
Encryption
Symmetric key example

Alice
- Encrypted data & id → Bob
- Decrypted data
- Regenerate key & guess

time

Bob
- Encrypted data & id
- Encryption key

Alice wins!
Encryption
Real-world example

• Solution: Also send a pseudo-unique identifier of the key/guess pair

• New problem:
  − If agent receives other agent's guess & key, they could cheat by generating a new guess and key that still has the same encrypted value (which they've already sent
Encryption
Symmetric key example

Alice

Encrypted data & id

Encrypted data & id

Decrypted data

Encryption key & guess

Encryption key & guess

Regenerate key & guess

Bob

Bob verifies Alice's data

Bob detects cheating!
Communications
Communications

- Communications can almost never be assumed to be simultaneous
  - Due to real-time constraints
  - Technical limitations

- Systems require synchronization
  - Handled with state machines
Communications

Between two agents

Alice

Bob

time
Communications
With a protocol

- Bob needs to ACK after receiving an IMP message

Alice ——— IMP ——— Bob

——— ACK ———

time
Communications
With a protocol

- But what if Bob is busy? Maybe receiving more data from Alice?
Communications
With a protocol

- An FSM can be used for remembering than an ACK needs to be sent

Alice  →  Bob

WAITING

time
Communications
With a protocol

- An FSM can be used for remembering than an ACK needs to be sent

Alice → Bob

WAITING

IMP

REC_IMP

time
Communications
With a protocol

- An FSM can be used for remembering than an ACK needs to be sent

Alice

Bob

time

WAITING

IMP

REC_IMP
Communications
With a protocol

- An FSM can be used for remembering than an ACK needs to be sent

Alice

Bob

WAITING

REC_IMP

SENT_ACK

time
Communications
With a protocol

• An FSM can be used for remembering than an ACK needs to be sent
CMPE-013/L

Toaster Oven Lab

Maxwell James Dunne
Integer Timing

Free running counters and precision

\[
\begin{align*}
time++; \\
\leq time; \\
if (time < n)
\end{align*}
\]
10 - 250
# OLED Display

Formatting and Update Cycles

```c
// Define "\x01\x0d\x01\x0d"
0% > 0% > 0% > 0% > 0%

Update OLED1())
```

Maxwell James Dunne
Switch

if (no update) { update OLED1; }
\[ C = 14326 \]

static constexpr = 14326

#define FOO 14326
\texttt{const Chor} *

2:30

150s \times 150 \% 60 \Rightarrow 30

150/60 \geq 2
Countdown
if(2Hz)

mond
if(2Hz)

ISR
2 Hz = TRIG
FSM
$$\frac{\text{Pot} \times \left[ \frac{\text{FP} \text{ division}}{\text{Shift and add} \ 100x} \right]}{60}$$
< 8
don't care