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);
}
```

\[
5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1
\]

\[
5! = 5 \cdot 4!
\]
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(5)

[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

[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

[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>2! = 2 * 1!</td>
<td>3! = 3 * 2!</td>
</tr>
<tr>
<td></td>
<td>3! = 3 * 2!</td>
<td>4! = 4 * 3!</td>
</tr>
<tr>
<td>Factorial(2)</td>
<td></td>
<td>5! = 5 * 4!</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>
</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>[0] 1! = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1] 2! = 2 * 1!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2] 3! = 3 * 2!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3] 4! = 4 * 3!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4] 5! = 5 * 4!</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>[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>
<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**

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

- [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! = 2 * 1 = 2
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

[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
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

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

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)

<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>1 = 1</td>
</tr>
<tr>
<td>[1]</td>
<td>2! = 2 * 1!</td>
<td>2 * 1 = 2</td>
</tr>
<tr>
<td>[2]</td>
<td>3! = 3 * 2!</td>
<td>3 * 2 = 6</td>
</tr>
<tr>
<td>[4]</td>
<td>5! = 5 * 4!</td>
<td>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
Recusion
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

F₄ = F₃ + F₂

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

\[
F_3 = F_2 + F_1
\]

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

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

Stack (top)

<table>
<thead>
<tr>
<th>Stack Contents</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(2)
Fibonacci(3)
Fibonacci(4)

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)

<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(1)</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

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

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

Stack (top)

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

<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>
</tbody>
</table>

Partial results pushed on stack

Function call replaced with result

\[
F_2 = 1 + F_0 \\
F_3 = F_2 + F_1 \\
F_4 = F_3 + F_2
\]

Maxwell J Dunne
Recursion
Evaluation of Recursive Functions

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

Stack (top)

<table>
<thead>
<tr>
<th>Fibonacci(0)</th>
</tr>
</thead>
<tbody>
<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

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

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

Stack (top)

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

Partial results pushed on stack

Function call replaced with result

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

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

Stack (top)

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

```
F_1 = 1
F_3 = 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)

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

F3 = 2 + 1
F4 = F3 + F2
Recursion
Evaluation of Recursive Functions

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

\[ F_4 = 3 + F_2 \]
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>
</tbody>
</table>

Fibonacci(2)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[
F_2 = F_1 + F_0 \\
F_4 = 3 + F_2
\]

Maxwell J Dunne
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)

Stack (top)

Fibonacci(2)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[
F_2 = 1 + F_0
\]

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

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

Stack (top)

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

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

Partial results pushed on stack

Function call replaced with result

\[
F_2 = 1 + 1 \\
F_4 = 3 + F_2
\]
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
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 1;
    }
    return 1 + WordCount(p + 1);
}
```
Recursion
Complex data example

Example

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

Diagram:
- `str` to `p+1`
- `This is the end. 1`
Recursion
Complex data example

Example

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

Example:

```
This is the end.
```

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

```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 1;
    }
    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);
}
```

This is the end.

1 1 1
Recursion
Complex data example

Example

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

This is the end.

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);
}
```

Example

```
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 1;
    }
    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 1;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

<table>
<thead>
<tr>
<th>This</th>
<th>is</th>
<th>the</th>
<th>end.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</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 1;
    }
    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

Traversals

- 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 binary tree serialization]
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 pointing to nodes, illustrating the serialization process.
Binary trees

Serialization

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

over
## Binary trees

**Serialization**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
</table>

![Binary tree diagram]
Binary trees

Serialization

A  B  C  D
Binary trees
Serialization

A   B   C

A
  /   \
 B   C
   /\  /
 C   D E
Binary trees
Serialization
Binary trees
Serialization

A | B | C | D |

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

A B C D E F G
## Binary trees

### Serialization

| A | B | C | D | E | F |

![Binary tree diagram](image-url)
Binary trees
Serialization

A B C D E F G
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
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>01</td>
<td>00</td>
</tr>
<tr>
<td>Lisa Smith</td>
<td>02</td>
<td>01</td>
</tr>
<tr>
<td>Sam Doe</td>
<td>03</td>
<td>02</td>
</tr>
<tr>
<td>Sandra Dee</td>
<td>04</td>
<td>03</td>
</tr>
<tr>
<td></td>
<td>05</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
Hashing

Uses

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

8-bit XOR

\[ 8 \oplus 080808 \]

16-bit data \rightarrow XOR() \rightarrow 8-bit hash

16-bit data
Checksums
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**
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]
Checksums
Checksum functions

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

<RC>32
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

Diagram:

```
Data → Checksum function → Checksum → ==
```
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);
```
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**

```c
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
// 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
Random number generation

Difference between random and pseudo-random (taken from Random.org)
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
6EB6957008E03CE4
Alice's public key

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
Encryption

Encryption function

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

```
0b11001010
```

```
xor()
```

```
0b01100000
```

```
0b10101010
```

```
0b11001010
```
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.

```
Alice  Encrypted data  Bob
       data            xor()
       Encryption key data
```

Time
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
  - Decrypted data
  - Regenerate key & guess

Bob
- Encrypted data & id
  - Encryption key

Time

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
Decrypted data
Regenerate key & guess
time

Bob

Encrypted data & id
Encryption key & guess
Bob verifies Alice's data

Encryption key & guess

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

Bob

IMP

ACK

time
Communications

With a protocol

- But what if Bob is busy? Maybe receiving more data from Alice?

Alice

Bob

IMP

ACK

time
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
Communications
With a protocol

• An FSM can be used for remembering than an ACK needs to be sent
Communications
With a protocol

- An FSM can be used for remembering than an ACK needs to be sent
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
* Indicates that you calculate the elapsed time by subtracting your stored time from the current Free Running Time.

2Hz timer triggered && Cooking time left == 0
Update the display
Clear 2Hz timer flag

reset 2Hz Timer
Save initial cook time
Turn oven on
Clear button event
Update the display

expanded time* >= LONG_PRESS
Switch input selection
Update time/temp from pot
Update the display
Clear button event

expanded time* < LONG_PRESS && BUTTON_EVENT_3UP
Reset time & temp, input selection
Rotate cooking mode
Update the display
Clear button event

COUNTDOWN

BUTTON EVENT 4DOWN
Store Free Running Time
Clear button event

2Hz timer triggered &&
Cooking time left > 0
Cooking time left -- 1
Update the display/LEDs
Clear 2Hz timer flag

2Hz timer triggered &&
Cooking time left == 0
Clear timer flag

2Hz timer triggered &&
Cooking time left > 0
Cooking time left -- 1
Update the display
Clear timer flag

elapsed time* >= LONG_PRESS
Integer Timing
Free running counters and precision

\[(\text{Adc read (\text{t})} \gg 2) << 1\]
100H₂ \rightarrow \text{Buttons}

2H₂ \rightarrow \text{event} = \text{TRUE};

S \rightarrow \text{free running timer}

\text{timer++};
Short = FRT;

FRT - Short D: Long-Press
while(1)
    printf("\ln/0x", event);
event
	time = 0;

if (time > Long)

event
	start + time = time;

if ((time - start + time) > Long)
32768 Hz

254 ← 255
255
0
1
2
3
4

-252

1 ms
16-bit
~90 s
32-bit
~30 days

\[
\frac{50}{2} = \frac{29}{2} = \frac{29}{2}
\]

30

29

29
OLED Display
Formatting and Update Cycles

#define OVON "\033[3m\033[1m\033[0m"

state machine
update OLED = TRUE;
Integermath

NO FP

Seconds >> 1

min = Seconds / 60
Sec = Seconds % 60

10 → 20
9.5 19
9 18
8.5 17

32768
```c
#define OVENLONTOP "\x3\x3\x3"

#define TIME: %0d

printf("0%s", ove)
printf("TIME: %0d")
```
"Hello"
"Goodbye"

Hello Goodbye
while (1):
    if (event != None):
        event = None

100H₂ → 100 H₂
    event = 'Button (EF)';
case Reset:
    // actions
    statement = Start;
    break;
    break;