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)

[4] \[ 5! = 5 \times 4! \]
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>4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factorial(5)</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)

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

Partial results pushed on stack

Factorial term replaced with result

[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>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

<table>
<thead>
<tr>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] 2! = 2 * 1!</td>
</tr>
<tr>
<td>[2] 3! = 3 * 2!</td>
</tr>
<tr>
<td>[3] 4! = 4 * 3!</td>
</tr>
<tr>
<td>[4] 5! = 5 * 4!</td>
</tr>
</tbody>
</table>
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] 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>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</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! = 1
- 2! = 2 * 1 = 2
- 3! = 3 * 2 = 6
- 4! = 4 * 3 = 24
- 5! = 5 * 4 = 120
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>Factorial(3)</td>
</tr>
<tr>
<td>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack

<table>
<thead>
<tr>
<th>Level</th>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>1! = 1</td>
<td>= 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>[3]</td>
<td>4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>[4]</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)

| Factorial(4) |
| Factorial(5) |

Partial results pushed on stack

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

Factorial term replaced with result

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>1</td>
</tr>
<tr>
<td>[1]</td>
<td>2 * 1 = 2</td>
</tr>
<tr>
<td>[2]</td>
<td>3 * 2 = 6</td>
</tr>
<tr>
<td>[3]</td>
<td>4 * 6 = 24</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</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>[3]</td>
<td>5! = 5 * 4!</td>
<td>= 5 * 24 = 120</td>
</tr>
</tbody>
</table>

Maxwell J Dunne
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

```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)

Fibonacci(4)
```

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)

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)

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

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)

<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

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

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

Stack (top)

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

Partial results pushed on stack

Function call replaced with result

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

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

Stack (top)

<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>
Fibonacci(2)
Fibonacci(3)
Fibonacci(4)

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

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

Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

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

\[
F_1 = 1 \\
F_3 = 2 + F_1 \\
F_4 = F_3 + F_2
\]

Maxwell J Dunne
Recursion
Evaluation of Recursive Functions

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

Stack (top)

Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_3 = 2 + 1 = 3 \]
\[ F_4 = F_3 + F_2 = 3 + 1 = 4 \]
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_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

\[
F_2 = F_1 + F_0 \\
F_4 = 3 + F_2
\]
Recursion
Evaluation of Recursive Functions

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

Stack (top)

<table>
<thead>
<tr>
<th>Fibbonacci(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibbonacci(2)</td>
</tr>
<tr>
<td>Fibbonacci(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)

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

F_0 = 1
F_2 = 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)

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

Partial results pushed on stack
Function call replaced with result

F₂ = \[ 1 + 1 \]
F₄ = \[ 3 + F₂ \]

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

\[ F_4 = 3 + 2 \]
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

Example:

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

```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

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

Example:

```
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);
}
```

This is the end.

1 1

str  p
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 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

```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

```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.

1 3
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
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>

```plaintext
          o
        /   
       o     o
      /   \
     o     o
    /   \
   o     o
  /   \
 o     o
```
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.
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 |

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

A   B   C

A   B
C
Binary trees

Serialization

A   B   C   D   

A
  /   
B   C
   /   
D
Binary trees

Serialization
Binary trees
Serialization

A  B  C  D  E  F  G
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

A  B  C  D  E  F  G
Binary trees

Serialization

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

Toaster Oven Lab

Maxwell James Dunne
Integer Timing
Free running counters and precision
$100 \text{H}_2 \rightarrow \text{Buttons}$

$2 \text{H}_2 \rightarrow \text{event} = \text{TRUE};$

$S \text{H}_2 \rightarrow \text{Free running timer}$

```c
++time;
```
\text{Short} = \text{FRT};

FRT - \text{Short} \text{ D: Long-Press}
while(1)
    printf("in\%0X", event);
event

time = 0;

if (time > Long)

event

start + time = time;

if ((time - start + time) > Long)
OLED Display
Formatting and Update Cycles

```
#define OVON "\033[33m\033[32m\033[33m"

state machine
Update Oled = TRUE;
```
Integer Math

NO FP

seconds >> 1

\[\text{min} = \frac{\text{seconds}}{60}\]

\[\text{Sec} = \text{seconds} \mod 60\]

\[\begin{array}{cccc}
\text{D} & \text{D} & \text{D} & \text{D} \\
\text{D} & \text{D} & \text{D} & \text{D} \\
\text{D} & \text{D} & \text{D} & \text{D} \\
\end{array}\]

\[\begin{array}{c}
10 \rightarrow 20 \\
9.5 \rightarrow 19 \\
9 \rightarrow 18 \\
8.5 \rightarrow 17 \\
32768
\end{array}\]
Sprintf
#define OVENU TIP "\x3\x3\x3"

Sprintf("0%s", ov
#define T"TIME: %0d"
Sprintf(T, 3)
"Hello"
In good bye

Hello
Goodbye
```python
main
while (1)
    if (event != None)
        event = None
```

```
100 H_2
```

```
[100 H_2
  event = Button(Ctrl);
]
case Reset:
    //actions
    store = Store;
    break;