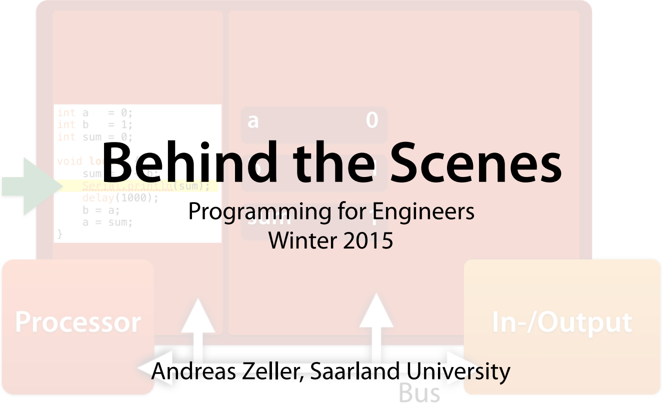
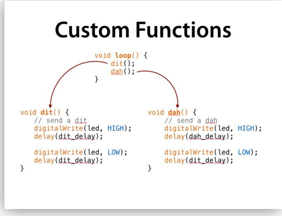


Behind the Scenes

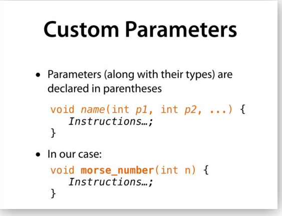
Programming for Engineers
Winter 2015

Andreas Zeller, Saarland University



Custom Functions

```
            void loop() {
              dit();
              dah();
            }

void dit() {                     void dah() {
    // send a dit                    // send a dah
    digitalWrite(led, HIGH);         digitalWrite(led, HIGH);
    delay(dit_delay);                delay(dah_delay);

    digitalWrite(led, LOW);          digitalWrite(led, LOW);
    delay(dit_delay);                delay(dit_delay);
}                                }
```



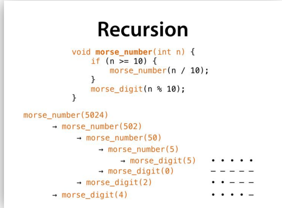
Custom Parameters

- Parameters (along with their types) are declared in parentheses

```
void name(int p1, int p2, ...) {
    Instructions…;
}
```

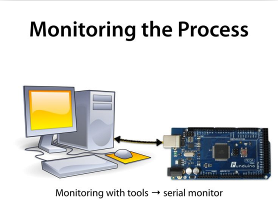
- In our case:

```
void morse_number(int n) {
    Instructions…;
}
```



Recursion

```
void morse_number(int n) {
    if (n >= 10) {
        morse_number(n / 10);
    }
    morse_digit(n % 10);
}

morse_number(5024)
    → morse_number(502)
        → morse_number(50)
            → morse_number(5)
                → morse_digit(5)      • • • • •
            → morse_digit(0)          – – – – –
        → morse_digit(2)              • • – – –
    → morse_digit(4)                  • • • • –
```



Monitoring the Process

Monitoring with tools → serial monitor

Today's Topics

- Variables
- Assignments
- Computing models

Storing Values

- We want to store values during computations
- We want to assign the result to a variable

Assignment

- The assignment

  ```
  name = value
  ```

  causes the variable *name* to have the new value *value*
- As execution resumes, every access to the variable *name* produces this value *value* (until the next assignment)

Assignment

```
int x = 0;

x = 1;
Serial.println(x);   - prints 1

x = 2;
Serial.println(x);   - prints 2

if (x > 2) {
   x = -1;
}
if (x > 1) {
   x = 100;
}
Serial.println(x);   - prints 100
```

Fibonacci

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

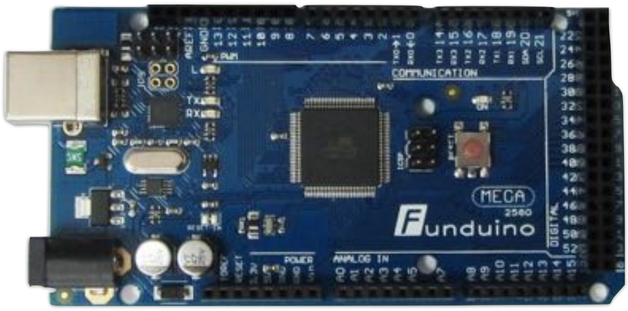
- prints 1 2 3 5 8 13 …

What Happens Here?

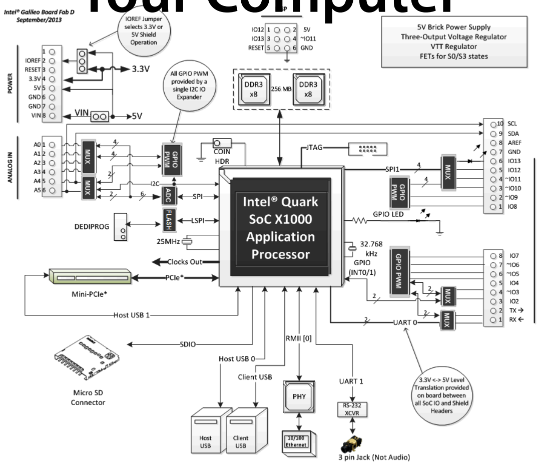
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

How does the computer execute the program?

Your Computer

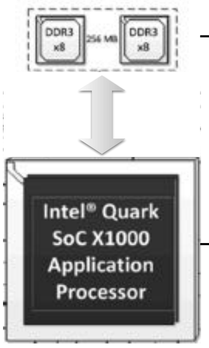


Your Computer



https://communities.intel.com/docs/DOC-21822

Your Computer



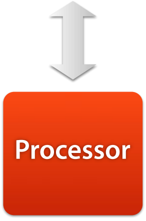
Memory
contains d*ata*

Processor
executes instructions

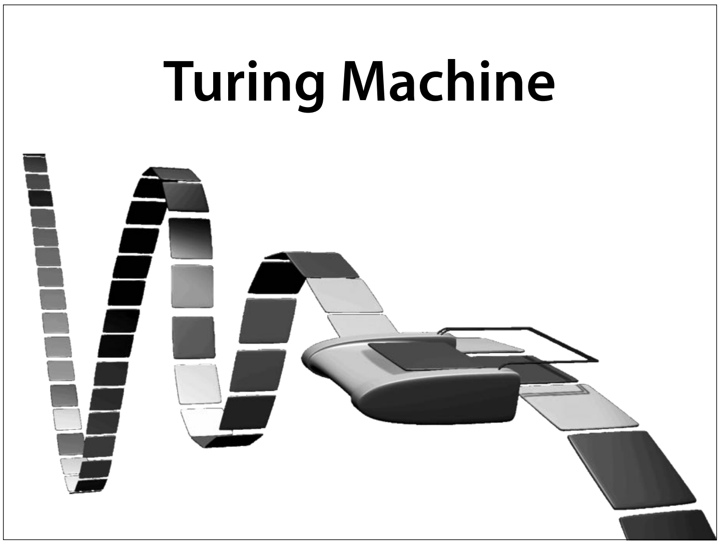
Your Computer

0	I	I	0	I	0	0	0	I	…

Memory



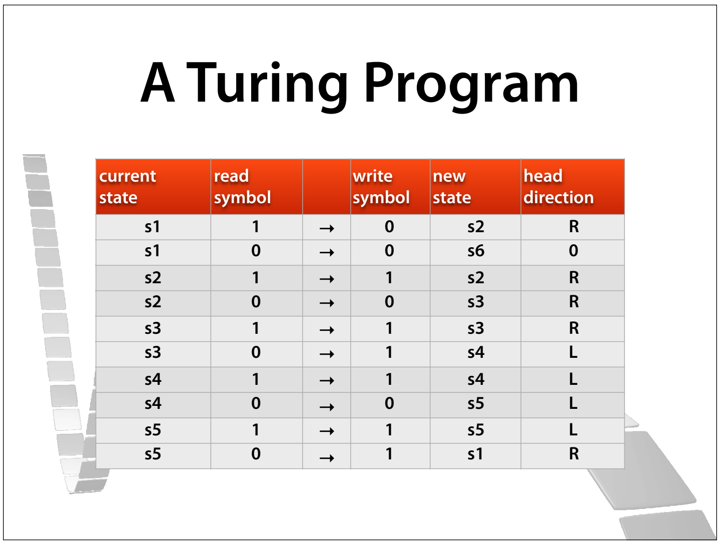
We can abstract further



Turing Machine

A Turing machine is an abstract "machine"[1] that manipulates symbols on a strip of tape according to a table of rules; to be more exact, it is a mathematical model that defines such a device.[2] Despite the model's simplicity, given any computer algorithm, a Turing machine can be constructed that is capable of simulating that algorithm's logic.[3]

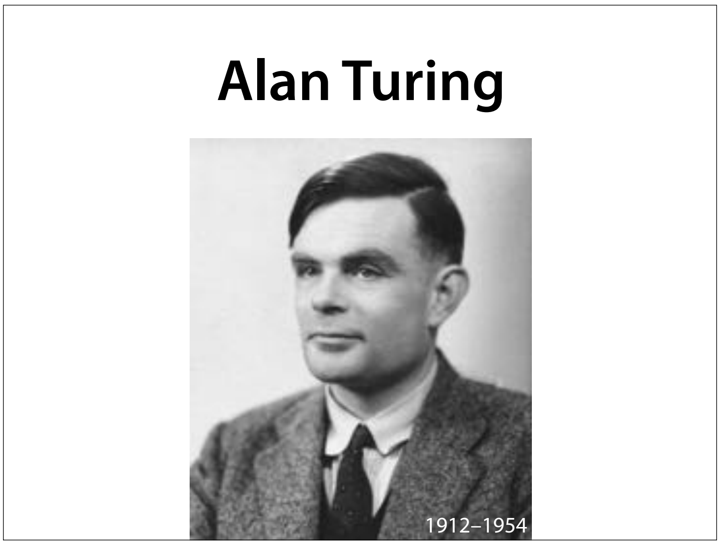
The machine operates on an infinite[4] memory tape divided into cells.[5] The machine positions its head over a cell and "reads" (scans[6]) the symbol there. Then per the symbol and its present place in a finite table[7] of user-specified instructions the machine (i) writes a symbol (e.g. a digit or a letter from a finite alphabet) in the cell (some models allowing symbol erasure[8] and/or no writing), then (ii) either moves the tape one cell left or right (some models allow no motion, some models move the head),[9] then (iii) (as determined by the observed symbol and the machine's place in the table) either proceeds to a subsequent instruction or halts[10] the computation. (Wikipedia)



A Turing Program

current state	read symbol		write symbol	new state	head direction
s1	1	→	0	s2	R
s1	0	→	0	s6	0
s2	1	→	1	s2	R
s2	0	→	0	s3	R
s3	1	→	1	s3	R
s3	0	→	1	s4	L
s4	1	→	1	s4	L
s4	0	→	0	s5	L
s5	1	→	1	s5	L
s5	0	→	1	s1	R

Doubles the number of 1’s on the tape (Wikipedia)



Alan Turing

1912–1954

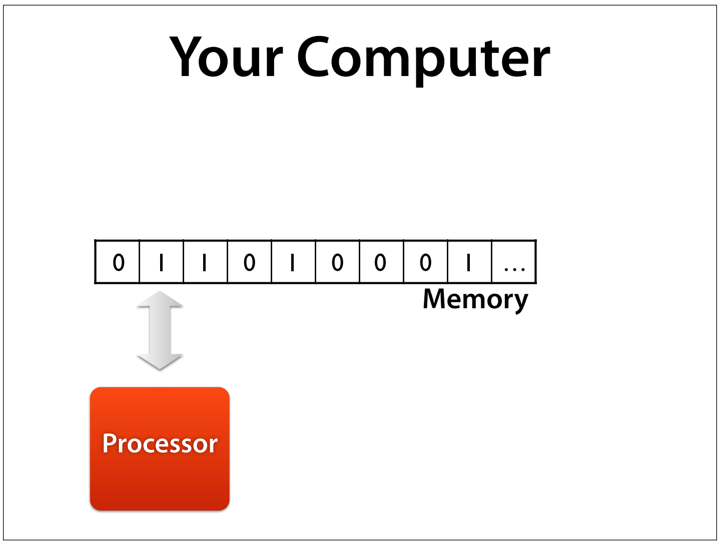
Alan Mathison Turing, OBE, FRS (/'tjʊərɪŋ/; 23 June 1912 – 7 June 1954) was a British pioneering computer scientist, mathematician, logician, cryptanalyst and theoretical biologist. He was highly influential in the development of computer science, providing a formalisation of the concepts of algorithm and computation with the Turing machine, which can be considered a model of a general purpose computer.[2][3][4] Turing is widely considered to be the father of theoretical computer science and artificial intelligence.

Enigma

An Enigma machine was a series of electro-mechanical rotor cipher machines developed and used in the early to mid twentieth century for commercial and military usage. Enigma was invented by the German engineer Arthur Scherbius at the end of World War I.[1] Early models were used commercially from the early 1920s, and adopted by military and government services of several countries, most notably Nazi Germany before and during World War II.[2] Several different Enigma models were produced, but the German military

The Bombe

The bombe was an electromechanical device used by British cryptologists to help decipher German Enigma-machine-encrypted secret messages during World War II.[2] The US Navy[3] and US Army[4] later produced their own machines to the same functional specification, but engineered differently from each other and from the British Bombe. (Wikipedia)



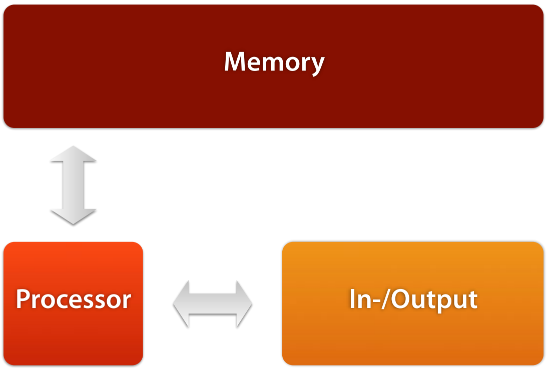
Your Computer

0	I	I	0	I	0	0	0	I	...

Memory

Processor

Your Computer

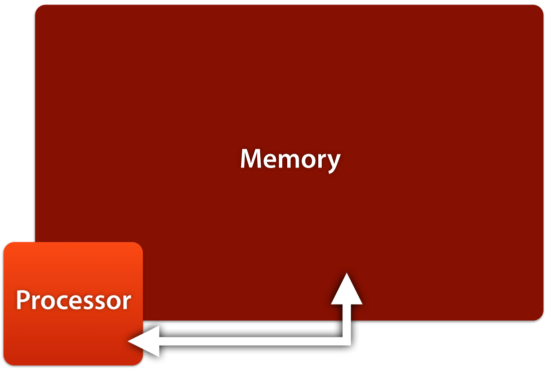


Memory

Processor

In-/Output

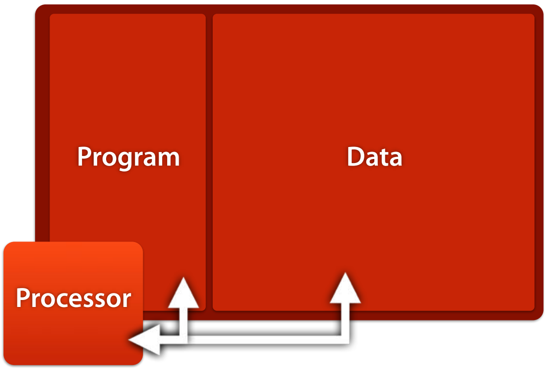
Memory



Memory

Processor

Memory

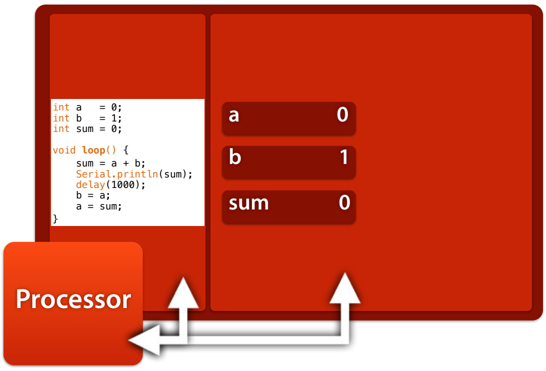


Program

Data

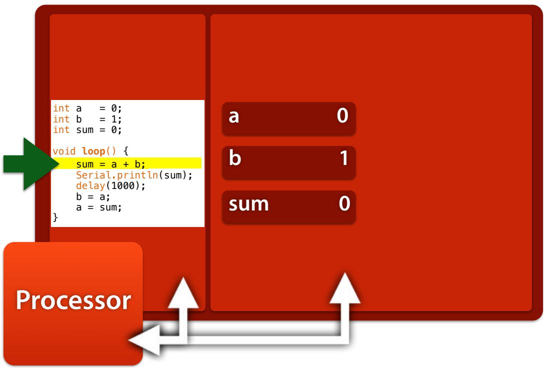
Processor

Memory



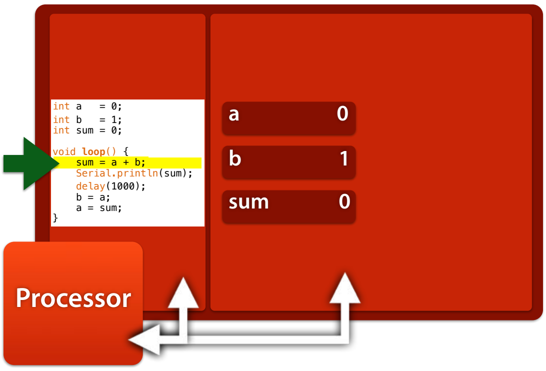
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Instruction Counter



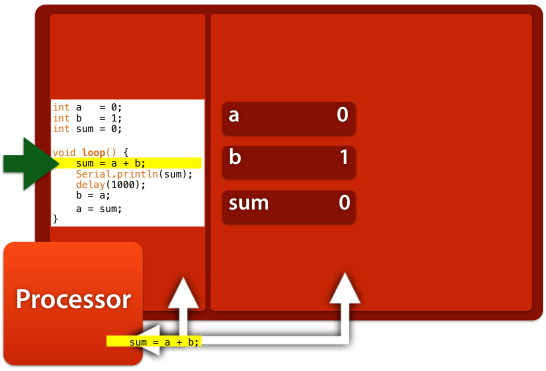
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Reading Instructions



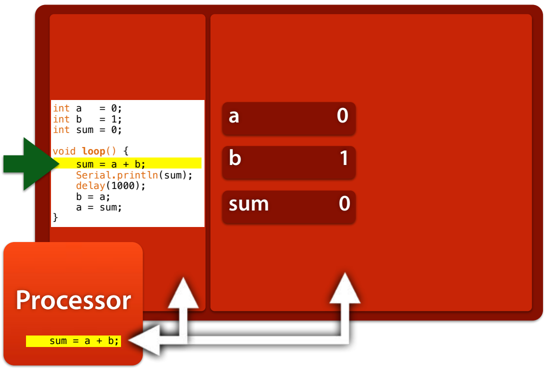
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Reading Instructions



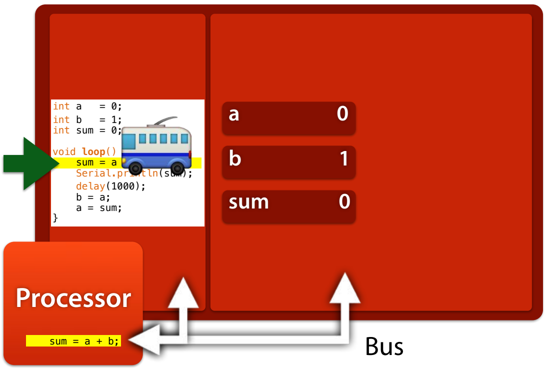
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Reading Instructions



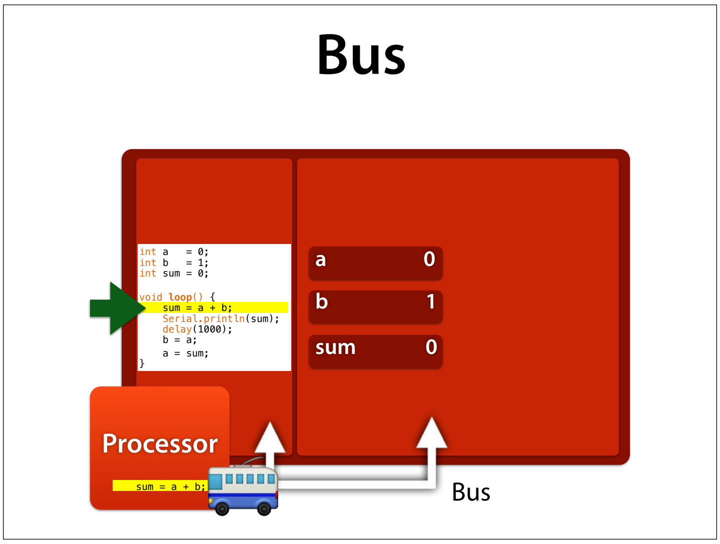
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Bus



```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

The bus moves instructions into the processor



Bus

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

a 0

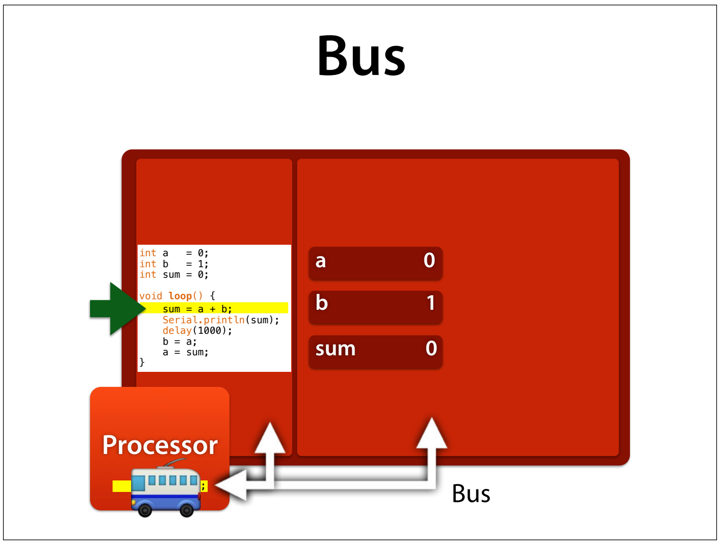
b 1

sum 0

Processor

sum = a + b;

Bus



Bus

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

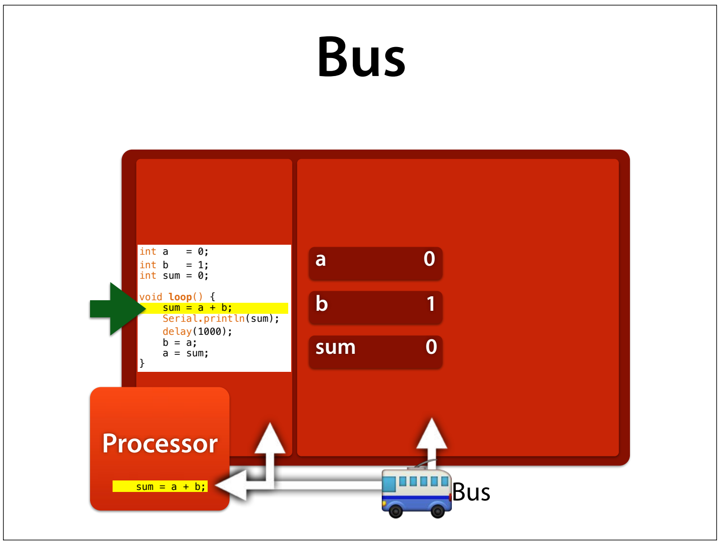
a 0

b 1

sum 0

Processor

Bus



Bus

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

a 0

b 1

sum 0

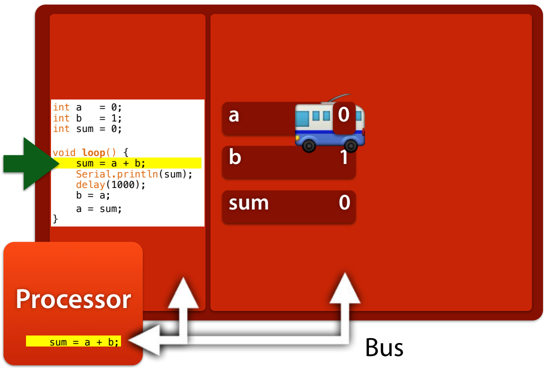
Processor

sum = a + b;

Bus

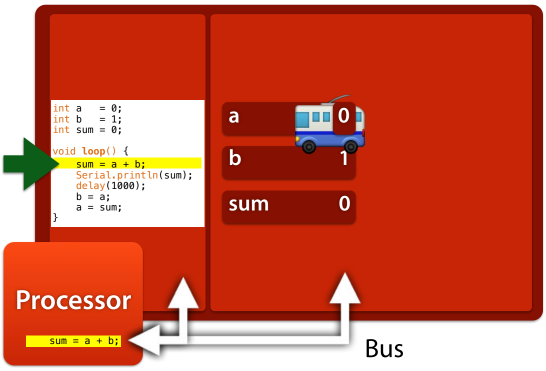
…but it also reads and writes data into memory

Bus



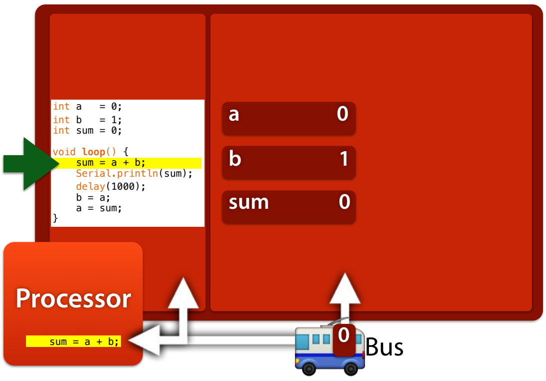
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Bus



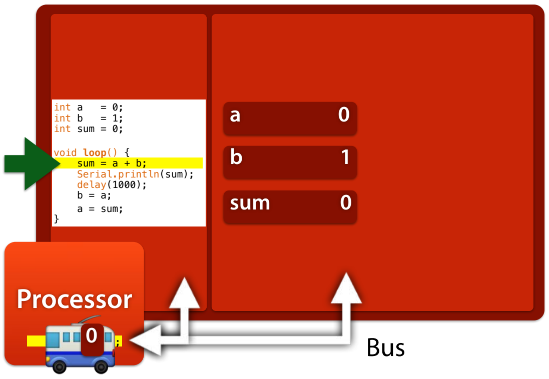
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Bus



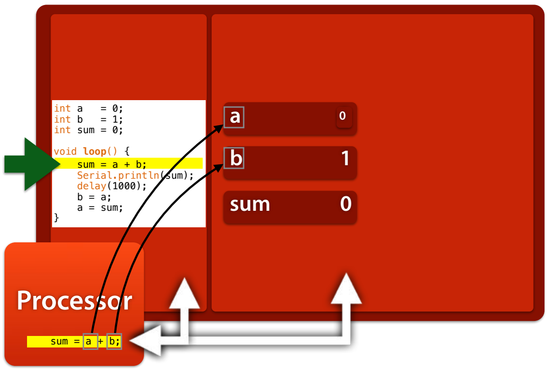
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Bus



```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

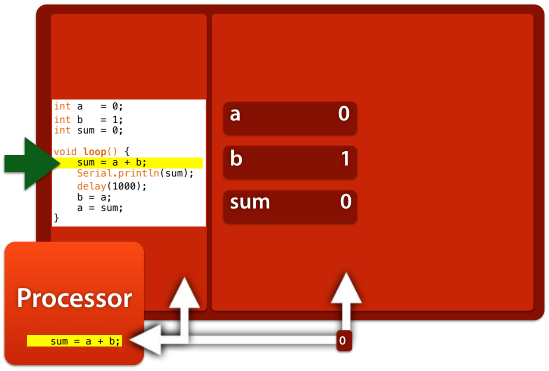
Addressing Data



```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

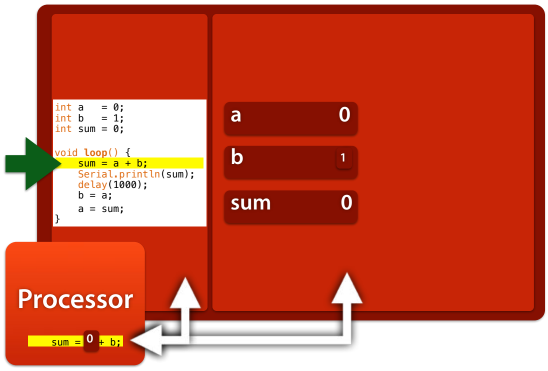
For this, the bus must know where to find the data

Read Data



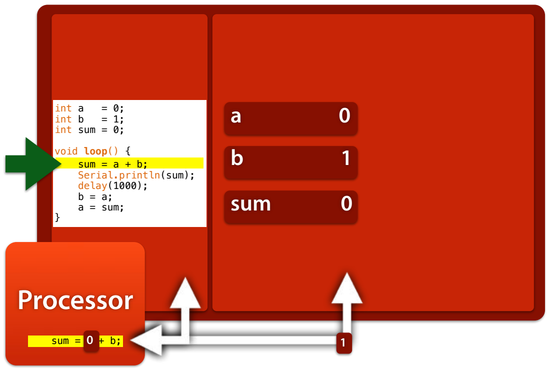
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Read Data



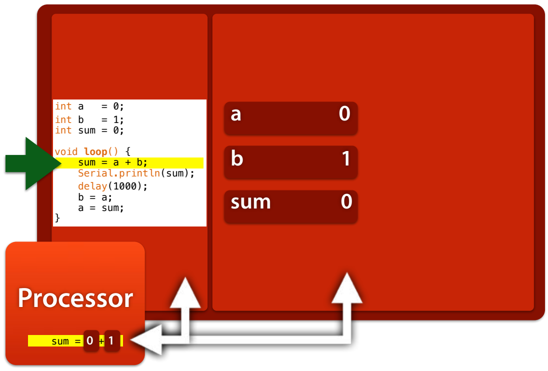
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Read Data



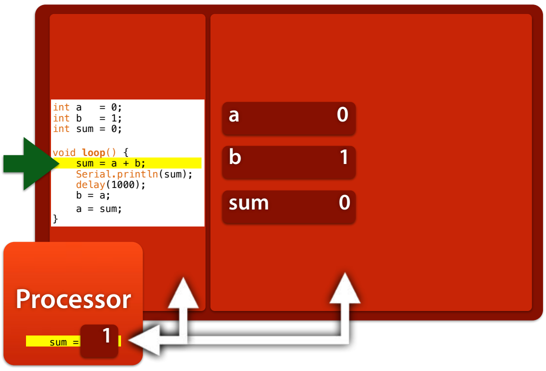
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Read Data



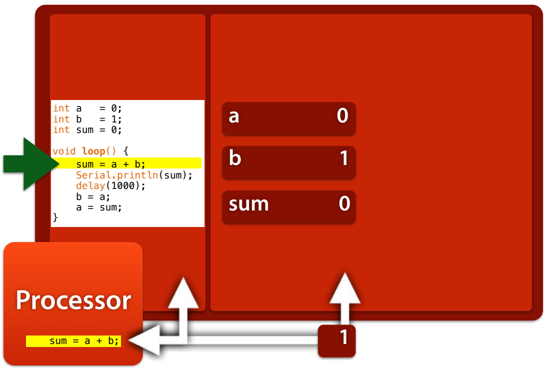
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Compute Data



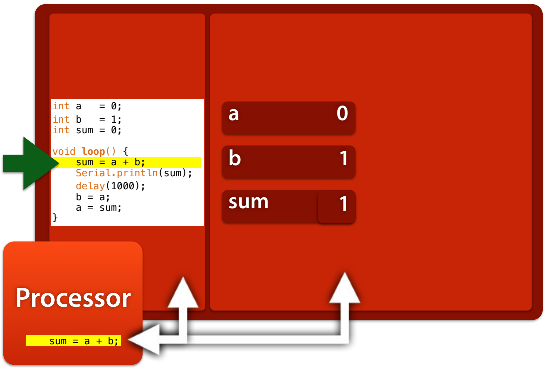
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Write Data



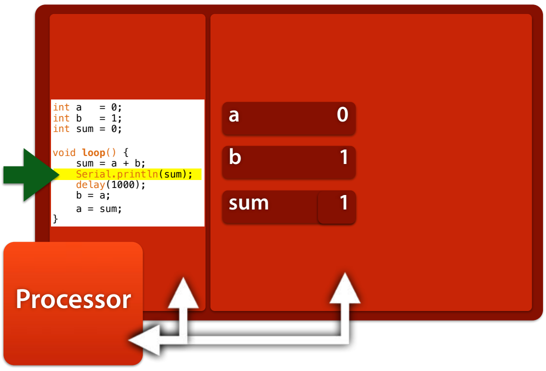
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Write Data



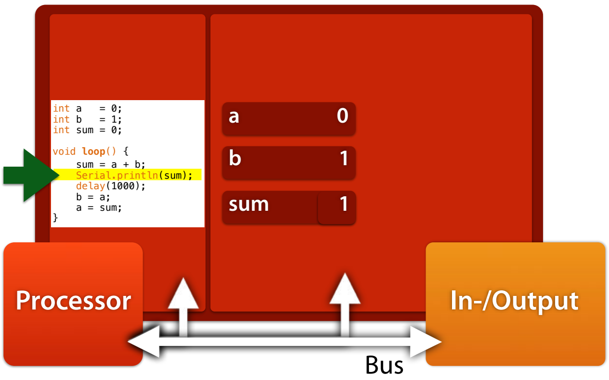
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Next Instruction



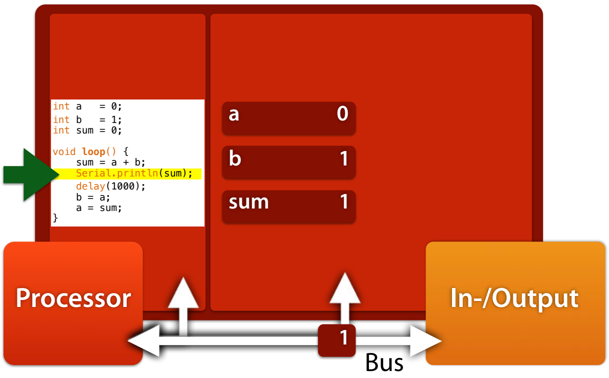
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

In-/Output



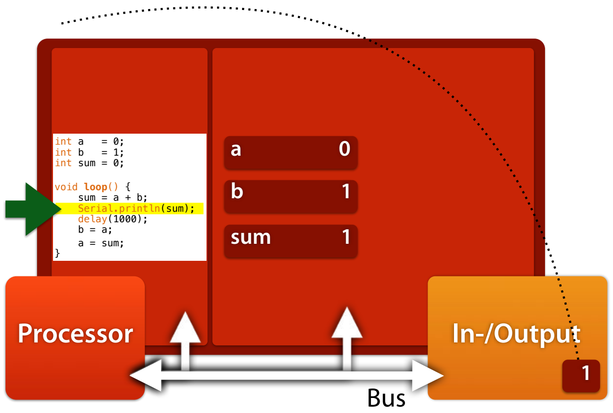
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

In-/Output



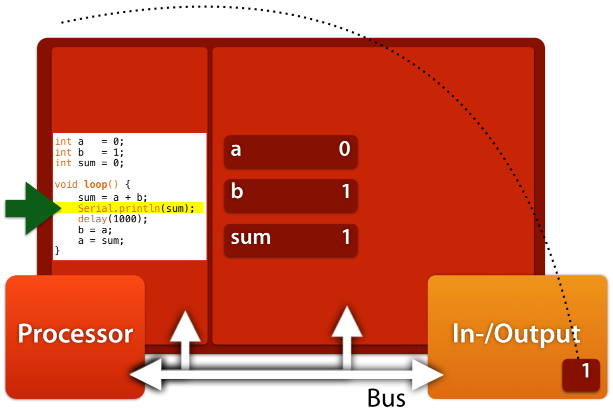
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

In-/Output



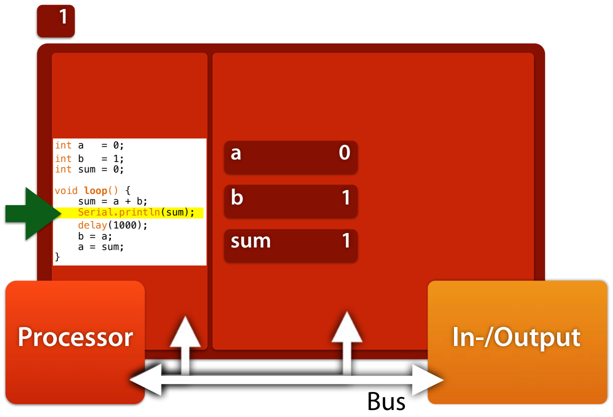
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

In-/Output



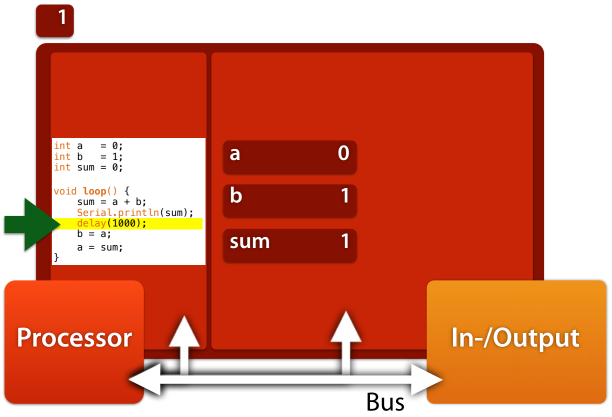
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

In-/Output



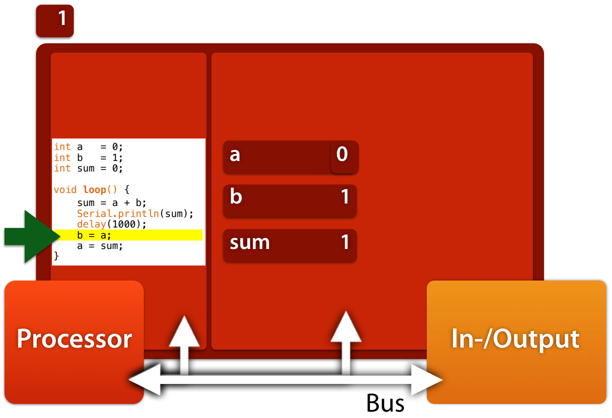
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Pause



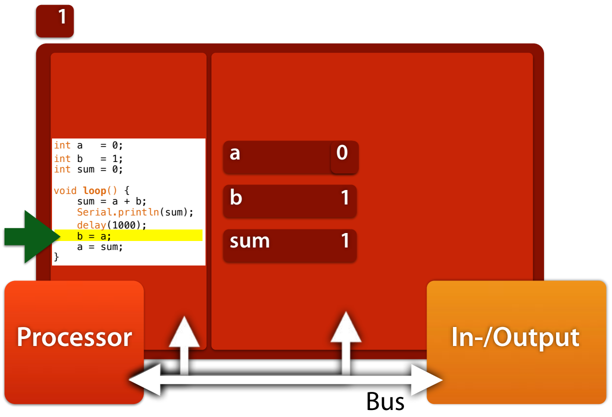
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Assignment

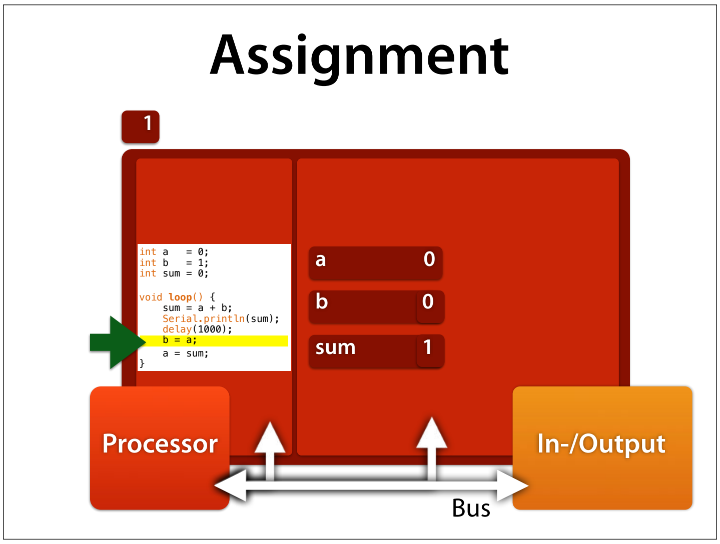


```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Assignment

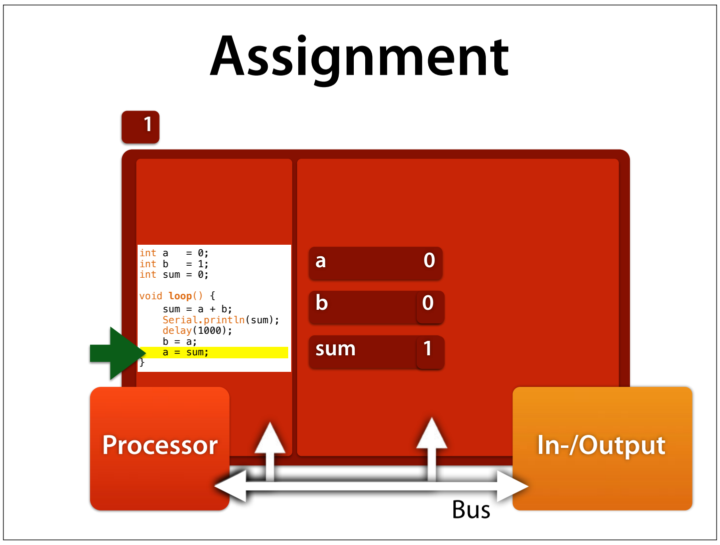


```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



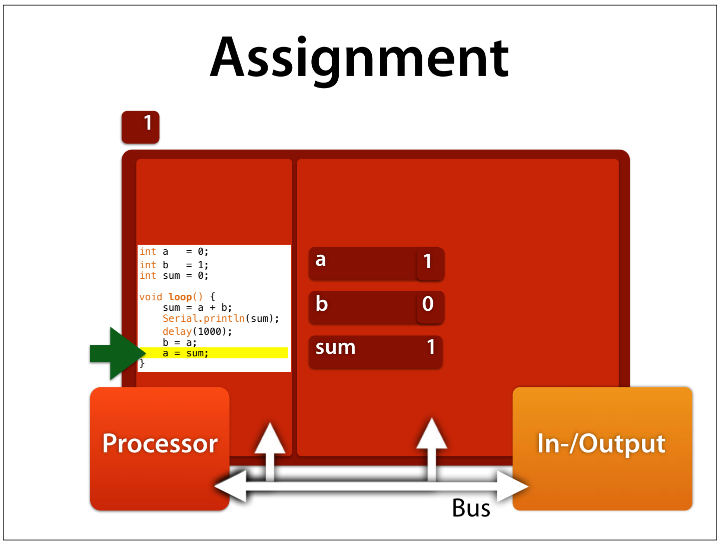
Assignment

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



Assignment

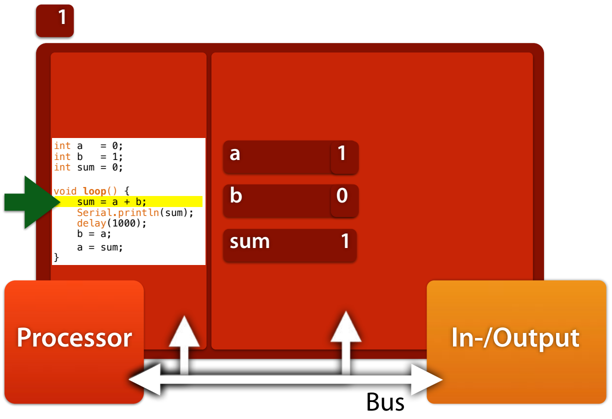
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



Assignment

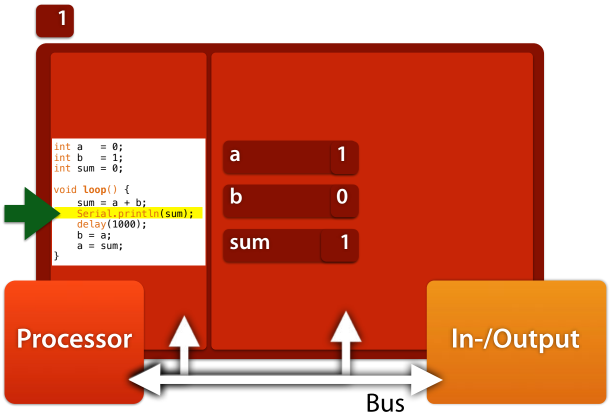
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Assignment



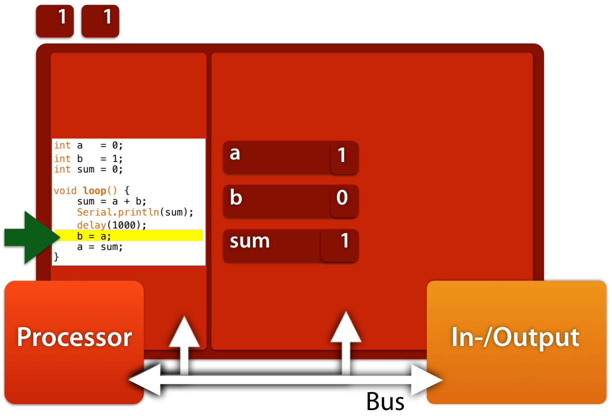
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Assignment



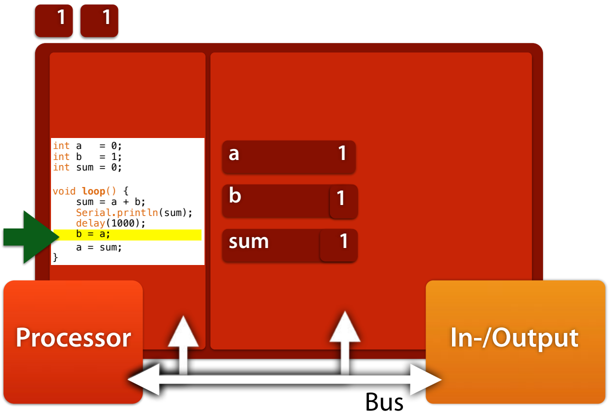
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Assignment



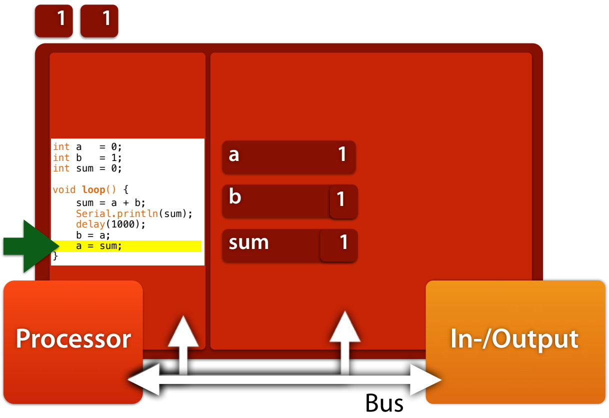
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Assignment



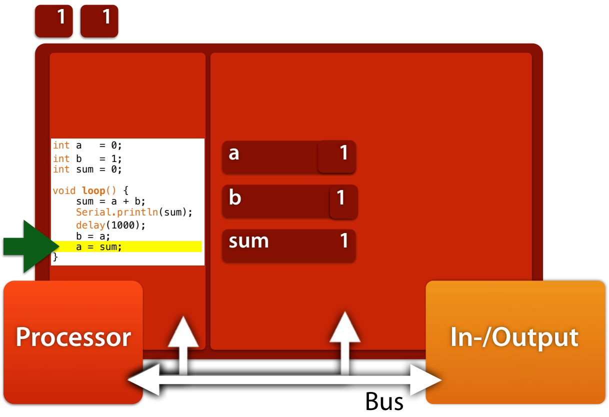
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Assignment



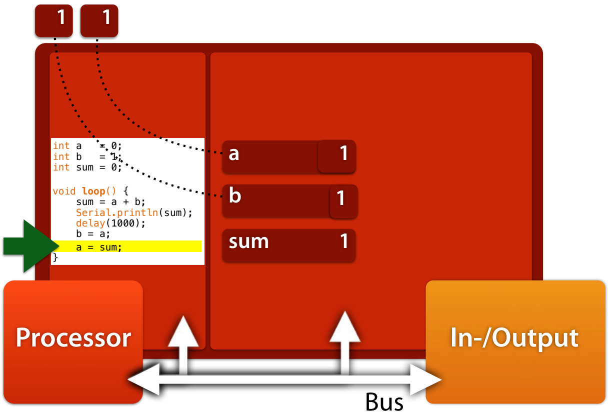
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Assignment



```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

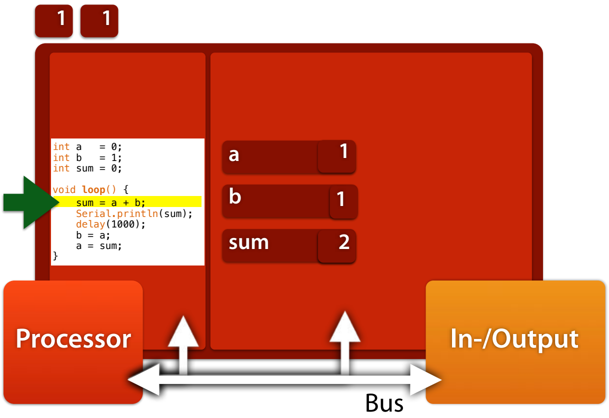
Assignment



```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

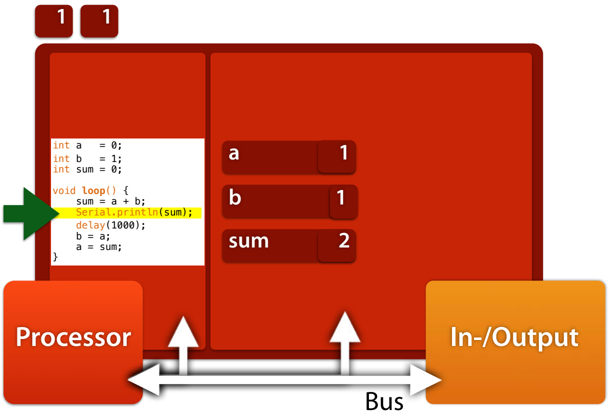
Observe: a is always the last, b the next-to-last element of the list

Next Iteration



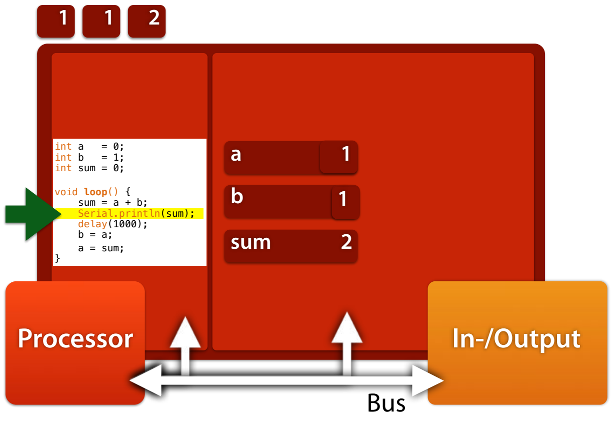
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Next Iteration



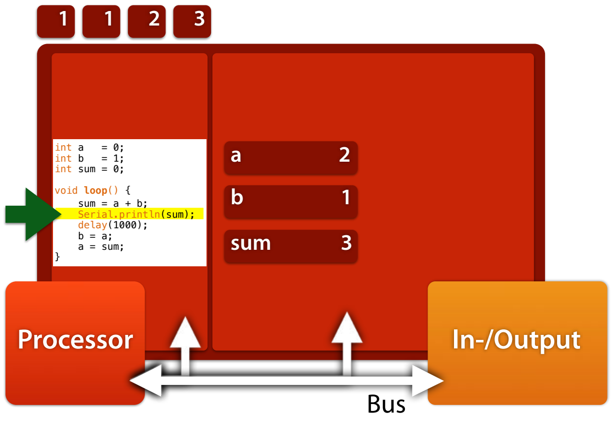
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Next Iteration



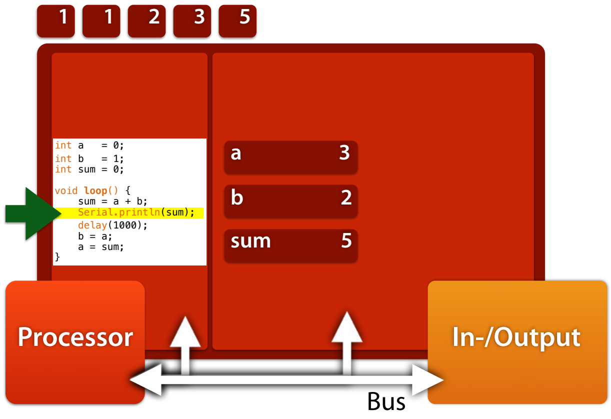
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Next Iteration



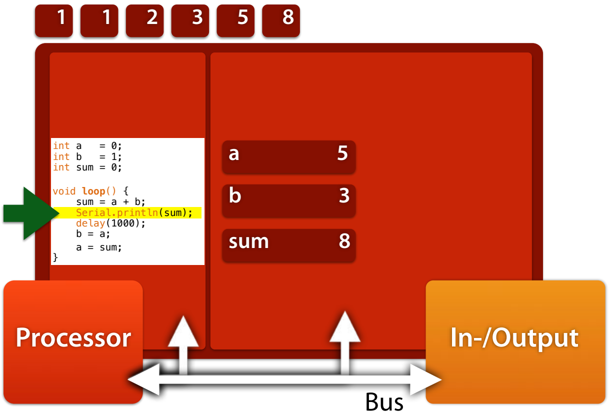
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Next Iteration



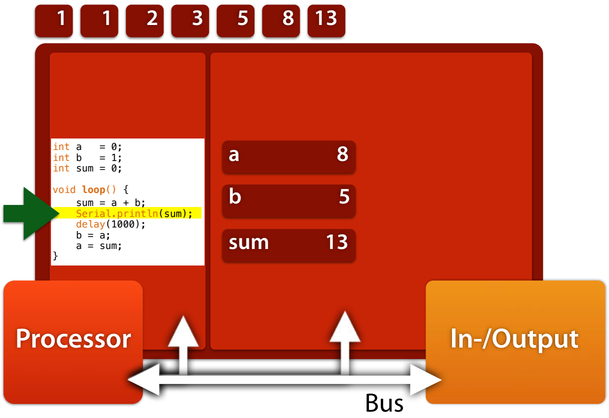
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Next Iteration



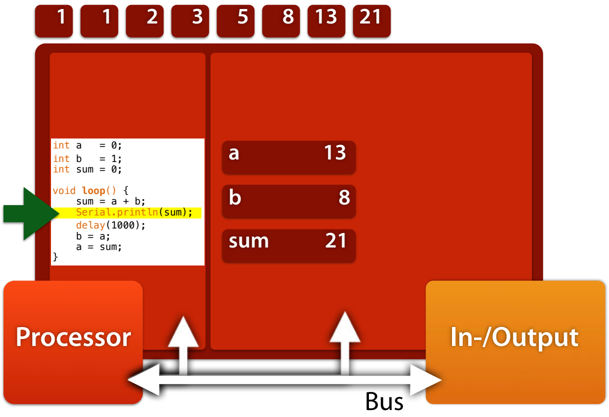
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Next Iteration



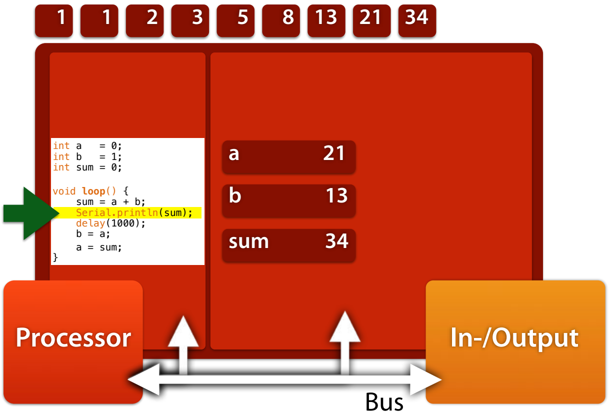
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Next Iteration



```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Next Iteration



```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

Fibonacci Sequence

1 1 2 3 5 8 13 21 34

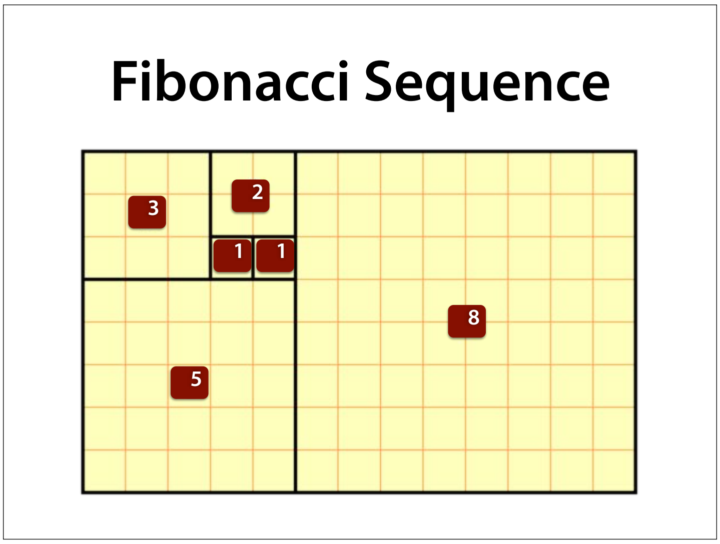
Fibonacci Sequence

1 1 2 3 5 8 13 21 34

Outside of India, the Fibonacci sequence first appears in the book Liber Abaci (1202) by Leonardo of Pisa, known as Fibonacci.[5] Fibonacci considers the growth of an idealized (biologically unrealistic) rabbit population, assuming that: a newly born pair of rabbits, one male, one female, are put in a field; rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits; rabbits never die and a mating pair always produces one new pair (one male, one female) every month from the second month on. The puzzle that Fibonacci posed was: how many pairs will there be in one year?

- At the end of the first month, they mate, but there is still only 1 pair.
- At the end of the second month the female produces a new pair, so now there are 2 pairs of rabbits in the field.
- At the end of the third month, the original female produces a second pair, making 3 pairs in all in the field.
- At the end of the fourth month, the original female has produced yet another new pair, the female born two months ago produces her first pair also, making 5 pairs.

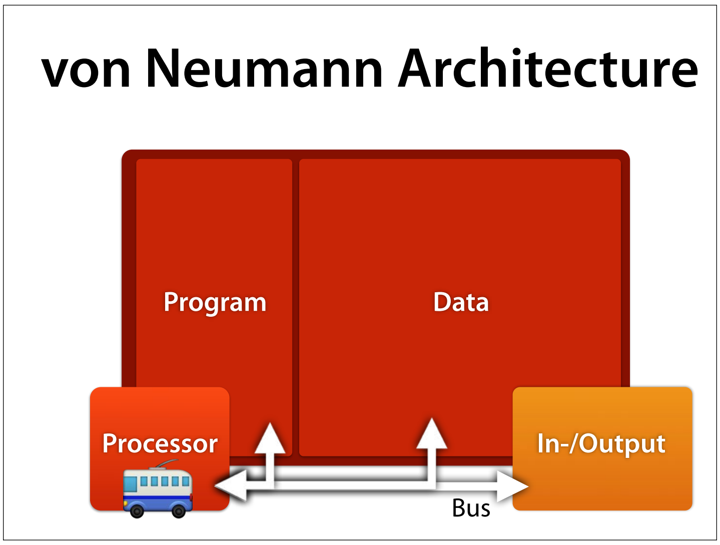
At the end of the nth month, the number of pairs of rabbits is equal to the number of new pairs (which is the number of pairs in month n − 2) plus the number of pairs alive last



Fibonacci Sequence

A tiling with squares whose side lengths are successive Fibonacci numbers (Wikipedia)

Such arrangements involving consecutive Fibonacci numbers appear in a wide variety of plants. (Wikipedia)

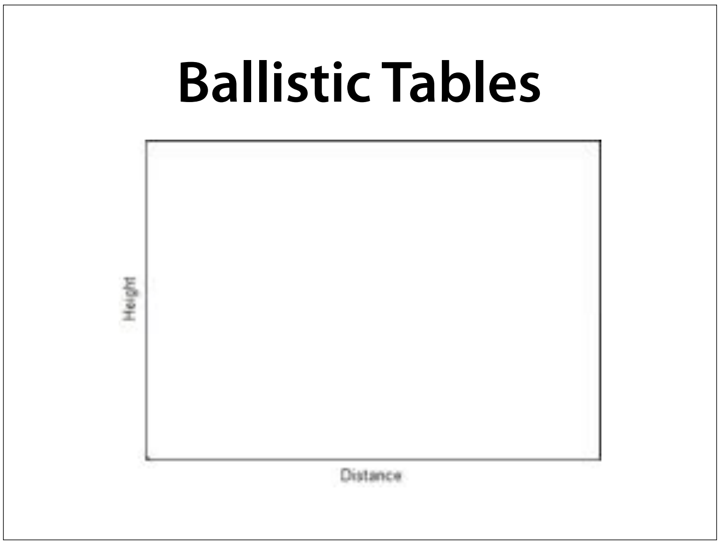


von Neumann Architecture

John von Neumann

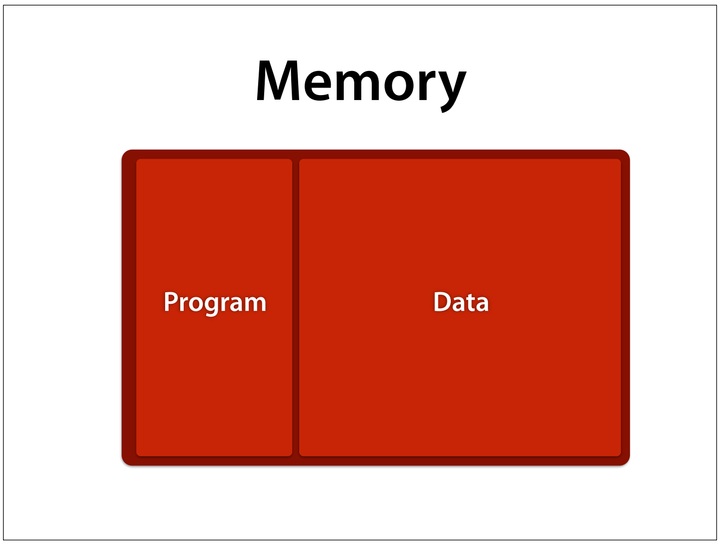
1903–1957

John von Neumann (Hungarian: Neumann János, /vɒn 'nɔɪmən/; December 28, 1903 – February 8, 1957) was a Hungarian-American pure and applied mathematician, physicist, inventor, polymath, and polyglot. He made major contributions to a number of fields,[3] including mathematics (foundations of mathematics, functional analysis, ergodic theory, geometry, topology, and numerical analysis), physics (quantum mechanics, hydrodynamics, fluid dynamics and quantum statistical mechanics), economics (game theory), computing (Von Neumann architecture, linear programming, self-replicating machines, stochastic computing), and statistics.[4] He was a pioneer of the application of operator theory to quantum mechanics, in the development of functional analysis, a principal member of the Manhattan Project and the Institute for Advanced Study in Princeton (as one of the few originally appointed), and a key figure in the development of game theory[3][5] and the concepts of cellular automata,[3] the universal constructor, and the digital computer. (Wikipedia)



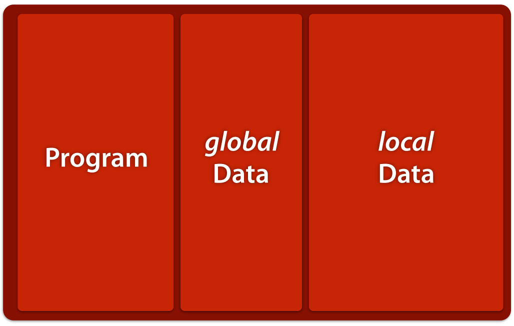
Ballistic Tables

What were these first computers being used for? Ballistic calculations.

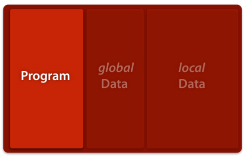


Memory

Memory

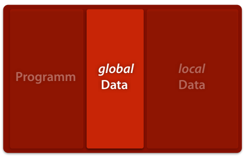


The Program



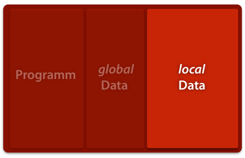
- Resides in memory (just like data)
- Cannot access or change itself
- The *operating system* loads and manages the programs in memory

Global Data



- Contains Variables with *global* visibility (= *defined outside of* functions)
- Accessible from all functions

Local Data



- Contains variables with *local visibility* (= *defined inside functions*)
- Local variables and parameters exist only during the execution of a function
- They are contained by the function frame

Global Variables

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

global variables

– produces 1 2 3 5 8 13 … aus

Local Variables

```
int a   = 0;
int b   = 1;


void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

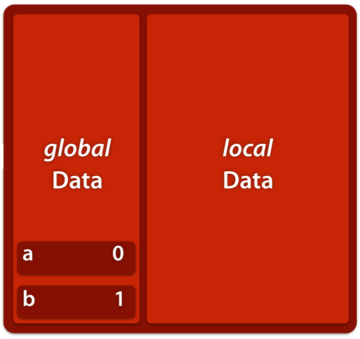
global variables

local variable

– gibt 1 2 3 5 8 13 … aus

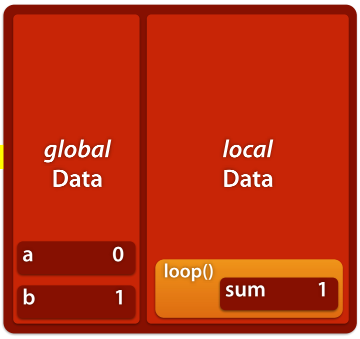
Local Variables

```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



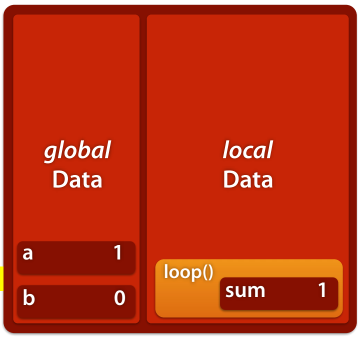
Local Variables

```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



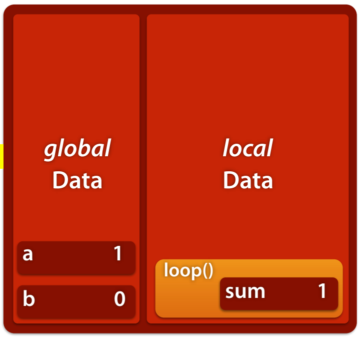
Local Variables

```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```



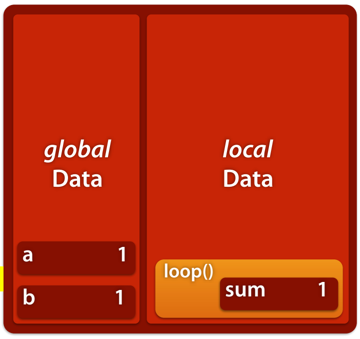
Local Variables

```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

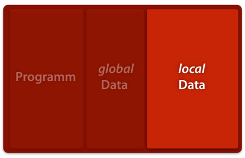


Local Variables

```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
```



Function Stack



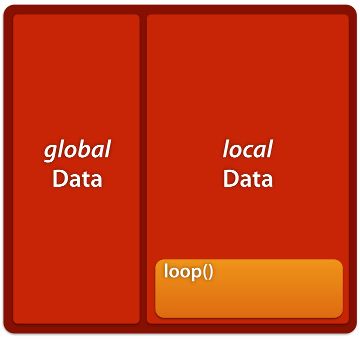
- Function frames are organised as a *stack*
- The *topmost* frame is *active*
- Upon returning from the topmost function, the underlying (calling) function is continued from the call site on

Morse code

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```

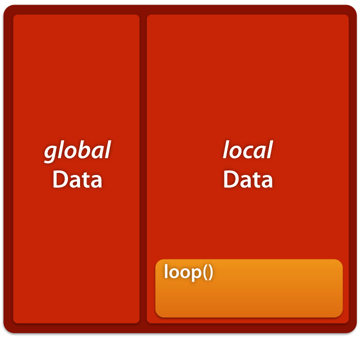
Function Stack

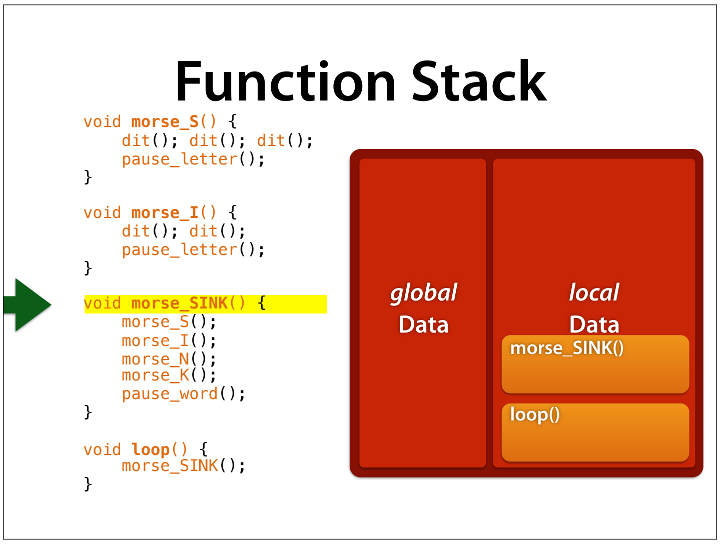
```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```



Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```





Function Stack

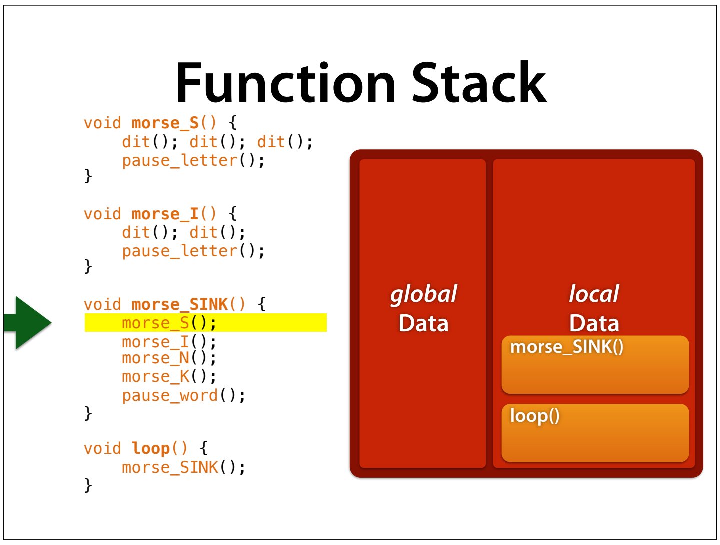
```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```

global Data

local Data

morse_SINK()

loop()



Function Stack

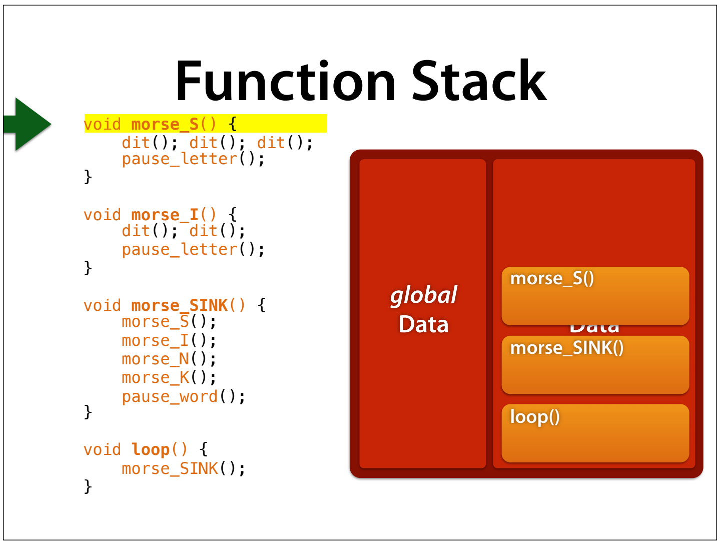
```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```

global Data

local Data

morse_SINK()

loop()



Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```

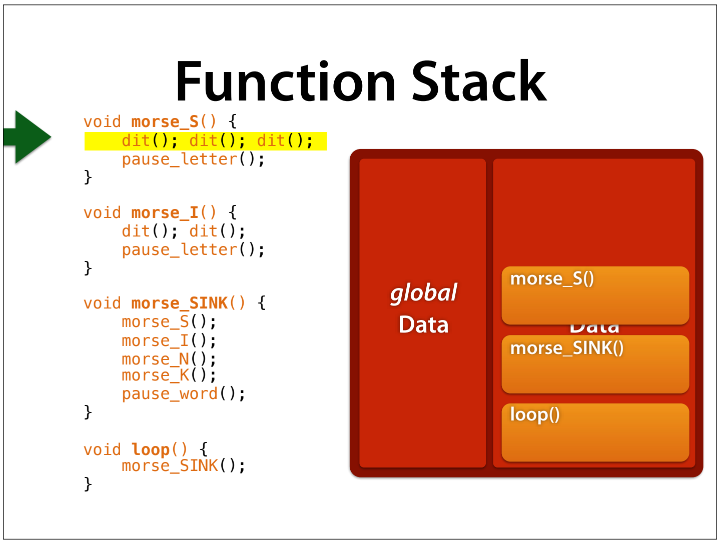
global Data

morse_S()

Data

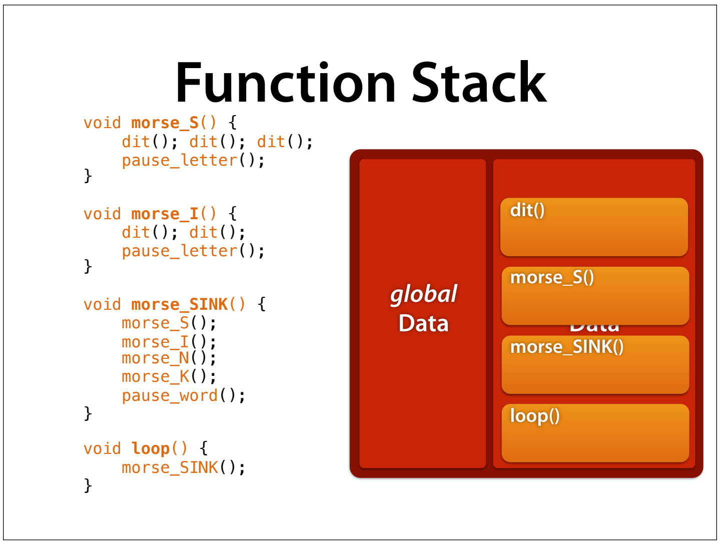
morse_SINK()

loop()



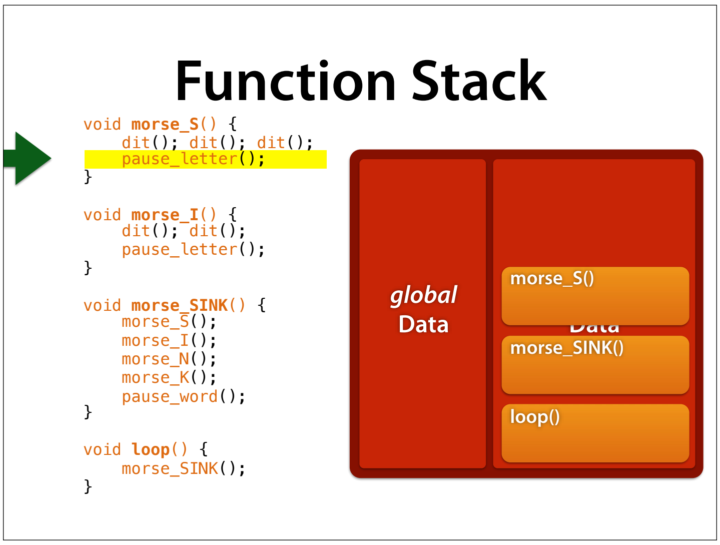
Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```



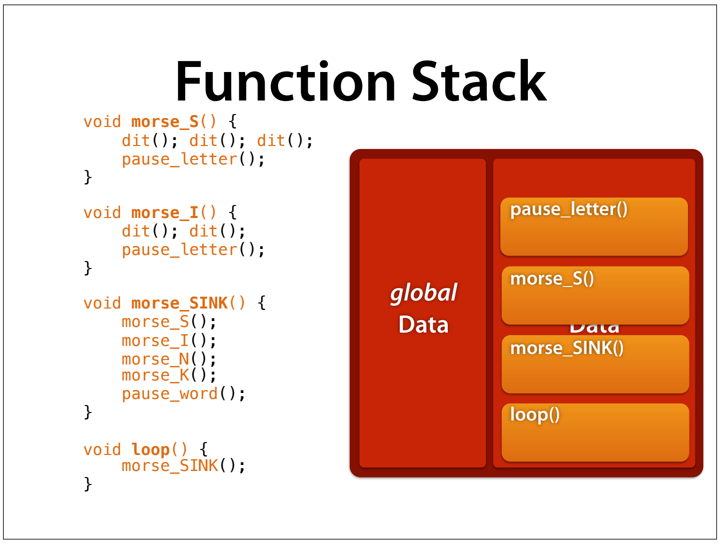
Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```



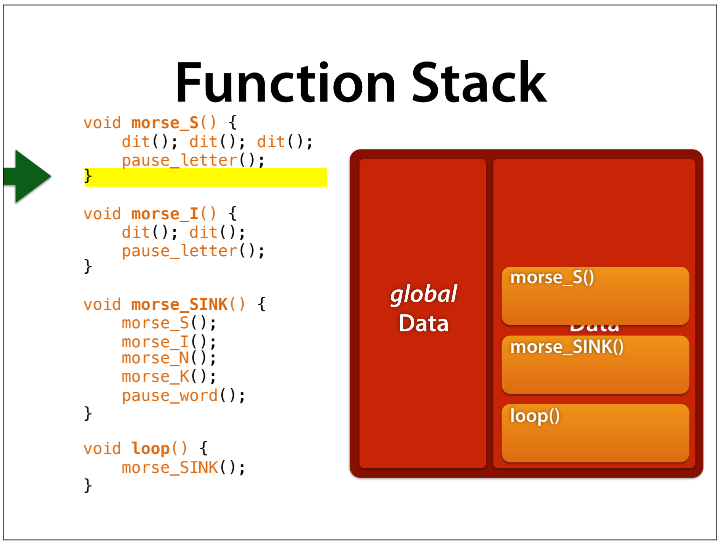
Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```



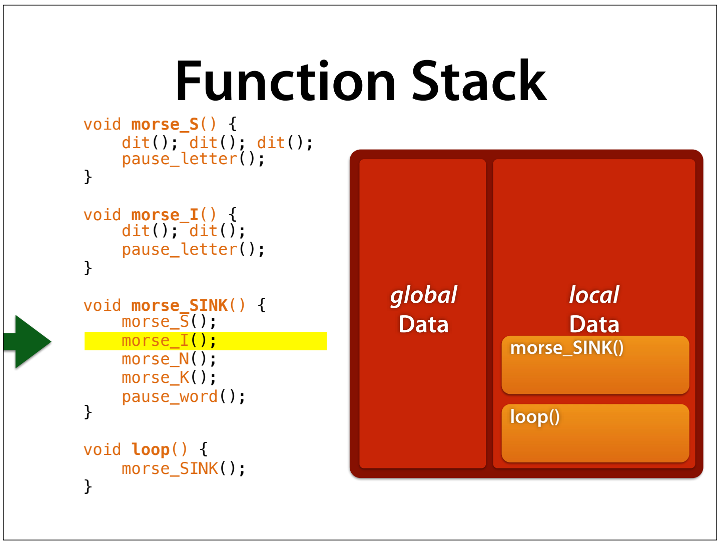
Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```



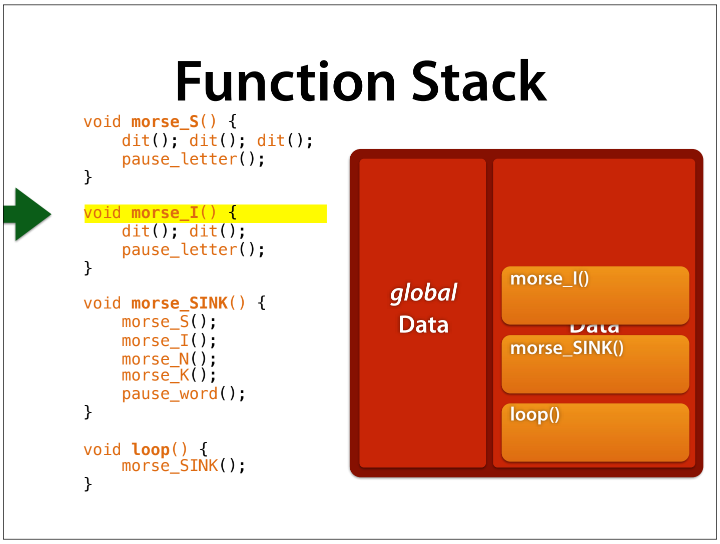
Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```



Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```



Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```

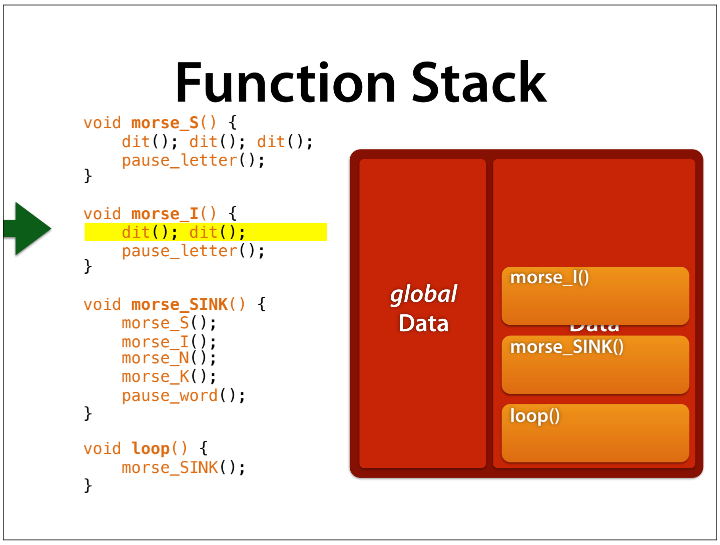
global Data

Data

morse_I()

morse_SINK()

loop()



Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```

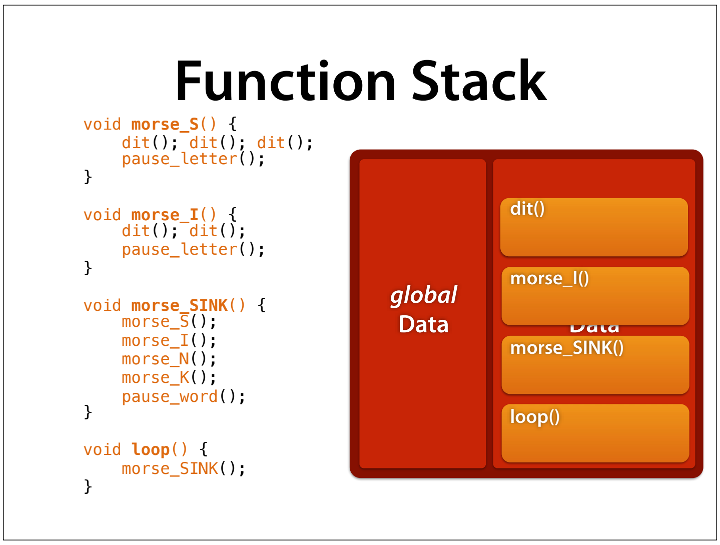
global Data

Data

morse_I()

morse_SINK()

loop()



Function Stack

```
void morse_S() {
    dit(); dit(); dit();
    pause_letter();
}

void morse_I() {
    dit(); dit();
    pause_letter();
}

void morse_SINK() {
    morse_S();
    morse_I();
    morse_N();
    morse_K();
    pause_word();
}

void loop() {
    morse_SINK();
}
```

global Data

Data

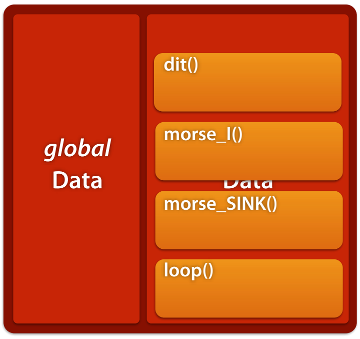
dit()

morse_I()

morse_SINK()

loop()

Function Stack

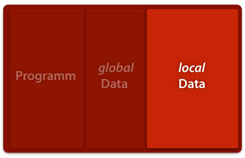


- Call: put on top *(push)*
- Return: remove *(pop)*

Function stack – like a tray stack in Mensa

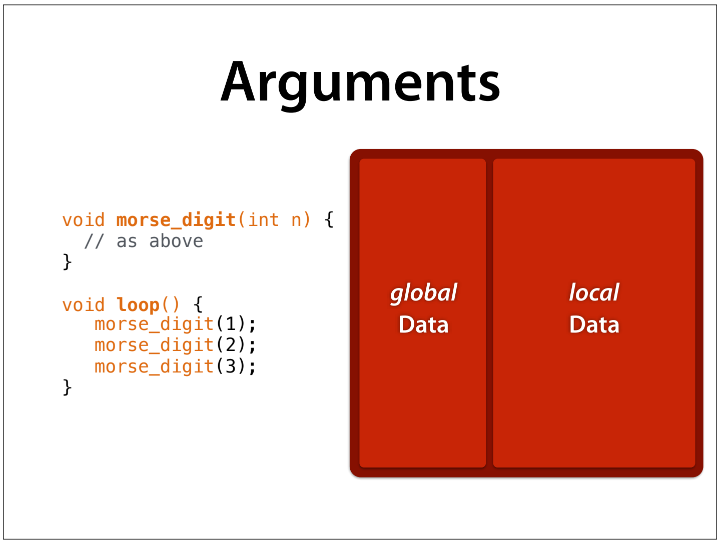
Source: http://www.blanco-professional.com/de/catering/produkte/blanco_spender/tablettspender.cfm

Arguments



- When a function *f is called,* its arguments are deposited like local variables – inside the function frame of *f*

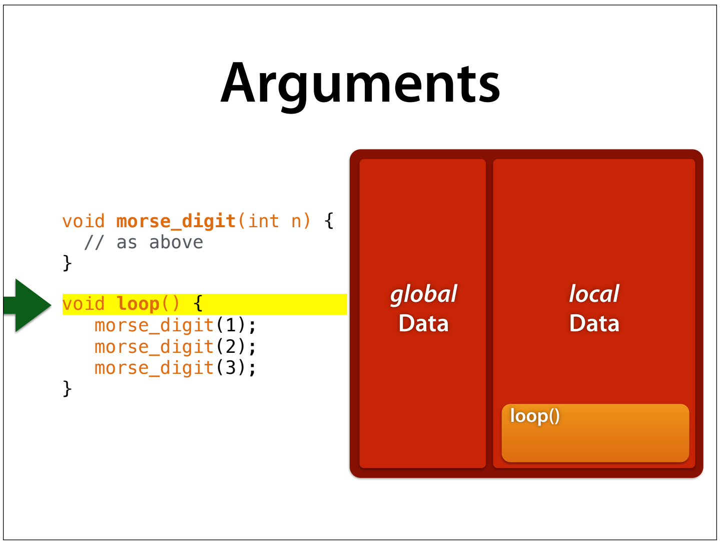
```
// send n in morse code
void morse_digit(int n) {
  if (n == 0) {
    dah(); dah(); dah(); dah(); dah();
  }
  if (n == 1) {
    dit(); dah(); dah(); dah(); dah();
  }
  // etc. for 2–8
  if (n == 9) {
    dah(); dah(); dah(); dah(); dit();
  }
  pause_letter();
}
```



Arguments

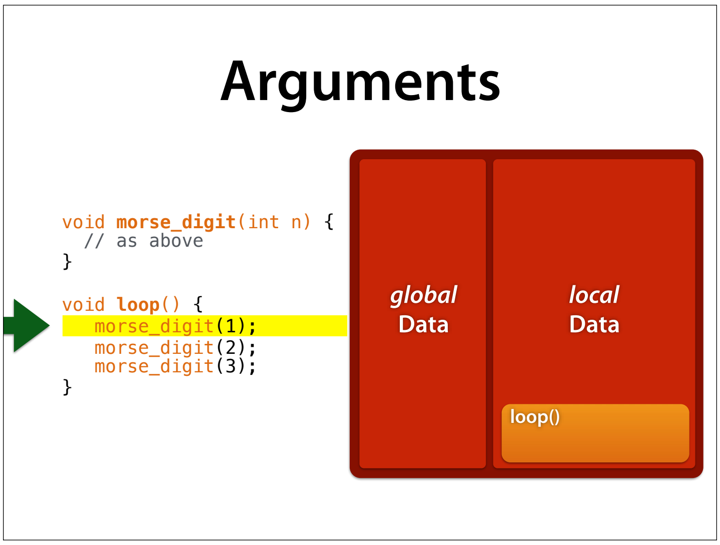
```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

Initially, there's no local data – these are being produced during the execution



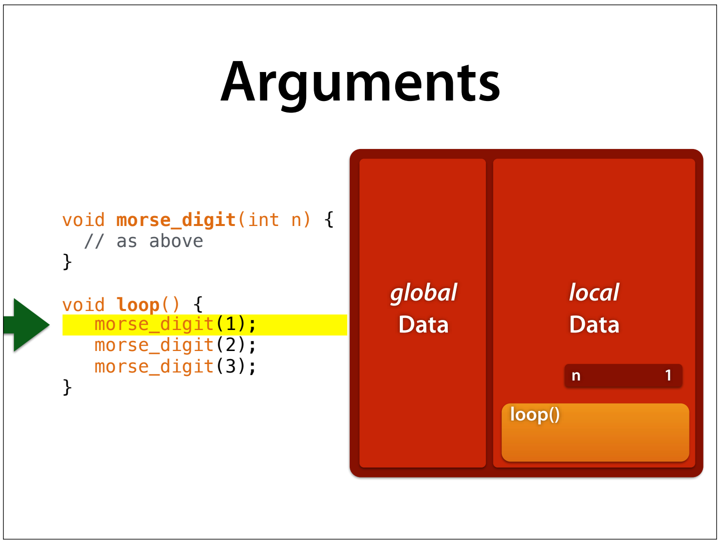
Arguments

```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```



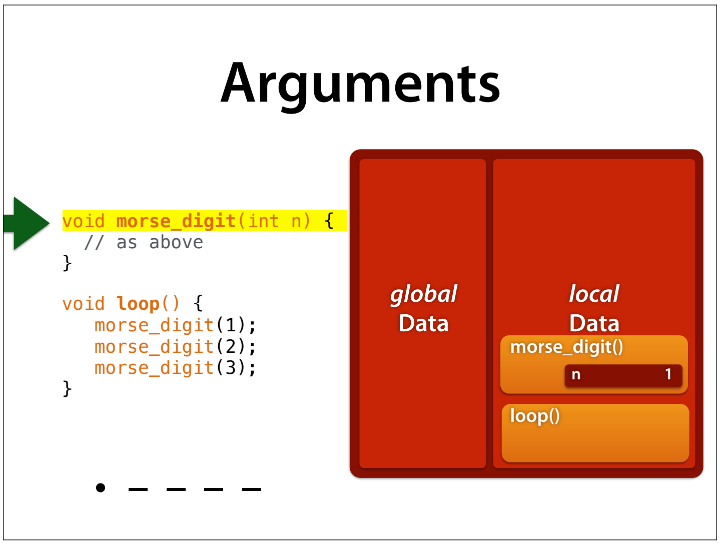
Arguments

```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```



Arguments

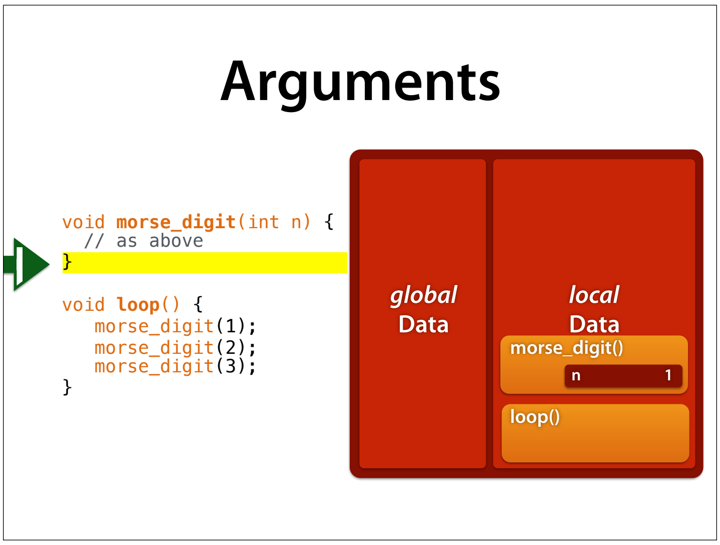
```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```



Arguments

```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

• – – – –

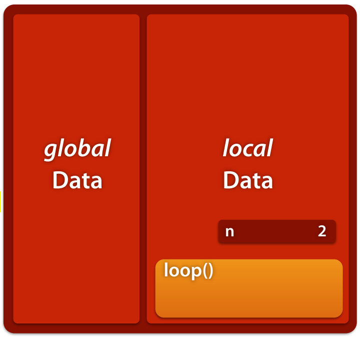


Arguments

```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

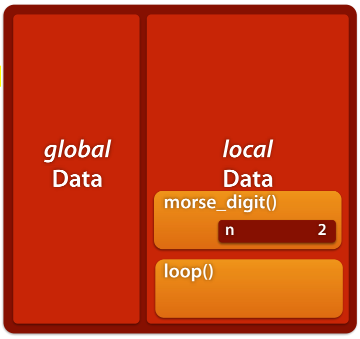
Arguments

```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```



Arguments

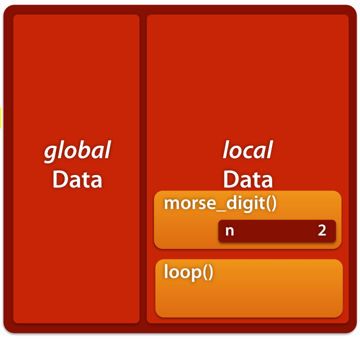
```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

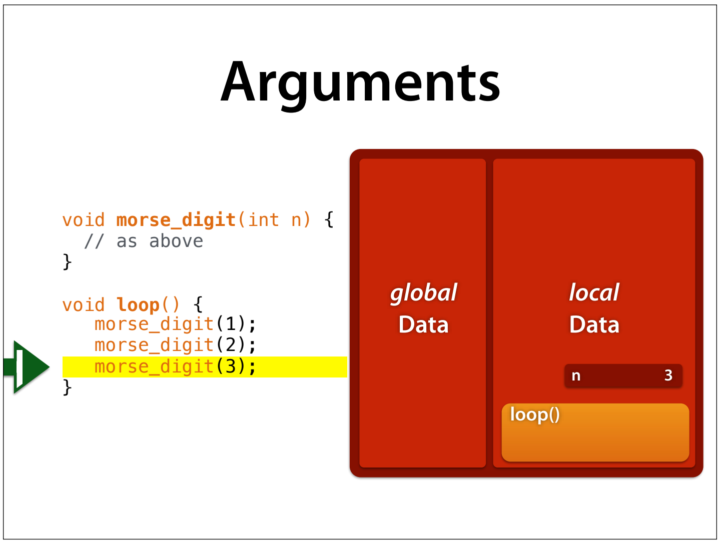


• • — — —

Arguments

```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```





Arguments

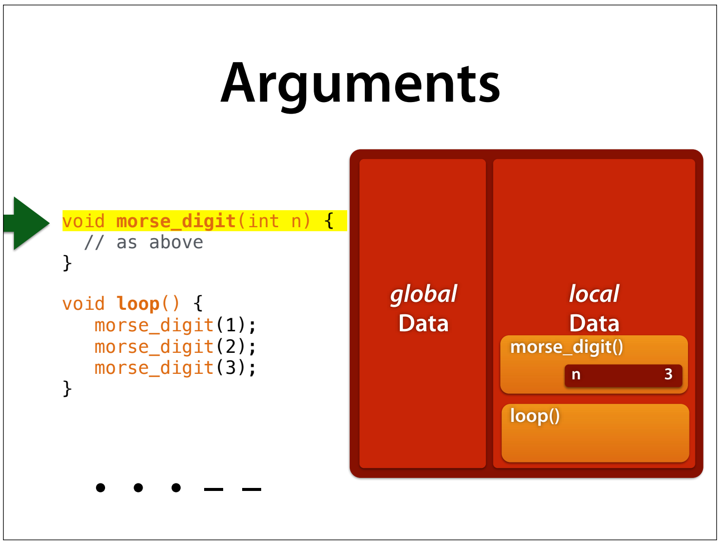
```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

global Data

local Data

n 3

loop()



Arguments

```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

• • • – –

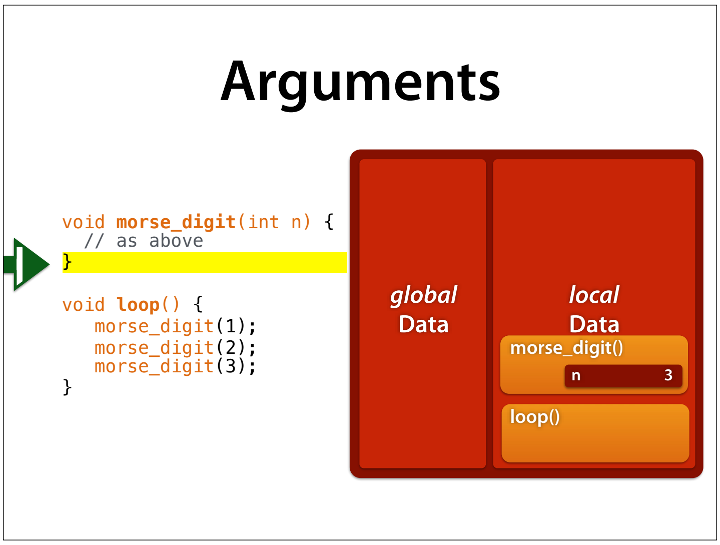
global Data

local Data

morse_digit()

n 3

loop()



Arguments

```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```

global Data

local Data

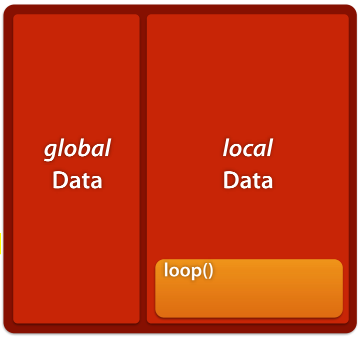
morse_digit()

n 3

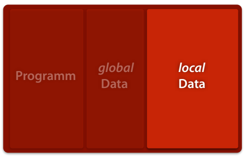
loop()

Arguments

```
void morse_digit(int n) {
  // as above
}

void loop() {
   morse_digit(1);
   morse_digit(2);
   morse_digit(3);
}
```



Active Function

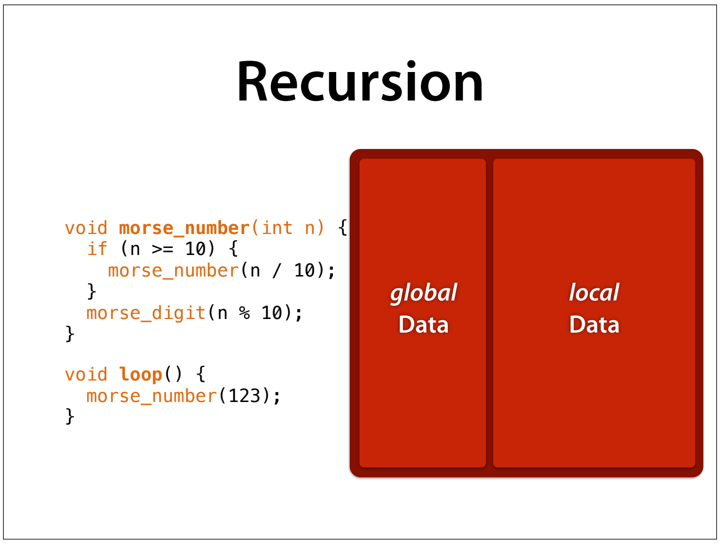


- The program can only ever access the active (= topmost) function frame
- A caller can be certain that his local variables remain unchanged

From Digits to Numbers

morse_number() prints a number recursively:

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

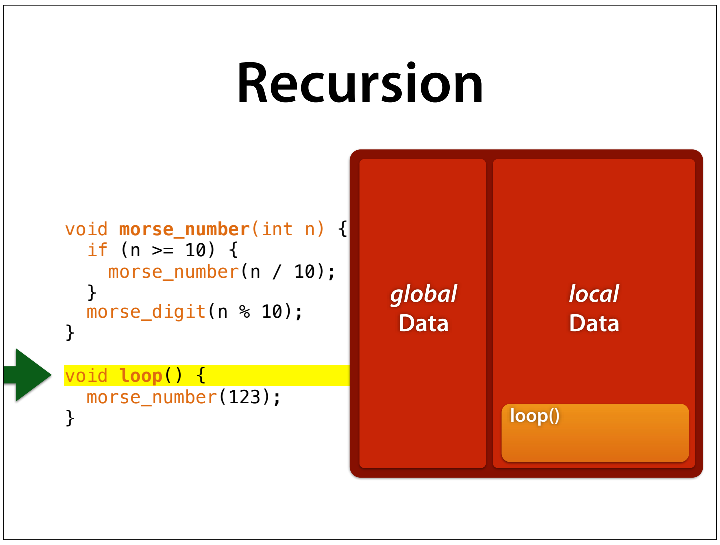


Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

global Data

local Data



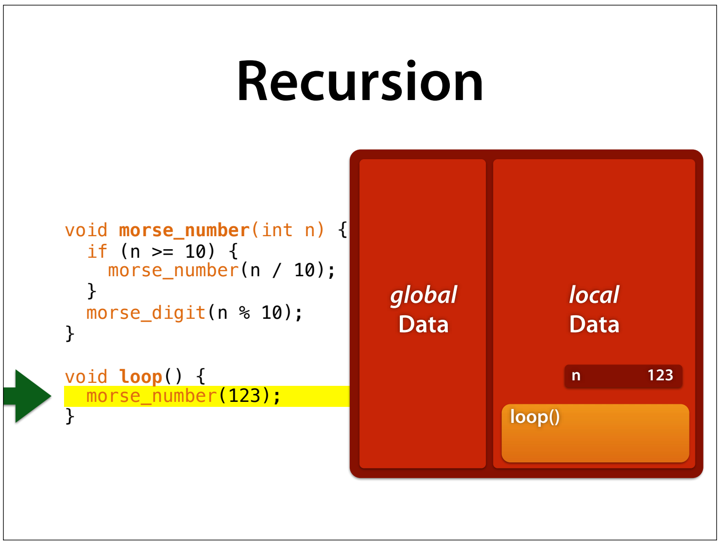
Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

global Data

local Data

loop()



Recursion

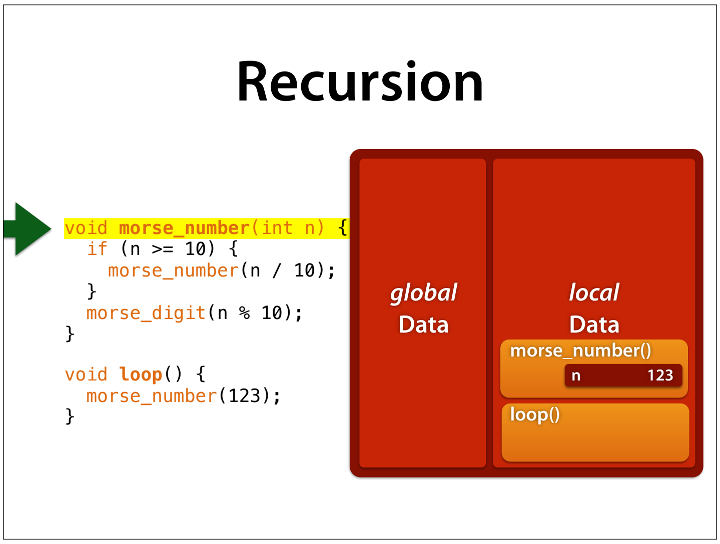
```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

global Data

local Data

n 123

loop()



Recursion

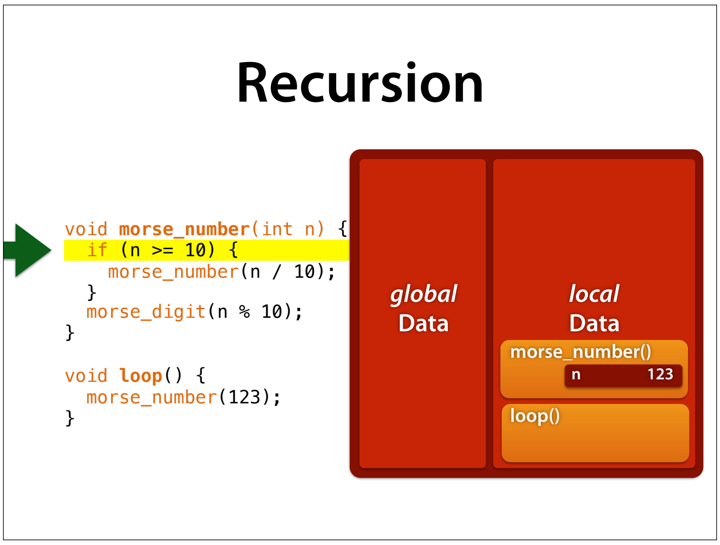
```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

global
Data

local
Data

morse_number()
n 123

loop()



Recursion

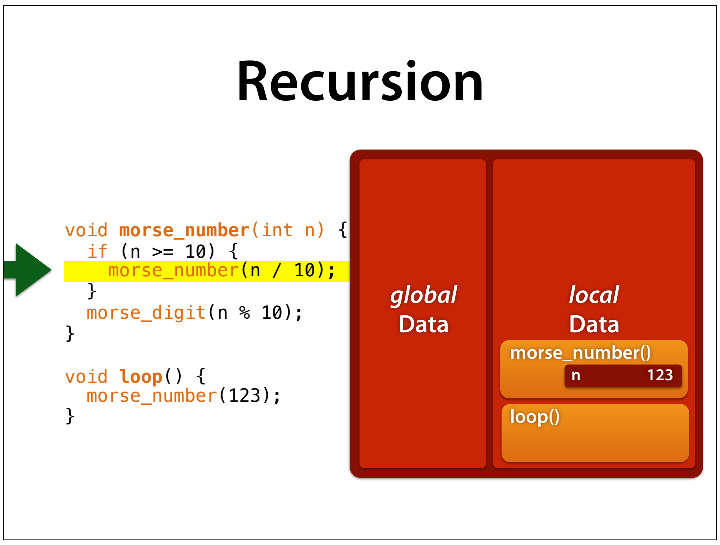
```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

global
Data

local
Data

morse_number()
n 123

loop()



Recursion

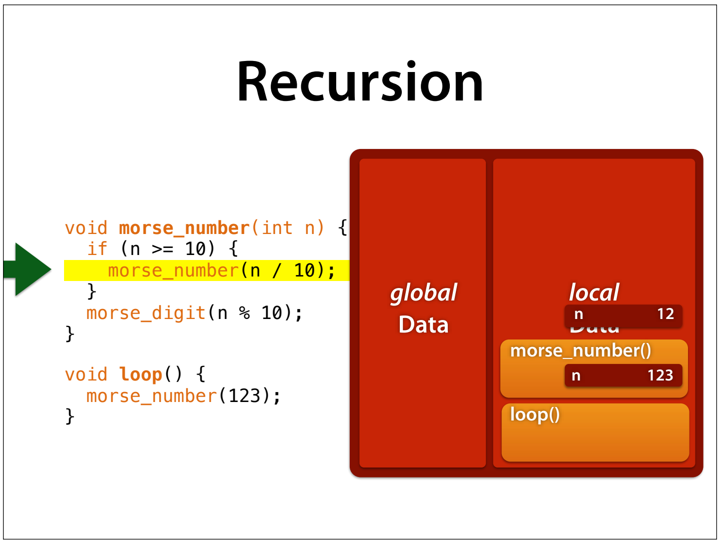
```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

global
Data

local
Data

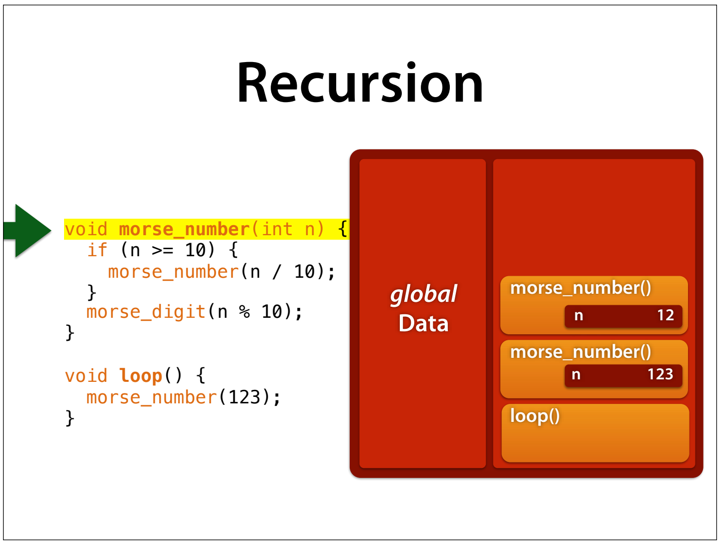
morse_number()
n 123

loop()



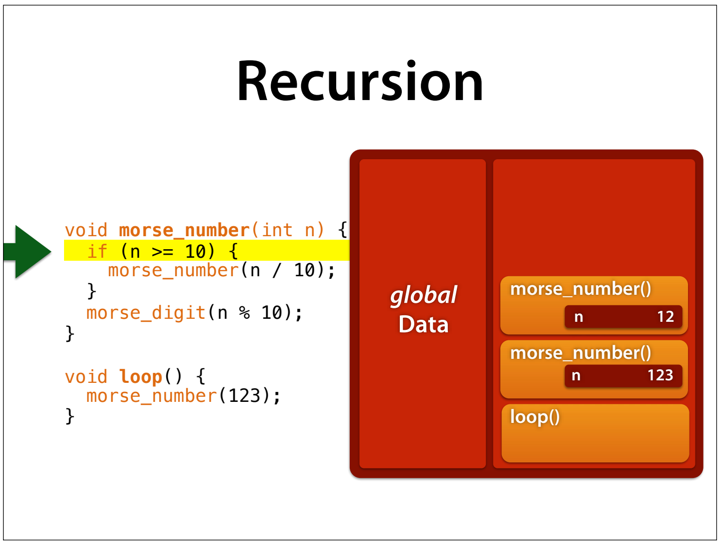
Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```



Recursion

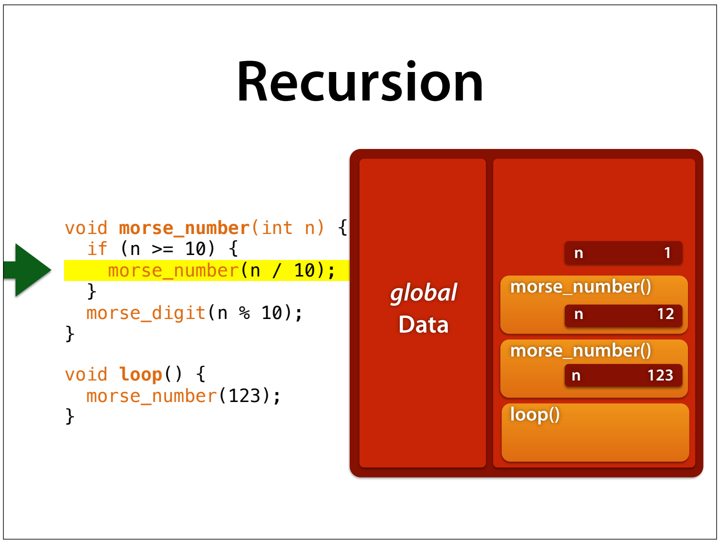
```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```



Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

Hier wird auf das **oberste** n der aktiven Funktion zugegriffen – die “unteren”, inaktiven sind nicht zugänglich



Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

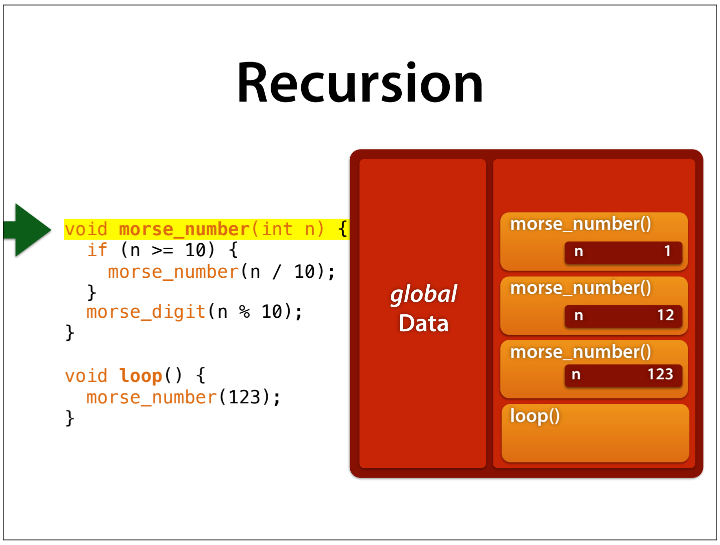
global Data

n 1

morse_number()
n 12

morse_number()
n 123

loop()



Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

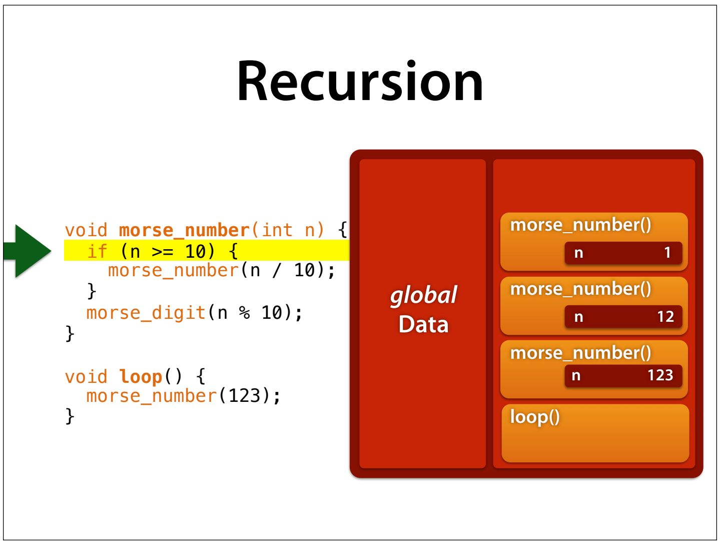
global Data

morse_number()
n 1

morse_number()
n 12

morse_number()
n 123

loop()



Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

global Data

morse_number()
n 1

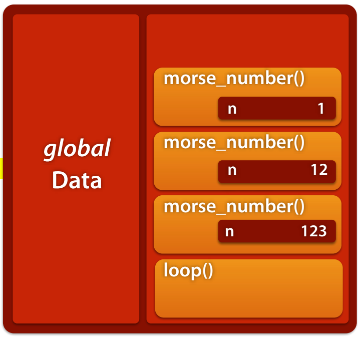
morse_number()
n 12

morse_number()
n 123

loop()

Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```



global Data

morse_number()
n 1

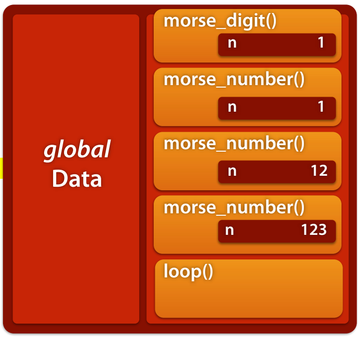
morse_number()
n 12

morse_number()
n 123

loop()

Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```



global Data

morse_digit()
n 1

morse_number()
n 1

morse_number()
n 12

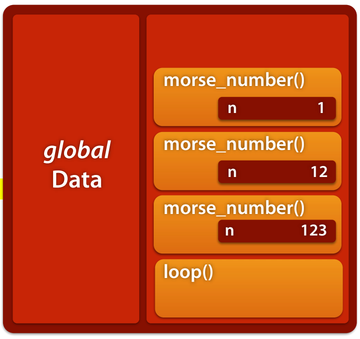
morse_number()
n 123

loop()

• — — — —

Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```



global Data

morse_number()
n 1

morse_number()
n 12

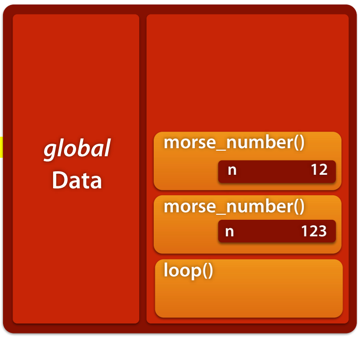
morse_number()
n 123

loop()

• — — — —

Recursion

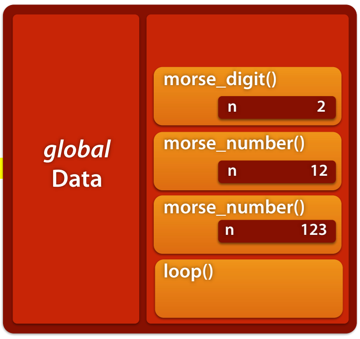
```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```



• — — — —   • • — — —

Recursion

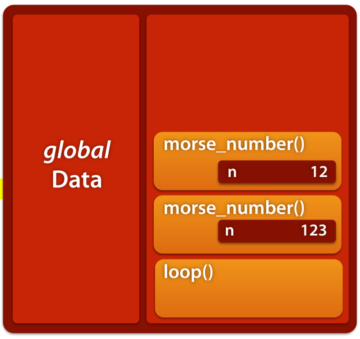
```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```



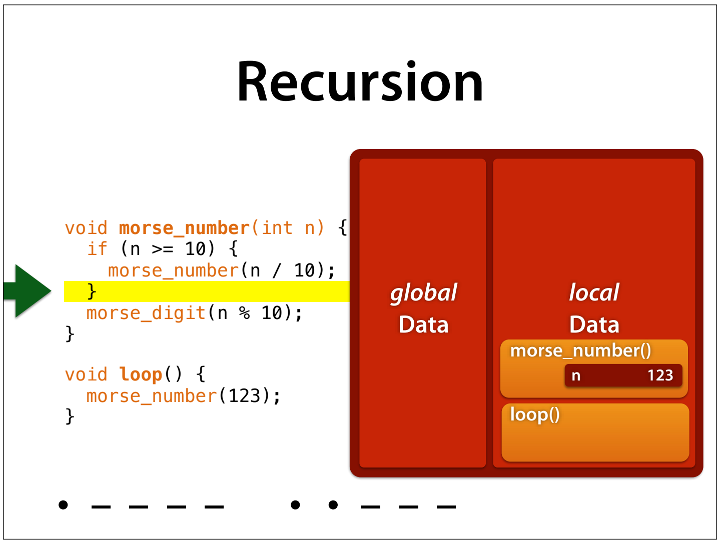
• — — — —   • • — — —

Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```



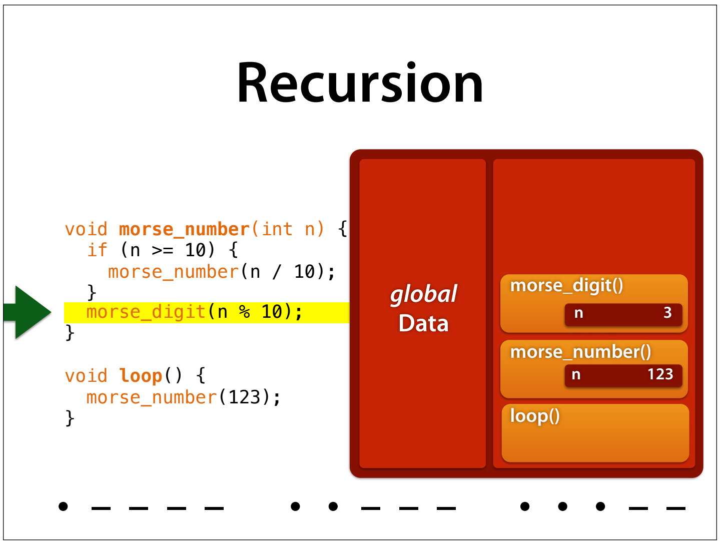
• — — — —   • • — — —



Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

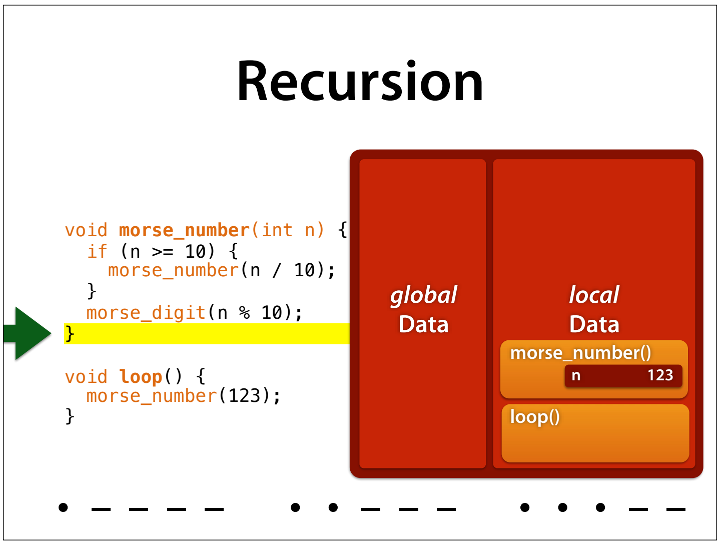
• — — — —   • • — — —



Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

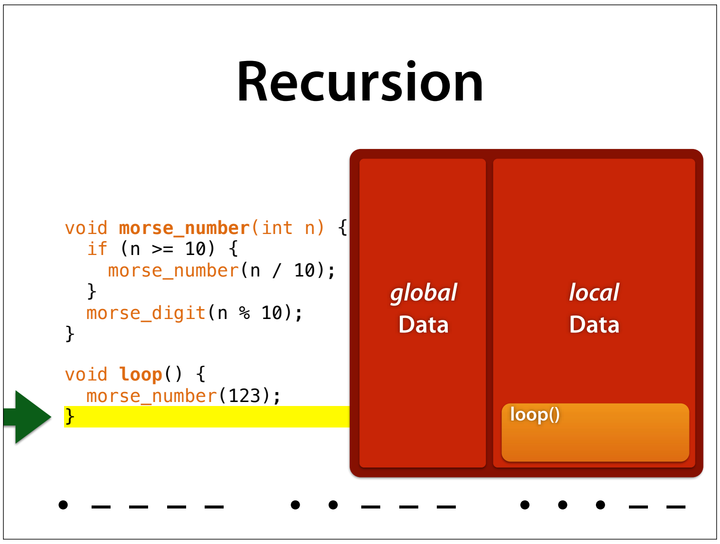
• — — — —   • • — — —   • • • — —



Recursion

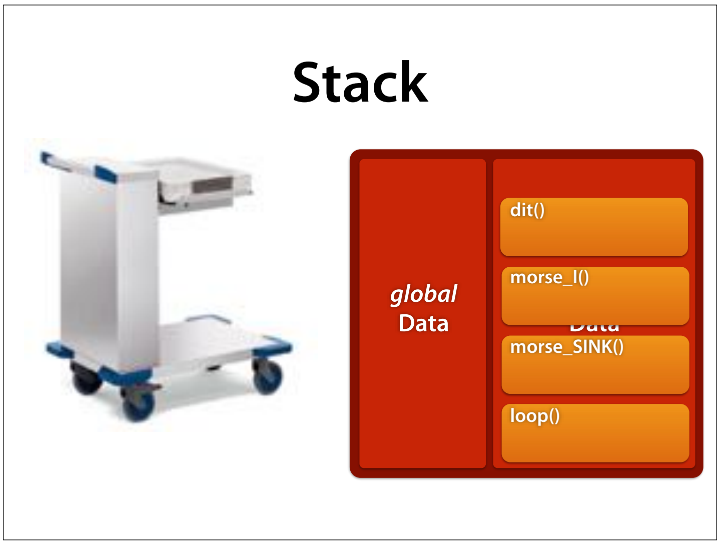
```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```

• — — — —   • • — — —   • • • — —

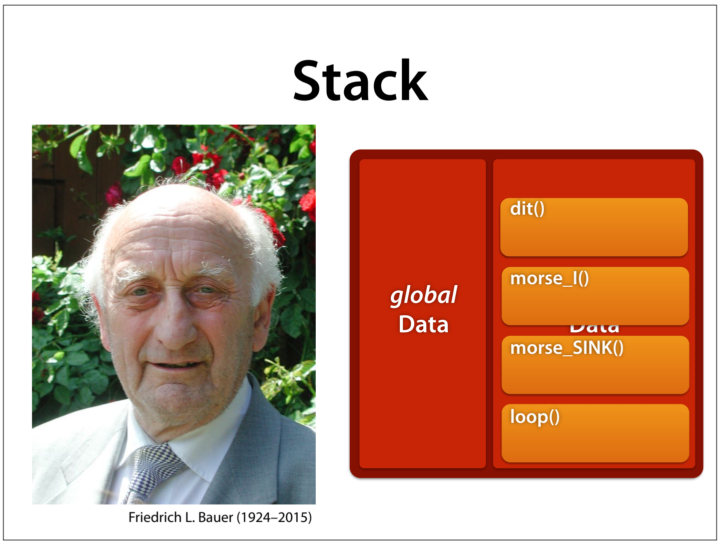


Recursion

```
void morse_number(int n) {
  if (n >= 10) {
    morse_number(n / 10);
  }
  morse_digit(n % 10);
}

void loop() {
  morse_number(123);
}
```



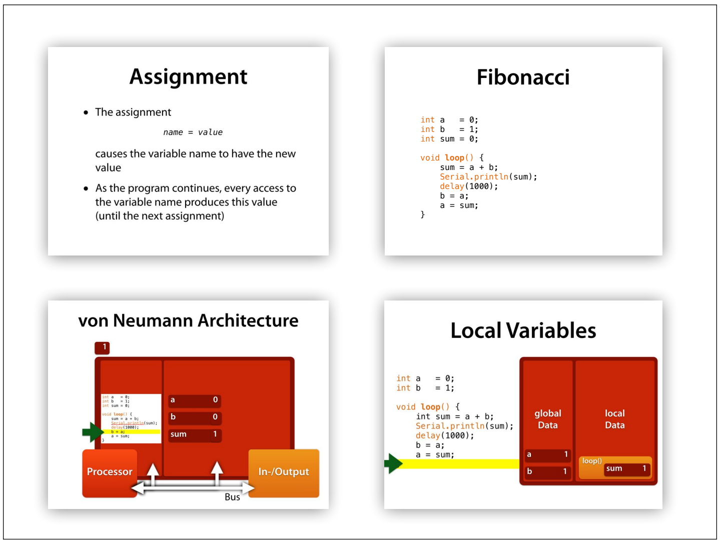
Stack



Stack

Friedrich L. Bauer (1924–2015)

Friedrich Ludwig Bauer (10 June 1924 – 26 March 2015) was a German computer scientist and professor emeritus at the Technical University of Munich. He was the first to propose the widely used stack method of expression evaluation. (Wikipedia)



Assignment

- The assignment

  name = value

  causes the variable name to have the new value

- As the program continues, every access to the variable name produces this value (until the next assignment)

Fibonacci

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

von Neumann Architecture

Local Variables

```
int a   = 0;
int b   = 1;

void loop() {
    int sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
```

Handouts

Assignment

- The assignment

  `name = value`

  causes the variable *name* to have the new value *value*

- As execution resumes, every access to the variable *name* produces this value *value* (until the next assignment)

Fibonacci

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

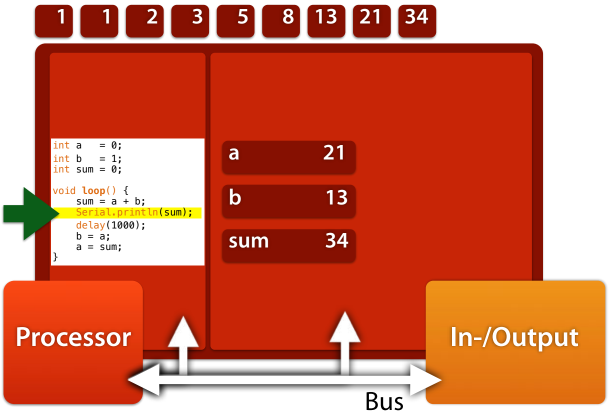
– prints 1 2 3 5 8 13 …

What Happens Here?

```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

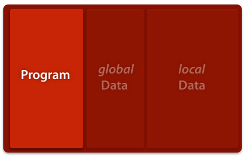
How does the computer execute the program?

Fibonacci Sequence



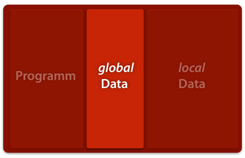
```
int a   = 0;
int b   = 1;
int sum = 0;

void loop() {
    sum = a + b;
    Serial.println(sum);
    delay(1000);
    b = a;
    a = sum;
}
```

The Program



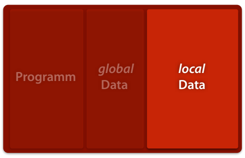
- Resides in memory
  (just like data)
- Cannot access or change itself
- The *operating system* loads and manages the programs in memory

Global Data



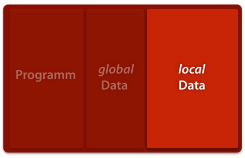
- Contains Variables with *global* visibility
  (= *defined outside of* functions)
- Accessible from all functions

Local Data



- Contains variables with *local visibility*
  (= *defined inside functions*)
- Local variables and parameters exist only during the execution of a function
- They are contained by the function frame

Function Stack



- Function frames are organised as a *stack*
- The *topmost* frame is *active*
- Upon returning from the topmost function, the underlying (calling) function is continued from the call site on