CSCI 334:
Principles of Programming Languages

Lecture 2: Language Models

Instructor: Dan Barowy

Williams

Topics

Language models / implementation

Pointer model / Breph

Pointers + stack machine model / C

Your to-dos

1. Reading response, **due Wednesday 2/9**.
2. Lab 1, **due Sunday 2/13** (partner lab)

Announcements

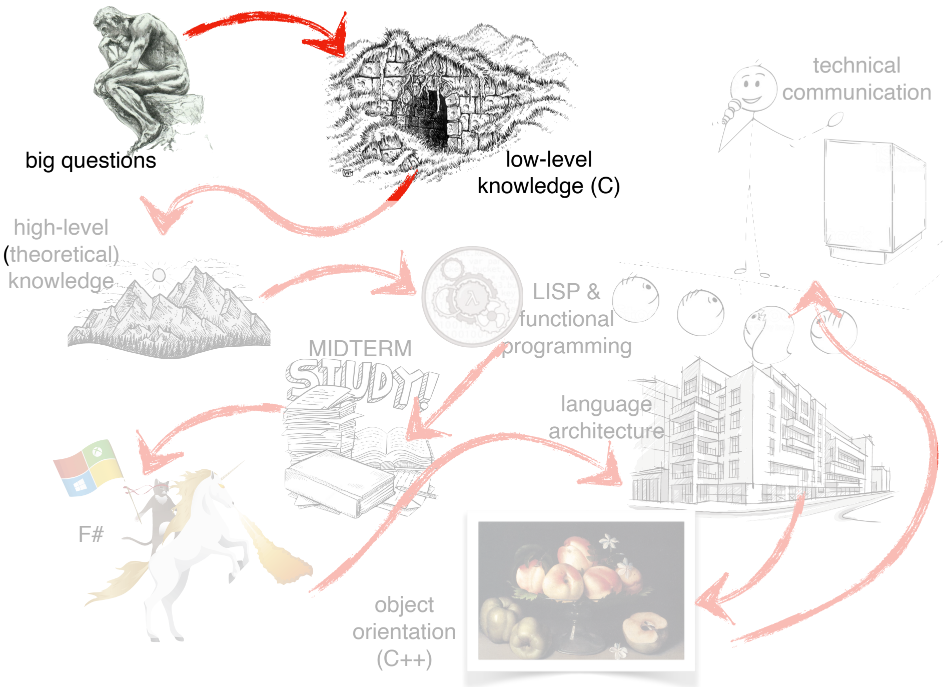
- Field trip to WCMA on **Thursday** (next class)
- Thesis Proposal Colloquium this **Friday, Feb 11**

Toyota Production System

Any worker can stop the line!

Toyota Production System

Stop me if you feel like something is missing!



big questions

low-level knowledge (C)

technical communication

high-level (theoretical) knowledge

LISP & functional programming

MIDTERM

language architecture

F#

object orientation (C++)

Why is computer science called “computer science”?

Why is computer science called "computer science"?

Let's start with the "computer" part.

Why is computer science called "computer science"?

How about the "science" part?

Science

Science, from the Latin *scientia* ('knowledge'), is a systematic enterprise that builds and organizes knowledge **in the form of testable explanations and predictions** about the universe.

(source: Wikipedia)

Science

theory

experiment

Broadly, the goal is to find a **simple explanation** that **accurately predicts the behavior** of a given phenomenon.

Models

zone of proximal development

ideal gas law

quantum mechanics

natural selection

carbon cycle

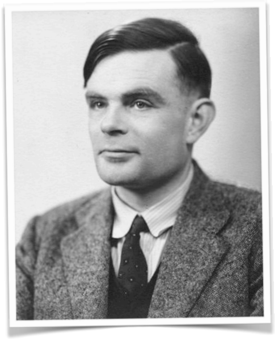
theory of mind

small world hypothesis

fermat’s last theorem

Simple explanations are often, but not always, mathematical.

Computer Science



theory

experiment

What are we **trying to explain** in this discipline?

Broadly: how **mechanical computation** can be **used effectively**.

What models do we have in CS?

Turing machine

Lambda calculus

calculus of constructions

nominal types

Von Neumann machine

object orientation

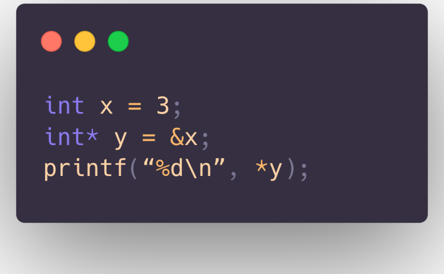
distributed system

n-player game

Programming languages are built on models.

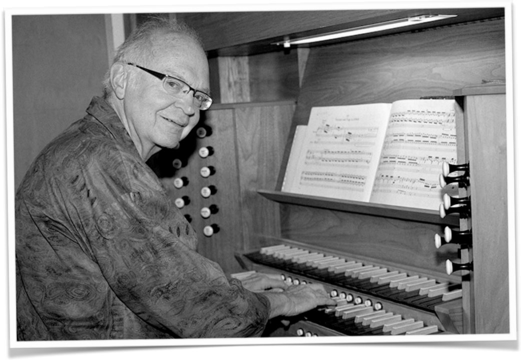
Programming language models

A programming language model says **what a program should do**.



```
int x = 3;
int* y = &x;
printf(“%d\n”, *y);
```

Pointer model



“The way C handles pointers […] was a brilliant innovation; it solved a lot of problems that we had before in data structuring and made the programs look good afterwards.” — Donald Knuth

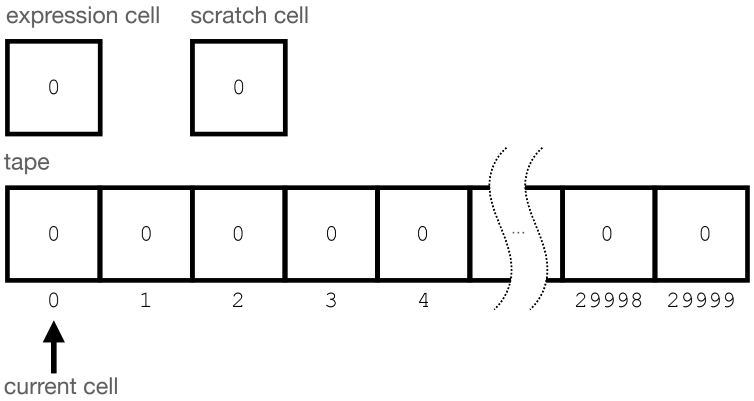
Why do we need pointers?

1. “Any problem in computer science can be solved with another level of indirection.” —Butler Lampson

2. They are necessary for building “persistent” data structures.

Breph machine

A simple Turing equivalent language with **pointers**.



expression cell

scratch cell

tape

current cell

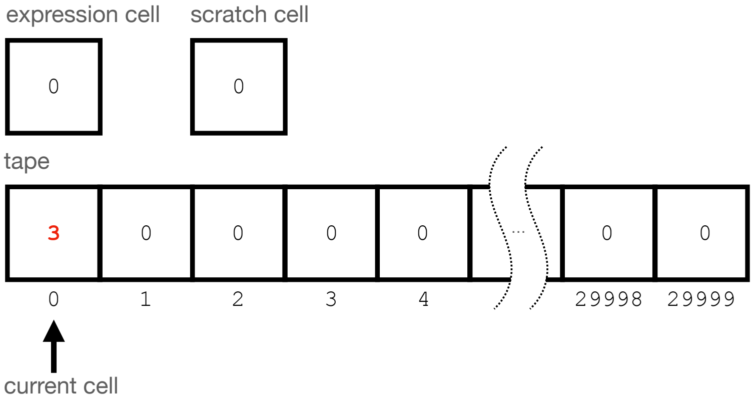
Breph machine

A Breph program changes the state of a Breph machine

```
*+++++++.
j
  &+!
  *+++++.
  &-!
  *-.
u
```

Breph machine: *dereference*

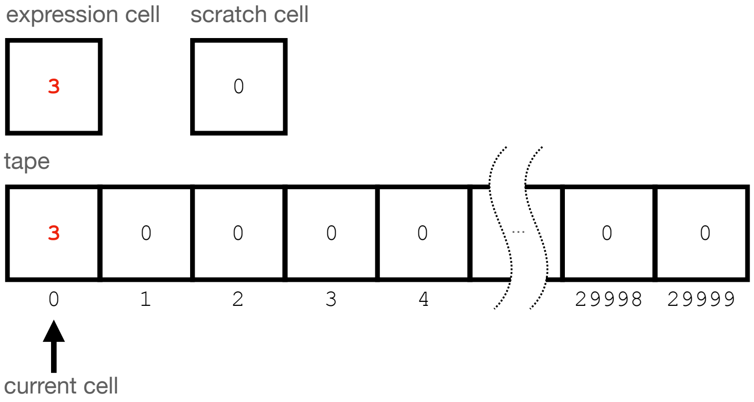
*



Copies current cell to expression cell.

Breph machine: *dereference*

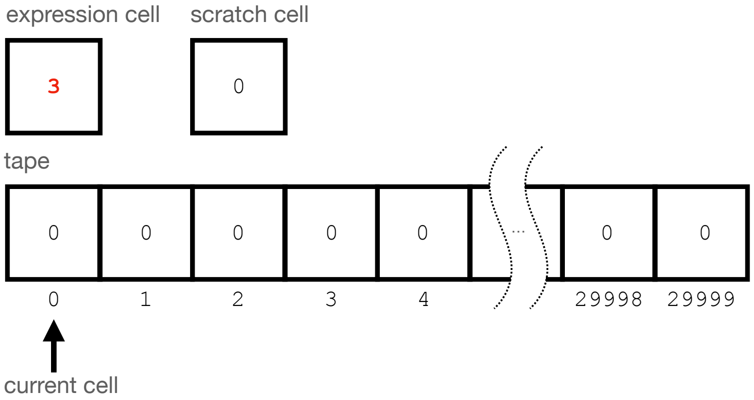
*



Copies current cell to expression cell.

Breph machine: *store*

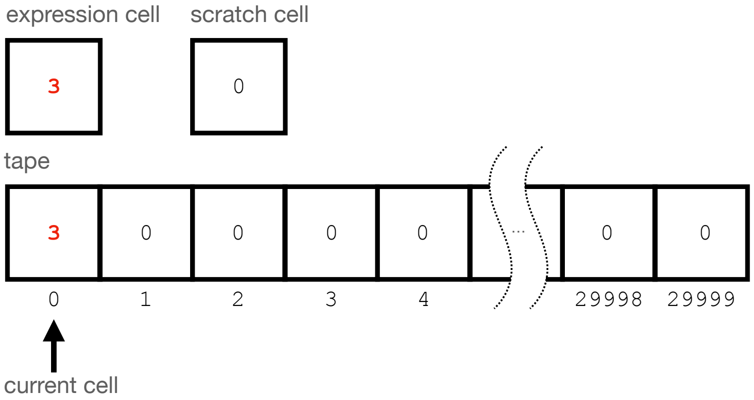
.



Copies expression cell to current cell.

Breph machine: *store*

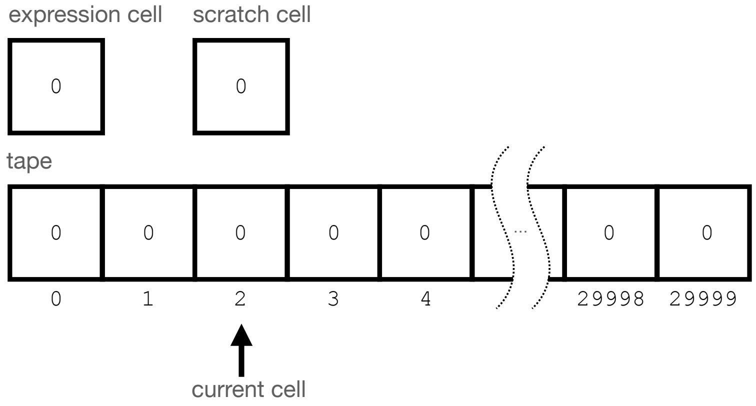
.



Copies expression cell to current cell.

Breph machine: *address of*

&



expression cell: 0

scratch cell: 0

tape

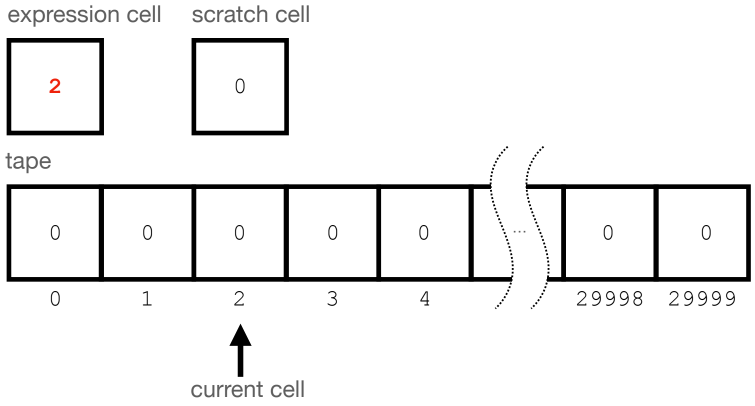
0	1	2	3	4	…	29998	29999
0	0	0	0	0	…	0	0

current cell: 2

Copies current location to expression cell.

Breph machine: *address of*

&



expression cell: 2

scratch cell: 0

tape

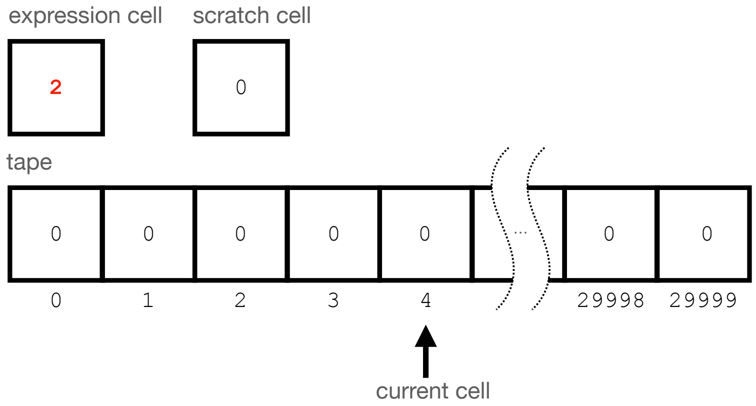
0	1	2	3	4	…	29998	29999
0	0	0	0	0	…	0	0

current cell: 2

Copies current location to expression cell.

Breph machine: *change location*

!



expression cell: 2

scratch cell: 0

tape

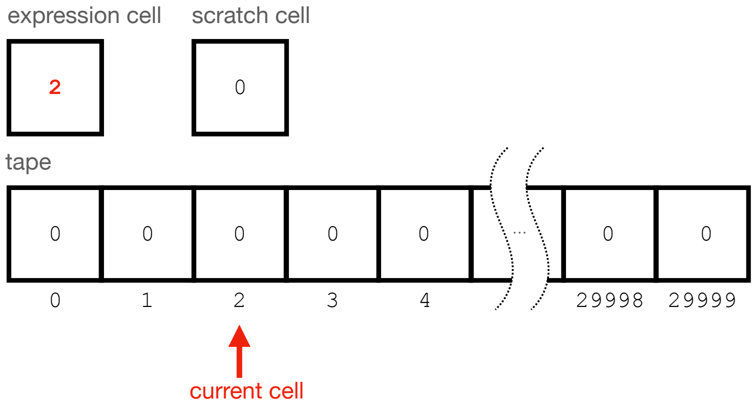
0	1	2	3	4	…	29998	29999
0	0	0	0	0	…	0	0

current cell: 4

Updates current cell location with value in expression cell.

Breph machine: *change location*

!



expression cell: 2

scratch cell: 0

tape

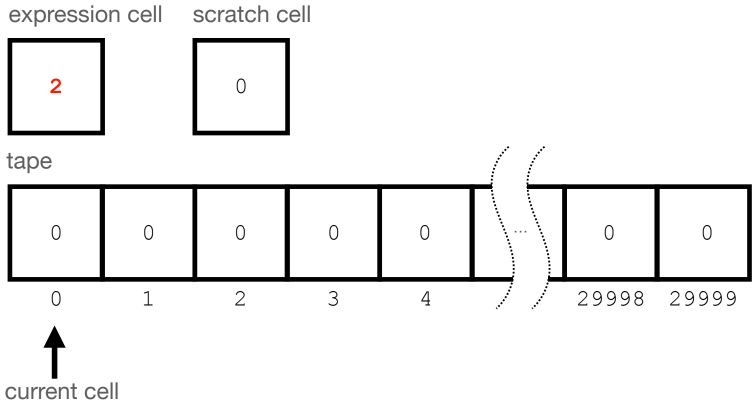
0	1	2	3	4	…	29998	29999
0	0	0	0	0	…	0	0

current cell: 2

Updates current cell location with value in expression cell.

Breph machine: *increment*

`+`



expression cell: 2

scratch cell: 0

tape

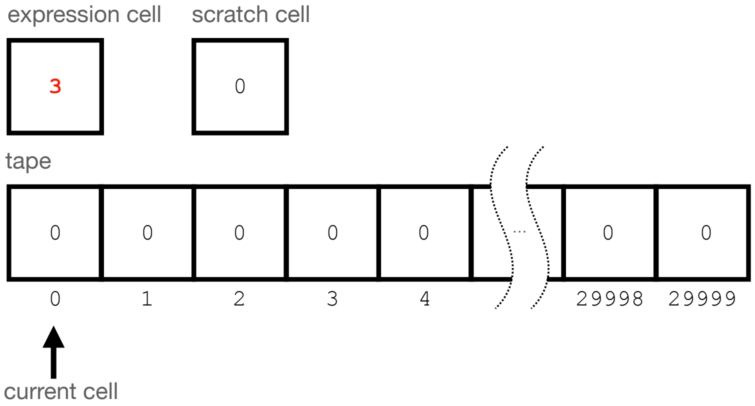
0	1	2	3	4	…	29998	29999
0	0	0	0	0	…	0	0

current cell

Increments the value in the expression cell.

Breph machine: *increment*

`+`



expression cell: 3

scratch cell: 0

tape

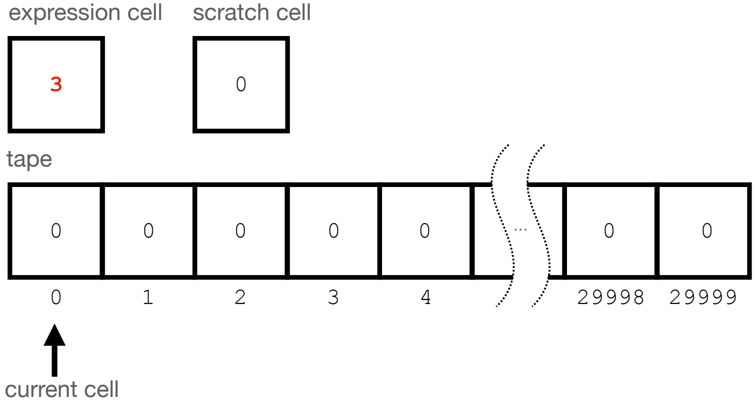
0	1	2	3	4	…	29998	29999
0	0	0	0	0	…	0	0

current cell

Increments the value in the expression cell.

Breph machine: *decrement*

`-`



expression cell: 3

scratch cell: 0

tape

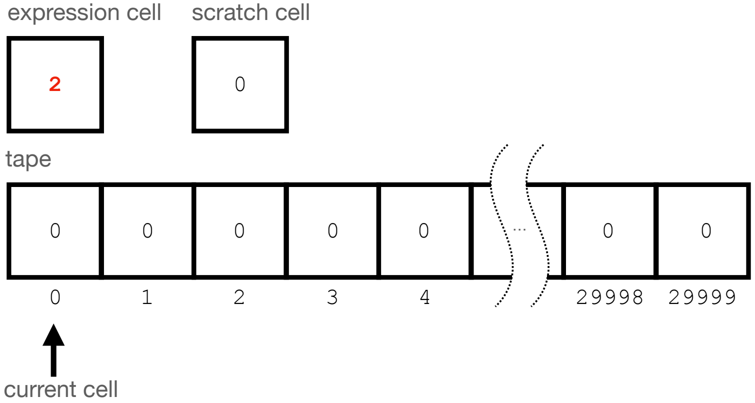
0	1	2	3	4	…	29998	29999
0	0	0	0	0	…	0	0

current cell

Decrements the value in the expression cell.

Breph machine: *jump*

`j`



expression cell: 2

scratch cell: 0

tape

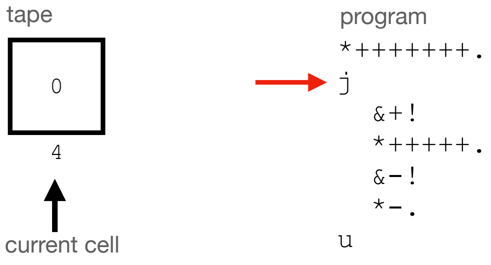
0	1	2	3	4	…	29998	29999
0	0	0	0	0	…	0	0

current cell

Jumps ahead to the *matching* `u` if current cell is 0.

Breph machine: *jump*

j

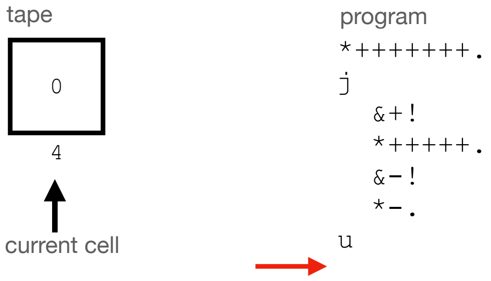


```
*+++++++.
j
   &+!
   *+++++.
   &-!
   *-.
u
```

Jumps after the *matching* `u` if current cell is 0.

Breph machine: *jump*

j

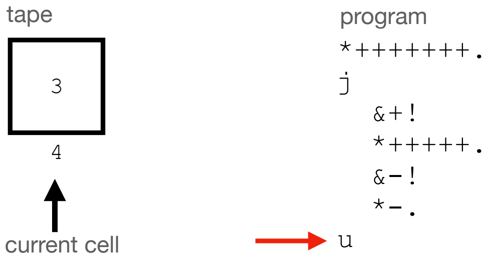


```
*+++++++.
j
   &+!
   *+++++.
   &-!
   *-.
u
```

Jumps after the *matching* `u` if current cell is 0.

Breph machine: *unjump*

u

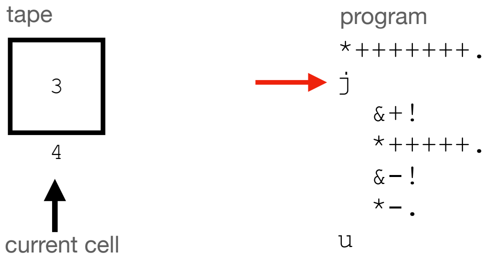


```
*+++++++.
j
   &+!
   *+++++.
   &-!
   *-.
u
```

Jumps to the previous *matching* `j` if current cell is not 0.

Breph machine: *unjump*

u

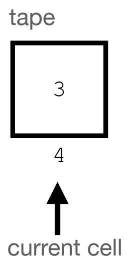


```
*+++++++.
j
   &+!
   *+++++.
   &-!
   *-.
u
```

Jumps to the previous *matching* `j` if current cell is not 0.

Breph machine: *unjump*

u



program

```
*+++++++.
j
   &+!
   *+++++.
   &-!
   *-.
u
```

Jumps to the previous *matching* `j` if current cell is not 0.

Breph machine

Syntax	Meaning	Returns
`i`	Prompt user to enter a character *c*.	ASCII code for *c*
`n`	Prompt for a character of input and interpret it as a number *n*.	*n*
`o`	Print the result of <*e*> as a character.	<*e*>
`#`	Print the result of <*e*> as a numeric character.	<*e*>
`+`	Add one to the result of <*e*>.	<*e*> + 1
`-`	Subtract one from the result of <*e*>.	<*e*> − 1
`*`	"Dereference" operator.	the contents of the current cell
`&`	"Address of" operator.	the location of the current cell
`.`	Stores the result of <*e*> in the current cell.	<*e*>
`!`	Changes the location of the current cell to the result of <*e*>.	<*e*>
`s`	Copies the value of the expression cell to the scratch cell.	<*e*>
`r`	Copies the value of the scratch cell to the expression cell.	the value in scratch
`\n`	Returns 0.	0
`j`	If the current cell is zero, jumps to the operation after the next `u`.	<*e*>
`u`	If the current cell is nonzero, jumps to the previous `j`.	<*e*>
`%`	A line starting with `%` is a comment and is ignored.	*n/a; removed from program*

Breph instruction **syntax** and its **semantics**.

C

C

Start reading "A Brief Overview of C."

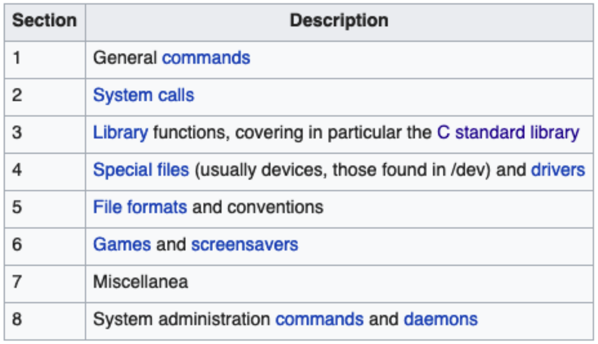
Some useful "functions" for Lab 2

```
malloc    atoi
 free     printf
sizeof    strncpy
memset    fscanf
 rand     fopen
srand     fclose
          rewind
```

Not a function; no man page

See the "man" pages

"man" pages



Section	Description
1	General commands
2	System calls
3	Library functions, covering in particular the C standard library
4	Special files (usually devices, those found in /dev) and drivers
5	File formats and conventions
6	Games and screensavers
7	Miscellanea
8	System administration commands and daemons

`sizeof` operator

`sizeof` is a compile-time **unary operator** which computes the **size of its operand in bytes**. The result of sizeof is of unsigned integral type (**`size_t`**).

`sizeof` can take two kinds of operands:

1. a **data type** (e.g., `int`, `float`, etc.), or
2. an **expression** (e.g., `2 + 1.07`).

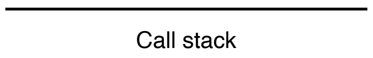
Manual memory management

- C was invented in 1972.
- It features **manual memory management**.
- **Automatic** memory management was invented in **1959!**
- So… **why** have manual management?
- In C, manual memory management is a **feature**, and it's the reason why it might be your language of choice.
- OS, high-performance code, etc.

Ken Thompson (inventor of C, sitting) and Dennis Ritchie (inventor of UNIX, standing).

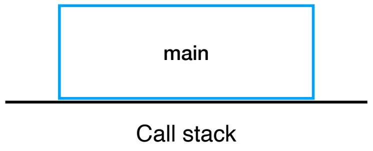
C is about memory

It uses a **model** of a computer that I call the "**boxes and arrows model**."

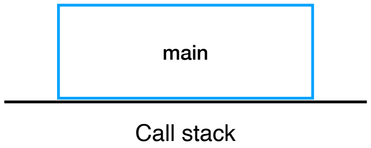


```
#include <stdio.h>

void hello() {
  printf("Hello world!\n");
}

int main() {
  hello();
  return 0;
}
```

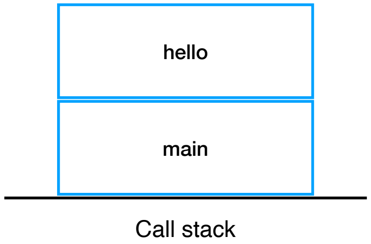
```
#include <stdio.h>

void hello() {
  printf("Hello world!\n");
}

int main() {
  hello();
  return 0;
}
```



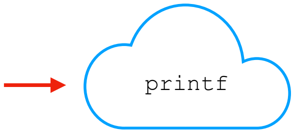
```
#include <stdio.h>

void hello() {
  printf("Hello world!\n");
}

int main() {
  hello();
  return 0;
}
```

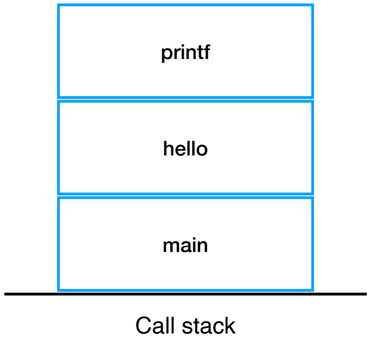


```
#include <stdio.h>

void hello() {
  printf(“Hello world!\n”);
}

int main() {
  hello();
  return 0;
}
```

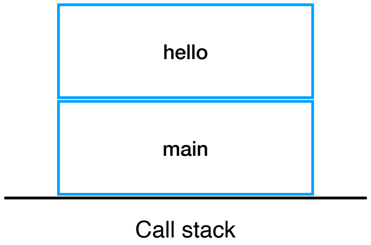


```
#include <stdio.h>

void hello() {
  printf(“Hello world!\n”);
}

int main() {
  hello();
  return 0;
}
```

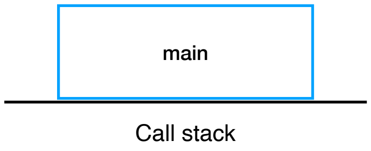




```
#include <stdio.h>

void hello() {
  printf(“Hello world!\n”);
}

int main() {
  hello();
  return 0;
}
```

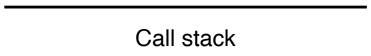


```
#include <stdio.h>

void hello() {
  printf(“Hello world!\n”);
}

int main() {
  hello();
  return 0;
}
```



```
#include <stdio.h>

void hello() {
  printf(“Hello world!\n”);
}

int main() {
  hello();
  return 0;
}
```

program is done



```
#include <stdio.h>

void hello() {
  printf(“Hello world!\n”);
}

int main() {
  hello();
  return 0;
}
```

Storage Duration

We will focus on two: **automatic** and **allocated**

You (the programmer) **choose** which one you **want**.

Rule:
Always choose **automatic duration** unless the lifetime of your data outlives its allocation site, in which case, you should choose **allocated duration**.

Storage Duration: Automatic

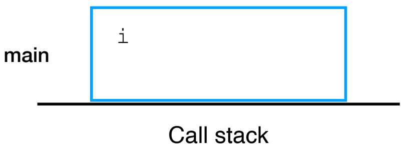
```
int i = 3;
```

`i` has automatic duration, because you didn’t specify anything.

C will automatically acquire (*allocate*)
and release (*deallocate*) memory for this variable.

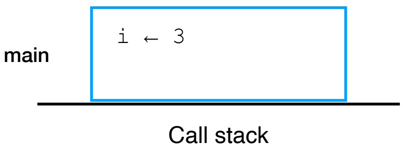
In reality, nearly every C implementation will store `i` *on the call stack*.

Storage Duration: Automatic



```
  #include <stdio.h>

  int main() {
→   int i = 3;
    return i;
  }
```

Storage Duration: Automatic



```
  #include <stdio.h>

  int main() {
    int i = 3;
→   return i;
  }
```

Storage Duration: Automatic

```
  #include <stdio.h>

  int main() {
    int i = 3;
    return i;
→ }
```

Call stack

Where does `i` get returned? How?

Storage Duration: Automatic

```
  #include <stdio.h>

  int main() {
    int i = 3;
    return i;
→ }
```

Call stack

`main`’s stack frame and all variables in it (i.e., `i`) are automatically deallocated when `main` *goes out of scope*.

Storage Duration: Allocated

```
int *i = malloc(sizeof(int));
```

the memory `i` points to has allocated duration, because you used `malloc`.

C will manually allocate *on request*
and deallocate memory *on request*.

In reality, nearly every C implementation will store `i` *on the heap*.

Storage Duration: Allocated

To deallocate, you must call `free`

```
int *i = malloc(sizeof(int));
free(i);
```

You have to do this even if `i` goes out of scope!

Failing to free when you are done is a bug called a *memory leak*.

Recap & Next Class

Today we covered:

Language models

Next class:

WCMA!