CSCI 334:
Principles of Programming Languages

Lecture 2

Instructor: Dan Barowy

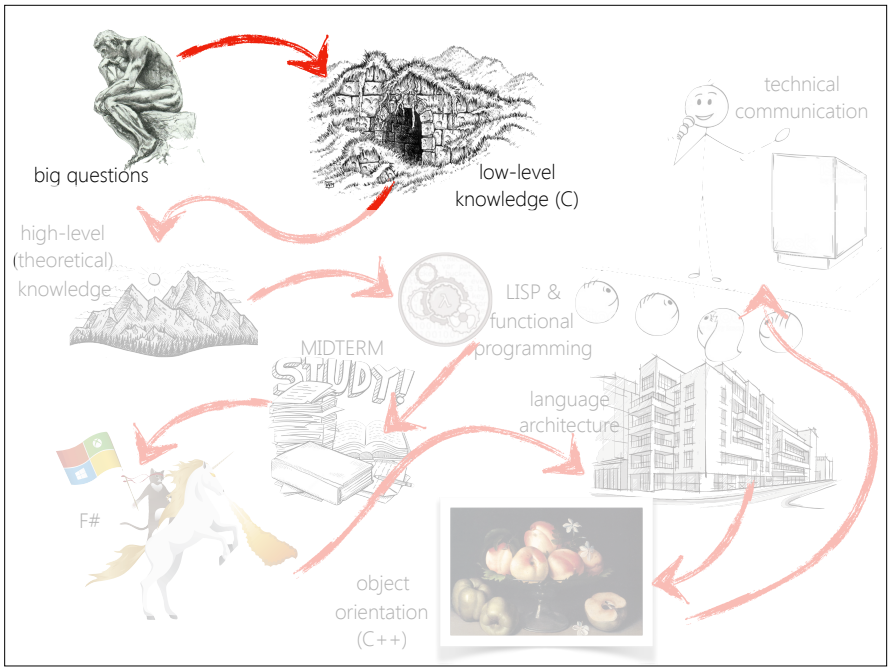
Williams

Announcements

Lab 1 due Sunday by 11:59pm

No class Friday (Winter Carnival)

No class Tuesday (PC meeting)



Outline

1. Quiz
2. LeWitt assignment
3. Boxes and arrows model

C

C

Read the “Intro” doc.

C

Read the “Intro” doc.

If you haven’t started, start now.

Some useful “functions” for Lab 1


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

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`).

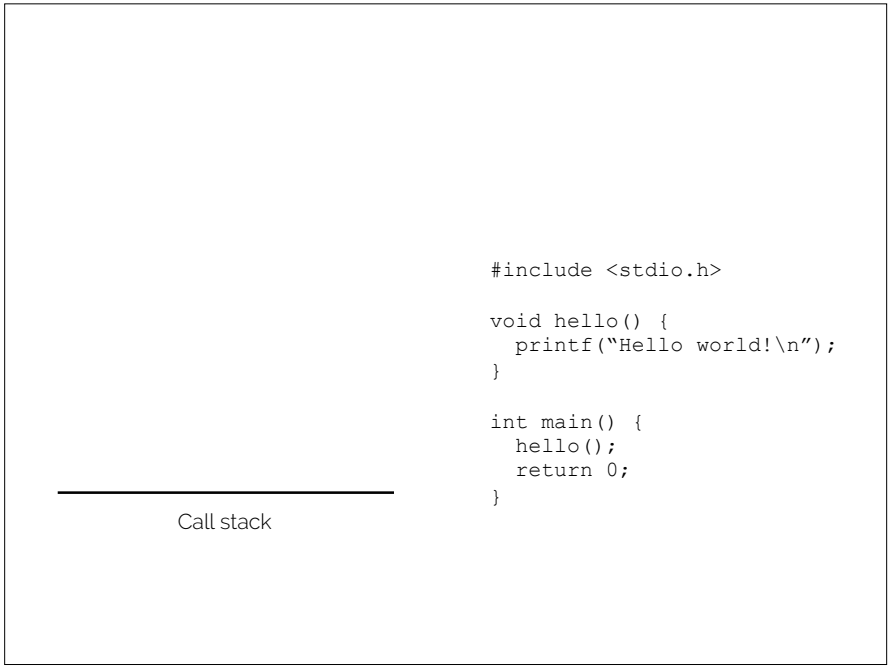
Manual memory management

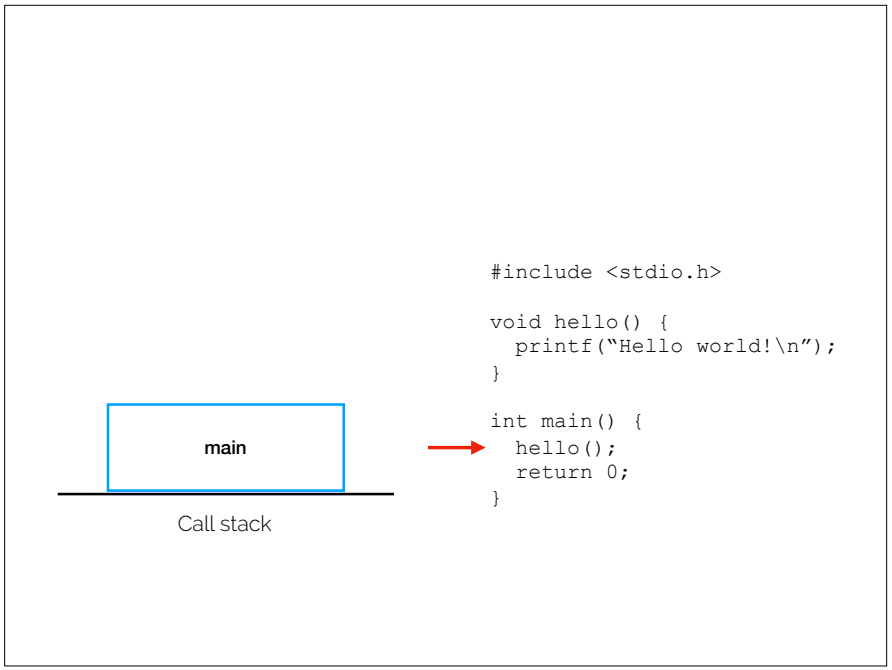
- C was invented in 1972.
- It features **manual memory management**.
- **Automatic** memory management was invented in **1959!**
- So… **why** have manual management?
- In C, manual memory management is a **feature**, and it's the reason why it might be your language of choice.
- OS, high-performance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

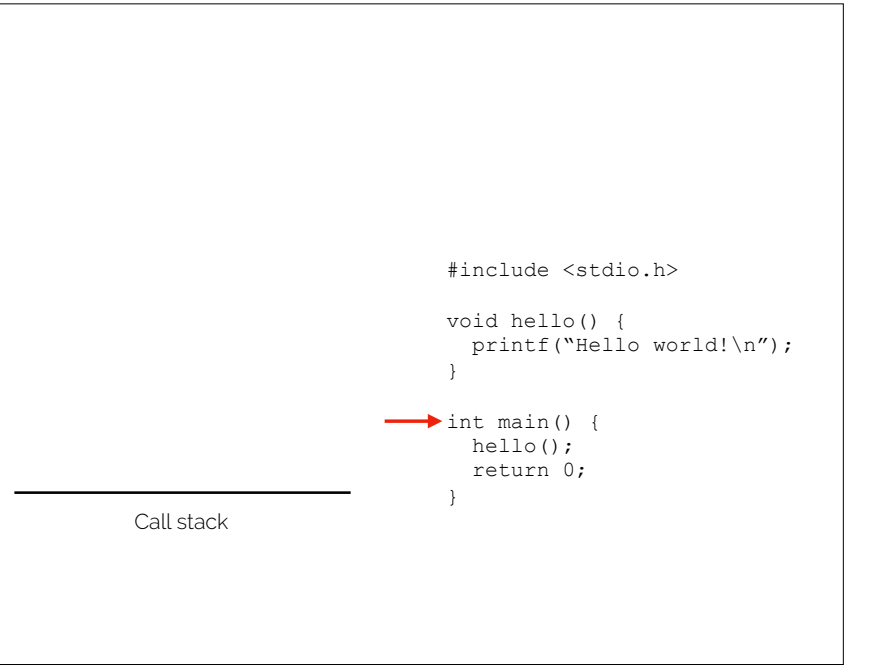
Ken Thompson (inventor of C, sitting) and Dennis Ritchie (inventor of UNIX, standing).

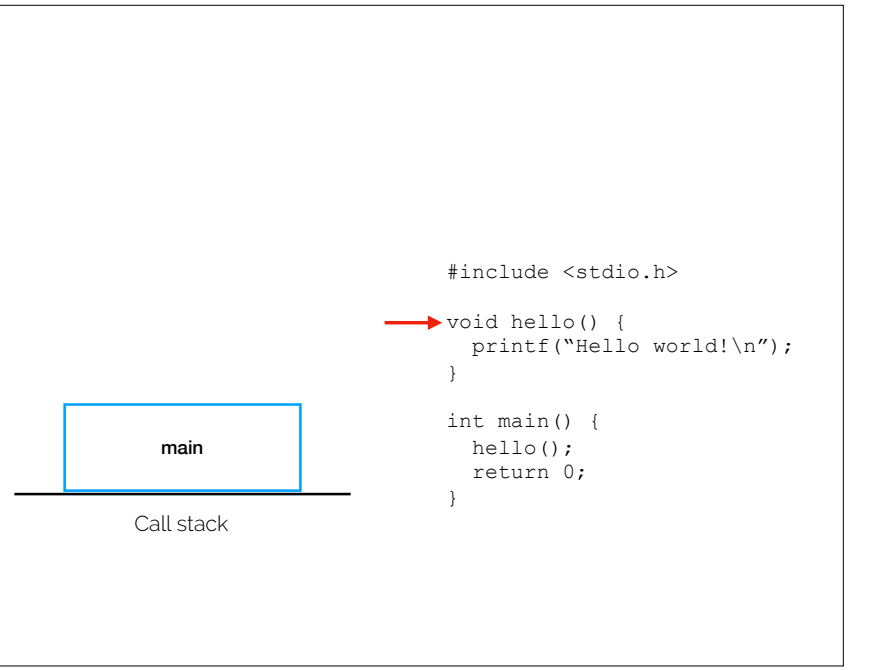
C is about memory

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


Call stack
#include <stdio.h>
void hello() {
printf("Hello world!\n");
}
int main() {
hello();
return 0;
}


main
Call stack
#include <stdio.h>
void hello() {
printf("Hello world!\n");
}
int main() {
hello();
return 0;
}


Call stack
#include <stdio.h>
void hello() {
printf("Hello world!\n");
}
int main() {
hello();
return 0;
}

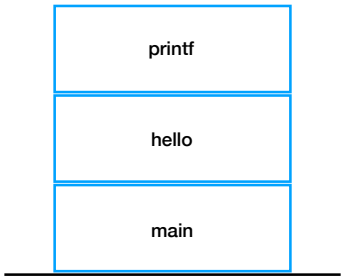

main
Call stack
#include <stdio.h>
void hello() {
printf("Hello world!\n");
}
int main() {
hello();
return 0;
}

Call stack

```
#include <stdio.h>

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

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

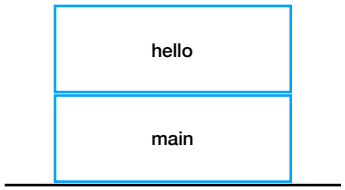


Call stack

```
#include <stdio.h>

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

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



Call stack

```
#include <stdio.h>

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

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

Call stack

```
#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;
}
```

Call stack

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.

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

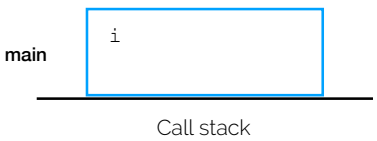
```
int i = 3;
```

`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

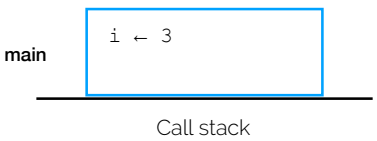


Call stack

```
#include <stdio.h>

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

Storage Duration: Automatic



Call stack

```
#include <stdio.h>

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

Storage Duration: Automatic

Call stack

```
#include <stdio.h>

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

Where does `i` get returned? How?

Storage Duration: Automatic

Call stack

```
#include <stdio.h>

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

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

Activity

```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

2 →

1 →

3 →

Diagram the stack and variables when the program is at the three points.

Storage Duration: Allocated

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

`i` 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

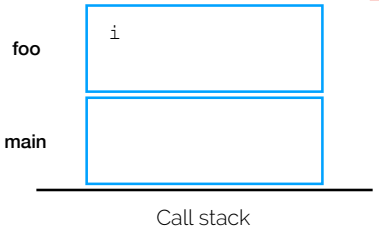
To deallocate, you must call `free`

```
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***.

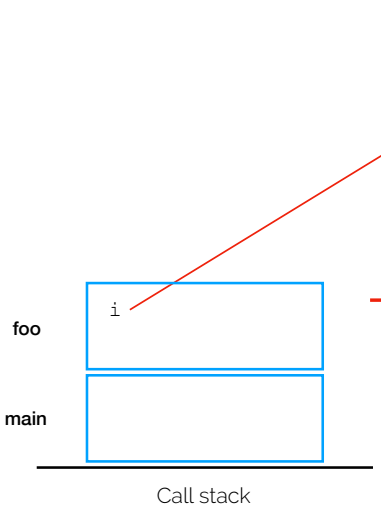
Storage Duration: Allocated




```
#include <stdio.h>
#include <stdlib.h>

int foo() {
  int i* = malloc(sizeof(int));
  *i = 3;
  return 0;
}
```

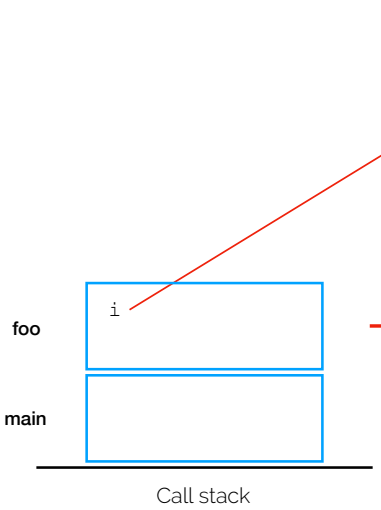
Storage Duration: Allocated




```
#include <stdio.h>
#include <stdlib.h>

int foo() {
  int i* = malloc(sizeof(int));
  *i = 3;
  return 0;
}
```

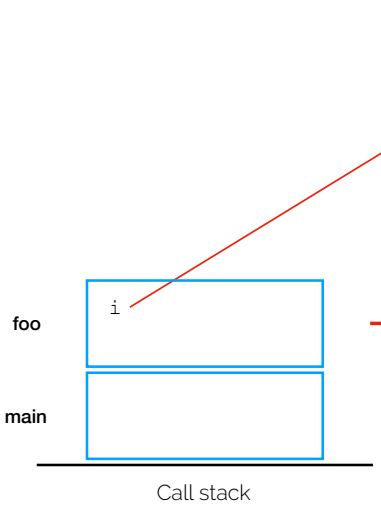
Storage Duration: Allocated




```
#include <stdio.h>
#include <stdlib.h>

int foo() {
  int i* = malloc(sizeof(int));
  *i = 3;
  return 0;
}
```

Storage Duration: Allocated

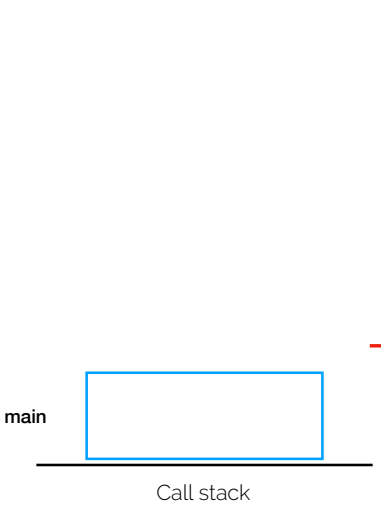



```
#include <stdio.h>
#include <stdlib.h>

int foo() {
  int i* = malloc(sizeof(int));
  *i = 3;
  return 0;
}
```

Anyone see a problem?

Storage Duration: Allocated




```
#include <stdio.h>
#include <stdlib.h>

int foo() {
  int i* = malloc(sizeof(int));
  *i = 3;
  return 0;
}
```

Anyone see a problem?

3 is now unreachable, and we cannot reclaim it. **Memory leak.**

Activity

```
#include <stdio.h>

void add(int *x, int *y, int *z) {
  *z = *x + *y;
}

int main() {
  int x = 1;
  int y = 3;
  int z;
  add(&x, &y, &z);
  return z;
}
```

2 → `}`

1 → `int z;`

3 → `return z;`

Diagram the stack and variables when the program is at the three points.

Call-by-value

(program evaluation strategy)

Examples:

C

Java

Python

```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

How does a function "obtain" a parameter value?

Call-by-value semantics: copying

Call-by-value

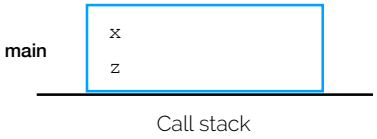
Call stack

```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

Call-by-value

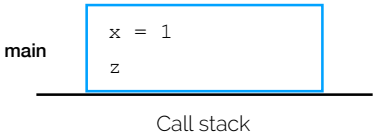



```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

Call-by-value

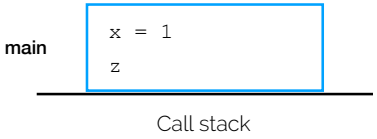



```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

Call-by-value



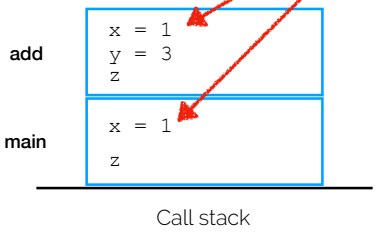

```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

Call-by-value

Not the same x!



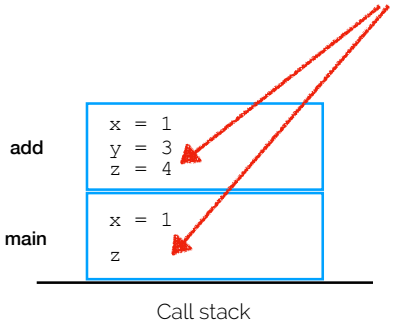

```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

Call-by-value

Not the same z!

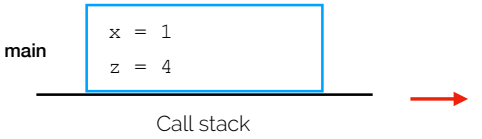



```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

Call-by-value




```
#include <stdio.h>

int add(int x, int y) {
  int z = x + y;
  return z;
}

int main() {
  int x = 1;
  int z = add(x, 3);
  return z;
}
```

What can a function return?

What can a function return?

C String Trick

Ensuring null termination is not always easy.

`memset` can make reasoning about C strings easier.

```
char *memset(char *buf, char c, size_t len)
```

e.g.,

```
memset(&dst,'\0',sizeof(dst))
```

Assuming that dst is an automatic buffer.

Recap & Next Class

Today we covered:

Boxes and arrows model

Next class:

Going deep with pointers

Final projects