CSCI 334:
Principles of Programming Languages

Lecture 13: More C; Debugging tips

Instructor: Dan Barowy

Williams

Topics

Basic C

Pointers + stack model = “boxes and arrows”

Makefiles

String pitfalls

Storage duration

Call-by-value evaluation

Your to-dos

1. Lab 7, **due Sunday 10/30** (partner lab)
2. Midterm review: **next Tuesday, 11/1**, in class
3. Midterm: next **Thursday, 11/3**, in class
4. Project brainstorms: **Sunday, 11/6**

Announcements

- CS Colloquium, Fri @ 2:35 in Wege Auditorium
  Pre-registration info session
  + cookies

Exam practice problems

Dealing with Bugs

Example from last time

```c
#include <stdio.h>

int main() {
  int data[10];
  for (int i = 0; i <= 10; i++) {
    data[i] = i;
  }
  for (int i = 0; i <= 10; i++) {
    printf("%d: %d\n", i, data[i]);
  }
  return 0;
}
```

Address Sanitizer

errno

Stepping debugger

Storage Duration

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 a duration**.

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

Nearly every C implementation stores `i` ***on the call stack***.

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*.

Nearly every C implementation stores `i` ***on the heap***.

Storage Duration: Allocated

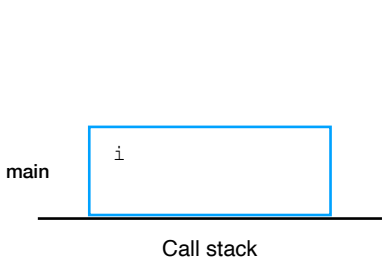
To **deallocate**, you must call **`free`**

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

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

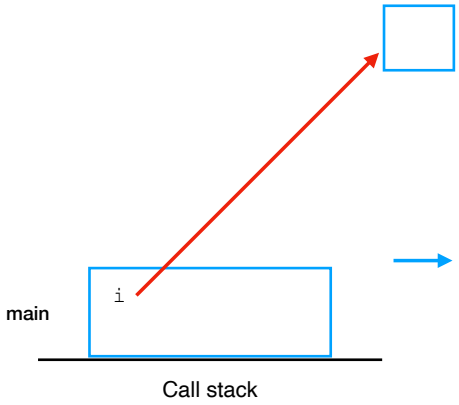
Failing to free when you are done is a **bug** called a ***memory leak***.

Storage Duration: Allocated



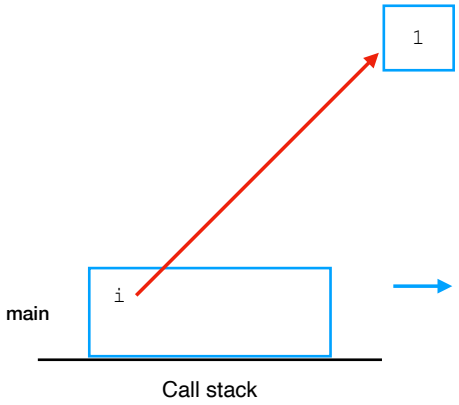

```
#include <stdio.h>

int main() {
  int* i = malloc(sizeof(int));
  *i = 1;
  free(i);
  return 0;
}
```

Storage Duration: Allocated



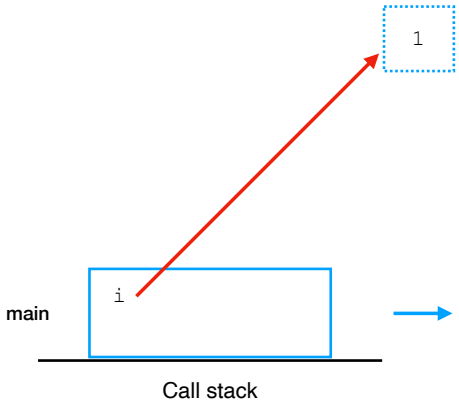

```
#include <stdio.h>

int main() {
  int* i = malloc(sizeof(int));
  *i = 1;
  free(i);
  return 0;
}
```

Storage Duration: Allocated




```
#include <stdio.h>

int main() {
  int* i = malloc(sizeof(int));
  *i = 1;
  free(i);
  return 0;
}
```

Storage Duration: Allocated




```
#include <stdio.h>

int main() {
  int* i = malloc(sizeof(int));
  *i = 1;
  free(i);
  return 0;
}
```

`free` **does not erase** the value stored in memory.

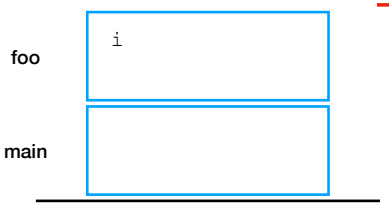
`free` **does not nullify** the pointer stored in `i`.

All `free` does is **mark** the allocated memory as "**reusable**."

Person example

Allocated Duration Pitfalls

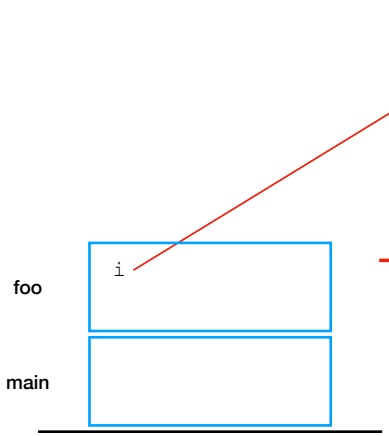
Allocated Duration Pitfalls



Call stack

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

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

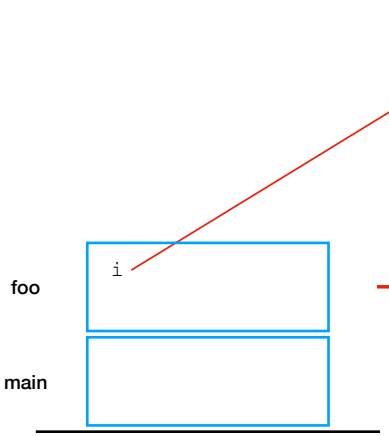
Allocated Duration Pitfalls



Call stack

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

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

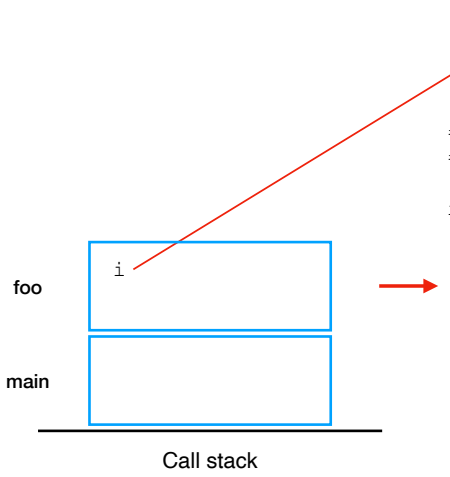
Allocated Duration Pitfalls



Call stack

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

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

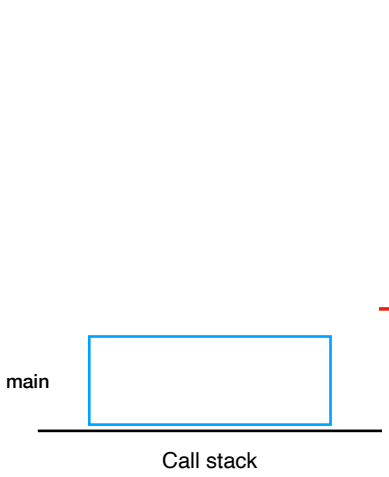
Allocated Duration Pitfalls




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

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

Anyone see a problem?

Allocated Duration Pitfalls




```
#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.**

Call-by-value

(program evaluation strategy)

Examples:

C

Java

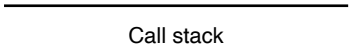
Python

```
#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?

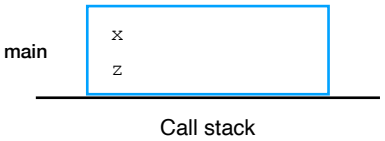
When using call-by-value semantics: **copying**

Call-by-value



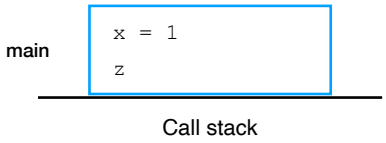

```
#include <stdio.h>

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



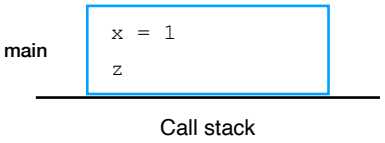

```
#include <stdio.h>

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



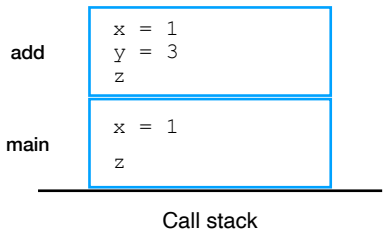

```
#include <stdio.h>

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




```
#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;
}
```

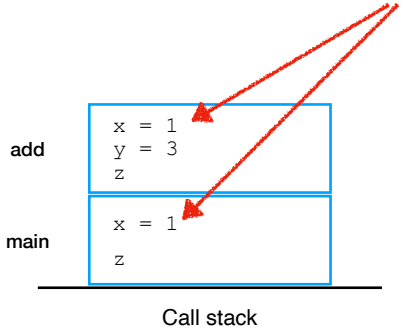
Call-by-value




```
#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;
}
```

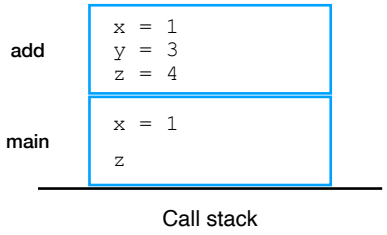
Call-by-value

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;
}
```

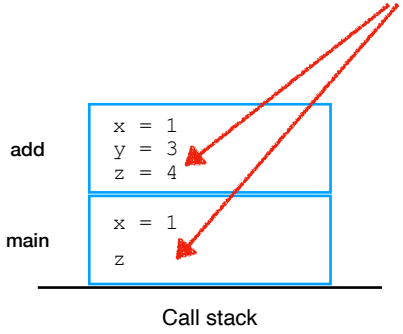
Call-by-value




```
#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;
}
```

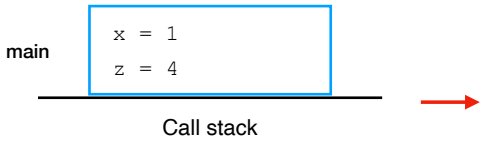
Call-by-value

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;
}
```

Partner activity

```
#include <stdio.h>

void add(int *x, int *y, int *z) {
  *z = *x + *y;
}                                  <- 2

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

Diagram the stack and variables when the program is **about to execute** the given line.

Call-by-reference

(C does not have this!)

Recap & Next Class

Today:

- Debugging C
- Storage duration
- Call-by-value evaluation

Next class:

- Exam review