CSCI 334:
Principles of Programming Languages

Lecture 4

Memory and call-by-value semantics

Instructor: Dan Barowy

Williams

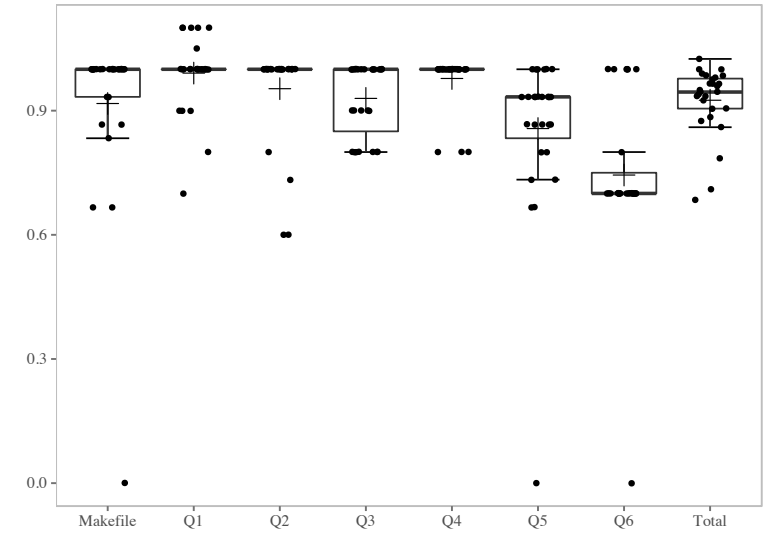
HW3 will be posted soon

HW1 solutions will be in my box by the end of the day.

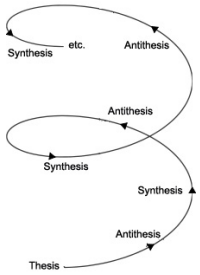
HW1 grades are back

Grades are sent back, with feedback,
as a "pull request" in Github.

HW1 grade distribution



Resubmissions



Resub Policy:

1. You have 5 to use as you wish (except the project), including the midterm exam.
2. You have 2 weeks from the time I hand back assignment to submit.
3. Resub must include original and updated solution.
4. Critically, it must explain, in plain English, *what* you did wrong, *why* you made the mistake, and *how* your new solution fixes the problem. In short, you need to tell me what you learned from your mistake.
5. Commit to your repo, and then open an issue called "Resubmission", and assign me to the issue.
6. You can earn back up to 50% of missing points.

Feedback

Feedback

1. Suggestion: more C practice.
2. Want: more time between lesson and hw due date.
   a. I hear you.
   b. Be an active learner.

This is the path to excellence

Reading Responses

Mental technique #2: motivation

Who do you want to be?

(Simone Biles)

You cannot be excellent until you commit to a goal.

Excellence requires *deliberate practice*.

You cannot commit to a goal unless you are motivated.

Why are you here?

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

This can be a tad complex.

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

Storage Duration: Automatic

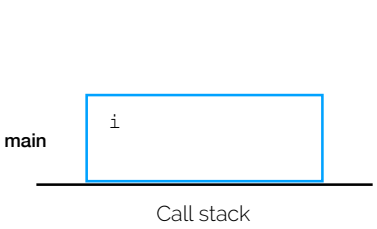
```
int i = 3;
```

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

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

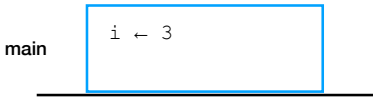
In reality, nearly every C implementation will store `i` ***on the call stack***.

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;
}
```

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***.

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;
}
```

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

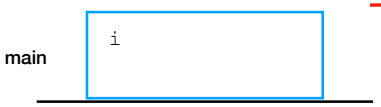
To deallocate, you must call `free`

```
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***.

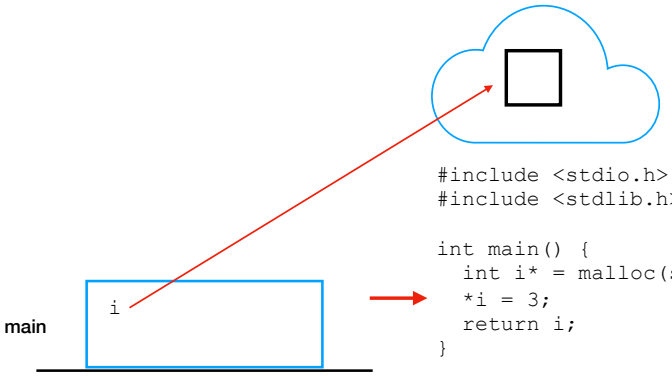
Storage Duration: Allocated



Call stack

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

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

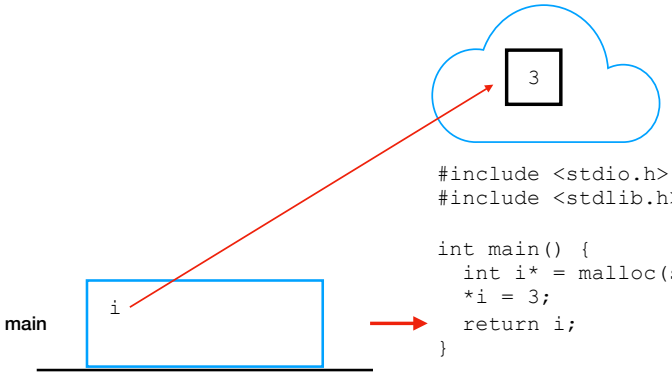
Storage Duration: Allocated



Call stack

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

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

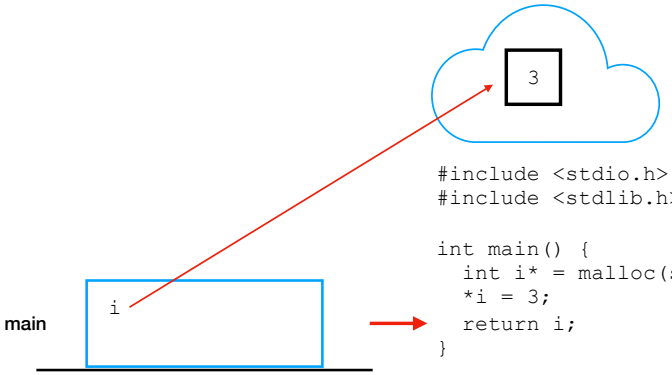
Storage Duration: Allocated



Call stack

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

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

Storage Duration: Allocated

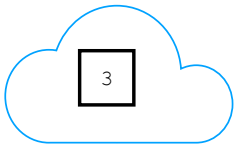


Call stack

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

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

Anyone see a problem?

Storage Duration: Allocated



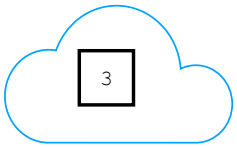

```
#include <stdio.h>
#include <stdlib.h>

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

Call stack

Anyone see a problem?

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

Storage Duration: Allocated




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

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

Call stack

Anyone see a problem?

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

How should we fix it?

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 → `}` (end of `add`)

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

```c
#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

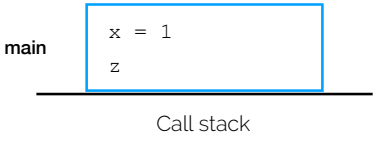
Call stack

```c
#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

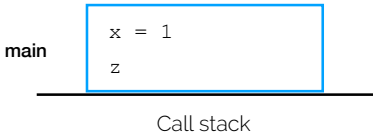

```c
#include <stdio.h>

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




```c
#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

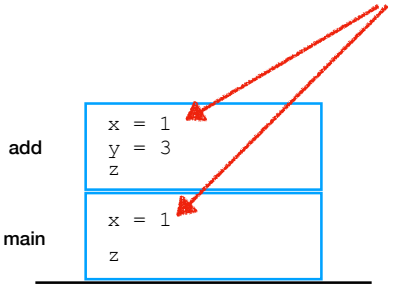


Call stack

```
#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

Not the same x!

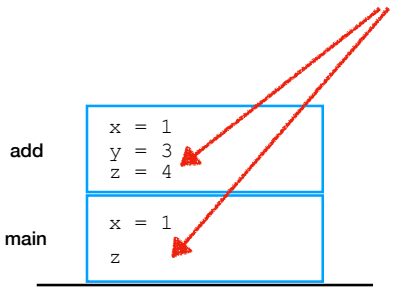


Call stack

```
#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

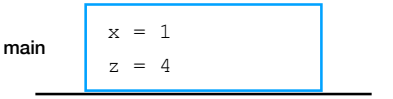
Not the same z!



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



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;
}
```

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.