CSCI 334:
Principles of Programming Languages

Lecture 5: The Rest of C

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 2, **due Sunday 2/20** (partner lab)
   a. You may want to start the reading early
2. Reading response, **due Wednesday 2/23**.

Announcements

- Partner?
- No office hours Friday (faculty “retreat”)

Visualizing programs

1. Seeing the **program** vs. seeing the **problem**
2. It is **hard to separate feelings** from programming
   a. Most vivid descriptions were about feeling bad during time pressure.

Growing a language

1. Isabel: Java **waiting was right** because "it was not hard to learn, and it was not hard to port." Scala: `https://www.scala-lang.org/api/current/scala/collection/immutable/Vector.html`
2. Go. No generics because it was designed for the "**lowest common denominator at Google**."

Makefiles

Makefiles

A `Makefile` is a **specification** used by the `make` tool to **automate** the compilation of programs.

Rationale

Programmers build software **frequently**.

Lazy
(don't want to retype)

Impatient
(don't want to wait for gcc)

Insight

The entire project does not need to rebuilt on every change.

a.h
```
int foo (int a);
```

a.c
```
int foo (int a) {
  return a + 1;
}
```

b.h
```
int bar (int b);
```

b.c
```
int bar (int b) {
  return b - 1;
}
```

depends on

depends on

c.c
```
#include <stdio.h>
#include "a.h"
#include "b.h"

int main() {
  int c = bar(foo(2));
  printf("c = %d\n", c);
  return 0;
}
```

Insight

The entire project does not need to rebuilt on every change.

a.h
```
int foo (int a);
```

a.c
```
int foo (int a) {
  return a + 1;
}
```

b.h
```
int bar (int b);
```

b.c
```
int bar (int b) {
  return b - 1;
}
```

depends on

depends on

c.c
```
#include <stdio.h>
#include "a.h"
#include "b.h"

int main() {
  int c = bar(foo(3));
  printf("c = %d\n", c);
  return 0;
}
```

make a change

a.c and b.c do not change.

Do we really need to rebuild them?

Makefile syntax

```
program: c.c b.o a.o
tab → gcc -o program c.c b.o a.o
```

```
target: dep1 … depn
tab → command
```

command should produce target.

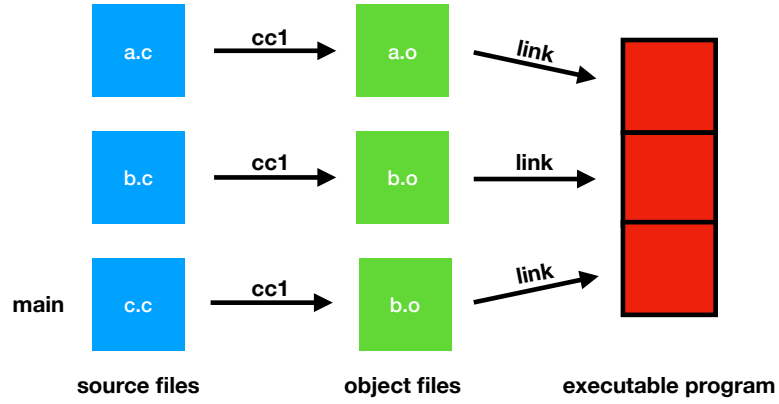
Partner activity

```
#include <stdio.h>

void something(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int x = 1;
  int y = 2;
  something(&x, &y);
  printf("x = %d, y = %d\n", x, y);
  return 0;
}
```

Produce a `Makefile` for this program.
Everything is in one file called `something.c`.

Separate compilation

Separate compilation



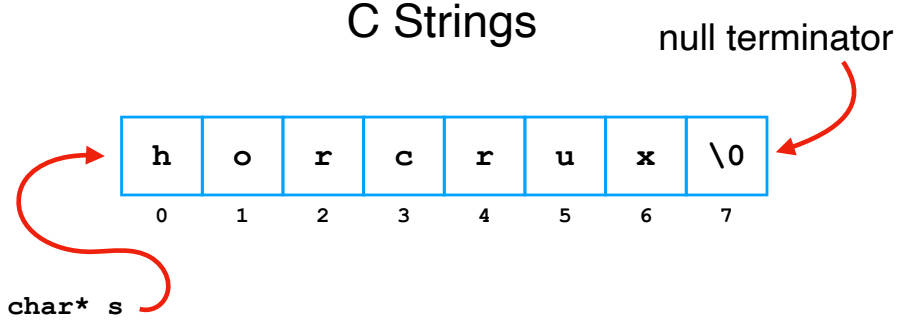
Makefile with separate compilation

```
#include <stdio.h>

void something(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int x = 1;
  int y = 2;
  something(&x, &y);
  printf("x = %d, y = %d\n", x, y);
  return 0;
}
```

Split your program into **two pieces**. `lib.c` should contain the `something` function and `something.c` should contain `main`. **Update** your `Makefile`.

More C

C Strings



C has **no string data type**!

C strings are "just" a convention.

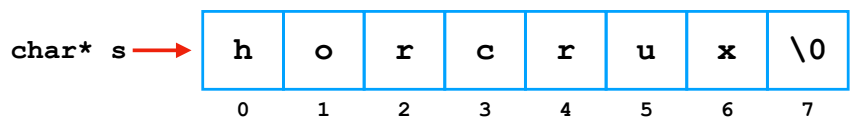
However, all string functions (`string.h`)
expect that you adhere to this convention.

Demonstration of badness

Copying Strings: watch out!

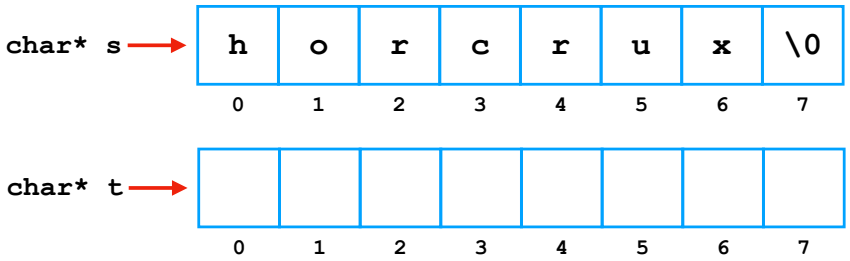
```
char *s = "horcrux";
char t[strlen(s)];
t = s;
```

Copying Strings: watch out!



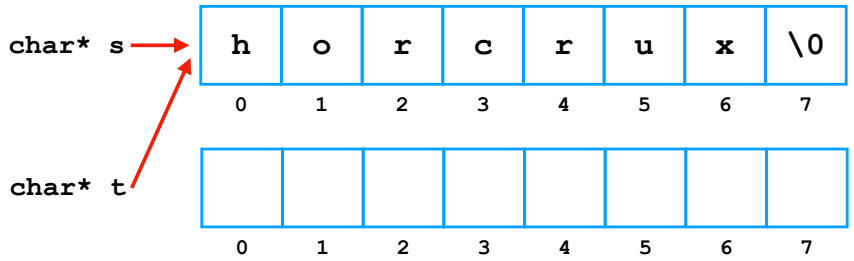

```
char *s = "horcrux";
char t[strlen(s) + 1];
t = s;
```

Copying Strings: watch out!



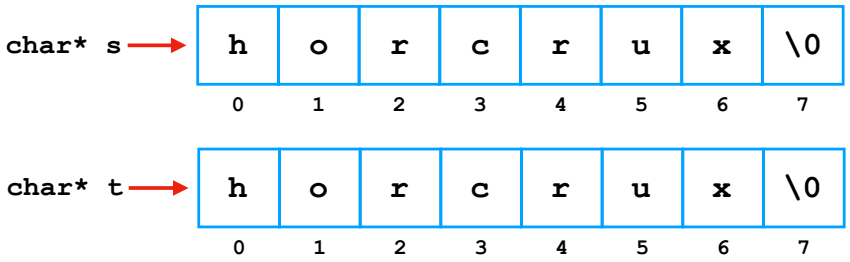

```
char *s = "horcrux";
char t[strlen(s) + 1];
t = s;
```

Copying Strings: watch out!




```
char *s = "horcrux";
char t[strlen(s) + 1];
t = s;
```

Instead, do:




```
char *s = "horcrux";
char t[strlen(s) + 1];
strcpy(t, s);
```

Storage Duration

Storage Duration

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

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

<u>Rule:</u>
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

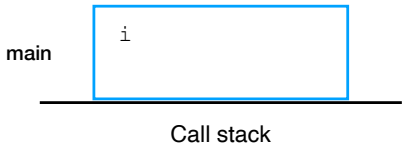
```
int i = 3;
```

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

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

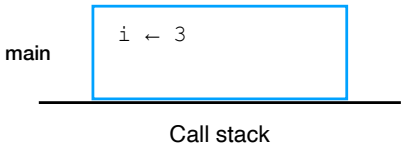
Nearly every C implementation stores `i` ***on the call stack***.

Storage Duration: Automatic



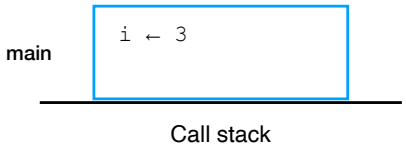

```
#include <stdio.h>

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

Storage Duration: Automatic



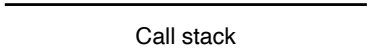

```
#include <stdio.h>

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

Storage Duration: Automatic




```
#include <stdio.h>

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

Storage Duration: Automatic




```
#include <stdio.h>

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

`i` goes away when `main` does.

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

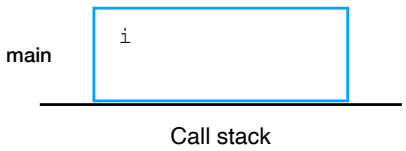
To **deallocate**, you must call **`free`**

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

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

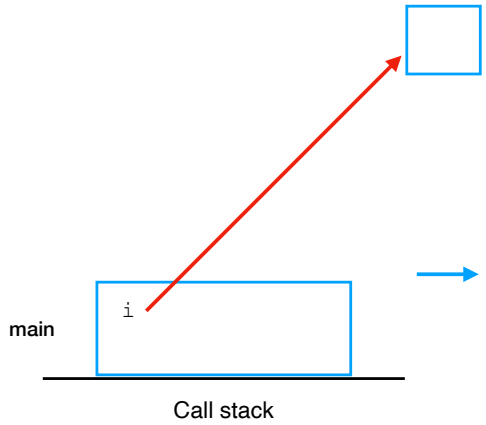
Failing to free when you are done is a **bug** called a ***memory leak***.

Storage Duration: Allocated



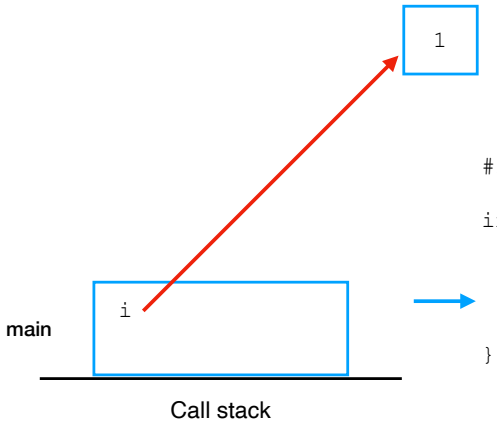

```
#include <stdio.h>

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

Storage Duration: Allocated



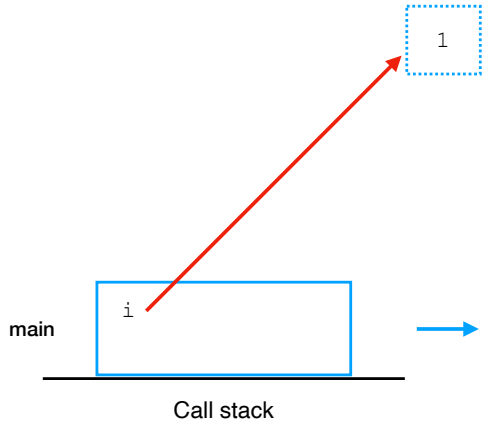

```
#include <stdio.h>

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.

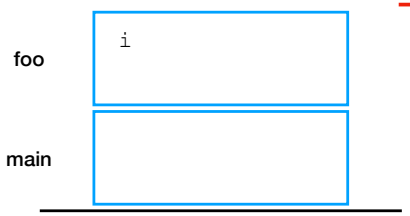
`free` **does not nullify** the pointer stored in `i`.

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

Person example

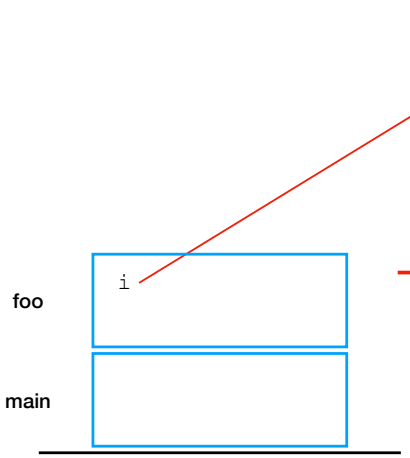
```
#include <stdio.h>
#include <stdlib.h>

struct person {
  char* fname;
  char* lname;
};

struct person* makePerson(char* first_name, char* last_name) {
  struct person* p = malloc(sizeof(struct person));
  if (!p) {
    fprintf(stderr, "Unable to allocate person.\n");
    exit(1);
  }
  p->fname = first_name;
  p->lname = last_name;
  return p;
}

int main(int argc, char** argv) {
  if (argc != 3) {
    printf("Usage:\n\t./person <first name> <last name>\n");
    exit(1);
  }

  struct person* p = makePerson(argv[1], argv[2]);
  printf("First name = '%s', Last name = '%s'\n", p->fname, p->lname);
  free(p);
  return 0;
}
```

Allocated Duration Pitfalls



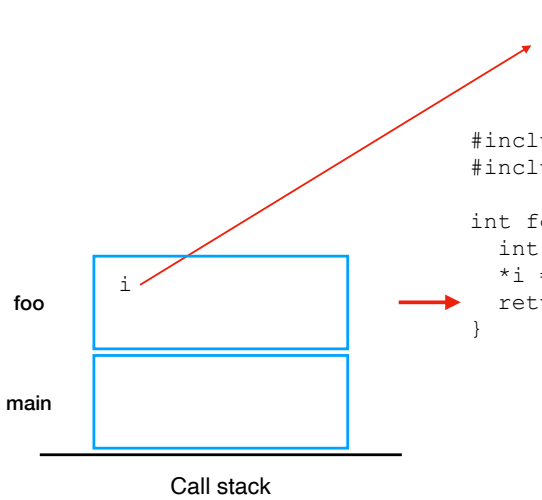

```
#include <stdio.h>
#include <stdlib.h>

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

Allocated Duration Pitfalls



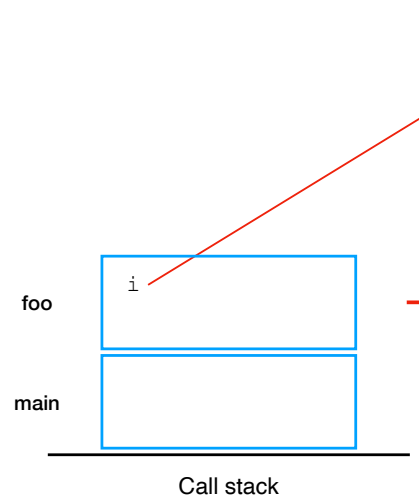

```
#include <stdio.h>
#include <stdlib.h>

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

Allocated Duration Pitfalls




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

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

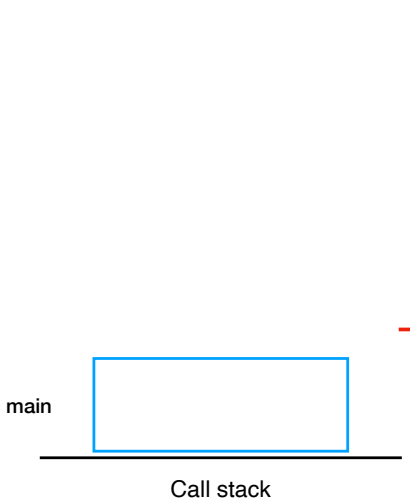
Allocated Duration Pitfalls




```
#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)

<u>Examples:</u>

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

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

main

x

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

main

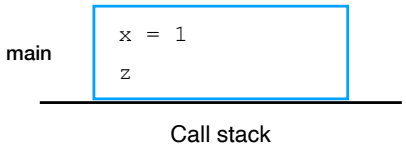
x = 1

z

Call stack

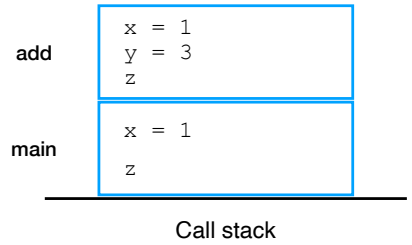
```
#include <stdio.h>

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




```
#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;
}
```

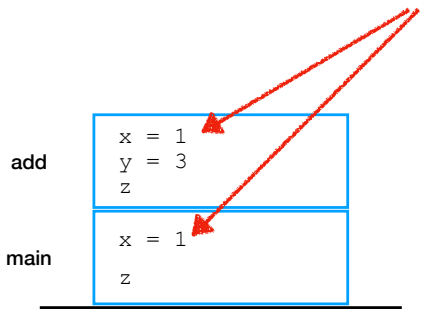
Call-by-value




```
#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;
}
```

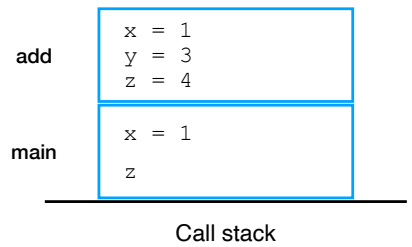
Call-by-value

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;
}
```

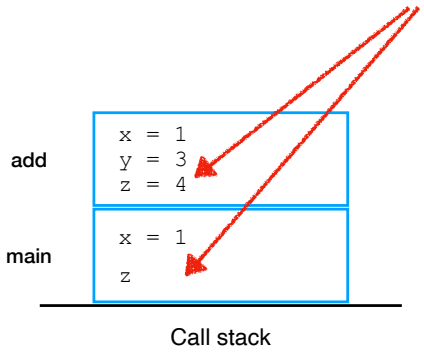
Call-by-value




```
#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;
}
```

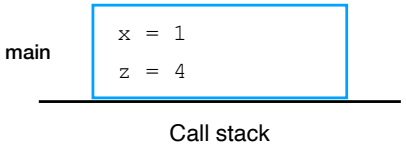
Call-by-value

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;
}
```

Recap & Next Class

Today:

Storage duration

Call-by-value evaluation

Next class:

Pointers: the key to abstraction

PL Fundamentals