CSCI 331: Introduction to Computer Security

Lecture 3: More C

Instructor: Dan Barowy

Williams

Announcements

- No CS Colloquium this week
- Instead, intro Women in CS event, Friday 2:35-4pm @ Eco Cafe (WITH SNACKS!!!)
- Lab 1/RR2 will be posted tonight.

Topics

Office hours today: 4-6pm in TBL 301

The Cuckoo's Egg discussion

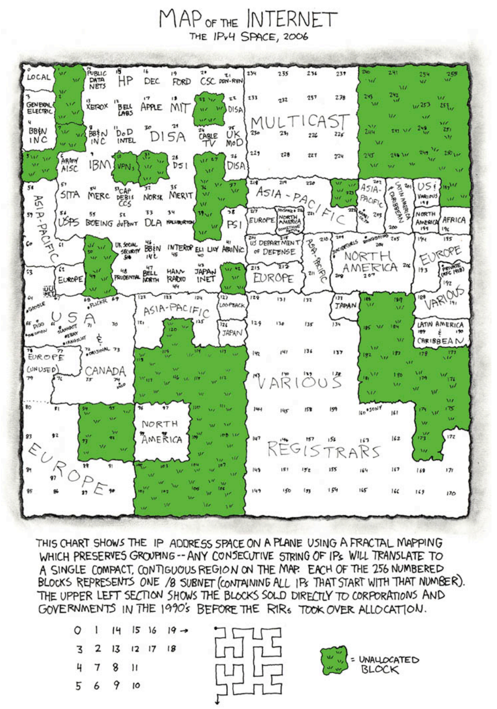
More C

Using a LaTeX template

Your to-dos

1. Lab 1 **out**.
   i. Note that it includes some reading.
   ii. Lab 1 **due Sunday 9/26** by **11:59pm**.
   iii. Be sure to get your RPi setup soon.
2. Reading response 2 (Schneier) **due Wed, 9/22**.
3. Keep on reading *The Cuckoo's Egg*.

Reading discussion



THIS CHART SHOWS THE IP ADDRESS SPACE ON A PLANE USING A FRACTAL MAPPING WHICH PRESERVES GROUPING -- ANY CONSECUTIVE STRING OF IPs WILL TRANSLATE TO A SINGLE COMPACT, CONTIGUOUS REGION ON THE MAP. EACH OF THE 256 NUMBERED BLOCKS REPRESENTS ONE /8 SUBNET (CONTAINING ALL IPs THAT START WITH THAT NUMBER). THE UPPER LEFT SECTION SHOWS THE BLOCKS SOLD DIRECTLY TO CORPORATIONS AND GOVERNMENTS IN THE 1990's BEFORE THE RIRs TOOK OVER ALLOCATION.

C rules from last class

0. Pointers are for **pointing at** other values in **memory**.
1. Whenever you **store** a **variable**, you **always** ask C to **reserve memory** for some **duration**.

Activity: What **effect** do these programs have on **memory**?

```c
#include <stdio.h>

int main() {
  int num = 331;
  printf("%d rocks!\n", num);
  return 0;
}
```

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

int main() {
  int *num_ptr = malloc(sizeof(int));
  if (!num_ptr) {
    printf("Unable to allocate.\n");
    exit(1);
  }
  *num_ptr = 331;
  printf("%d rocks!\n", *num_ptr);
  return 0;
}
```

Rule 2:

All long duration storage needs to be both **allocated** and **deallocated**.

Last class we spotted what was wrong here…

```c
free(num_ptr);
```

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

int main() {
  int *num_ptr = malloc(sizeof(int));
  if (!num_ptr) {
    printf("Unable to allocate.\n");
    exit(1);
  }
  *num_ptr = 331;
  printf("%d rocks!\n", *num_ptr);
  return 0;
}
```

Does this bug "matter" for this program?

Rule 3:

Always initialize variables.

What does this program print?

```c
#include <stdio.h>

int main() {
  int num;
  printf("%d rocks!\n", num);
  return 0;
}
```

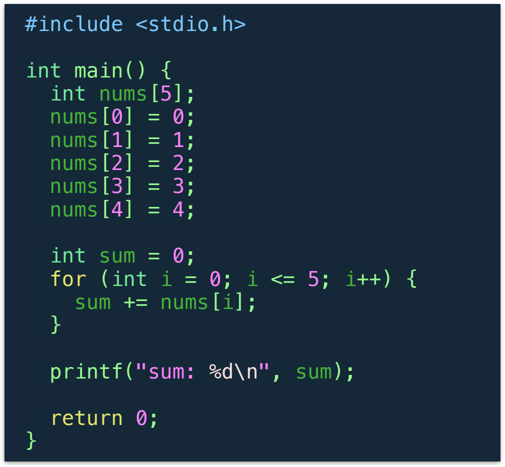
(always? are you sure?)

This program prints `"331 rocks!"`

```c
#include <stdio.h>

int foo() {
  int a = 331;
  return a;
}

int bar() {
  int b;
  return b;
}

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

Rule 4:

Watch out for **off-by-one** errors.



```
#include <stdio.h>

int main() {
  int nums[5];
  nums[0] = 0;
  nums[1] = 1;
  nums[2] = 2;
  nums[3] = 3;
  nums[4] = 4;

  int sum = 0;
  for (int i = 0; i <= 5; i++) {
    sum += nums[i];
  }

  printf("sum: %d\n", sum);

  return 0;
}
```

Effects range from **subtle** to **catastrophic**!

Rule 5:

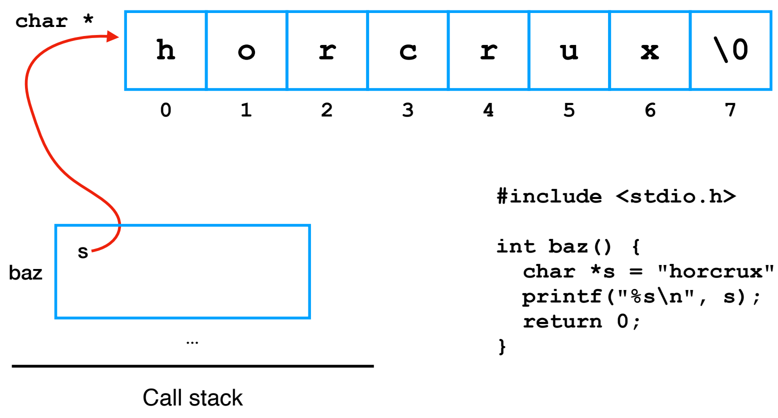
Always **null-terminate** "**C strings**."

C has **no `String` data type**.
Instead, it has **character arrays**.
Character arrays must always be **null-terminated**.

(otherwise **bad things** happen)

C Strings

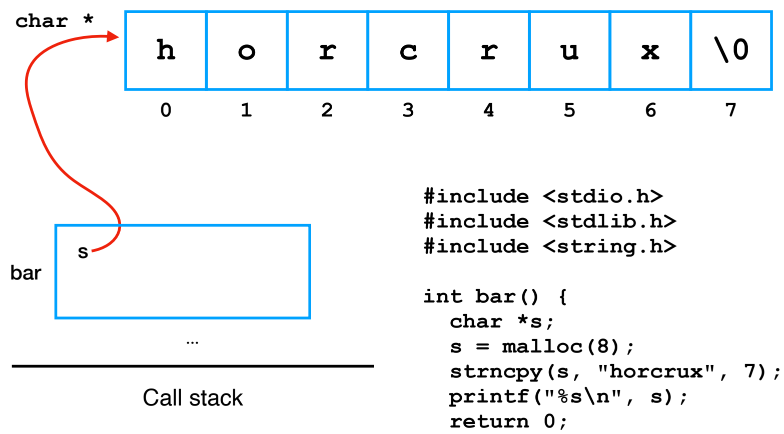
What is the type of s? What does s store? How do I know that s points to an array?
Where in memory does the data `"horcrux\0"` live?



```
#include <stdio.h>

int baz() {
  char *s = "horcrux";
  printf("%s\n", s);
  return 0;
}
```

String: just a null-terminated array of chars.
There is *no* string type in C.

C Memory

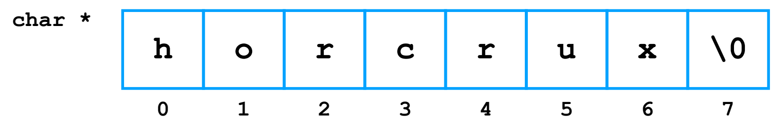


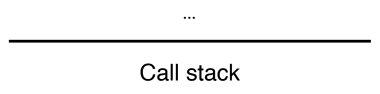
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int bar() {
  char *s;
  s = malloc(8);
  strncpy(s, "horcrux", 7);
  printf("%s\n", s);
  return 0;
}
```

What happens to `s` when `bar` returns?

What happens to the thing `s` pointed to?

C Memory





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

int bar() {
  char *s;
  s = malloc(8);
  strncpy(s, "horcrux", 7);
  printf("%s\n", s);
  return 0;
}
```

Answer: nothing. Memory leak!

C Rules

0. Pointers are for **pointing at** other values in **memory**.
1. Remember, when using a variable, you're **always** ask C to **reserve memory** for some **duration**.
2. **Always allocate** and **deallocate** long duration storage.
3. **Always initialize** variables.
4. **Watch out** for **off-by-one** errors.
5. **Always null-terminate** "C strings."

Recap & Next Class

Today we discussed:

The Cuckoo's Egg

More C

Next class:

Virtual memory

Segmentation Faults

Pseudoterminals