Lecture 04
Pointers

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

Slides adapted from:
Jesse Tov

Administrivia

- EX3 due today
- EX4 available
  - Slowing down. Not due until next week Tuesday
  - This is the last set of C exercises. They'll pick up again in week 6

- Quiz today
  - Setting an alarm for 3:00 pm

- Homework 1 due Thursday
  - Warning: **much** more work than the exercises are!

Gradescope demo

- Submitting code from terminal

- Seeing results in Gradescope
  - Be sure to either follow link or navigate to assignment again

- Can submit as many times as you want
  - We may later rate-limit your submissions
  - Later assignments WILL have hidden tests

  - Use the tests you fail on Gradescope to write your own tests!

Example Gradescope output

Unit test: charseq_length("abc") (0/0.5)

```
#Test: charseq_length("abc")
#Input:
abc
#Expected Output:
3
#Received Output:
0

#[X] FAILED
```

- Failure is that Expected and Received Output did not match

- You can duplicate this test locally, which is easier to fix!
  - Create a new test that runs `charseq_length()` on "`abc`"

Test code locally and submit to Gradescope when ready

- Just running `make` compiles *and* runs tests

- I'll recompile my code every few lines
  - That way there are never too many bugs to fix at once

- Then I make sure that I'm passing all the tests before uploading
  - And I add new tests whenever I see something weird I'm failing on Gradescope

Today’s Goals

- Introduce pointers in C
  - Why do they exist?
  - What are they useful for?
  - How do we use them?
  - How do they connect to arrays?

- Explore AddressSanitizer:
  - A tool that helps explain pointer errors

Getting files for today's lecture

```
cd ~/cs211/lec/
```
(or wherever you put stuff)

```
tar -xkvf ~cs211/lec/04_pointers.tgz
cd 04_pointers/
```

- A couple people asked for me to share the code from lectures
  - It's already shared! You can grab your own copy whenever
  - I included "finished" versions of code we write
  - Usually has working versions of code from slides too

Outline

- **Pointers**
  - **What are pointers?**
  - Why are pointers?
  - Pointers & Arrays
- Address Sanitizer
- Arguments to main

Remember: values, objects, and variables

- **Values** are the actual information we want to work with
  - Numbers, Strings, Images, etc.
  - Example: 3 is an `int` value

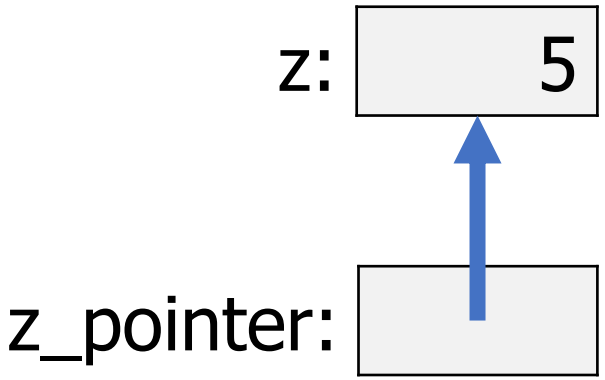
- An **object** is a chunk of memory that can hold a value of a particular type.
  - Example: function `f` has a parameter `int x`
    - Each type `f` is called, a "fresh" object that can hold an int is "created"

- A **variable** is the name of an object

- Assigning to a variable changes the *value* stored in the object named by the variable

Pointers are another type of value

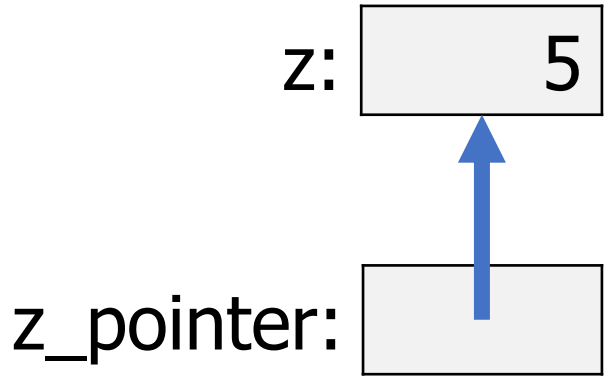
- Values could be a number, like 5 or 6.27
- Or they could be a "pointer" to an **object**
  - Points at the object, not the variable or value
  - It points at the "chunk of memory"
    - Technically, in C it holds the address of that memory



C syntax for pointers

- Pointers are a family of types
  - Each pointer is an existing C type, followed by a *
- To get the pointer to an existing variable, use the & operator
  - Returns the address of that variable
- Example:

```
int z = 5;
int* z_pointer = &z;
```



Longer pointer example

longer_pointers.c

```
1. double alpha;
```

What is the initial value of `alpha`?

Longer pointer example

longer_pointers.c

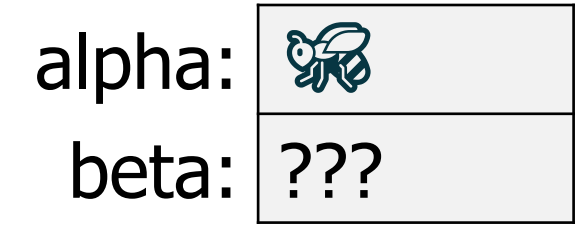
```
1. double alpha;
```

alpha: 

Longer pointer example

longer_pointers.c

```
1. double alpha;
2. double* beta;
```



What is the initial value of `beta`?

Longer pointer example

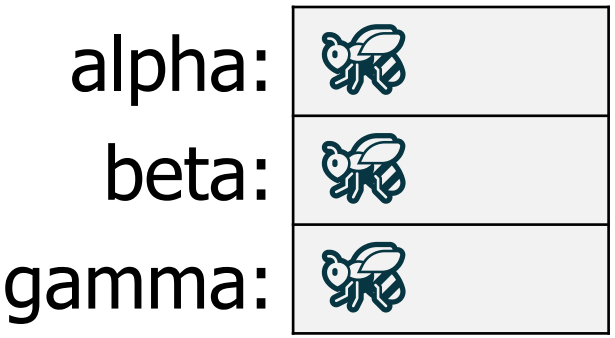
longer_pointers.c

```
1. double alpha;
2. double* beta;
```

Longer pointer example

longer_pointers.c

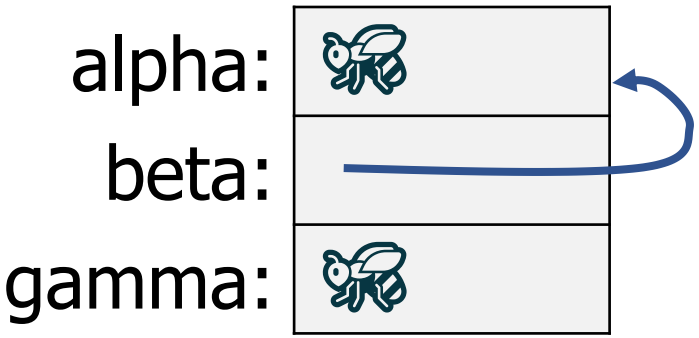
```
1. double alpha;
2. double* beta;
3. double* gamma;
```



Longer pointer example

longer_pointers.c

```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
```



Longer pointer example

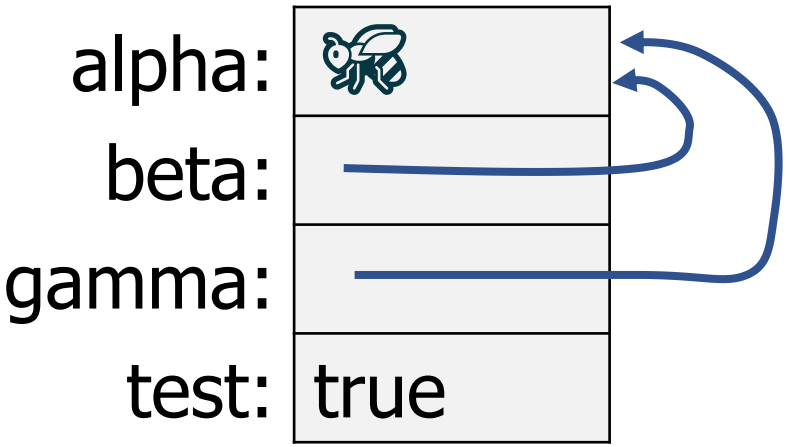
longer_pointers.c

```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
```

Longer pointer example

longer_pointers.c

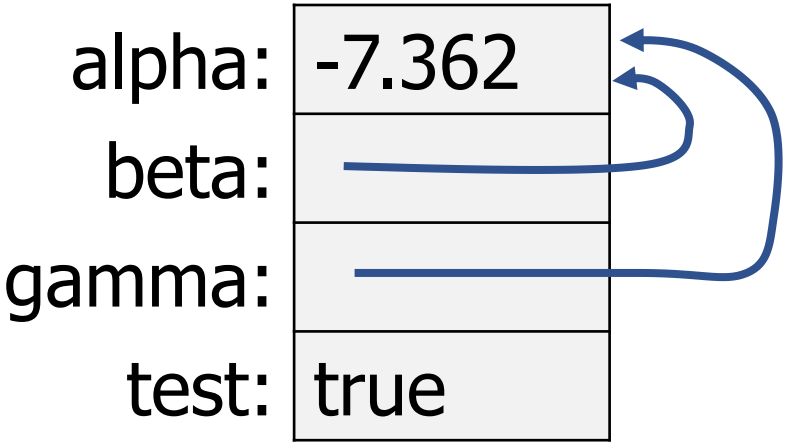
```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
6. bool test = (beta == gamma && beta == &alpha);
```



Longer pointer example

longer_pointers.c

```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
6. bool test = (beta == gamma && beta == &alpha);
7. alpha = -7.362;
```



Dereferencing a pointer

- Pointers can be used to read or modify the value in the object pointed at

- The * operator is used for getting/setting the value in the object
  - This is called “dereferencing” the pointer
  - Not multiply in this context

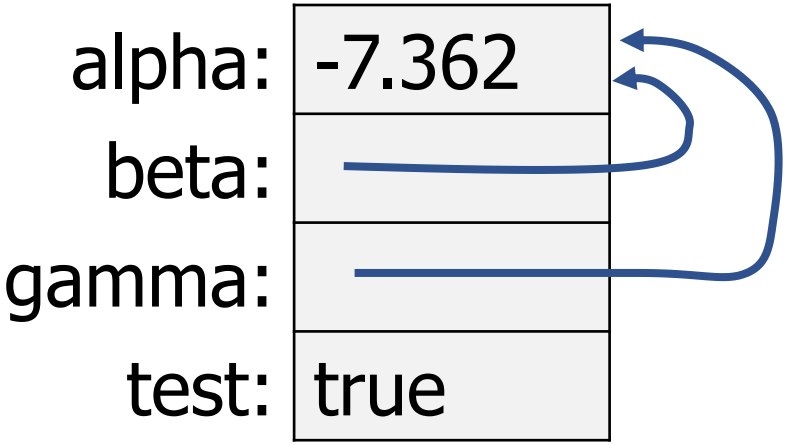
- Examples:

```
printf(“%d\n”, *my_int_pointer);

*my_int_pointer = 15;
```

Longer pointer example

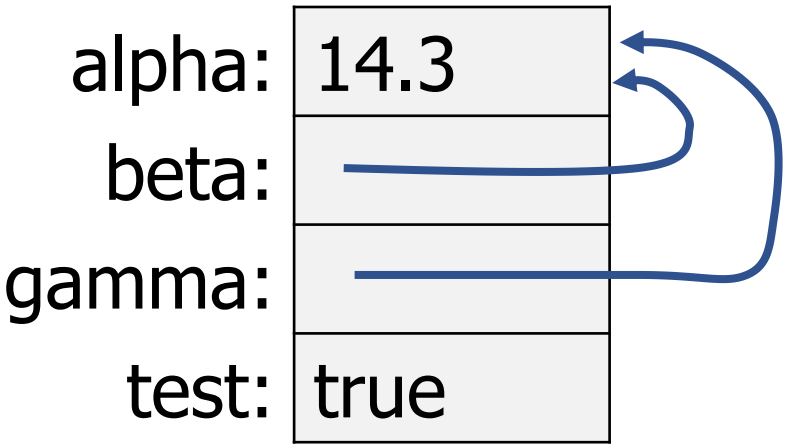
longer_pointers.c

```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
6. bool test = (beta == gamma && beta == &alpha);
7. alpha = -7.362;
8. test = (*beta < 0); // still true!
```



Longer pointer example

longer_pointers.c




```
1. double alpha;
2. double* beta;
3. double* gamma;
4. beta = &alpha;
5. gamma = &alpha;
6. bool test = (beta == gamma && beta == &alpha);
7. alpha = -7.362;
8. test = (*beta < 0);
9. *gamma = 14.3
```

Possible pointer values

- Uninitialized

```
unsigned long* zeta;
```

- Pointing at an existing object

```
char* letter_ptr = &my_char;
```

- Null (explicitly pointing at nothing)

```
int* p = NULL;
bool* b = NULL;
double* d = NULL;
```

  - NULL works for any pointer type
  - NULL is NOT the same as uninitialized ()
  - Dereferencing a null pointer is an error (segfault)

Some things to remember about pointers

1. Remember that a pointer is a type
   - int*, char*, short*, bool*, double*, size_t*, etc.

2. Think carefully about whether the pointer is being modified or the value in the object it points to
   - my_pointer = &x; // modifies which object we are pointing at
   - *my_pointer = x; // modifies the value in the object we are pointing at

3. Remember that pointer variables are themselves variables
   - They have values: the address of the object being pointed at
   - They name objects: memory is allocated to hold the address

Break + Question

```
int a = 15;
int* b = &a;
int* c = b;
*c = 7;
```

What are the values of:

```
a   =
*b  =
c   =
```

Break + Question

```
int a = 15;
int* b = &a;
int* c = b;
*c = 7;
```

What are the values of:

```
a    = 7        // set by *c=7
*b   =
c    =
```

Break + Question

```
int a = 15;
int* b = &a;
int* c = b;
*c = 7;
```

What are the values of:

```
a    = 7       // set by *c=7
*b   = 7       // points to value of a
c    =
```

Break + Question

```
int a = 15;
int* b = &a;
int* c = b;
*c = 7;
```

What are the values of:

```
a   = 7      // set by *c=7
*b  = 7      // points to value of a
c   = &a     // holds the address of a
```

C things that make pointers annoying

- For pointer types, the * doesn't have to be next to the type
  - These three all mean exactly the same thing:

```
1. int* x; // I strongly recommend you use this

2. int * x;

3. int *x;
```

C things that make pointers annoying

- For pointer types, the * doesn't have to be next to the type
  - These three all mean exactly the same thing:

```
1. int* x; // I strongly recommend you use this

2. int * x;

3. int *x;
```

- The * operator also means multiplication

```
signed long w = *t * *v; // multiply values referenced
                         // by the pointers t and v
```

Never define multiple variables at once

- You can define multiple variables at once in C

```
double x, y, radius;
```

Equivalent code:

```
double x;
double y;
double radius;
```

Never define multiple variables at once

- But this breaks when you're using pointers

```
double* x, y, radius;
```

Equivalent code:

```
double* x;
double y;
double radius;
```

} Not pointers!!! 😱

- To write that line correctly, you need to write:

```
double *x, *y, *radius;
```
OR
```
double * x, * y, * radius;
```

  - Or just never ever declare multiple variables in the same line!
    - That's the CS211 style rule

Full CS211 C style guidelines

- https://nu-cs211.github.io/cs211-files/cstyle.html

- Read them and make sure you follow them for homework
  - 5-10% of your grade for each homework is based on style
  - We'll be gentler about it on this first homework

Outline

- **Pointers**
  - What are pointers?
  - **Why are pointers?**
  - Pointers & Arrays
- Address Sanitizer
- Arguments to main

Pointers functions directly modify values inside variables

- Normally, functions get a copy of the value inside the variable

- With pointers, functions can directly modify the variable
  - The function gets a copy of the pointer to the variable

Example programming

add-starter.c

1. Add two to a variable with and without pointers

Adding two to a variable WITHOUT pointers

add_without_pointers.c

```
int add_two(int n) {
   return n+2;
}

int main(void) {
   int x = 15;
   x = add_two(x);
   printf("%d\n", x);
   return 0;
}
```

Adding two to a variable WITH pointers

add_with_pointers.c

```
void add_two(int* n) {
   *n += 2;
}

int main(void) {
   int x = 15;
   add_two(&x);
   printf("%d\n", x);
   return 0;
}
```

Side-by-side comparison of without/with pointers

```c
int add_two(int n) {
  return n+2;
}

int main(void) {
  int x = 15;
  x = add_two(x);
  printf("%d\n", x);
  return 0;
}
```

```c
void add_two(int* n) {
  *n += 2;
}

int main(void) {
  int x = 15;
  add_two(&x);
  printf("%d\n", x);
  return 0;
}
```

Example programming

struct-starter.c

1. Add two to a variable with and without pointers

2. Use pointers to initialize a struct

Another example: what if we want to pass a struct

struct_with_pointers.c

```c
typedef struct plants {
  bool is_watered;
  double height;
  int num_leaves;
} plant_t;
```

```c
void initialize_oak_tree(plant_t* plant){
  (*plant).is_watered = true;
  (*plant).height = 10;
  (*plant).num_leaves = 100000;
}

int main(void){
  plant_t plant_a;
  initialize_oak_tree(&plant_a);
  return 0;
}
```

Shortcut for pointers to structs

- C programs end up using pointers to structs A LOT

- It's annoying to type (*struct).field all the time
  - So we made a shortcut. These two mean exactly the same thing:

```
(*struct).field

struct->field
```

(that's dash and greater than)

  - This is known as "syntactic sugar"
    - Bonus syntax to make common things easier

Example programming

struct-starter.c

1. Add two to a variable with and without pointers

2. Use pointers to initialize a struct

3. Use pointers to print a struct

Adding a function to print the struct

struct_with_pointers.c

```c
typedef struct plants {
  bool is_watered;
  double height;
  int num_leaves;
} plant_t;
```

```c
void initialize_oak_tree(plant_t* plant){
  (*plant).is_watered = true;
  (*plant).height = 10;
  (*plant).num_leaves = 100000;
}

void print_plant(plant_t* plant){
  printf("Plant is %d meters tall and "
         "has %d leaves.\n",
         plant->height, plant->num_leaves);

  if (!plant->watered) {
    printf("\tIt needs to be watered!\n");
  }
}
```

Scanf example

- `scanf()` uses pointers to write to the variables you pass it

```
int x = 0;
int count = scanf(“%d”, &x);
```

  - Pointers allow `scanf()` to read results directly into your variable
  - Pointers also `scanf()` to simultaneously return the number of arguments matched

Break + Question

```
double x = 7.0;
double* xptr = &x;
*xptr += 3.0;
x = x / 4.0;
printf("%f\n", *xptr);
```

What value prints?

Break + Question

```
double x = 7.0;
double* xptr = &x;
*xptr += 3.0;
x = x / 4.0;
printf("%f\n", *xptr);
```

What value prints? **2.5**

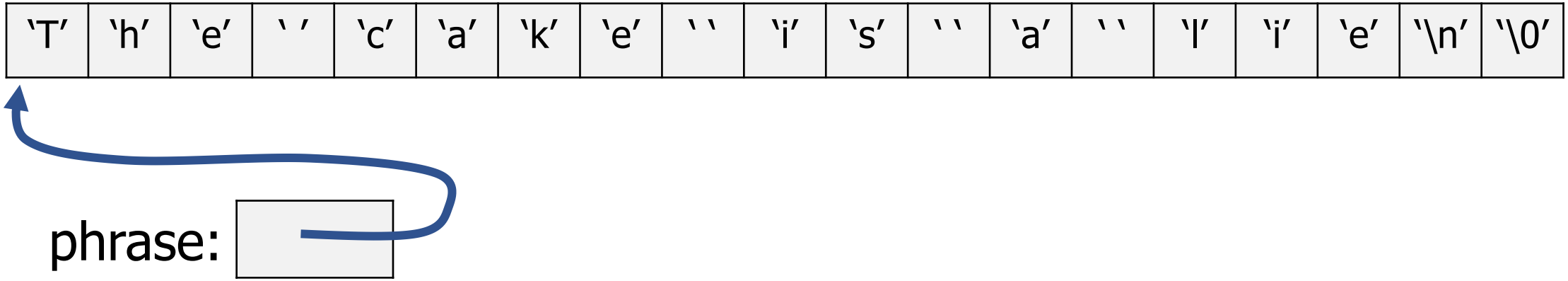
Outline

- **Pointers**
  - What are pointers?
  - Why are pointers?
  - **Pointers & Arrays**
- Address Sanitizer
- Arguments to main

Reminder: arrays and strings

```
int array_x[5] = {1, 2, 3, 4, 5};
```


```
const char* phrase = "The cake is a lie";
```



The name of the array is like a pointer to the first element

- You can treat the name of the array like a pointer
  - It basically is one

- You could dereference it, and you'll get the value in the first slot of the array

- Two ramifications of this:
  - You can't pass arrays into functions, only pointers

  - Array indexing is identical to pointer arithmetic

Arrays passed into functions are just pointers

array_print.c

- When you pass an array into a function, you don't pass a copy of the values
  - Instead you pass a pointer to the start of the array
  - Be sure to pass a length as well! (no way to determine that in C)

```
void print_array(int* values, int count) {
  . . .
}

int main(void) {
  int array[10] = {1, 2, 3, 4, 5, 5, 4, 3, 2, 1};
  print_array(array, 10);
  return 0;
}
```

Square brackets are the same as adding to the pointer

- Indexing into arrays is just adding to the pointer value
  - Example, these two are equivalent:

```
array[10]              // array indexing

*(array+10)            // pointer arithmetic
```

  - As are these two: (both result in a pointer)

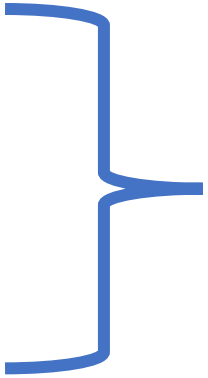
```
&(array[7])

array+7
```

A note on writing meaningful code

- Technically, NULL pointers and null terminators are both implemented as a value zero (on any modern system)
  - `false` is implemented as zero as well
  - So, technically, you could use any to mean any

- But humans will be the ones reading your code
  - NULL '\0', 0, and `false` all have different meanings

  - NULL means pointers
  - '\0' means the end of strings
  - `false` means a Boolean value
  - 0 means a number



Use the one that is appropriate to the situation!

Outline

- Pointers
  - What are pointers?
  - Why are pointers?
  - Pointers & Arrays
- **Address Sanitizer**
- Arguments to main

DANGER! Nothing stops you from going past the end of an array

array_print.c

- C does not check whether your array accesses are valid
  - It just tries to grab the value in the memory you asked for
- Going past the end (or before the beginning) of an array is **UNDEFINED BEHAVIOR**
  - Could result in *anything* happening
- If you're lucky, the code will crash
  - But you will not always get lucky
  - Be sure to always check if you're going past the end of the array

Address Sanitizer

- Automatically compiled in as part of your homework code

- Checks various accesses to memory for validity
  - Produces long error messages that can be scary at first! But are really helpful!

  - Error locations: (more on these "locations" on Thursday)
    - Stack – local variable
    - Global – global variable (usually a string)
    - Heap – variable created with `malloc()`

  - Error types:
    - buffer-overflow – past the end of an array of memory
    - buffer-underflow – before the beginning of an array of memory (rare)
    - various others

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn’t fit on the slide)
```

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn't fit on the slide)
```

Error is coming from AddressSanitizer

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn't fit on the slide)
```

Heap-buffer-overflow means past the end of an array created with `malloc()`

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn't fit on the slide)
```

The error happened in `expand_charseq()` in src/translate.c line 74

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn't fit on the slide)
```

Full "stack trace" of functions that were called to get to where the error happened

Example address sanitizer error

```
=================================================================
==238==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000016 at pc 0x55a44c0d8243
bp 0x7ffd8caf8c10 sp 0x7ffd8caf8c00
WRITE of size 1 at 0x602000000016 thread T0
SCARINESS: 31 (1-byte-write-heap-buffer-overflow)
    #0 0x55a44c0d8242 in expand_charseq src/translate.c:74
    #1 0x55a44c0d6c23 in gr_expand_charseq harness/hw02_tester.c:37
    #2 0x55a44c0d7394 in main harness/tester.c:28
    #3 0x7fa42386fbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x55a44c0d6699 in _start (/autograder/source/compile/tester+0x4699)

0x602000000016 is located 0 bytes to the right of 6-byte region [0x602000000010,0x602000000016)
allocated by thread T0 here:
    #0 0x7fa4248b8c68 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x10bc68)
    #1 0x55a44c0d8006 in expand_charseq src/translate.c:62

SUMMARY: AddressSanitizer: heap-buffer-overflow src/translate.c:74 in expand_charseq
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  . . .
  (more here that wouldn't fit on the slide)
```

Where the array was created in the first place (`expand_charseq()` in translate.c line 62)

Live demos of AddressSanitizer

- array_print.c

- string_print.c

Where the error happened may not but where the bug is

- AddressSanitizer usually points to a line where the array is being accessed

- But the bug is often because an index is out of bounds
- Or because the pointer passed in was invalid to begin with

- This is a new class of problem you'll all have to deal with
  - Errors that occur because of bugs elsewhere

Other AddressSanitizer errors

string_print.c

- Dereferencing a NULL pointer

```
src/string_print.c:4:28: runtime error: load of null pointer of type 'const char'
AddressSanitizer:DEADLYSIGNAL
=================================================================
==2838978==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x000000400912 bp
0x000000000000 sp 0x7ffe1379cec0 T0)
==2838978==The signal is caused by a READ memory access.
==2838978==Hint: address points to the zero page.
SCARINESS: 10 (null-deref)
    #0 0x400911 in print_string_chars src/string_print.c:4
    #1 0x400a33 in main src/string_print.c:12
    #2 0x7fefdbf5a492 in __libc_start_main ../csu/libc-start.c:314
    #3 0x40082d in _start (/home/branden/cs211/f21/lec/04_arrays_strings/string_print+0x40082d)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV src/string_print.c:4 in print_string_chars
==2838978==ABORTING
```

Outline

- Pointers
  - What are pointers?
  - Why are pointers?
  - Pointers & Arrays
- Address Sanitizer
- **Arguments to main**

Passing arguments to main

- We've been using "`int main(void);`" as `main()`'s signature

- Actually, `main()` can receive arguments, which are what the user called the program with

```
% ./programname arg1 arg2 arg3
```

Real signature for main

- The real signature for `main()` is:

```
int main(int argc, char* argv[]);
```

- `argc` – the number of strings in `argv` (length of `argv`)
- `argv` – an array of strings (array of char*)
  - The first string is the name of the program itself
  - The remaining strings are the arguments to the function

- By using `main(void)`, we've just been ignoring these
  - Which is fine, because they aren't always useful

Working with argv

argv_print.c

- Let's print out all the arguments to the function

```
int main(int argc, char* argv[]) {
  for (int i=0; i<argc; i++) {
    printf("Argument %d: \"%s\"\n", i, argv[i]);
  }

  return 0;
}
```

Outline

- Pointers
  - What are pointers?
  - Why are pointers?
  - Pointers & Arrays
- Address Sanitizer
- Arguments to main