Lecture 04
Arrays and Strings

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Winter 2022

Slides adapted from:
Jesse Tov

Regarding in-person classes returning

- As currently scheduled, we'll be back in person on Tuesday
  - Tech, LR2 (just left/North of the Ryan Auditorium)

- Rationale from the University is that classrooms are safe due to:
  - Vaccination mandate
  - Mask mandate
  - Testing strategy

- CS211 is going to roll with whatever we've got to do
  - We'll do our best to make sure you have an environment for learning
  - Definitely need buy-in from you all too
    - Wear masks, Don't come in when you're sick, Support your classmates

Administrivia

- Homework 1 is due today

- Lab03 will be released late tonight (due Monday)
    - Practice manipulating strings
    - Probably the *most useful* lab

- Homework 2 will be released late tonight (due next Thursday)
    - Lots of string manipulation
    - Get started early!

    - Partner assignment (work with 1 or 0 other people)
        - We put out a survey for people who want to be matched (see Campuswire)

    - Includes “hidden tests”

Always use Make, rather than calling the compiler yourself

- Make is our tool for compiling programs
  - It has rules for how to build the programs using the compiler

- You *could* compile your programs manually
  - But you would need to know the proper flags for the compiler to do so
  - Some programs rely on class-specific libraries for testing and memory management

  - This is a big pain, so just you `make` instead
    - And if you’re curious, you can look at the Makefile to see what the flags we’re providing are

Today’s Goals

- More practice with pointers and why they are useful

- Introduce Arrays and how they work
  - One variable that holds multiple values (like lists)
  - Related strongly with pointers

- Demonstrate Strings which are arrays of characters
  - How do they work in C?
  - How do we use them?

Getting the code for today

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

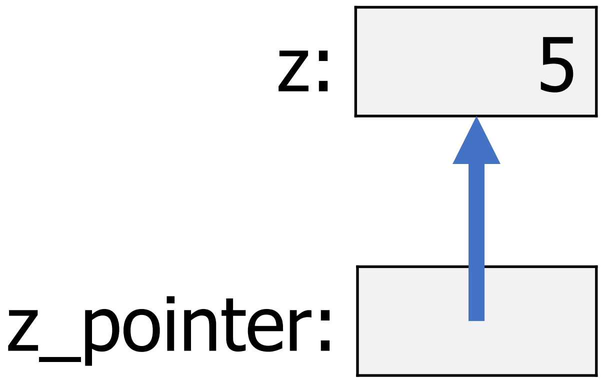
```
tar -xkvf ~cs211/lec/04_arrays_strings.tgz
cd 04_arrays_strings/
```

Outline

- **What are pointers?**
- Why are pointers?
- Arrays
- Characters
- Strings
- Arguments to main

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



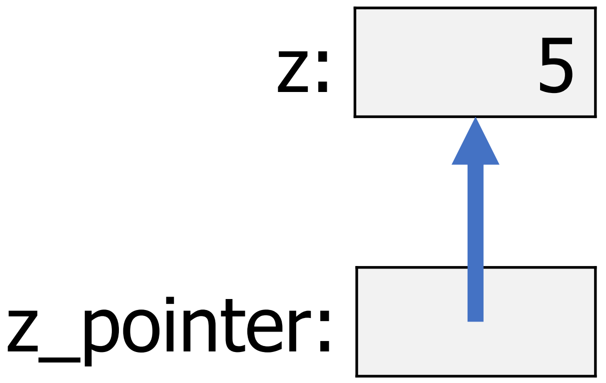
C syntax for pointers

- 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;
```



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

Outline

- What are pointers?
- **Why are pointers?**

- Arrays

- Characters
- Strings
- 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

Adding two to a variable WITHOUT pointers

pointers_examples/
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

pointers_examples/
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
void 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;
}
```

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
- `scanf()` also simultaneously returns the number of arguments matched
- For homework 1, for example, `scanf()` needs to match three inputs
  - Without pointers, you would only be able to match one

Another example: what if we want to pass a struct

pointers_examples/
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

Adding a function to print the struct

pointers_examples/
struct_with_pointers.c

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

```
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

- What are pointers?
- Why are pointers?
- **Arrays**
- Characters
- Strings
- Arguments to main

Array types

- Let’s talk about some ideas that really rely on the existence of pointers

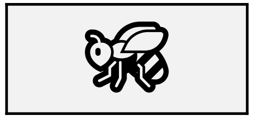
- The first of these is arrays
  - Arrays: a variable that holds multiple of a type

  - Example: one horizontal shelf
    - Can hold multiple books

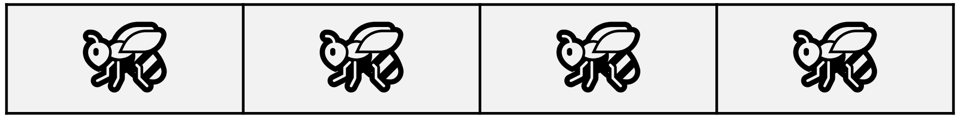
    - A shelf is an “array of books”

Arrays in C

```
int x;
```

x: 

```
int array_x[4];
```

array_x: 

Multiple **objects**
for a single **variable**,
each with their own **value**

- Generally:

  `type variable_name[N];` (array of type with length N)

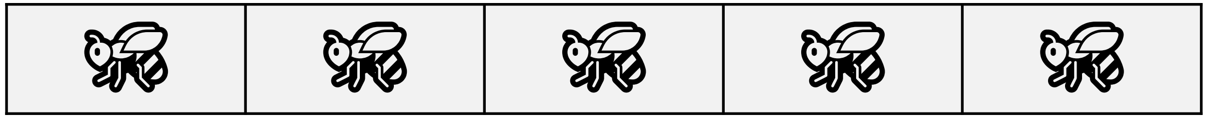
Working with values in arrays

- Every array has one or more objects, each with their own values
  - Like fields in a struct
- The “slots” in an array are numbered from zero
  - Arrays in C are zero-indexed

```
double values[3] = {1.2, -3.5623, 0.0};
double x = values[0];
```

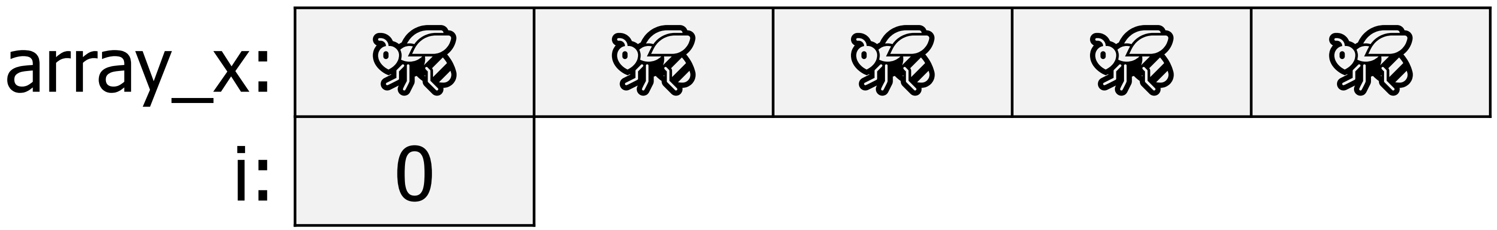
values:	1.2	-3.5623	0.0
x:	1.2		

Array assignment example

array_x: 

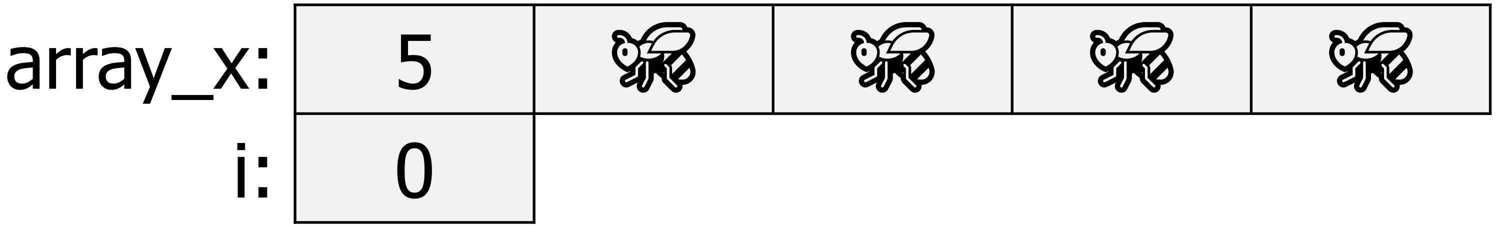
```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example



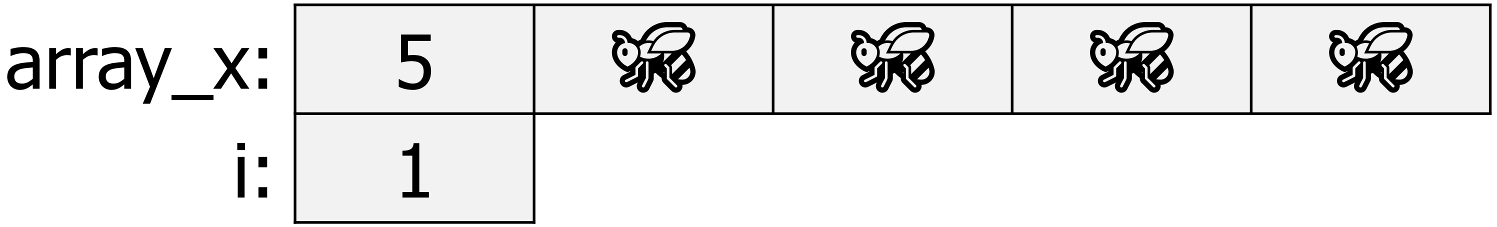

```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example



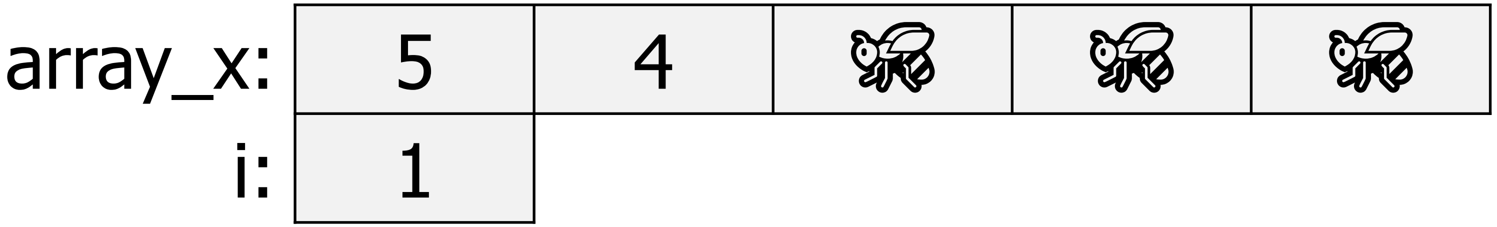

```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example



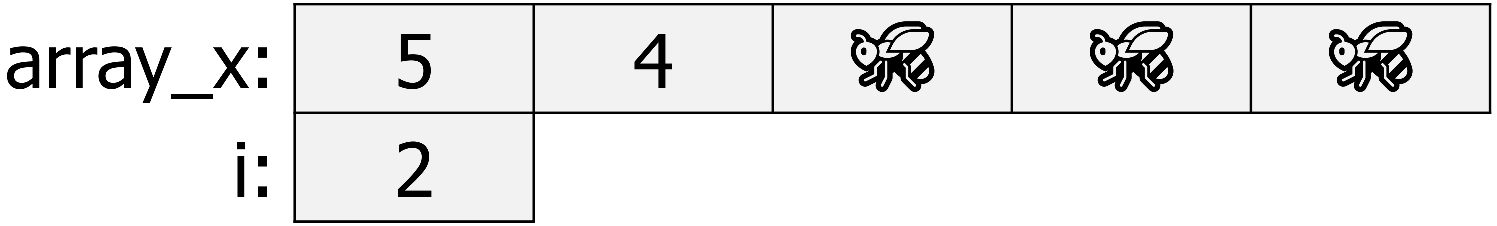

```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example




```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example



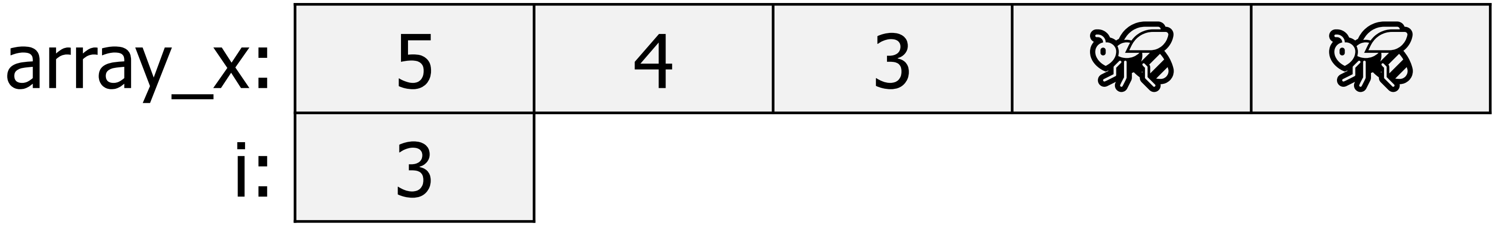

```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example

array_x:	5	4	3		
i:	2				

```
int data[5];
for (int i=0; i<5; i++) {
    data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example




```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example

array_x:	5	4	3	2	
i:	3				

```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example

array_x:	5	4	3	2	
i:	4				

```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example

array_x:	5	4	3	2	1
i:	4				

```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example

array_x:	5	4	3	2	1
i:	5				

```
int data[5];
for (int i=0; i<5; i++) {
   data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Array assignment example

array_x:				
5	4	3	2	9

```
int data[5];
for (int i=0; i<5; i++) {
  data[i] = 5-i;
}
data[4] = data[0] + data[1];
```

Remember `array[N-1]` is the last slot in an array of length `N`

Lengths of arrays

- How do you determine how long an array is?

- You cannot in C
  - Hopefully, you remember
  - Or someone told you

- This is an example of C giving you “full control”
  - Why bother storing the length of the array? That wastes memory

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-starter.c
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
```

Sidebar: Stopping a running program

- What do you do if a program has an infinite loop?
  - Or if it’s just taking a long time to run

- You can interrupt a running program with Ctrl+c
  - That will stop the program from running forcibly

- Irrelevant technical details
  - This sends the Interrupt signal to the program, which causes it to exit by default
  - See CS343 for more details

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

- 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

Ways of creating arrays

- Statically sized “local variable” (a variable inside a function)

```
int array[10];
```

- Dynamically sized local variable

```
int data_size;
scanf(“%d”, &data_size);
int data[data_size]; // probably should have checked
                     // the value in data_size first...
```

One more way to create arrays

- Using a library that gives you a chunk of memory for the objects

- Example

```
double* array = malloc(4 * sizeof(double));
```

  - `malloc()` returns a pointer to an amount of memory requested
  - `sizeof()` returns the size of a type in bytes
  - 4 slots, each of which can hold a double

  - MUCH more about malloc next week

C arrays cannot change length

- Once an array is created, its length cannot be changed
  - You cannot grow or shrink the number of slots

- You can make a whole new array that's bigger
  - Copy over elements from the old array

- `malloc()` and dynamic memory are a way to create new arrays
  - We'll talk about this more next week

Array of structs example

array_structs.c

- Arrays can be made of any type
  - int, float, bool, char, etc.
  - Also structs!

```
struct circle {
  double x;
  double y;
  double radius;
};

struct circle many_circles[5] = {0};
many_circles[1].x = 1;
many_circles[1].y = 1;
many_circles[1].radius = 2;
```

Special syntax to initialize all values as zero within the array. Only works for zero.

Break + Question

- Fill in the remaining code to sum an array in C

```
int sum_array(int* array, size_t length) {
  int sum = 0;
  for (size_t i=0; ________; ___) {
    sum += ________;
  }
  return sum;
}
```

Break + Question

- Fill in the remaining code to sum an array in C

```
int sum_array(int* array, size_t length) {
  int sum = 0;
  for (size_t i=0; i<length; i++) {
    sum += array[i];
  }
  return sum;
}
```

Outline

- What are pointers?
- Why are pointers?

- Arrays

- **Characters**
- Strings
- Arguments to main

Character types

- `char`, `signed char`, `unsigned char`
  - Capable of holding numbers from 0 to 255 or -128 to 127

- Also capable of holding single “characters”
  - A letter, a digit, a symbol

```
char letter = ‘a’;

char number = ‘1’;

char symbol = ‘~’;
```

MUST use single quotes in C when referring to characters

Characters are both numbers and letters

- How can a `char` hold either a letter or a number?
  - Each number represents a certain character
  - Example:
    - 33 is ‘!’
    - 65 is ‘A’
    - 66 is ‘B’
    - 97 is ‘a’
    - 50 is ‘2’
    - 51 is ‘3’
    - ‘2’ + ‘3’ == 101 (‘e’)

ASCII character encoding

- Mappings from number to letter
  - ASCII is one such mapping (https://www.asciitable.com/)
  - Maps American keyboard characters and symbols
    - Also special characters like tab, newline, or backspace

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p

Other encoding systems

- ASCII was made in 1961 and was never meant to encompass everything (American Standard Code for Information Interchange)

- Modern systems use Unicode
  - Which includes letters in other alphabets
    - 144762 characters from 159 modern and historic written languages

  - Also includes various symbols like emoji

  - Doesn't fit in a `char` though, that's only 256 options
    - We'll stick to simple ASCII for this class

Escape sequences

- The first part of the ASCII table was various special sequences
  - Most of which aren't relevant anymore, but some are
  - We need a way to type those "characters"
  - Also sometimes want to write normal characters that would break C syntax

- Escape sequences: \ followed by another symbol (only counts as one character)
  - Common examples:
    - `\n` – newline
    - `\t` – tab
    - `\\` – backslash
    - `\'` – single quote
    - `\"` – double quote

Dec	Hx	Oct	Char	
0	0	000	NUL	(null)
1	1	001	SOH	(start of heading)
2	2	002	STX	(start of text)
3	3	003	ETX	(end of text)
4	4	004	EOT	(end of transmission)
5	5	005	ENQ	(enquiry)
6	6	006	ACK	(acknowledge)
7	7	007	BEL	(bell)
8	8	010	BS	(backspace)
9	9	011	TAB	(horizontal tab)
10	A	012	LF	(NL line feed, new line)
11	B	013	VT	(vertical tab)
12	C	014	FF	(NP form feed, new page)
13	D	015	CR	(carriage return)
14	E	016	SO	(shift out)
15	F	017	SI	(shift in)
16	10	020	DLE	(data link escape)
17	11	021	DC1	(device control 1)
18	12	022	DC2	(device control 2)
19	13	023	DC3	(device control 3)
20	14	024	DC4	(device control 4)
21	15	025	NAK	(negative acknowledge)
22	16	026	SYN	(synchronous idle)
23	17	027	ETB	(end of trans. block)
24	18	030	CAN	(cancel)
25	19	031	EM	(end of medium)
26	1A	032	SUB	(substitute)
27	1B	033	ESC	(escape)
28	1C	034	FS	(file separator)
29	1D	035	GS	(group separator)
30	1E	036	RS	(record separator)
31	1F	037	US	(unit separator)

Outline

- What are pointers?
- Why are pointers?
- Arrays
- Characters
- **Strings**
- Arguments to main

Strings in C

- C strings are arrays of characters, ending with a null terminator
  - Null terminator: ‘\0’ character, which is the integer value zero
  - No relation to NULL pointers

- String literals in code are arrays of characters
  - And a ‘\0’ is placed at the end of them automatically

“Hello!\n”

MUST use double quotes in C when referring to strings

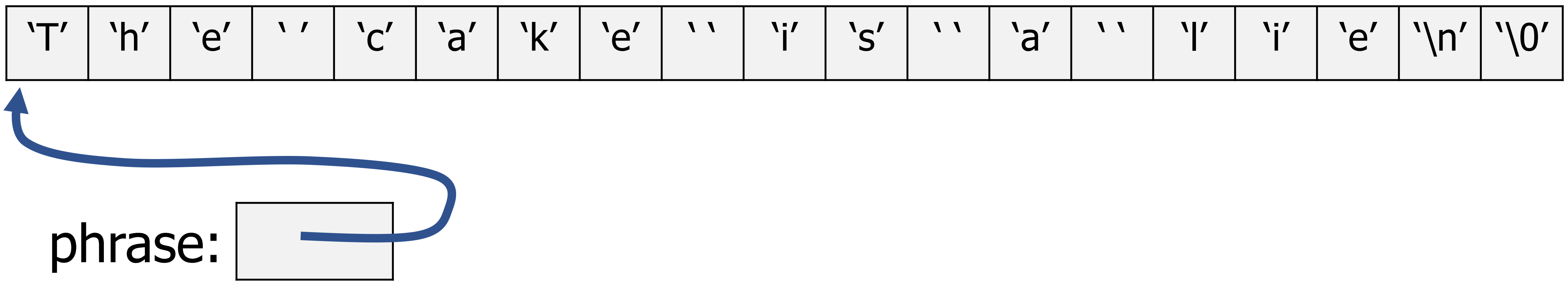
‘H’	‘e’	‘l’	‘l’	‘o’	‘!’	‘\n’	‘\0’

Working with strings

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

printf(“%s\n”, phrase);     // prints “The cake is a lie\n”
printf(“%c\n”, phrase[0]); // prints “T\n”


char letter = phrase[2];
```

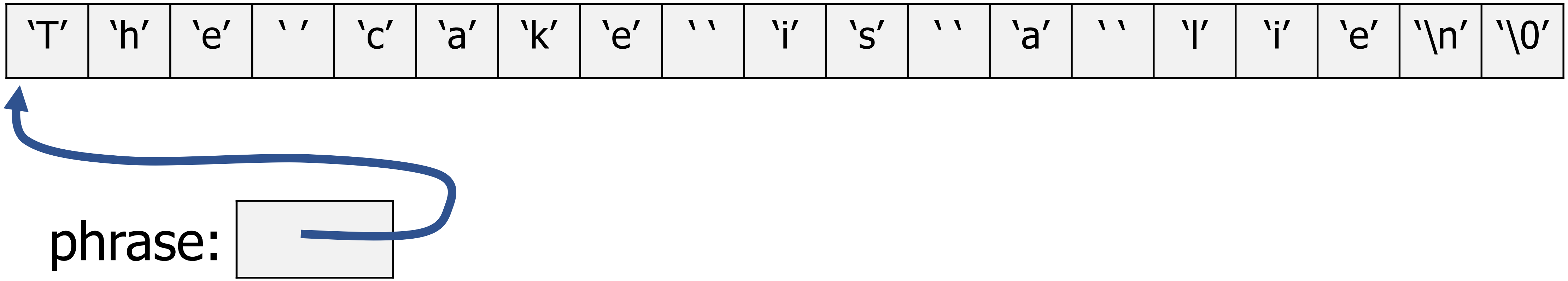


Working with strings

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

printf(“%s\n”, phrase);     // prints “The cake is a lie\n”
printf(“%c\n”, phrase[0]); // prints “T\n”


char letter = phrase[2];
```

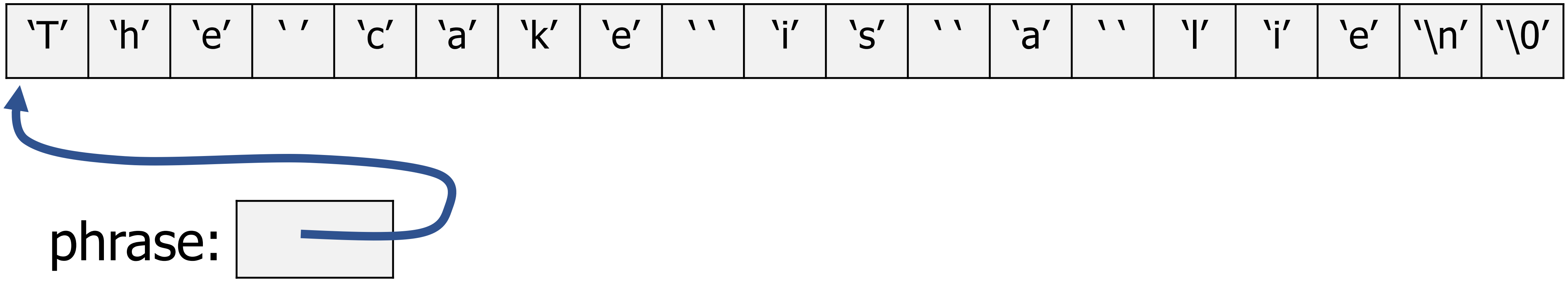


Working with strings

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

printf(“%s\n”, phrase);     // prints “The cake is a lie\n”
printf(“%c\n”, phrase[0]); // prints “T\n”

char letter = phrase[2];
```



Working with strings

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

printf(“%s\n”, phrase);     // prints “The cake is a lie\n”
printf(“%c\n”, phrase[0]); // prints “T\n”

char letter = phrase[2];
```

‘T’	‘h’	‘e’	‘ ’	‘c’	‘a’	‘k’	‘e’	‘ ‘	‘i’	‘s’	‘ ‘	‘a’	‘ ‘	‘l’	‘i’	‘e’	‘\n’	‘\0’

phrase: [ ]

letter: ‘e’

WARNING! Single quotes versus double quotes

- Single quotes mean single characters

```
'a'
'\n'
'&'
```

- Double quotes mean strings (zero or more characters)

```
"a"
"alpha"
""
"She-Ra is the best show ever!\n"
```

- Be really careful not to mix them up!
  - Especially because in many other languages they are identical
  - And the error message you'll get is hard to understand

The null terminator marks the end of the string

- So, strings are arrays of characters
- And there's no way to know the length of an array in C
- So how does `printf` know when to *stop* printing characters?

- It looks for the null terminator!

Iterating through a string

string_print.c

```
void print_string_chars(char* string) {
  for (size_t i=0; string[i] != '\0'; i++) {
    printf("String[%d] = '%c'\n", i, string[i]);
  }
}
```

- Note that we didn't need a length this time!
  - Just iterate until you find the null terminator

String literals cannot be modified

const_strings.c

- `const` in C marks a variable as constant (a.k.a. immutable)
  - Example:

```
const int x = 5;
x++; // Compilation error!
```

- String literals in C are of type `const char*`

```
const char* mystr = “Hello!\n”;
mystr[1] = ‘B’;  // Compilation error!
```

  - Just removing the “`const`” will result in a runtime crash instead…

Making modifiable strings

mutable_strings.c

Two options

1. Create a new character array with enough room for the string and then copy over characters from the string literal
   - Need to be sure to copy over the '\0' for it to be a valid string!

2. Initialize an array with a string literal

```
char mystr[] = "abc";
```

Creates a character array of length 4 ('a', 'b', 'c', and '\0')

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!

C has a library for working with strings

`#include <string.h>`

- https://www.cplusplus.com/reference/cstring/
  - Particularly useful:
    - `strlen()` finds the length of a string (not including null terminator)
    - `strcpy()` copies the characters of a string
    - `strcmp()` compares two strings to determine alphabetic order
      - Note: you cannot compare two strings with ==
      - That would just check if the pointers are the same

Outline

- What are pointers?
- Why are pointers?

- Arrays

- Characters
- Strings
- **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

- What are pointers?
- Why are pointers?
- Arrays
- Characters
- Strings
- Arguments to main

Outline

- Bonus: Variable Lifetimes

  (We'll get to this in class at some point, but I suspect not today)

When is a pointer “valid”?

1. If it is initialized

2. If the variable it is referencing still has a valid lifetime
    - Variables “live” until the end of the scope they were created in
    - Scopes are defined by { }

    - Example:

```
void some_function(void) {
    int a = 5;
}
```

`a` goes “out of scope” here
The variable stops being “alive”

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

a: 5

Examples of variable lifetimes

```c
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

a: 5

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

a: 5

Examples of variable lifetimes

```
int main(void) {
  int a = 5;
  printf("%d\n", a);

  return 0;
}
```

~~a~~: 

- Variable `a` is no longer "alive" at this point
  - It "poofs" out of existence
  - The variable is no longer valid

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

n: 17

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

n:	17
a:	5

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

n:	17
a:	5

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

n:	17
a:	5
b:	16

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```

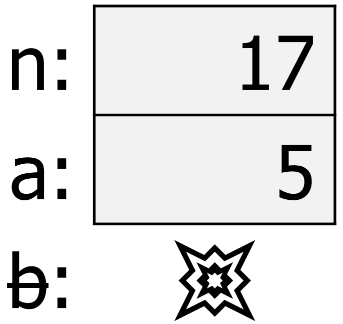
n:	17
a:	5
b:	16

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf(“%d\n” , b);
  }

  printf(“%d\n”, n);
}
```



Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

n:	17
a:	5

Referring to variable `b` at this point would be a compilation error

Lifetimes go from creation to end brace }

```
test(17);

void test(int n) {
  int a = 5;
  if (n >= a) {
    int b = 16;
    printf("%d\n" , b);
  }

  printf("%d\n", n);
}
```

~~n~~:

~~a~~:

Variable lifetimes are what makes loops work

- Variables created inside of loops only exist until the end of that iteration of the loop
  - i.e. they only exist until the next end curly brace }

```
while (n < 5) {
  int i = 1;
  n += i;
}
```

A new variable `i` is created each time the loop repeats

Dangling pointers reference invalid objects

```
int* get_pointer_to_value(void) {
  int n = 5;
  return &n;
}

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

Dangling pointers reference invalid objects

```
int* get_pointer_to_value(void) {
  int n = 5;
  return &n;
}

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

`n` goes out of scope at the end of this function

So what does the pointer point to???

Dangling pointers are especially dangerous

- Accessing a dangling pointer is *undefined behavior*
  - Anything could happen!

- If you are lucky: segmentation fault (a.k.a. SIGSEGV)
  - The OS kills your program because it accesses invalid memory

- If you are unlucky: *anything at all*
  - Including returning the correct result the first time you run it and an incorrect result the second time

String literals are an exception to scoping rules

string_lifetime.c

- String literals always exist
  - This is why they cannot be modified. They might be reused later

```
const char* get_pointer_to_string(void) {
  return “oh, hello!”; // this is okay for string literals
}

int main(void) {
  const char* string = get_pointer_to_string();
  printf(“%s\n”, string);
  return 0;
}
```