Lecture 03
Arrays and Strings

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Winter 2023

Slides adapted from:
Jesse Tov

Northwestern

Administrivia

- Lab1 is due today (76% of the class already done)
- EX2 is due today (71% of the class already done)

- EX3 available now (due Tuesday)

- Homework 1 will be released late tonight (due next Thursday)
  - Lots of string manipulation
  - Get started early!

Quiz 1 details

- Next week Tuesday during class
  - We’ll stop lecture near the end and give you fifteen minutes to work on it

- Bring a pencil
  - No notes allowed
  - No calculators, laptops, headphones, etc.

- Covers
  - Material from the first three lectures (includes today)
  - Won’t expect you to memorize shell commands

Today’s Goals

- Finally get to some miscellaneous C syntax we haven’t covered

- Introduce more complex types in C
  - Structs and Arrays

- 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/03_arrays_strings.tgz
cd 03_arrays_strings/
```

Outline

- **More C syntax**
  - **Iteration**
  - Miscellaneous syntax

- Complex data types
  - Structs
  - Arrays

- Text
  - Characters
  - Strings
  - Arguments to main

Definition of Fibonacci Function

- $fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$

n	**fib(n)**
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21

Implementing Fibonacci in C

fib.c

```c
long fib(int n){
 if (n < 2) {
  return n;
 } else {
  return fib(n - 2) + fib(n - 1);
 }
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Statements and Conditions aren’t enough

- Not all problems are easily solved with recursion

- C, like many programming languages, also has loops
    - Repeats the statements inside it until some condition is met

Iteration with the While Statement

- Syntax

```
while (⟨test-expression⟩) {
  ⟨body-statements⟩
}
```

- Semantics
  1. Evaluate `⟨test-expression⟩` to a `bool`
  2. If the `bool` is *false* then skip to the statement after the `while` loop
  3. Execute `⟨body-statements⟩` (if the `bool` was true)
  4. Go back to step 1

Let’s reimplement fib using a while loop

- To the shell!

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        n = n - 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

For loops

- For loops allow you to combine iteration and incrementing
    - When you write a for statement like this:

```
for (⟨start-decl⟩; ⟨test-expr⟩; ⟨step-expr⟩) {
  ⟨body-stms⟩
}
```

- It's as if you'd written this while statement:

```
{
  ⟨start-decl⟩;
  while (⟨test-expr⟩) {
    ⟨body-stms⟩
    ⟨step-expr⟩;
  }
}
```

Modify fib to use a for loop

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        n = n - 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Modify fib to use a for loop

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;
    int i = 0;

    while (i < n) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        i = i + 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Complete: modify fib to use a for loop

fib.c

```c
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    for (int i = 0; i < n; i = i + 1) {
        long prev = curr;

        curr = next;
        next = prev + curr;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Outline

- **More C syntax**
  - Iteration
  - **Miscellaneous syntax**
- Complex data types
  - Structs
  - Arrays
- Text
  - Characters
  - Strings
  - Arguments to main

C comments

- `//` means a single-line comment
- `/*` starts a multiline comment, which continues until `*/`

- How to use comments effectively
    - Comment “blocks” of code with their purpose
        - Every line is too much
        - Often helpful to write the comments before the code as planning
    - Comment tricky bits of code so you know what it means
        - You + several weeks = “what does that code mean?!”

Logical operators

- || &&
  - Logical OR, and Logical AND
  - `a < 5 && b > 12`

- !
  - Logical NOT
  - `!(a < 5)` equivalent to `(a >= 5)`

- ==
  - Equality test
  - `5 == 5` -> TRUE
  - `16 == -3` -> FALSE

  - Don’t mix it up with assignment (single equals sign)
    - Really common new C programmer mistake

Other operators you'll see around

- `+=   *=   -=   /=`
  - Perform the action of `VAR = VAR operator ARG`
  - `a += 5` -> `a = a + 5`
  - `a *= b` -> `a = a * b`

- %
  - Modulus operator
  - Returns the remainder of division
  - `12 % 10` -> 2

- ~ | & ^
  - Bitwise NOT, OR, AND, and XOR (you'll learn these in CS213)
  - Importantly, ^ is not exponentiation!!!

Adding and Subtracting one

- ++ --
  - Shorthand for plus 1 or minus 1
  - `++a` **->** `a += 1` **->** `a = a + 1`

- The auto-increment/decrement operators can go before or after the variable
  - (`--x`) subtracts one and returns the new value of x from the expression
  - (`x--`) subtracts one but returns the *old* value of x from the expression

  - Usually, this doesn't matter, unless you write complicated statements that combine assignment and conditions
  - `if (--x > 0)` … (please just never do this)

Implementing Fibonacci in C

fib.c

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    for (int i = 0; i < n; ++i) {  // i++ also works
        long prev = curr;

        curr = next;
        next = prev + curr;
    }

    return curr;
}
```

Break + Question

- What value will this code return when called as:
  - loop_function(6)
  - loop_function(5)
  - loop_function(3)

```
int loop_function(int test) {
  int retval = 0;
  while (test < 5) {
    retval++;
    test++;
  }
  return retval;
}
```

Break + Question

- What value will this code return when called as:
  - loop_function(6) **returns 0**
  - loop_function(5)
  - loop_function(3)

```
int loop_function(int test) {
  int retval = 0;
  while (test < 5) {
    retval++;
    test++;
  }
  return retval;
}
```

Break + Question

- What value will this code return when called as:
  - loop_function(6) **returns 0**
  - loop_function(5) **returns 0**
  - loop_function(3)

```
int loop_function(int test) {
  int retval = 0;
  while (test < 5) {
    retval++;
    test++;
  }
  return retval;
}
```

Break + Question

- What value will this code return when called as:
    - loop_function(6) **returns 0**
    - loop_function(5) **returns 0**
    - loop_function(3) **returns 2**

```
int loop_function(int test) {
  int retval = 0;
  while (test < 5) {
    retval++;
    test++;
  }
  return retval;
}
```

Outline

- More C syntax
  - Iteration
  - Miscellaneous syntax

- **Complex data types**
  - **Structs**
  - Arrays

- Text
  - Characters
  - Strings
  - Arguments to main

Working with more complex data

- Sometimes it makes sense to collect multiple variables together
  - Coordinate in 2D space: {x, y}
  - Multiple attributes that describe a user: Name, ID, Email

- Structs are a collection of fields, each of which has its own type and name
  - First, you define a type and what fields it has
  - Then, you can create a struct and initialize the fields

Struct definitions

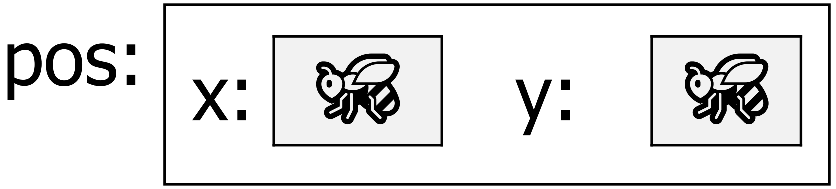
```
struct coordinate {
  int x;
  int y;
};
```

- Creates a new type that can be used in code “struct coordinate”
  - With fields “x” and “y” which are accessed with .
- Any type can be a struct field
  - int, unsigned int, char, double, another struct, array, etc.

Initializing a struct

```
struct coordinate pos; // uninitialized for now
```

- Can initialize fields individually

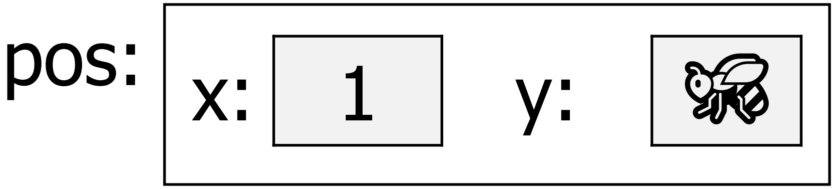


Initializing a struct

```
struct coordinate pos; // uninitialized for now
```

- Can initialize fields individually

```
pos.x = 1;
```

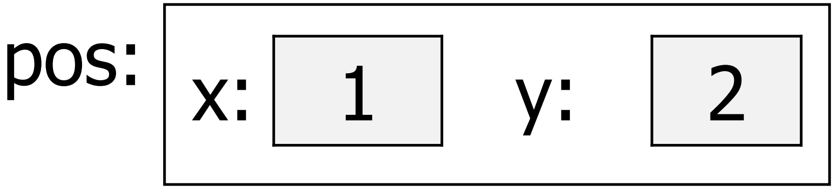

pos: x: 1 y: 🐞

(period operator accesses individual fields)

Initializing a struct

```
struct coordinate pos; // uninitialized for now
```

- Can initialize fields individually

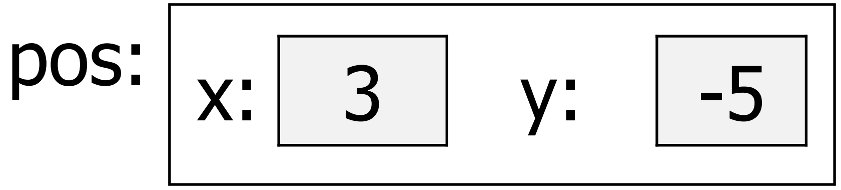


```
pos.x = 1;
pos.y = 2;
```

(period operator accesses individual fields)

Can initialize all fields of a struct at once

```
struct coordinate pos = {3, -5};
```



pos: x: 3 y: -5

OR

```
struct coordinate pos = {.x=3, .y=-5};
```

`typedef` can be used to make new C type names

- Typedef creates a new type name that is a copy of an existing type

- Typedef keyword is followed by two types
    - First type: the original type name
    - Second type: the new type name

- Example:

```
typedef int x_coordinate_t;
x_coordinate_t my_variable = 5;
```

Struct definitions usually use typedef

- Defining

```
typedef struct {
  int x;
  int y;
} coordinate_t;
```

- Initializing

```
coordinate_t pos = {1, 2};
```

Outline

- More C syntax
  - Iteration
  - Miscellaneous syntax
- **Complex data types**
  - Structs
  - **Arrays**
- Text
  - Characters
  - Strings
  - Arguments to main

Array types

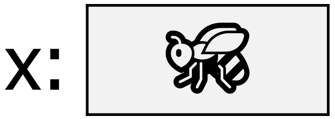
- Arrays are another way to store more complex data
  - They hold many instances of a single type

- Analogy: one horizontal shelf
  - Can hold multiple books

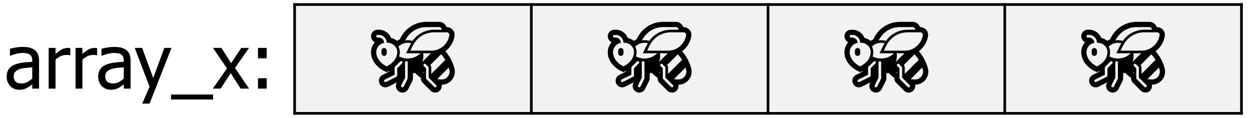
  - A shelf is an “array of books”

Arrays in C

```
int x;
```



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



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

- Generally:

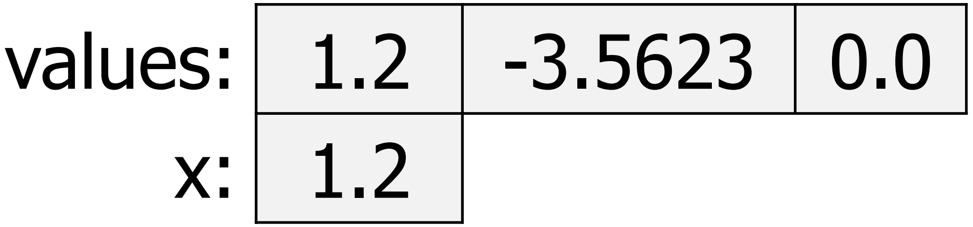
  `type variable_name[N];`   (array of type with length N)

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

array_x:	🐝	🐝	🐝	🐝	🐝
i:	0				

```
  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	🐝	🐝	🐝	🐝
i:	0				

```
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	🐝	🐝	🐝	🐝
i:	1				

```
  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	🐝	🐝	🐝
i:	1				

```
  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	🐝	🐝	🐝
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

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

array_x:	5	4	3	🐝	🐝
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:	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

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

Passing arrays into functions

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 the **location** of the start of the array (a pointer)
  - Be sure to pass a length as well! (no way to determine that in C)

```
void print_array(int* values, int count) {
  // can still access with square brackets
  // values[0] == 1, values[1] == 2, values[4] == 5, etc.
  . . .
}

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

Let’s print the contents of an array

- To the shell!

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 pointers and 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.

Struct with an array example

- Structs can hold any type
    - int, float, bool, char, etc.
    - Also arrays!

```
struct samples {
  int id;
  double data[100];
};

struct samples raw_samples = {0};
raw_samples.id = 5;
raw_samples.data[0] = 1.5;
```

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

- More C syntax
  - Iteration
  - Miscellaneous syntax

- Complex data types
  - Structs
  - Arrays

- **Text**
  - **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 = ‘~’;
```

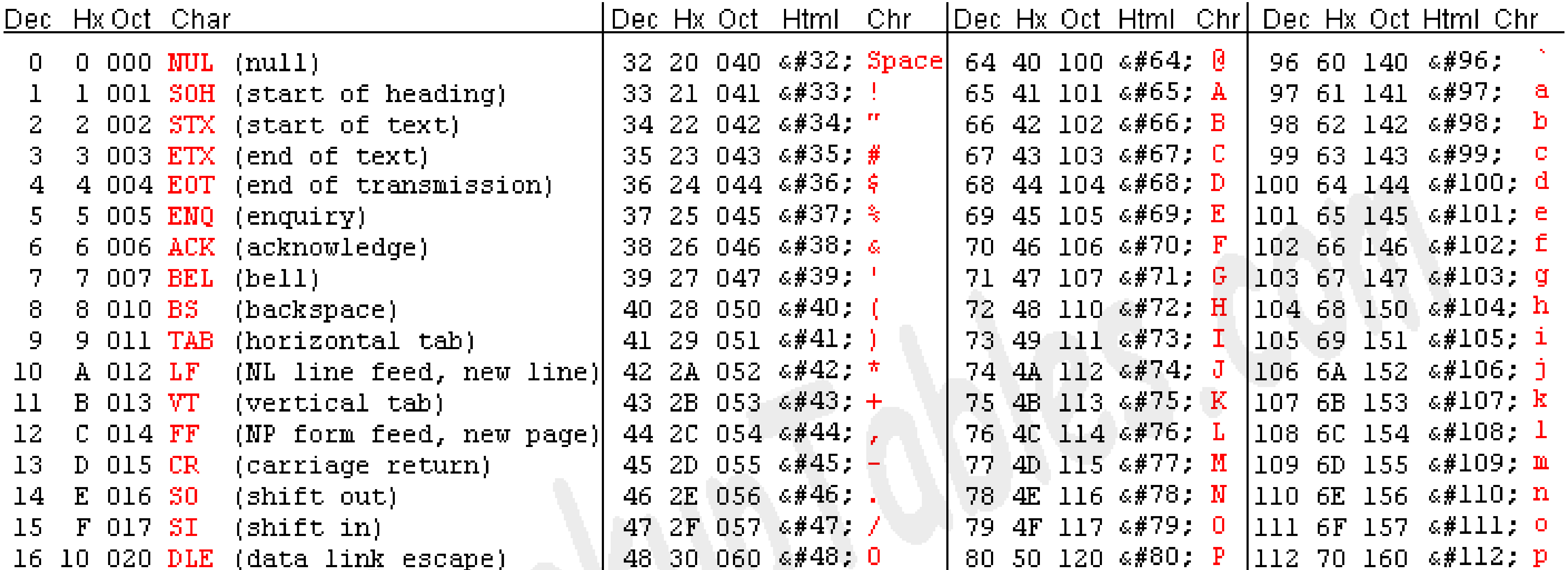
MUST use single quotes in C when referring to characters

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

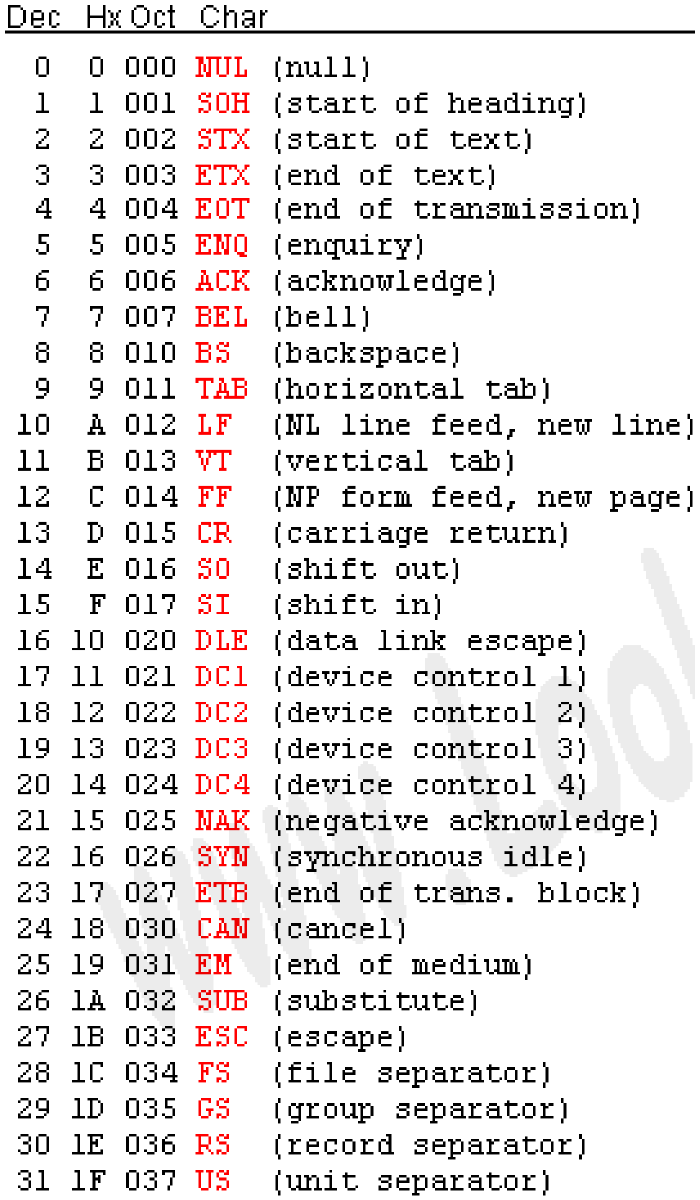
  - Doesn’t fit in a `char` though, that’s only 256 options
    - We’ll stick to simple ASCII for this class

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

- More C syntax
  - Iteration
  - Miscellaneous syntax
- Complex data types
  - Structs
  - Arrays
- **Text**
  - 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

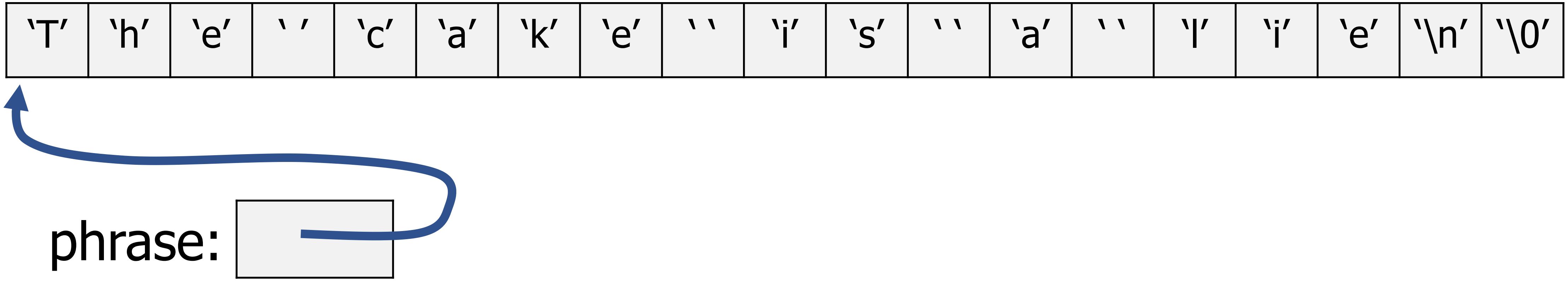
‘H’	‘e’	‘l’	‘l’	‘o’	‘!’	‘\n’	‘\0’

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];
```



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

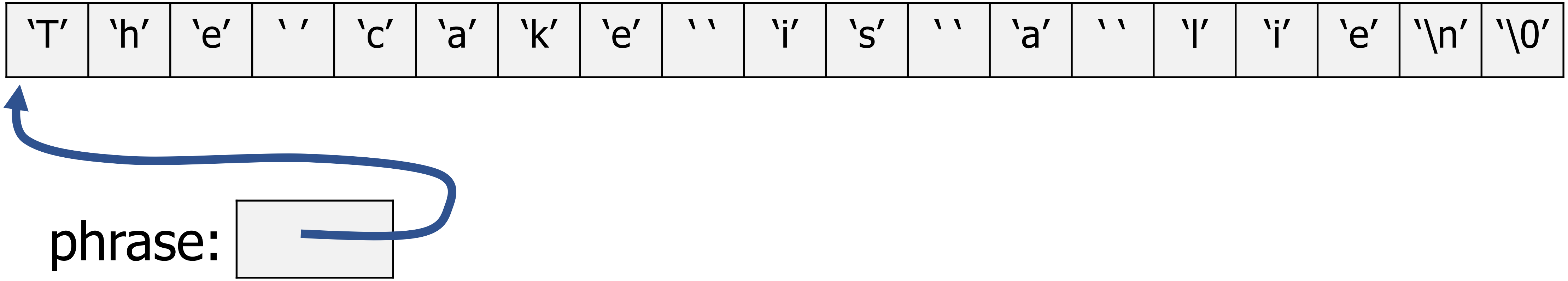
phrase:

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];
```



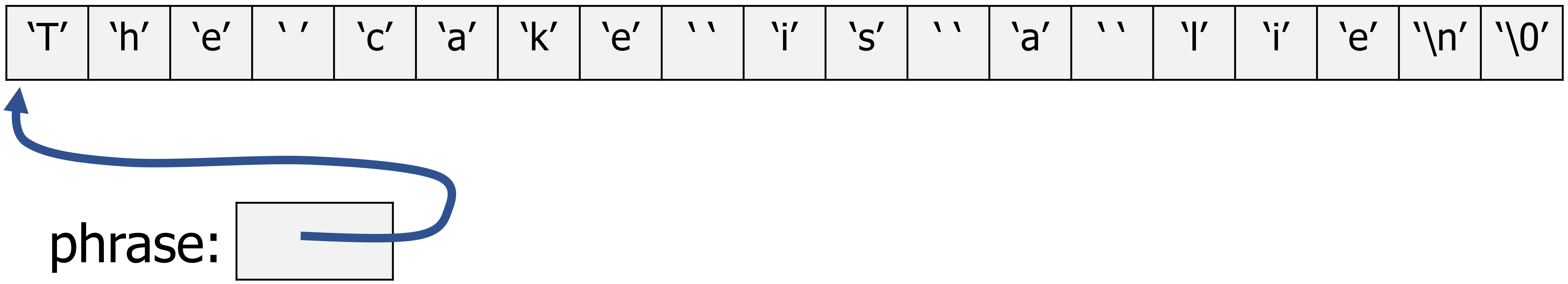
‘T’	‘h’	‘e’	‘ ’	‘c’	‘a’	‘k’	‘e’	‘ ‘	‘i’	‘s’	‘ ‘	‘a’	‘ ‘	‘l’	‘i’	‘e’	‘\n’	‘\0’

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’

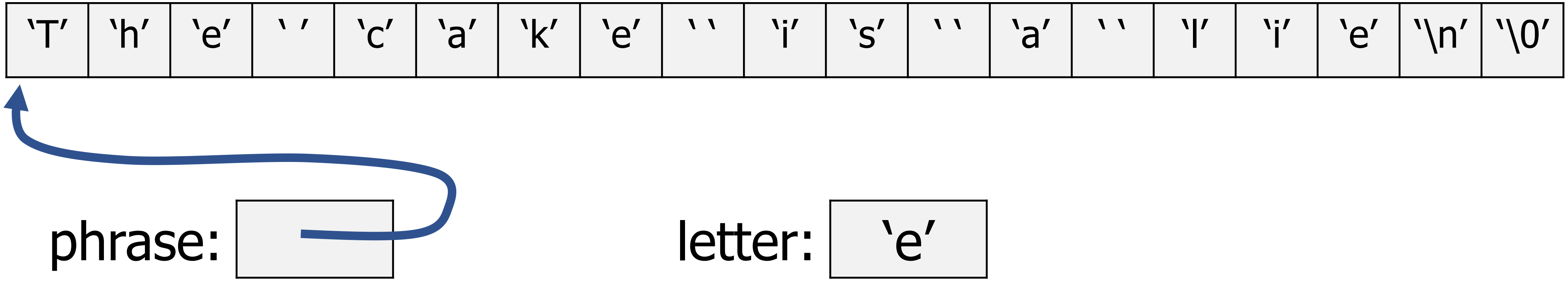
phrase:

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')

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

- More C syntax
  - Iteration
  - Miscellaneous syntax
- Complex data types
  - Structs
  - Arrays
- **Text**
  - 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

- More C syntax
  - Iteration
  - Miscellaneous syntax

- Complex data types
  - Structs
  - Arrays

- Text
  - Characters
  - Strings
  - Arguments to main