C, Pointers, Arrays, Strings

Introduction to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema
Andrew Rigas
Anisha Bhatia
Audrey Yang
Craig Lee
Daniel Duan
David LuoZhang
Eddy Yang
Ernest Ng
Heyi Liu
Janavi Chadha
Jason Hom
Katherine Wang
Kyrie Dowling
Mohamed Abaker
Noam Elul
Patricia Agnes
Patrick Kehinde Jr.
Ria Sharma
Sarah Luthra
Sofia Mouchtaris

How is/was your Halloween?

Logistics

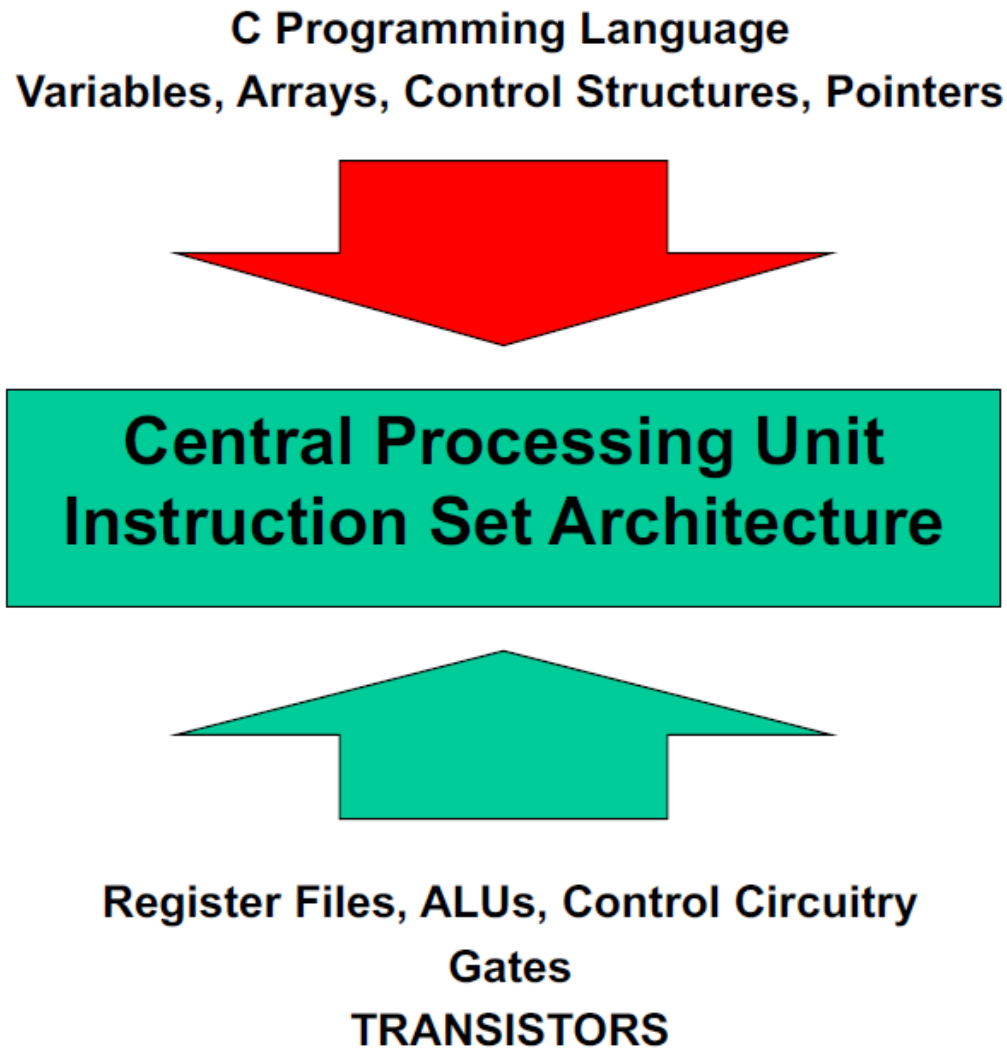
- Check-in06 Due Wednesday 11/2 @ 4:59 pm
- HW06 (Video Game) Due Friday 11/4 @ 11:59 pm
  - Should have everything you need after this lecture
- Midsemester Survey Due Wednesday 11/9 @ 11:59 pm
- HW03 Regrade Requests are open
  - Close at 11:59 pm Friday 11/4

Lecture Outline

- **Intro to C**
- Pointers
- Arrays
- Strings
- Formatted I/O
  - printf & scanf

CIS 2400

- First half of the course is more hardware focused
- Second Half is from a more software / programming perspective
- Later, we will connect these two halves



Brief C History

- The history of C is closely tied to UNIX
  - UNIX is an OS family/design, C is a programming language
  - C was developed alongside UNIX for writing various UNIX utilities and UNIX was eventually re-written in C. This made UNIX one of the first Operating Systems not written in just assembly
  - C allows users to have direct control over memory and expects most users to have knowledge of the underlying architecture
  - Unix and C are extremely influential.
    - Part of this is due to Bell Labs (where C and UNIX were made) not being allowed to copyright it. C and UNIX were "Open Source"
    - Most OS's are "Unix-like" (Android OS, Chrome OS, macOS, iOS, Linux)
    - Linux is sort of the "successor" of UNIX

C Language family

- Many languages adopted similar syntax to C due to it's success. (curly braces, function definitions, if/while/for syntax, variable declarations etc.)
- Examples
  - C (1969)
  - C++ (1979)
  - Objective-C (1986)
  - Perl (1988)
  - Java (1991)
  - Javascript (1995)
  - Rust (2010)
  - …

This means Java code can look very similar to C code. A lot of C code is readable if you are comfortable with Java

First C program: Hello World

Similar to import statements. Allows us to use the std I/O and std library modules of C

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

int main() {
  printf("Hello World!\n");
  return EXIT_SUCCESS;
}
```

Still have a main() function to indicate where the program starts execution.

Function is wrapped in curly braces

Return statements

Double quotes to indicate a string literal. parenthesis to call a function

Second C program: sum evens

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

int sum_evens(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    if (i % 2 == 0) {
      sum += i;
    }
  }
  return sum;
}

int main() {
  int sum = sum_evens(5);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

Function declarations & parameters

Variables local to the function

For loops & if statements look similar

Print statements are different to format output. This replaces %d with the value of sum, more later in lecture

Another Similarity: Scope

- Variables declared inside of a function are local to that function and are not visible outside of that scope.

- Some older C compilers, **like lcc**, are picky about how you initialize variables. Lcc won't let you initialize variable in a for loop declaration (`for(int i = 0; …)`. Newer c compilers like clang and gcc do not require this.

- Variables can also be declared outside of a function – these variables typically have global scope but there are some subtleties

C vs Java Similarities Overview

- C and Java are very similar syntactically

- Similarities:
  - Control Structures (if/else/for/while/…)
  - Variables and data types (int/char/float/double/…)
  - Arrays and strings exist in both
    (but are also different implementation wise)
  - Statements & Expressions
    x = (y + z) / 2
  - Proper functions

C vs Java

- C and Java are Syntactically Similar, but …
  - **do not assume everything in C is like Java**
  - **do not assume everything in C is like Java**
  - **do not assume everything in C is like Java**
  - **do not assume everything in C is like Java**
  - **do not assume everything in C is like Java**

- From my experience, a common source for making mistakes in C is forgetting that things are not like Java

C vs Java: Differences

- C is functionally very different than Java

- Some differences:
  - C doesn't default initialize anything
  - C doesn't have objects
  - C compiles down to machine code
  - C runs really fast
  - C doesn't check much in terms of safety, no nice error messages like Java has
  - C is "just above" assembly in terms of abstraction
  - C allows for direct memory access

C vs Java: Motivations

- Java aims to shield the programmer from the details of machine, including memory management
  - Garbage Collection
  - Default Initialization

- C expects you to be intimately familiar with how the machine works. Allows you to manipulate machine state directly.
  - Directly access memory locations
  - Store and manipulate addresses
  - Allocate and deallocate resources

Today's topic, extremely important in C

Wednesday's topic

Lecture Outline

- Intro to C
- **Pointers**
- Arrays
- Strings
- Formatted I/O
  - printf & scanf

Pointers

POINTERS ARE EXTREMELY IMPORTANT IN C

- Variables that store addresses
  - It stores the address to somewhere in memory
  - Must specify a type so the data at that address can be interpreted

- Generic definition: `type* name;` or `type *name;` (equivalent)
  - Example: `int *ptr;`
    - Declares a variable that can contain an address
    - Trying to access that data at that address will treat the data there as an int

Pointer Operators

- *Dereference* a pointer using the unary * operator
  - Access the memory referred to by a pointer
  - Can be used to read or write the memory at the address
  - Example:

```
int *ptr = ...; // Assume initialized
int a = *ptr; // read the value
*ptr = a + 2; // write the value
```

- Get the address of a variable with &
  - `&foo` gets the address of foo in memory
  - Example:

```
int a = 240;
int *ptr = &a;
*ptr = 2; // 'a' now holds 2
```

Pointer Example

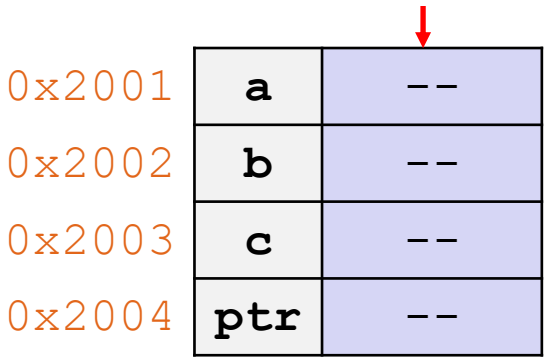
```
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;   // ptr is a pointer to an int

  a = 5;
  b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```

Initial values are garbage



Assuming that integers and pointers each fit into a single memory location

Pointer Example

```
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;   // ptr is a pointer to an int

  a = 5;
  b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```

0x2001	a	5
0x2002	b	3
0x2003	c	--
0x2004	ptr	--

Assuming that integers and pointers each fit into a single memory location

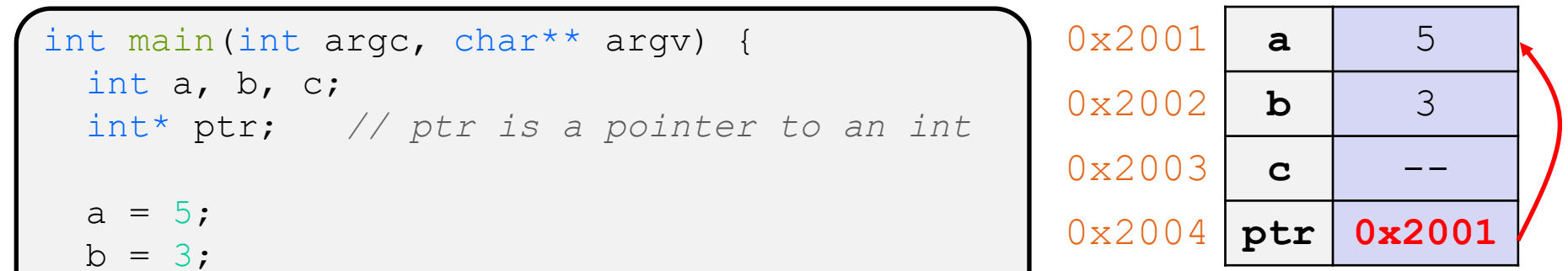
Pointer Example

```
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;    // ptr is a pointer to an int

  a = 5;
  b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```



Assuming that integers and pointers each fit into a single memory location

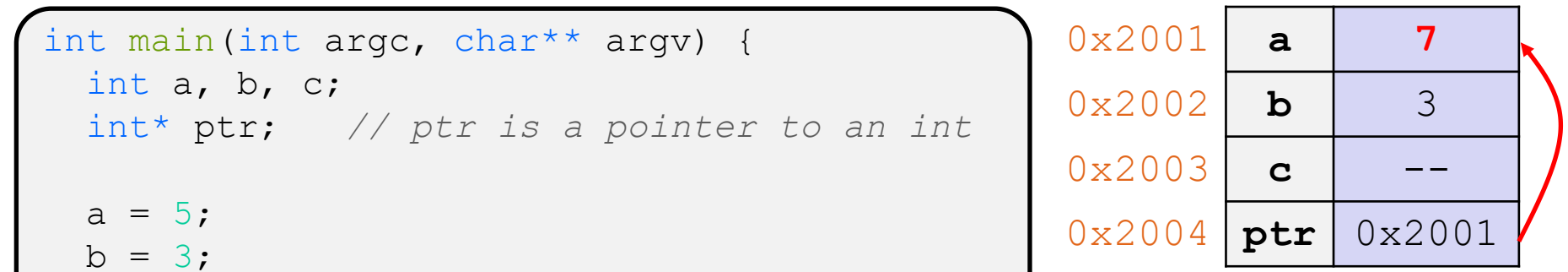
Pointer Example

```
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;    // ptr is a pointer to an int

  a = 5;
  b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```



Assuming that integers and pointers each fit into a single memory location

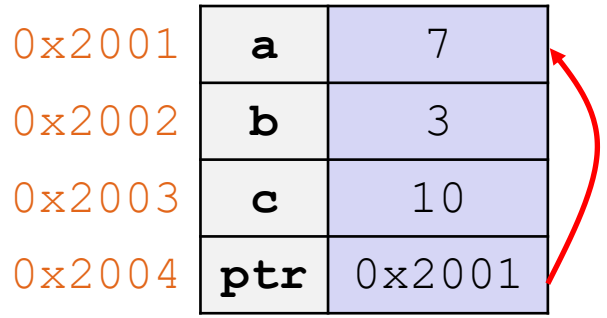
Pointer Example

```
int main(int argc, char** argv) {
  int a, b, c;
  int* ptr;    // ptr is a pointer to an int

  a = 5;
  b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```



Assuming that integers and pointers each fit into a single memory location

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```

Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```

main

soln1	?
soln2	?

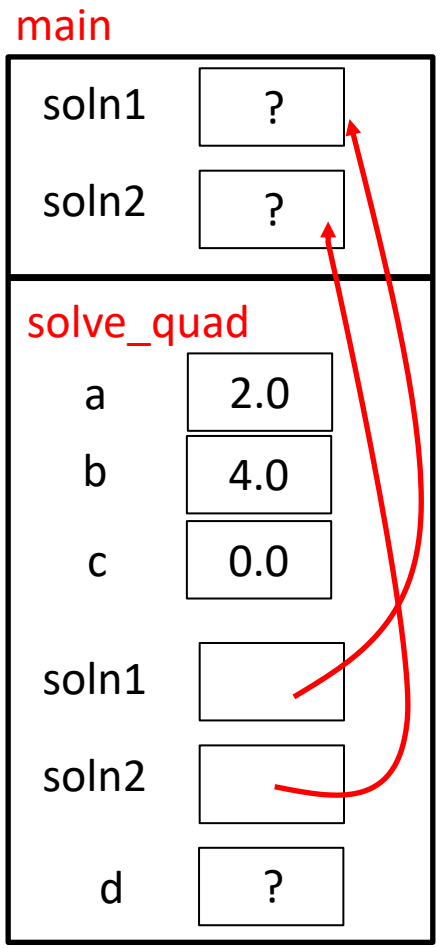
Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```



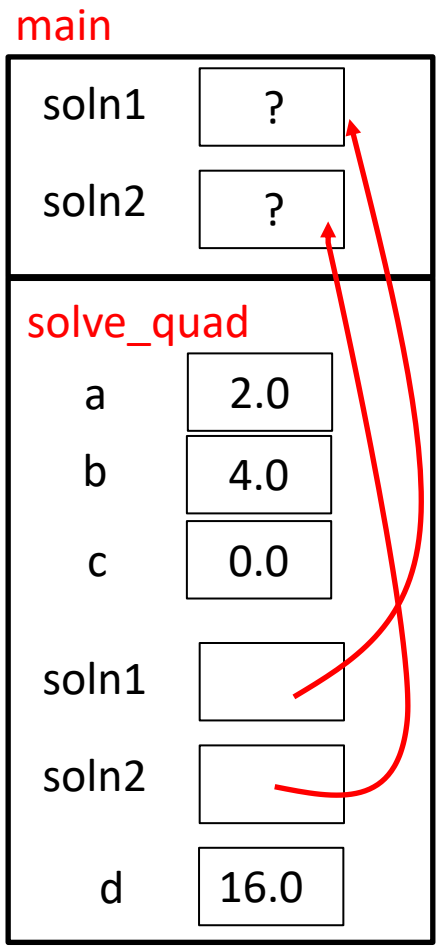
Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```



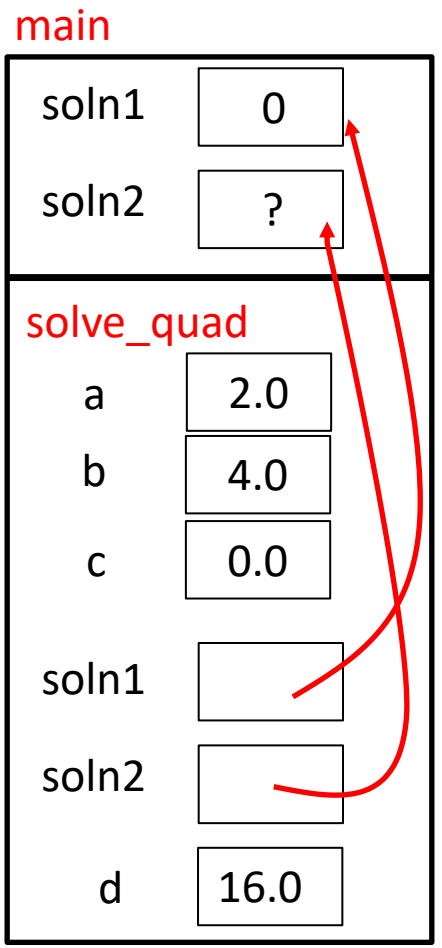
Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```



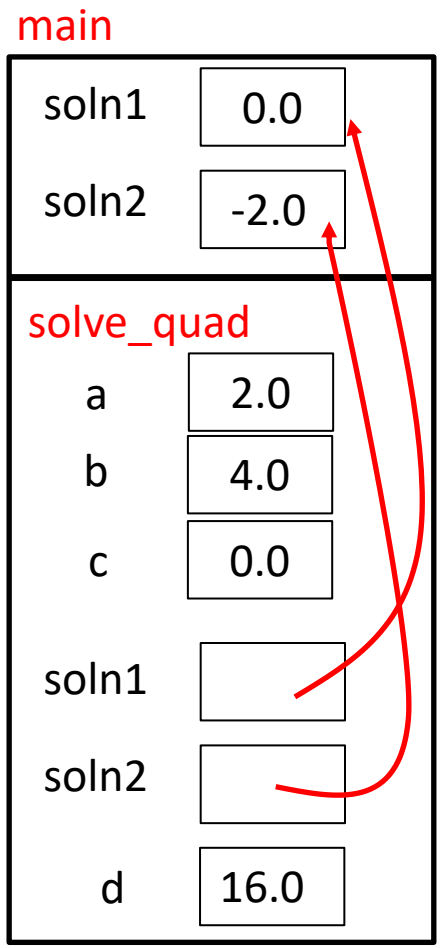
Red arrow indicates the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```



Red arrow indicate the NEXT line to execute

Output Parameters

- Pointers can be used to "return" more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                    double* soln1, double* soln2){
  double d = b*b - 4 * a * c;
  if (d >= 0) {
    *soln1 = (-b + sqrt(d)) / (2*a);
    *soln2 = (-b - sqrt(d)) / (2*a);
    return 1;
  } else {
    return 0;
  }
}

int main(int argc, char** argv) {
  double soln1, soln2; // populated by function call
  solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
  // ...
}
```

main

soln1	0.0
soln2	-2.0

Poll Everywhere

pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

A. 5, 22, 42

B. 42, 42, 37

C. 42, 22, 37

D. 5, 42, 37

E. I'm not sure

Poll Everywhere

pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute

main

a	5	b	22	c	42

Poll Everywhere

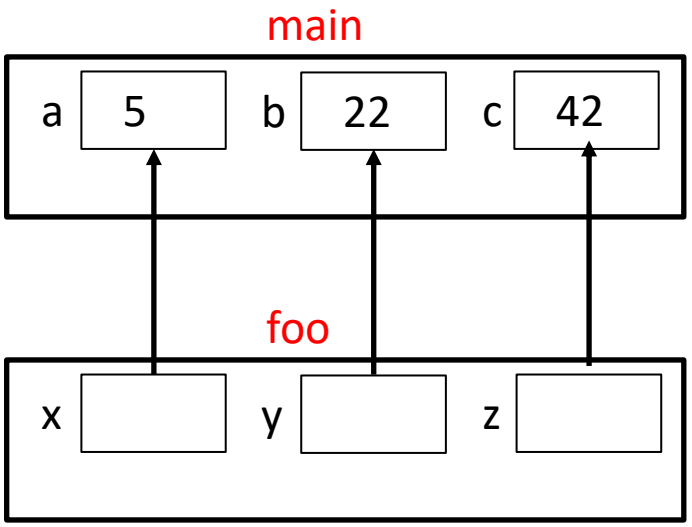
pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



Poll Everywhere

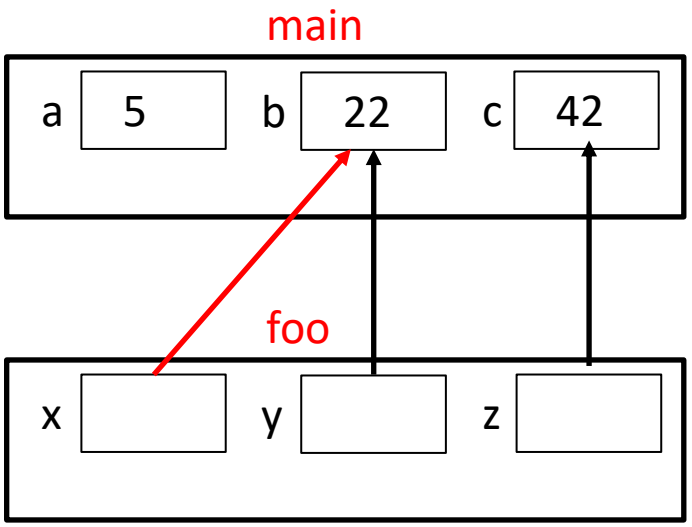
pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



Poll Everywhere

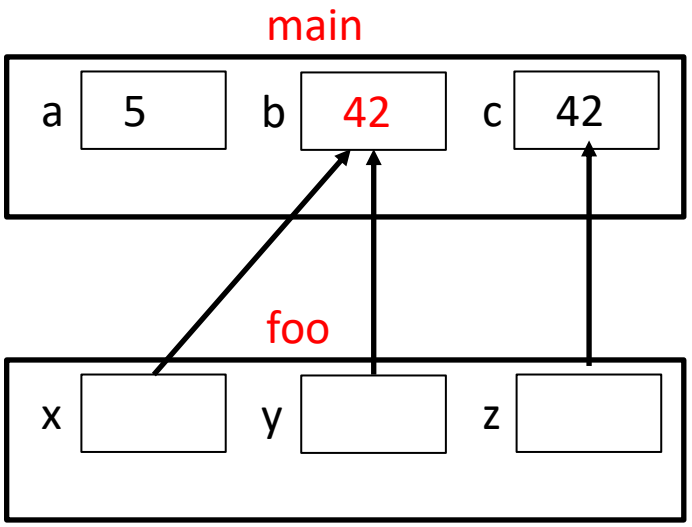
pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



Poll Everywhere

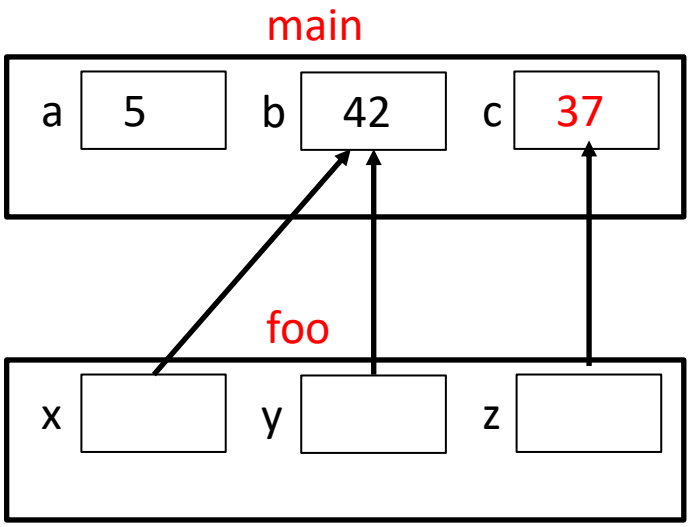
pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



Poll Everywhere

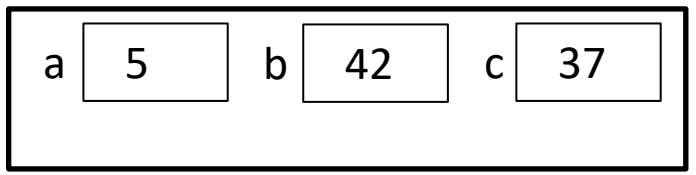
pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main() {
  int a = 5, b = 22, c = 42;
  foo(&a, &b, &c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

Red arrow indicates the NEXT line to execute



D. 5, 42, 37

Lecture Outline

- Intro to C
- Pointers
- **Arrays**
- Strings
- Formatted I/O
  - printf & scanf

Arrays

- Definition: `type name[size]`
  - Allocates `size*sizeof(type)` bytes of *contiguous* memory
  - Normal usage is a compile-time constant for `size` (*e.g.* `int scores[175];`)
  - Initially, array values are "garbage"

- Size of an array
  - **Not stored anywhere** – array does not know its own size!
  - The programmer will have to store the length in another variable or hard-code it in

Using Arrays

Optional when initializing

- Initialization: `type name[size] = {val0,…,valN};`
  - `{}` initialization can *only* be used at time of definition
  - If no size supplied, infers from length of array initializer

- Array name used as identifier for "collection of data"
  - `name[index]` specifies an element of the array and can be used as an assignment target or as a value in an expression
  - Array name (by itself) produces the address of the start of the array
    - Cannot be assigned to / changed

```
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0;  // memory smash!
```

No IndexOutOfBounds
Hope for segfault

Multi-dimensional Arrays

- Generic 2D format:

```
type name[rows][cols];
```

  - Still allocates a single, contiguous chunk of memory
  - C is *row-major*
  - Can access elements with multiple indices
    - `A[0][1] = 7;`
    - `my_int = A[1][2];`

  - The entries in this array are stored in memory in row major order as follows:
    - A[0][0], A[0][1], A[0][2], A[1][0], A[1][1], A[1][2]

  - 2-D arrays normally only useful if size known in advance. Otherwise use dynamically-allocated data and pointers (later)

Arrays as Parameters

- It's tricky to use arrays as parameters
  - What happens when you use an array name as an argument?
  - Arrays do not know their own size

Passes in address of start of array

```
int sumAll(int a[]) {
  int i, sum = 0;
  for (i = 0; i < ...???
}
```

Equivalent

```
int sumAll(int* a) {
  int i, sum = 0;
  for (i = 0; i < ...???
}
```

- Note: Array syntax works on pointers
  - E.g. `ptr[3] = ...;`

Solution: Pass Size as Parameter

```
int sumAll(int* a, int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += a[i];
  }
  return sum;
}
```

- Standard idiom in C programs

Pointer Arithmetic

- In LC4, we did arithmetic on addresses to iterate through arrays. We can do the same in C

```
double my_array[10]; // create an array of 10 doubles
double *ptr = my_array; // ptr has the address of the
                        // first element
ptr = ptr + 1; // increment ptr to point to
               // the next elemetnt
ptr[2] = 3.14; // equivalent to *(ptr + 2) = 3.14
```

- Pointers are *typed*
  - Tells the compiler the size of the data you are pointing to
- Pointer arithmetic is scaled by `sizeof(*ptr)`
  - Sometimes a single array element can span multiple addresses
  - Works nicely for arrays

Size (number of bytes) of thing being pointed at

Pointer Square Brackets

- We can use the "array syntax" on pointers

```
ptr[3] = ...;
```

- This syntax is the same as

```
*(ptr + 3) = ...;
```

- Fun Fact, these are all the same in C:

```
ptr[3] = ...;
```

```
*(ptr + 3) = ...;
```

```
*(3 + ptr) = ...;
```

```
3[ptr] = ...;
```

Poll Everywhere

pollev.com/tqm

- What are the final values of nums?

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```

A. 2, 3, 5, 6

B. 38, 38, 5, 6

C. 2, 38, 5, 2

D. 2, 38, 5, 5

E. I'm not sure

Poll Everywhere

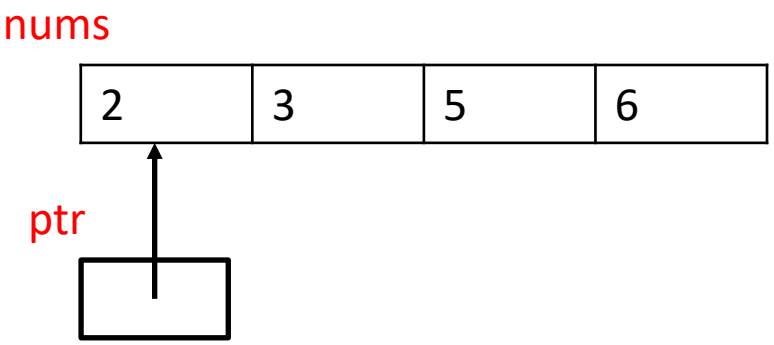
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



Poll Everywhere

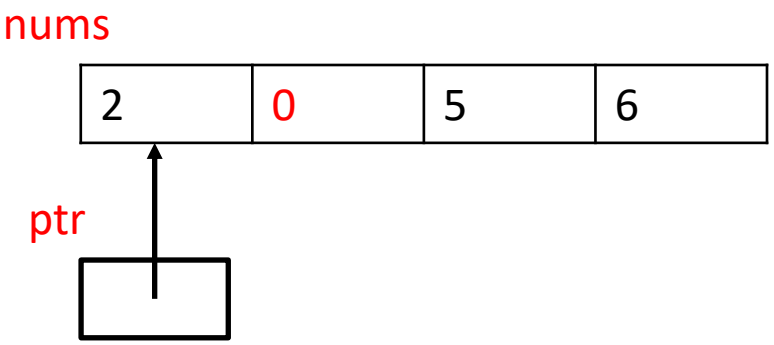
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



Poll Everywhere

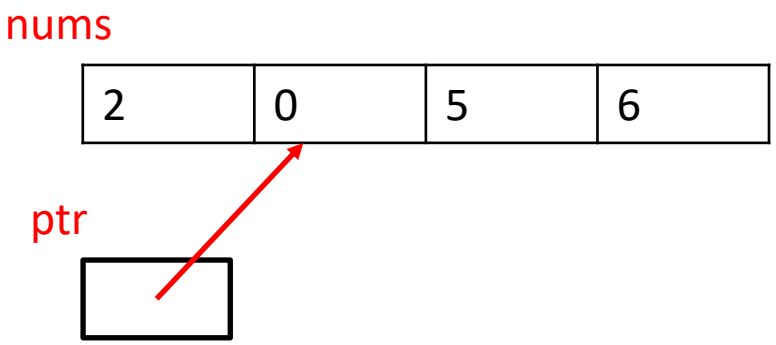
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



Poll Everywhere

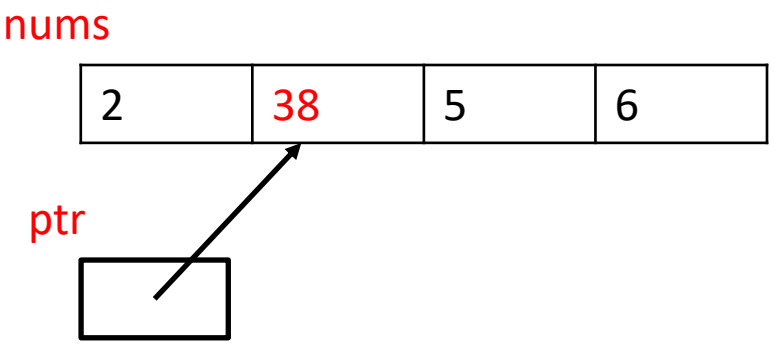
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



Poll Everywhere

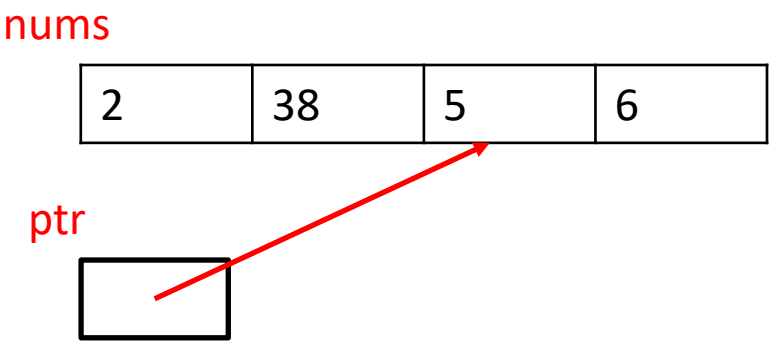
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



Poll Everywhere

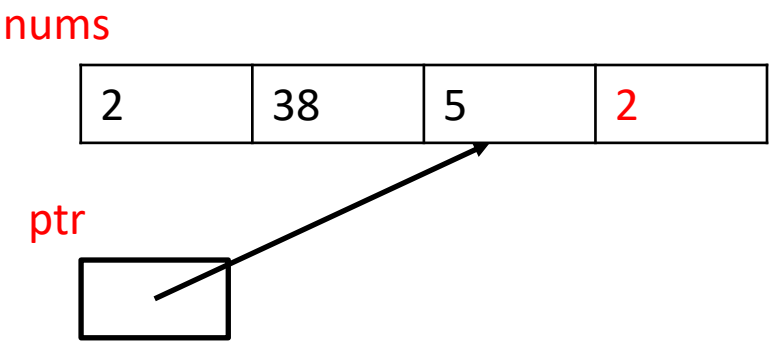
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



C. 2, 38, 5, 2

Lecture Outline

- Intro to C
- Pointers
- Arrays
- **Strings**
- Formatted I/O
  - printf & scanf

Strings without Objects

- Strings are central to C, very important for I/O
- In C, we don't have Objects but we need strings
- If a string is just a sequence of characters, we can have use array of characters as a string

- Example:

```
char str_arr[] = "Hello World!";
char *str_ptr = "Hello World!";
```

Null Termination

DO NOT FORGET THIS. THIS IS THE CAUSE OF MANY BUGS

- Arrays don't have a length, but we **mark the end of a string with the null terminator character.**
  - The null terminator has value **`0x00`** or **`'\0'`**
  - Well formed strings ***MUST*** be null terminated

- Example: `char str[] = "Hello";`
  - Takes up 6 characters, 5 for "Hello" and 1 for the null terminator

address	0x2000	0x2001	0x2002	0x2003	0x2004	0x2005
value	'H'	'e'	'l'	'l'	'o'	'\0'

String library Functions

- Many Library functions are provided for processing strings
- Most are found in the header file `<string.h>`
  - `strlen(char* str)` – returns the number of characters in the string **excluding** the null terminator.
  - `strcpy(char *s1, char *s2)` - copies the string in s2 into s1. Assumes that s1 has enough space to store the copy.
  - `strcmp(char *s1, char *s2)` – compares two strings and returns < 0 if s1 < s2, > 0 if s1 > s2 and 0 if they are the same string

More Library functions

- There are also other useful functions defined in <ctypes.h>
  - `isalnum(int c)` returns non-zero if c is an alphanumeric character
  - `isspace(int c)` returns non-zero if c is a space character
  - `tolower(int c)` if c is an uppercase letter, returns the lowercase counterpart. If c is not an uppercase letter, c is returned.
- There are more functions that exist that you may find useful.

C Standard Library

- Not as big as Java standard libraries but has many useful functions.
- Don't reinvent something that already exists

- Examples:
  - `stdio.h` useful for I/O, printing, reading input, etc.
  - `ctypes.h` functions for converting and testing char's
  - `math.h` mathematical functions (pow, sqrt, etc.)
  - `stdlib.h` general purpose functions
  - `string.h` functions for using strings

Lecture Outline

- Intro to C
- Pointers
- Arrays
- Strings
- **Formatted I/O**
  - **printf & scanf**

Formatted I/O

- Many programs need to convert between the bit values a computer manipulates and something a human can read
  - Example: converting between the binary encoding for an int into readable string

- Often done by the following functions or variants of them
  - `printf`
    - Prints a formatted string to the console
  - `scanf`
    - Reads a formatted string from the console

Formatted I/O

- Remember that **EVERYTHING** is stored as bits.
- Do not confuse what you read on the terminal with the actual representation of data in memory
- Converting bits to be human readable is a big part of formatted output

Formatting Example

- Do you recognize the following 32 bit single precision value?:
  - 01000000010010010000111111011011
- Let's run a number to string procedure to convert it into sequence of ASCII characters
  - 0x33, 0x2E, 0x31, 0x34, 0x31, 0x35, 0x39
- What about now?
  - 3.14159

Another Formatting Example

- Recognize the following 16 bit 2C integer value?
  - 1111001001110001

- Here are the ASCII characters in its decimal representation
  - 0x2D, 0x33, 0x34, 0x37, 0x30

- Here is what it would look like printed out.
  - -3470

Formatting Strings

- To specify how bits should be interpreted for printing and scanning. We must use a **format string**
- A format string is just a string with formatting specifiers:
  - `%d` – a decimal integer value
  - `%x` – a hexadecimal value
  - `%s` – a string
  - `%f` – a floating point value

- For `printf`- these formatting subsequences may be accompanied by field width and precision specifiers
  - `%12.3f` – prints a floating-point number using 12 characters with three digits after the decimal place

Special Characters

- There are also special characters that can show up in any string which have special meaning
  - `'\n'` : newline character
  - `'\t'` : tab character

Formatting Printing Example

```c
int main(int argc, char** argv) {
  int a = 27;
  double b = 3.14159;

  // simple string output with a newline at the end
  printf("Hello World\n");

  // formatted output that will perform
  // numerical conversions
  printf("a = %d, b = %7.3f\n", a, b);
}
```

Outputs:

```
Hello World
a = 27, b =   3.142
```

Formatted input

- Just as how we can format output, we can convert strings to binary representation when we read input with `scanf`
  - Example converts "134" to 16 bit integer 0000000010000110

- There are similar functions to `scanf` such as `sscanf`
  - `sscanf` takes a string as input for scanning rather than reading from the terminal

```
int x;
char* to_scan = "value: 240";
sscanf(to_scan, "value: %d", &x); // sets x to 240
```

Input Mismatch

- Input may not always match the expected string, function will parse as many as it can. Function returns the number of arguments successfully decoded

- Examples:

```
int x, y;
char* to_scan = "-108 97";
sscanf(to_scan, "%d %d", &x, &y); // returns 2
```

```
int x, y;
char* to_scan = "203 wtf";
sscanf(to_scan, "%d %d", &x, &y); // returns 1
```

Formatted I/O

Basic Data types – machine representations