Software Systems

Day 15 - Functions as Parameters, Variadic Functions

Agenda

- Time Zones
- Announcements
- Functions as Parameters
- Variadic Functions
- (if time allows) Open time for Project 1

Time Zones

- During the project, some of you had to "roll your own" data structures, hash functions, etc.
- It's a good thing none of you tried to do time zones.
- https://www.youtube.com/watch?v=-5wpm-gesOY

Announcements

- Project 1 is due tonight.
    - See Canvas for the rubric.
- We'll start up readings/quizzes/assignments this week.
    - There's a reading/quiz due next time.
- Project 2 milestones:
    - Fri 4/7: Team Sign-Up
    - Thu 4/13: Proposal
    - Thu 4/20: Architecture Review
    - Thu 4/27: Code Review
    - Thu 5/4: Slides
    - Fri 5/5: Final Draft

Functions as Parameters

- In Python, you might be familiar with list comprehensions:

```
pi_digits = [3, 1, 4, 1, 5, 9]
square_digits = [digit ** 2 for digit in pi_digits]
# square_digits is [9, 1, 16, 1, 25, 81]
```

- A (somewhat outdated) way of doing this in Python is:

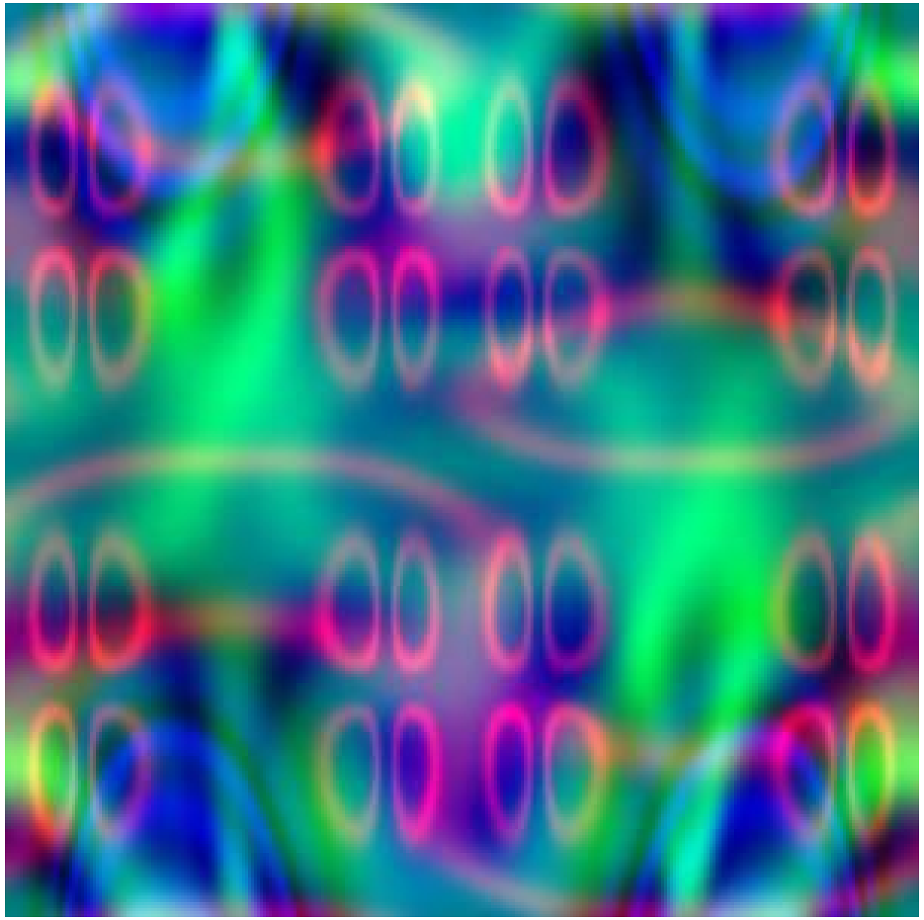
```
def square(digit):
    return digit ** 2

square_digits = list(map(square, pi_digits))
```

This is a function!

Functions as Parameters

- Passing functions as parameters can be useful in a variety of situations:
  - Applying a function to every member of a list (see example)
  - Taking an action in response to an event (e.g., JavaScript callbacks)
  - Recursively generating functions



Functions as Parameters

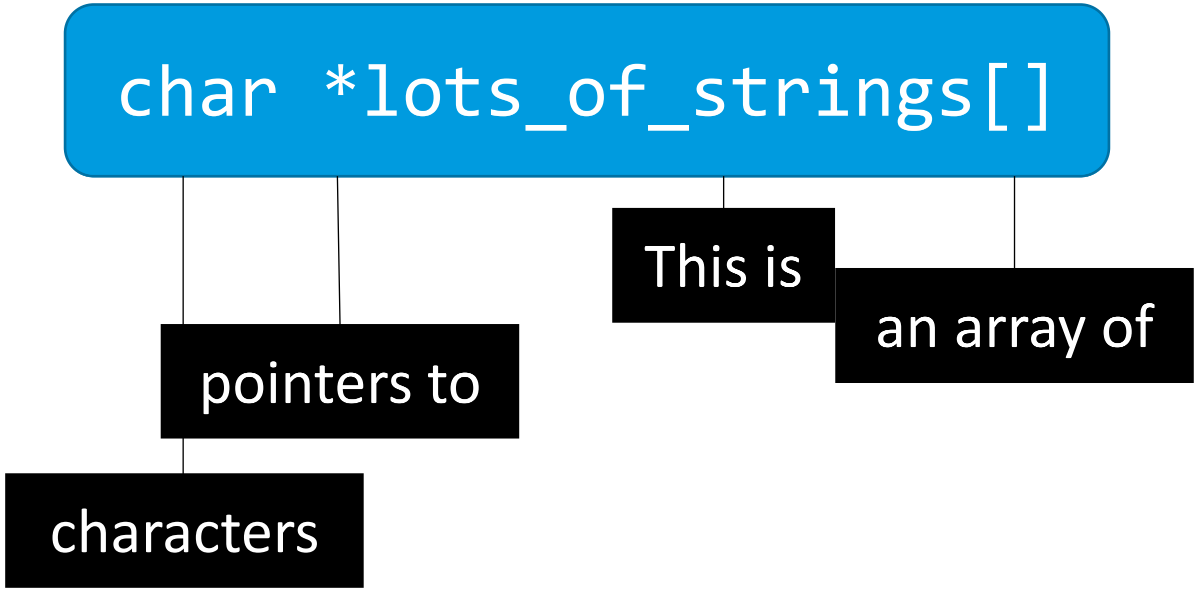
- Exercise:
  - At your tables, come up with a realistic example of how you might use functions as parameters in C.
  - Explain alternatives to using a function as a parameter, and why a function as a parameter might be useful versus these alternatives.
  - If you have time, sketch out a quick example in code.

Functions as Parameters

- If you try to declare a function that takes another function as a parameter, getting the type right can be tricky.
- C doesn't have a "generic function" type like Python does.
- A function has to specify what parameter types it takes and what type it returns.
- So how do you represent "a function that takes an int and returns an int"?

Functions as Parameters

- Let's come back to the right-to-left rule.
  - Start from the variable name and read right.
  - Then go back to the variable name and read left.



```
char *lots_of_strings[]
```

This is

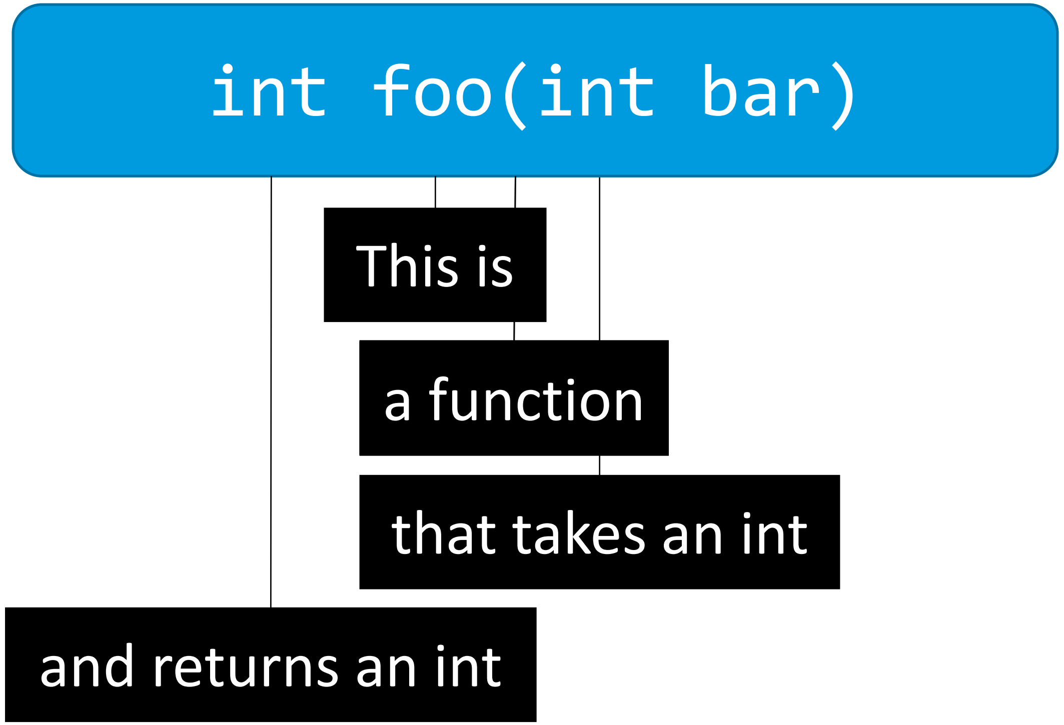
an array of

pointers to

characters

Functions as Parameters

- This works for functions, too.



```
int foo(int bar)
```

This is

a function

that takes an int

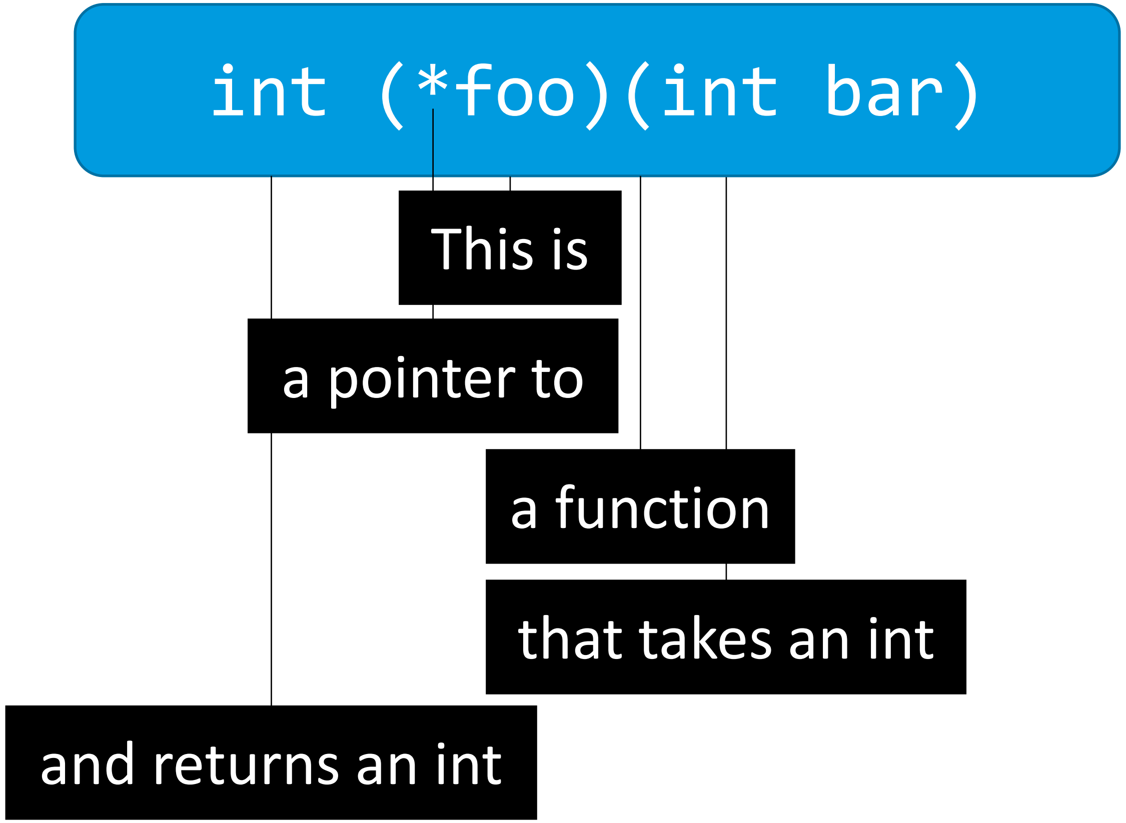
and returns an int

Functions as Parameters

- But this doesn't work:
  ```
  void baz(int x, int foo(int bar));
  ```
- You actually need to pass the pointer to the function.
- And for that, we'll need parentheses.

Functions as Parameters

- The right-to-left rule follows parentheses.



```
int (*foo)(int bar)
```

This is

a pointer to

a function

that takes an int

and returns an int

Functions as Parameters

- Then, you can do this:

```
void baz(int x, int (*foo)(int));
```

- Or this:

```
int (*f)(int) = foo;
int y = f(42);
```

- In this case, the variables `f` and `foo` are function pointers - they represent the address of the start of the function code.

Functions as Parameters

- Exercise:
  - Declare a function pointer called strproc that takes a string (char*) and and int, and returns another string.

Functions as Parameters

- Having to declare functions with their exact types can be tedious - what if we want something more general?
- It's possible, but you have to be careful.
- Here, there be void pointers: https://en.cppreference.com/w/c/algorithm/qsort

Variadic Functions

- Even with extensive use of void pointers, it looks like functions still need to take a fixed number of parameters.
- But functions like `printf` can take any number of parameters:

  ```
  printf("The answer is %d\n.", x);
  printf("Cool, right?\n");
  ```
- For that type of behavior, you can use **variadic functions**: functions that take a *variable number of parameters*.

Variadic Functions

- Exercise:
  - At your tables, come up with a realistic use case for a variadic function.
  - Are there any other ways that you could implement this function without making it variadic?
  - Which do you think is better (variadic or non-variadic), and why?

Variadic Functions

- Variadic functions could be used for something like concatenating an unknown number of strings together:
  `concat(size_t num_strings, ...);`
- But you could also pass a list of strings:
  `concat(size_t num_strings, char** string_list);`
- Ultimately, there are pros and cons to each approach - some use cases are better suited for variadic functions, while others are better suited for lists.

Variadic Functions

- The machinery for variadic functions is in `stdarg.h`.
- It defines some types and macros for creating variadic functions.
  - `va_list`
  - `va_start`
  - `va_arg`
  - `va_end`

Variadic Functions

- Example:

```
#include <stdarg.h>

int add_ints(size_t count, ...) {
  int sum = 0;
  va_list args;  // Declare variable arg list
  va_start(args, count); // Set up arg list
  for (size_t i = 0; i < count; ++i)
    sum += va_arg(args, int);  // Get the next arg
  va_end(args);  // Stop reading variable args
  return sum;
}
```

Variadic Functions

- Remember that va_start, va_arg, and va_end are preprocessor macros, not functions.
  - This means that you can get weird compile errors if you write them the wrong way.
  - Also, make sure you get the types right - if you meant to read in `va_arg(args, int)` and wrote `va_arg(args, size_t)` instead, weird things can happen.

Variadic Functions

- Exercise:
  - Write a variadic function called `print_lines` that takes a `size_t` and some number of strings, and then prints each of those strings on a new line.
- Example if you need it again:

```
#include <stdarg.h>

int add_ints(size_t count, ...) {
  int sum = 0;
  va_list args;  // Declare variable arg list
  va_start(args, count); // Set up arg list
  for (size_t i = 0; i < count; ++i)
    sum += va_arg(args, int);  // Get the next arg
  va_end(args);  // Stop reading variable args
  return sum;
}
```

Variadic Functions

- Variadic functions have to have at least one named parameter.
- So you can do this:
  ```
  int add_ints(size_t count, ...);
  ```
- But not this:
  ```
  int add_ints(...);
  ```
- (This will change in the next C standard.)

Variadic Functions

- Do you have to pass a parameter that tells you how many extra arguments to read?
  ```
  int add_ints(size_t count, ...);
  printf("%d %d %d\n", a, b, c);
  ```
- There's another way - null pointer termination!
  ```
  print_lines(char* initial_str, ...);
  ```
- Then you can pass NULL as the last item in that list to signal that you're done passing strings.

Function Pointers and Variadic Functions

- Remember, getting the types and syntax right is a huge part of using these features.
- Also, don't feel obligated to use these features if you don't have to - they're sometimes more trouble than they're worth.
- You'll get more chances to explore these in the assignment.