



[pollev.com/naomila](https://pollev.com/naomila)



**Name a *value* that you feel is embedded in the C language.**

**(open-ended survey question)**

**By “value” we mean an adjective describing the relative worth, merit, or importance of something (*e.g.*, loyalty, kindness), NOT a number or constant.**

# CSE333 Systems Programming

## Makefiles

### Instructors:

Naomi Alterman

### Teaching Assistants:

Ann Baturytski

Barbora Buzkova

Mendel Carroll

Camden Harris

Hannah Hempstead

Rishabh Jain

Irene Lau

Jonathan Nister


Advay Patil

Aviel Wood

Angela Wu

Jennifer Xu

# Administrivia

- ❖ **No more leniency with assignment submission** – messed up tag or file locations = no submission
- ❖ Exercise 4 ongoing, due on Monday
  - Refer to section materials about how to deal with directories
- ❖ ~~Homework 1 due last night (4/16)~~
  - Lol attu went down last night 
  - **HW1 due date pushed later by 24hr (tonight, 4/17 at 11:59p)**
- ❖ Homework 2 out today
  - Builds on top of Homework 1 data structures to create search engine!

# Lecture Outline

- ❖ **Make and Build Tools**
- ❖ Makefile Basics

# make

- ❖ make was the first coordinate software **build system** – a program for controlling what gets (re)compiled and how
  - Many other such programs now exist (*e.g.*, ant, maven, IDE “projects”)
  - But make is the OG and still in heavy use today
- ❖ make has tons of fancy features, but only two basic ideas:
  - 1) Be a **centralized space** that holds the commands used to compile our project
  - 2) Express **dependencies** for those commands so they only run when the source code they process changes (avoid unnecessary work)
- ❖ To avoid “just teaching make features” (boring and narrow), let’s focus more on the concepts...

# Building Software (1/2)

- ❖ Programmers spend a lot of time “building”
  - Creating programs from source code
  - Both programs that they write and other people write



<https://xkcd.com/303/>

# Building Software (2/2)

- ❖ Programmers spend a lot of time “building”
  - Creating programs from source code
  - Both programs that they write and other people write
- ❖ Programmers like to automate repetitive tasks
  - Repetitive: `gcc -Wall -g -std=c17 -o widget foo.c bar.c baz.c`

- Retype this every time:



- Use up-arrow or history:



(still retype after logout)

- Have an alias or bash script:



- Have a Makefile:



(you're ahead of us)

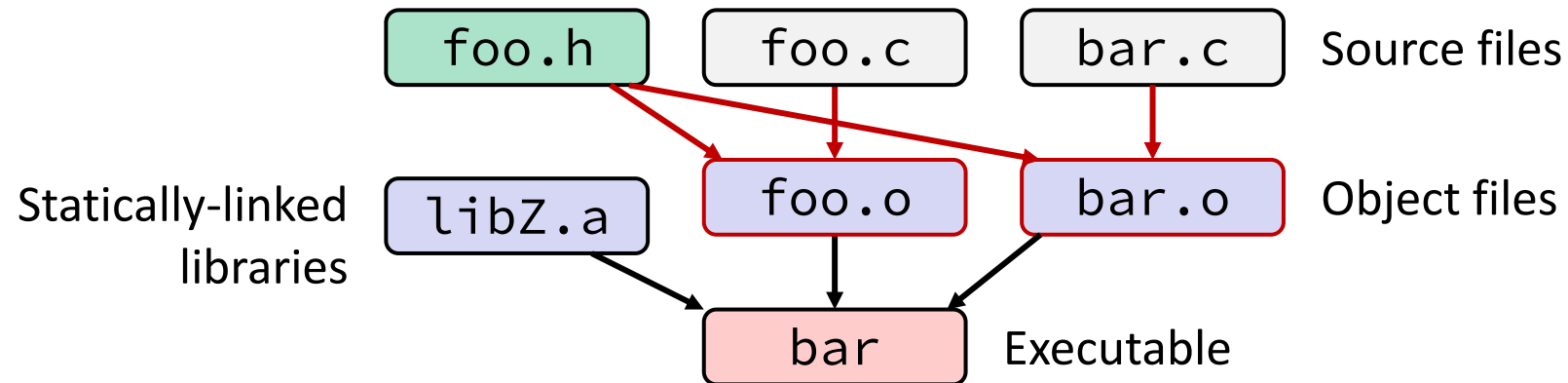
# “Real” Build Process

- ❖ On larger projects, you can't or don't want to have one big (set of) command(s) that are all run every time you change anything. To do things “smarter,” consider:
  - 1) It could be worse: If gcc didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
  - 2) Source files could have multiple outputs (*e.g.*, javadoc). You may have to type out the source file name(s) multiple times
  - 3) You don't want to have to document the build logic when you distribute source code; make it relatively simple for others to build
  - 4) You don't want to recompile everything every time you change something (especially if you have  $10^5$ - $10^7$  files of source code)
- ❖ A traditional shell script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

# Recompilation Management

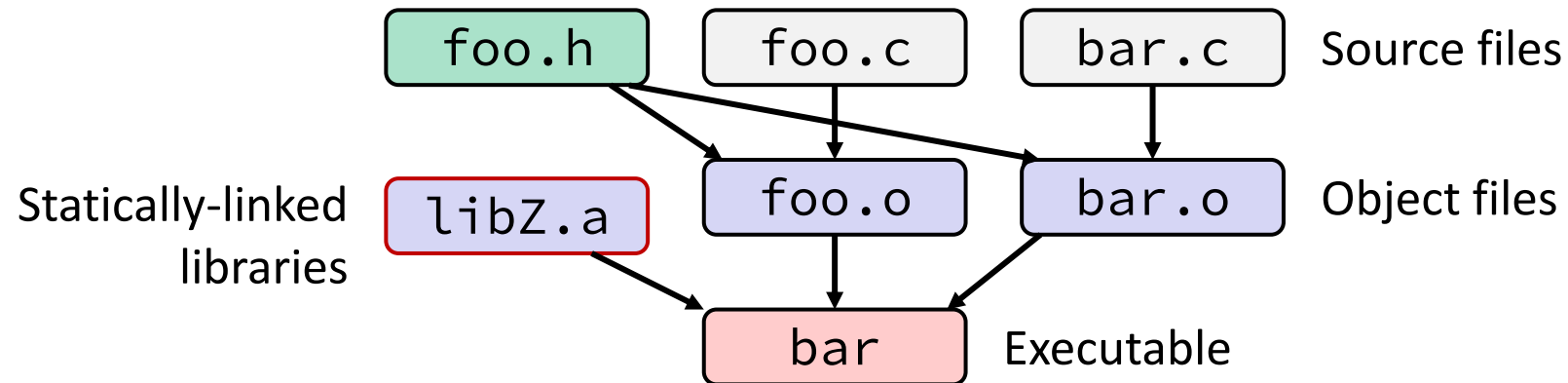
- ❖ The “theory” behind avoiding unnecessary compilation is a *dependency dag* (directed, acyclic graph)
- ❖ To create a target  $t$ , you need sources  $s_1, s_2, \dots, s_n$  and a command  $c$  that directly or indirectly uses the sources
  - If  $t$  is newer than every source (file-modification times), assume there is no reason to rebuild it
  - Recursive building: if some source  $s_i$  is itself a target for some other sources, see if it needs to be rebuilt...
  - Cycles “make no sense”!

# Theory Applied to C (1/4)



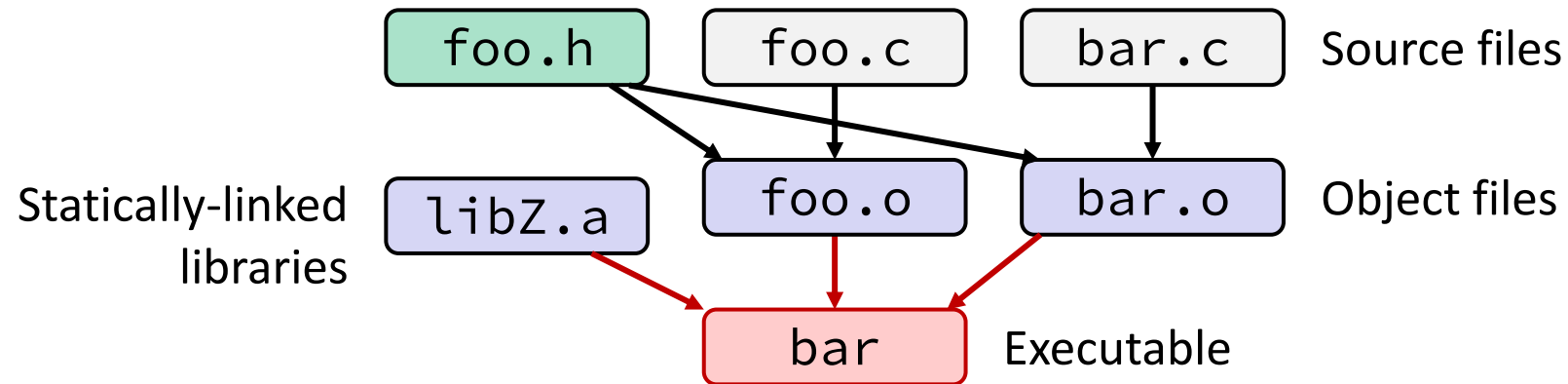
- ❖ Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)

# Theory Applied to C (2/4)



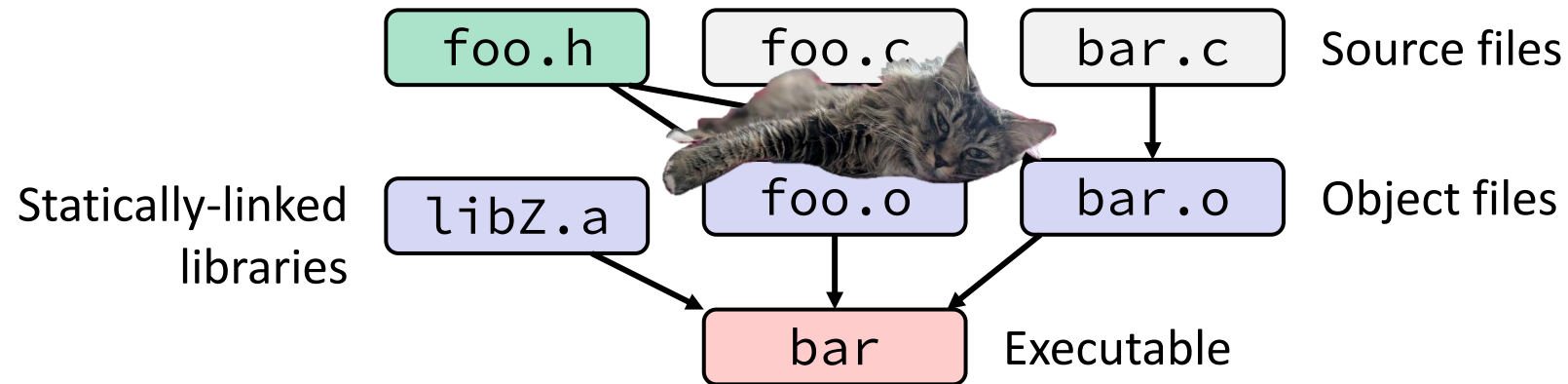
- ❖ Compiling a .c creates a .o – the .o depends on the .c and all included files (.h, recursively/transitively)
- ❖ An archive (library, .a) depends on included .o files

# Theory Applied to C (3/4)



- ❖ Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- ❖ An archive (library, `.a`) depends on included `.o` files
- ❖ Creating an executable (“linking”) depends on `.o` files and archives
  - Archives linked by `-L<path> -l<name>`  
(*e.g.*, `-L. -lfoo` to get `libfoo.a` from current directory)

# Theory Applied to C (4/4)



## ❖ Effects of code changes:

- If one `.c` file changes, just need to recreate one `.o` file, maybe a library, and re-link
- If a `.h` file changes, may need to rebuild more
- Many more possibilities!

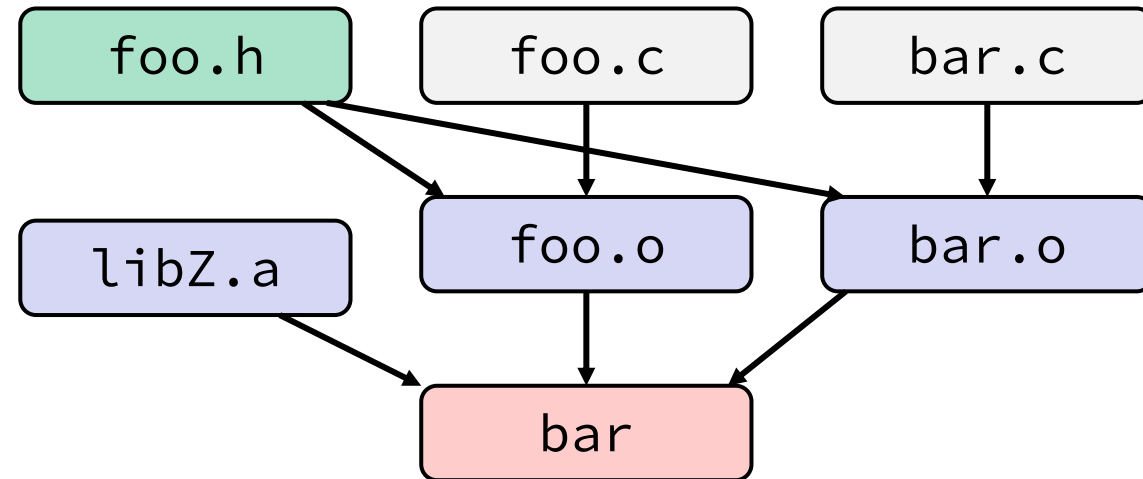
# Poll Everywhere

[pollev.com/naomila](https://pollev.com/naomila)



## If foo.h changes, which files get re-built?

- A. foo.o, bar.o
- B. foo.o, bar.o, bar
- C. foo.c, foo.o, bar
- D. foo.c, foo.o
- E. None of them



# Lecture Outline

- ❖ Make and Build Tools
- ❖ **Makefile Basics**

# make Basics

- ❖ A makefile contains a collection of **targets**:

```
Target-name: sources  
← Tab → command
```

- Colon after target name is *required*
- Command lines must start with a **TAB**, NOT SPACES
- Multiple commands for same target are executed *in order*
  - Can split commands over multiple lines by ending lines with ‘\’
- If target produces an output file, the target name must be exactly the output’s filename (this is how Make builds its dependency tree)

- ❖ Example:

```
foo.o: foo.c foo.h bar.h  
      gcc -Wall -o foo.o -c foo.c
```

# Using make

```
$ make -f <makefileName> target
```

## ❖ Defaults:

- If no `-f` specified, use a file named `Makefile` in current dir
- If no `target` specified, will use the first one in the file
- Will interpret commands in your default shell
  - Set `SHELL` variable in makefile to ensure

## ❖ Target execution:

- Check each source in the source list:
  - If the source is a target in the makefile, then process it recursively
  - If some source does not exist, then error
  - If any source is newer than the target (or target does not exist), run command (presumably to update the target)

# “Phony” Targets

- ❖ A make target whose command does not create a file of the target’s name (*i.e.*, a “recipe”)
  - As long as target file doesn’t exist, the command(s) will be executed because the target must be “remade”
- ❖ *e.g.*, target `clean` is a convention to remove generated files to “start over” from just the source

```
clean:
```

```
    rm foo.o bar.o baz.o widget *~
```

- ❖ *e.g.*, target `all` is a convention to build all “final products” in the makefile
  - Lists all of the “final products” as sources

# “all” Example (make or make all)

```
1 all: prog B.class someLib.a
2 # notice no commands this time
prog: foo.o bar.o main.o
3 gcc -o prog foo.o bar.o main.o
B.class: B.java
javac B.java
someLib.a: foo.o baz.o
4 ar r foo.o baz.o
5
6
foo.o: foo.c foo.h header1.h header2.h
7 gcc -c -Wall foo.c
8
# similar targets for bar.o, main.o, baz.o, etc...
```

# make Variables

- ❖ You can define variables in a makefile:
  - All values are strings of text, no “types”
  - Variable names are case-sensitive and can't contain ':', '#', '=', or whitespace

- ❖ Example:

```
CC = gcc
CFLAGS = -Wall -std=c17
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
           $(CC) $(CFLAGS) -o widget $(OBJFILES)
```

- ❖ Advantages:
  - Easy to change things (especially in multiple commands)
    - It's common to use variables to hold lists of filenames
  - Can also specify/overwrite variables on the command line:  
(*e.g.*, `make CC=clang CFLAGS=-g`)

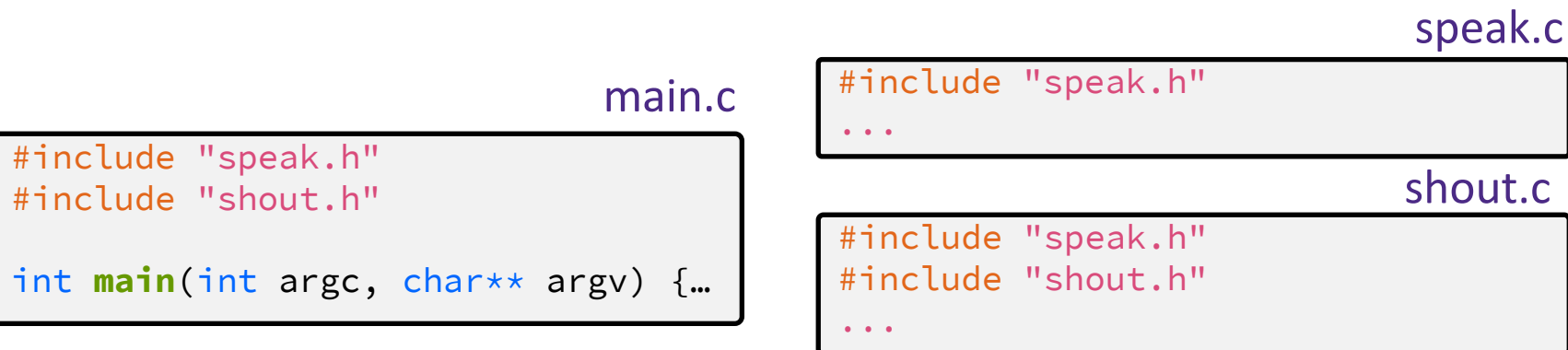
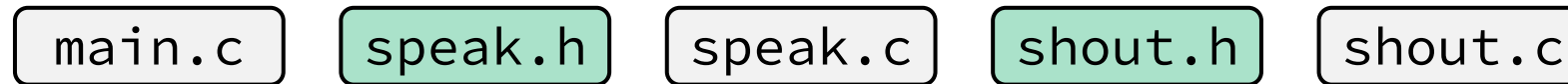


# Makefile Writing Tips

- ❖ *When creating a Makefile, first draw the dependencies!!!!*
  
- ❖ C Dependency Rules:
  - .c and .h files are never targets, only sources
  - Each .c file will be compiled into a corresponding .o file
    - Header files will be implicitly used via #include
  - Executables will typically be built from one or more .o file
  
- ❖ Good Conventions:
  - Include a clean rule
  - If you have more than one “final target,” include an all rule
  - The first/top target should be your singular “final target” or all

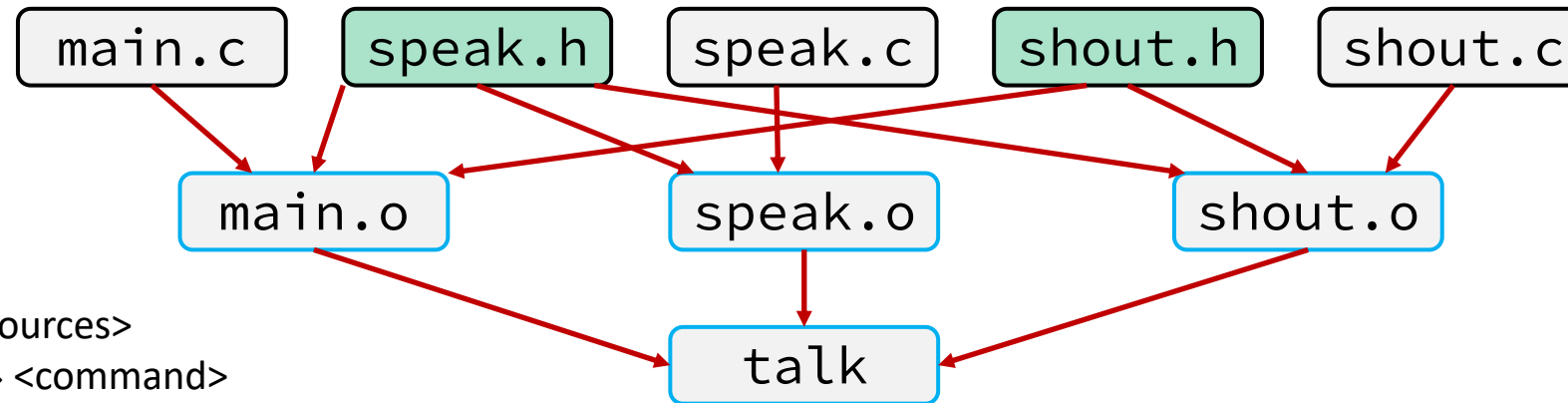
# Writing a Makefile Example: DAG

- ❖ “talk” program (find files on web with lecture slides)



# Writing a Makefile Example: Makefile

- ❖ “talk” program (find files on web with lecture slides)



<target>: <sources>  
←tab character→ <command>

# Revenge of the Funny Characters

- ❖ Special variables:
  - `$$` for target name
  - `$$^` for all sources
  - `$$<` for left-most source
  - Lots more! – see the documentation
- ❖ Examples:

```
# CC and CFLAGS defined above
widget: foo.o bar.o
          $(CC) $(CFLAGS) -o $$ $^
foo.o: foo.c foo.h bar.h
          $(CC) $(CFLAGS) -c $$<
```

# And more...

- ❖ There are a lot of “built-in” rules – see documentation
- ❖ There are “suffix” rules and “pattern” rules
  - Example:

```
%.class: %.java  
    javac $< # we need the $< here
```

- ❖ Remember that you can put *any* shell command – even whole scripts!
- ❖ You can repeat target names to add more dependencies
- ❖ Often this stuff is more useful for reading makefiles than writing your own (until some day...)

# On the topic of compilation...

*This is cool trivia, but totally unnecessary to know. Hopefully you enjoy it :)*

- ❖ What's the best compilation message?

```
$ gcc -g -Wall -std=c11 -o msg not_a_function.c
not_a_function.c:1:6: warning: 'main' is usually a
                             function [-Wmain]
```

- ❖ C will let you do many things, many of which you should not do.
  - This is an example of something you should NEVER do.
- ❖ You can write a program that runs and prints!

- Not portable though...

```
not_a_function.c
const int main[] = {
    -443987883, 440, 113408, -1922629632,
    4149, 2013696, 84869120, 15544,
    266023168, 1818576901, 1126199148, 857752915,
    169947955, 1936269379, 1701344288, 1936024096,
    -1878384268, 521126749, 1096237312, 1447165833
}
```

Credit to this blog:

<http://jroweboy.github.io/c/asm/2015/01/26/when-is-main-not-a-function.html>