C isn't that hard:

void ((*f[])()) ();*

Defines *f* as an array of unspecified size of ponters to functions that return pointers to functions with a return type of *void*.

Section 10
Function Pointers

CS 107A, Autumn 2021
Andrew Benson (adbenson@)

Don’t forget to start recording

Unix Tip Spotlight

- gdb TUI mode
  - The secret mode you probably saw CAs use
  - Enter via one of
    - <CTRL+X><CTRL+A> (hold down CTRL in between)
    - layout src
  - Known issue: display gets garbled sometimes when printing
    - Fix with either <CTRL+L> or refresh

Announcements

- Midterm Review Session
  - Saturday 10/23 4-6pm in Huang (ideally we'll takeover a room)
  - Planning to rewrite one problem tonight and release problems by Friday morning, which we'll go over
  - Snacks
- Next week Tuesday: Midterm
- Next week Thursday: Feedback session

Bird's Eye View

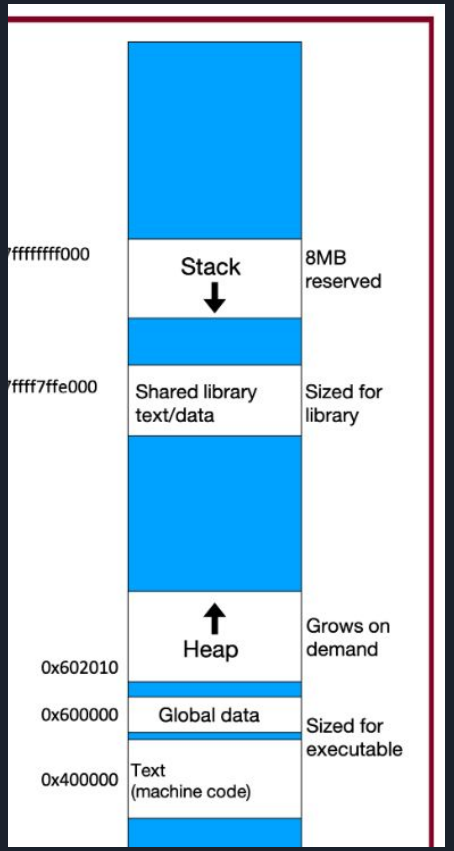
Day	Week 5 Wednesday	Thursday	Friday	Week 6 Monday	Tuesday	Wednesday	Thursday	Friday
CS 107A		Section: Function Pointers			Section: Intro x86 / Last-minute Practice (?)		Section: x86-64 ALU / Feedback Session	
CS 107	Lab 4: `void*` / Function Pointers		Lecture: Intro x86	Lecture: x86-64 ALU		Lab 5: Assembly		Lecture: x86-64 Control Flow
CS 107 assignments	assign3 due, assign4 released				Midterm			assign4 due next Wed

Agenda

- Function Pointers (+exercises)
- Function Pointers for void*
- Comparison Functions (+exercises)
- Coding Practice

Function Pointers (+exercises)

Your code is data too



How do you get a pointer to a function?

- Use &

```
int sum(int a, int b) {
  return a+b;
}
int main() {
  int (*my_sum_function_pointer)(int, int) = &sum;
  return 0;
}
```

Function Pointer Types

- https://goshdarnfunctionpointers.com/
- There's a variant of this with a name you'd expect
- https://cdecl.org/
- It can get pretty bad

```
#include <stdlib.h>

void *(*return_my_function_pointer_arg(void *(*arg)(size_t)))(size_t) {

  return arg;

}

int main() {

  void *(*malloc_ptr)(size_t) = return_my_function_pointer_arg(&malloc);

  return 0;

}
```

Fill in the correct type

```
char gimme_b() {
  return ‘b’;
}

int main() {
  ??? = &gimme_b;
  return 0;
}
```

Fill in the correct type

```
char gimme_b() {
  return ‘b’;
}

int main() {
  char (*gimme_b_ptr)() = &gimme_b;
  return 0;
}
```

Fill in the correct type

```
char *binky(struct foo *bar) {
  return bar->some_char_ptr_field;
}


int main() {
  ??? = &binky;
  return 0;
}
```

Fill in the correct type

```
char *binky(struct foo *bar) {
  return bar->some_char_ptr_field;
}

int main() {
  char *(*binky_ptr)(struct foo*) = &binky;
  return 0;
}
```

What gets printed?

```
int sum(int a, int b) {
  return a+b;
}
int square(int x) {
  return x*x;
}
int apply_the_things(int (*f)(int, int), int (*g)(int), int x, int y) {
  return f(g(x), g(y));
}
int main() {
  printf("%d\n", apply_the_things(&sum, &square, 2, 3));
  return 0;
}
```

Function Pointers for `void*`

From last section: `void*`

- You can't do anything with void* unless you're told more information about the type you're pointing to

  1) Maybe you're told the size of the type you point to – allows you to use memcpy to copy or move the values around

  2) Maybe you're given a function pointer that you're told you can use to manipulate values of this mysterious type

Example functions that allow you to use void*

```
void print_mysterious_value(void *value);

void *combine_two_mysterious_values(void *a, void *b);

char *mysterious_value_to_string(void *value);

int compare_mysterious_value(void *a, void *b);
```

Comparison Functions (+exercises)

Usage of comparison function

```
int compare_mysterious_value(void *a, void *b);
```

Mysterious Type Owner

```
// We know it's a string!
int compare_mysterious_value(
    void *a, void *b) {
  char *a_int = *(char**)a;
  char *b_int = *(char**)b;
  return strcmp(a, b);
}
```

User of generic interface

```
// No idea what type it is
void *value1 = get_mysterious_value_ptr();
void *value2 = get_mysterious_value_ptr();
if (compare_mysterious_value(v1, v2)
    == 0) {
  printf("EQUAL!\n");
}
```

Comparison Functions

- Pointers to these are always of the form
  - `int (*compare_fn)(void *a, void *b);`
- Arguments: pointers to the values being compared
- Return value:
  - Negative if a less than b
  - 0 if a equal to b
  - Positive if a more than b
- The comparison does not have to be based on literal value!

Normal Int Compare Function

```
int int_compare(void *a, void *b) {

  ???

}
```

Normal Int Compare Function

```
int int_compare(void *a, void *b) {
  int x = *(int*)a;
  int y = *(int*)b;
  if (x < y) return -1;
  if (x > y) return 1;
  return 0;
}
```

Normal Int Compare Function

```
int int_compare(void *a, void *b) {
  int x = *(int*)a;
  int y = *(int*)b;
  return x - y;
}
```

Magic Int Compare Function

```
int int_magic_compare(void *a, void *b) {
  ???
}
```

An `int` is magic if it is 7, otherwise it isn't.

```
        All other integers                    7
<---------EQUALLY NOT MAGIC-----------------MAGIC----->
```

The Magic Number Scale

Magic Int Compare Function

```
int int_magic_compare(void *a, void *b) {
  int x = *(int*)a;
  int y = *(int*)b;
  if (x != 7 && y == 7) return -1;
  if (x == 7 && y != 7) return 1;
  return 0;
}
```

An `int` is magic if it is 7, otherwise it isn't.

```
          All other integers                    7
<---------EQUALLY NOT MAGIC-----------------MAGIC----->
              The Magic Number Scale
```

Code Exercises

```
git clone /afs/ir/class/cs107a/WWW/git/section10
```