CS 241: Systems Programming
Lecture 25. Function Pointers

Spring 2020

Prof. Stephen Checkoway

Function pointers

Function pointers are pointers that point to…functions

Syntax

- `return_type (*var)(parameters);`
- `int (*f1)(void);` // f1 is a pointer to a function returning an int
- `struct foo *(*f2)(double, size_t) = blah;`

Calling a function pointer (two options)

- Pretend it's a function: `int x = f1();`
- Dereference it first: `struct foo *p = (*f2)(2.3, 82);`

Aside: C is *super* weird

Function call operator `(...)` only applies to function pointers

Functions **decay** to pointers to the function

When calling `foo(5)`, `foo` decays to a pointer and then the call happens

Assuming we have a function `void foo(int x)`, these are identical

- `foo(3)` // decay -> call
- `(&foo)(3)` // address of -> call
- `(*foo)(3)` // decay -> dereference -> decay -> call
- `(*&foo)(3)` // address of -> dereference -> decay -> call
- `(&*foo)(3)` // decay -> dereference -> address of -> call

Example

```c
#include <stdio.h>

void foo(void) { puts("foo"); }
void bar(void) { puts("bar"); }
void qux(void) { puts("qux"); }

// An array of function pointers
void (*table[])(void) = { foo, bar, qux };

int main(int argc, char *argv[argc]) {
  void (*ptr)(void) = table[argc % 3];
  ptr();
  return 0;
}
```

An actual use case

```
int atexit(void (*handler)(void));
```

- Call `atexit` and pass it a function (pointer)
- When the program exits normally (via `exit(3)` or returning from `main`), the function is called
- `_exit(2)` [defined by POSIX] or `_Exit(3)` [defined by C] don't call the atexit handlers
- Atexit handlers are called in reverse order
- Atexit handlers must not call `exit(3)`

What does this code print?

A. 1
   2
   3
   4

B. 1
   3
   2

C. 3
   1
   2

D. 3
   4
   2
   1

E. 3
   2
   1

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

void foo(void) { puts("1"); }
void bar(void) { puts("2"); }

int main(void) {
  atexit(foo);
  puts("3");
  atexit(bar);
  exit(0);
  puts("4");
  return 0;
}
```

Generic sorting

```
void qsort(void *base, size_t nel, size_t width,
           int (*compare)(void const *, void const *));
```

Takes an array, `base`, of `nel` elements, each of size `width` and a comparison function, `compare` and sorts the array

compare gets a pointer to two elements x and y and returns <0, 0, or >0 depending on the x < y, x = y, or x > y

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

enum Rank { ASSISTANT, ASSOCIATE, FULL };
char const *const ranks[] = { "Assistant", "Associate", "Full" };

struct Professor {
  enum Rank rank;
  char const *name;
};

struct Professor profs[] = {
  { .rank = ASSISTANT, .name = "Roberto Hoyle" },
  { .rank = ASSISTANT, .name = "Adam Eck" },
  { .rank = FULL,      .name = "John Donaldson" },
  { .rank = ASSISTANT, .name = "Sam Taggart" },
  { .rank = FULL,      .name = "Bob Geitz" },
  { .rank = ASSISTANT, .name = "Cynthia Taylor" },
  { .rank = ASSISTANT, .name = "Stephen Checkoway" },
};
```

```c
// Compare by descending rank and then ascending names.
int compare_profs(void const *x, void const *y) {
  struct Professor const *p1 = x;
  struct Professor const *p2 = y;
  if (p1->rank > p2->rank)
    return -1;
  if (p1->rank < p2->rank)
    return 1;
  return strcmp(p1->name, p2->name);
}

int main(void) {
  size_t num_profs = sizeof profs / sizeof profs[0];

  qsort(profs, num_profs, sizeof profs[0], compare_profs);

  for (size_t i = 0; i < num_profs; ++i)
    printf("%s, %s Professor\n", profs[i].name, ranks[profs[i].rank]);

  return EXIT_SUCCESS;
}
```

```
$ ./profs
Bob Geitz, Full Professor
John Donaldson, Full Professor
Adam Eck, Assistant Professor
Cynthia Taylor, Assistant Professor
Roberto Hoyle, Assistant Professor
Sam Taggart, Assistant Professor
Stephen Checkoway, Assistant Professor
```

```c
// Compare by names only.
int compare_by_names(void const *x, void const *y) {
  struct Professor const *p1 = x;
  struct Professor const *p2 = y;
  return strcmp(p1->name, p2->name);
}
```

```
$ ./profs
Adam Eck, Assistant Professor
Bob Geitz, Full Professor
Cynthia Taylor, Assistant Professor
John Donaldson, Full Professor
Roberto Hoyle, Assistant Professor
Sam Taggart, Assistant Professor
Stephen Checkoway, Assistant Professor
```

Generic binary search

```
void *bsearch(void const *key, void const *base,
              size_t nel, size_t width,
              int (*compare)(void const *, void const *));
```

Takes a `key`; a sorted array, `base`, of `nel` elements each of size `width`; and a comparison function and returns a pointer to the element matching the key or **`NULL`** if none do

```
int compare(void const *key, void const *elem);
```

- Compares the key with the element, returning <0, 0, or >0
- `key` and `elem` need not point to the same type

```c
int find_by_name(void const *key, void const *elem) {
  char const *name = key;
  struct Professor const *p = elem;
  return strcmp(name, p->name);
}

// Assuming profs is sorted according to name.
struct Professor *steve;
steve = bsearch("Stephen Checkoway", profs, num_profs,
                sizeof profs[0], find_by_name);
if (steve)
  puts(ranks[steve->rank]); // Prints "Assistant".
```

What happens if we call bsearch() on an array that isn't sorted? Assume that the array contains an element that matches the given key.

A. A pointer to the matching element is returned.

B. `NULL` is returned.

C. Either a pointer to the matching element or `NULL` is returned.

D. Undefined behavior

Signals (brief intro)

Signals are the mechanism the OS uses to communicate with UNIX processes

There are a whole bunch of signals (see signal(7) or run `$ kill -l`)

SIGINT is the signal that is sent when the user presses control-c

A signal handler can be installed for many (but not all) signals

- Signal handlers are ***extremely*** limited
- They can't call most library functions (including `malloc(3)` and `printf(3)`)
- They should essentially set a variable of type `volatile sig_atomic_t` and return

C is ridiculous again

The signal function takes an int and a function pointer as arguments and returns a function pointer:

```
void (*signal(int signum, void (*handler)(int)))(int);
```

This is totally unreadable.

Use a typedef!

- ```
  typedef void (*sighandler_t)(int);
  sighandler_t signal(int signum, sighandler_t handler);
  ```
  ```

```c
#include <signal.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

static volatile sig_atomic_t done;
static void handler(int signum) { done = 1; }

int main(void) {
  signal(SIGINT, handler);

  time_t start_time = time(0);
  time_t now = start_time;
  while (!done) {
    printf("The current time is %s", ctime(&now));
    sleep(10);
    now = time(0);
  }
  long diff = now - start_time;
  printf("\e[G\e[K%ld seconds elapsed\n", diff);
  return 0;
}
```

```c
#include <signal.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

static volatile sig_atomic_t done;
static void handler(int signum) { done = 1; }

int main(void) {
  signal(SIGINT, handler);

  time_t start_time = time(0);
  time_t now = start_time;
  while (!done) {
    printf("The current time is %s", ctime(&now));
    sleep(10);
    now = time(0);
  }
  long diff = now - start_time;
  printf("\e[G\e[K%ld seconds elapsed\n", diff);
  return 0;
}
```

```
$ ./a.out
The current time is Sun Nov  3 18:36:43 2019
The current time is Sun Nov  3 18:36:53 2019
The current time is Sun Nov  3 18:37:03 2019
26 seconds elapsed
```

In the previous example, after the signal handler runs, the code essentially performs

```
long diff = time(0) - start_time;
printf("seconds elapsed\n", diff);
exit(0);
```

Could this code be placed into the signal handler instead and would that be a better approach? (Assume start_time were changed to be global.)

A. Yes, that would be better

B. Yes, but it's not any better

C. Yes, but it would be worse

D. No

In-class exercise

https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-25.html

Grab a laptop and a partner and try to get as much of that done as you can!