Function Pointers in C

Function

Set of executable statements performing a definite task.

Pointer

Variable that contains the address of a variable.

Function Pointers

A function pointer points to the address of a function instead of the address of data.

Functions are defined by their return value and their signature. So in order to fully describe a function, we must include its return value and the type of each parameter it accepts.

Declaring a function pointer

```
int add(int n, int m) { return n + m; } // function

int (*function_ptr) (int, int);
function_ptr = &add;

int sum = (*function_ptr)(2, 3); // sum == 5
```

Passing function pointers

```
int add_1_and_2(int (*function_ptr)(int, int)) {
    return (*function_ptr)(1, 2);
}

int (*function_ptr) (int, int) = &add;

int ret = add_1_and_2(function_ptr); // ret == 3
```

Returning function pointers

So far so good... Now it gets confusing.

```
int (*function_factory(int n))(int, int) {
    printf("Got parameter %d", n);
    int (*function_ptr)(int, int) = &add;
    return function_ptr;
}
```

“C is sometimes castigated for the syntax of its declarations, particularly ones that involve pointers to functions. The syntax is an attempt to make the declaration and the use agree...”

-- K & R

Returning function pointers

Returning a function pointer appears messy. It's nice to be able to use typedefs.

```
typedef int (*my_func_def)(int, int);

my_func_def function_factory(int n) {
   printf("Got parameter %d", n);
   my_func_def function_ptr = &add;
   return function_ptr;
}
```

Array of function pointers

Like normal pointers, we may have an array of function pointers.

```
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

int (*function_ptr_arr[])(int, int) = {add, sub};

int res_add = function_ptr_arr[0](1, 2); // res_add == 3
int res_sub = function_ptr_arr[1](1, 2); // res_sub == -1
```

A Brief Summary

1. Function pointers point to code, not data.
2. They point to a single instance of a function already present in the memory.
3. They can be passed to and returned from functions.
4. Like normal pointers, we can have an array of function pointers.

Uses of function pointers

The most common use of function pointers is **callback**.

This is a way to **parameterize** a function: some part of its behavior is not hard-coded into the function, but into the callback function. Callers can make the function behave differently by passing different callback functions.

Callback example

```
struct student {
   int roll_num, marks;
};

int student_roll_comp( void * s1, void * s2) {
   int roll1 = (( struct student *) s1)->roll_num,
       roll2 = (( struct student *) s2)->roll_num;
   if (roll1 < roll2)       return -1;
   else if (roll1 > roll2)  return 1;
   else                     return 0;
}

int student_marks_comp( void * s1, void * s2) {
   int marks1 = (( struct student *) s1)->marks,
       marks2 = (( struct student *) s2)->marks;
   if (marks1 < marks2)         return -1;
   else if (marks1 > marks2)    return 1;
   else                         return 0;
}
```

Callback example

```
void ins_sort(void * arr, int len, int size, int (*comp) (void *, void *)) {
   char *ptr1, *ptr2, *start = arr, *end = arr + len * size,
           *temp = malloc(size);

   for (ptr1 = start + size; ptr1 <= end - size; ptr1 += size) {
       ptr2 = ptr1 - size;
       memcpy(temp, ptr1, size);
       while (ptr2 >= start && comp(ptr2, temp) >  0) {
           memcpy(ptr2 + size, ptr2, size);
           ptr2 -= size;
       }
       memcpy(ptr2 + size, temp, size);
   }
}
```

Callback example

```
void disp_students(struct student arr[], int len) {
   int i;
   for (i = 0; i < len; i++) {
       printf("{Roll=%2d, Marks=%2d}, " , arr[i].roll_num, arr[i].marks);
   }
   printf("\n");
}

struct student students[] = {{ 6, 56}, {2, 32}, {3, 4}, {1, 86}};
disp_students(students, 4);
ins_sort(students, 4, sizeof(struct student), &student_roll_comp);
disp_students(students, 4);
ins_sort(students, 4, sizeof(struct student), &student_marks_comp);
disp_students(students, 4);
```

```
{Roll= 6, Marks=56}, {Roll= 2, Marks=32}, {Roll= 3, Marks= 4}, {Roll= 1, Marks=86},
{Roll= 1, Marks=86}, {Roll= 2, Marks=32}, {Roll= 3, Marks= 4}, {Roll= 6, Marks=56},
{Roll= 3, Marks= 4}, {Roll= 2, Marks=32}, {Roll= 6, Marks=56}, {Roll= 1, Marks=86},
```

Callback in the C library

Callback is also employed in the C library. Some notable examples include the following:

1. Qsort

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))
```

2. Bsearch

```
void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *))
```

Uses of function pointers

Another common use of function pointers is in jump tables. They are commonly used in finite state machines and also in compiler optimizations.

```
switch (state)
    case A:
        switch (event):
            case e1: ....
            case e2: ....
    case B:
        switch (event):
            case e3: ....
            case e1: ....
```

We can make a 2D array of function pointers and simply call

`handle_event[state][event]`

Sources

- Google
- Wikipedia
- Stackoverflow
- Oracle Blogs
- K & R

Thank you