Pointers to Functions

Based on slides from K. N. King and Dianna Xu
Bryn Mawr College
CS246 Programming Paradigm

Pointers to Functions

• C doesn’t require that pointers point only to data; it’s also possible to have pointers to functions.
  • Function pointers point to memory addresses where functions are stored.
    ○ int (*fp) (void);
    ○ A function pointer determines the prototype of a function, but not its implementation.
    ○ Any function of the identical prototype can be assigned to the function pointer.
    ○ A function without its argument lists becomes its own pointer.
    ○ Function pointers do not need & or *

Function Pointer: Example

```c
#include <stdio.h>

int main() {
    int i = 1;
    int (*fp) (const char *, ...) = printf;
    fp("i == \%d\n", i);
    fp("i == \%d\n", i);
    return 0;
}
```

• Notice no need for &printf or (*fp)
• But I like to stick with (*fp)

Overriding Functions

• Also known as late-binding, this is emulated in C with function pointers.
• Together with generic pointers (void *), one can have typeless parameters and functions.

```c
void fd (void *base, size_t n, size_t size){
    double *p = base;
    for (p = base; p < (double*) (base+(n*size)); p++) ;
}
```

```c
int main() {
    double a[5] = {0, 1, 2, 3, 4};
    if (type == DOUBLE) {
        void (*f) (void *, size_t, size_t) = fd;
        (*f)(a, 5, sizeof(double));
    }
}
```

Printing of Generic Arrays

```c
typedef struct {
    double x;
    double y;
} Point;

int main() {
    double a[5] = {0, 1, 2, 3, 4};
    int b[5] = {5, 6, 7, 8, 9};
    Point ps[2] = {{0.5, 0.5}, {1.5, 2.5}};
    gp(a, 5, sizeof(double));
    gp(b, 5, sizeof(int));
    gp(ps, 2, sizeof(Point));
}
```

```c
void gp (void *b, size_t n, size_t size){
    char *p;
    for (p = (char *)b; p < ((char *)b) + n*size; p += size){
        switch (size) {
            case sizeof(double):
                printf("%.2f ", *(double*)p);
                break;
            case sizeof(int):
                printf("%d ", *(int*)p);
                break;
            case sizeof(Point):
                printf("x = %.2f\n", ((Point *)p)->x);
                printf("y = %.2f\n", ((Point *)p)->y);
                break;
        }
    }
    printf("\n");
}
```
The **qsort** Function

- Some of the most useful functions in the C library require a function pointer as an argument.
- One of these is `qsort`, which belongs to the `<stdlib.h>` header.
- `qsort` is a general-purpose sorting function that’s capable of sorting any array.

The **qsort** Algorithm

- Assume that the array to be sorted is indexed from 1 to `n`.

**qsort algorithm**

1. Choose an array element `e` (the “partitioning element”), then rearrange the array so that elements 1, …, `i – 1` are less than or equal to `e`, element `i` contains `e`, and elements `i + 1`, …, `n` are greater than or equal to `e`.
2. Sort elements 1, …, `i – 1` by using Quicksort recursively.
3. Sort elements `i + 1`, …, `n` by using Quicksort recursively.

The **qsort** Function

- Example of partitioning an array:

The **qsort** Example

```c
int vs[] = {40, 10, 100, 90, 20, 25};
int comp ( const void *a, const void *b){
return ( *(int*)a - *(int*)b );
}
int main () {
qsort (vs, 6, sizeof(int), comp);
}
```
**Compar**

```c
int comp_nodes (const void *a, const void *b) {
    struct Node *n1 = a; struct Node *n2 = b;
    if ( (n1->num < n2->num) return -1;
    else if (n1->num > n2->num) return 1;
    else return 0;
/* or
    return ((struct Node *)n1)->num - ((struct Node
    *)n2)->num; */
qsort(nodes, 10, sizeof(struct Node), comp_nodes);
```

**The qsort Example**

- Recall in Chapter 16, we have the inventory array:

```c
struct part {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} inventory[MAX_PARTS];
```

- To sort the inventory array using qsort:

```c
qsort(inventory, num_parts, 
    sizeof(struct part), compare_parts);
```

- compare_parts is a function that compares two part structures.
- Writing the compare_parts function is tricky.
- qsort requires that its parameters have type void *, but we can’t access the members of a part structure through a void * pointer.
- To solve the problem, compare_parts will assign its parameters, p and q, to variables of type struct part *.

**The qsort Function**

- A version of compare_parts that can be used to sort the inventory array into ascending order by part number:

```c
int compare_parts(const void *p, const void *q) {
    const struct part *p1 = p;
    const struct part *q1 = q;
    if (p1->number < q1->number)
        return -1;
    else if (p1->number == q1->number)
        return 0;
    else
        return 1;
}
```

- Most C programmers would write the function more concisely:

```c
int compare_parts(const void *p, const void *q) {
    if (((struct part *) p)->number <
    ((struct part *) q)->number)
        return -1;
    else if (((struct part *) p)->number ==
    ((struct part *) q)->number)
        return 0;
    else
        return 1;
}
```

- compare_parts can be made even shorter by removing the if statements:

```c
int compare_parts(const void *p, const void *q) {
    return ((struct part *) p)->number -
    ((struct part *) q)->number;
}
```
**The qsort Function**

- A version of `compare_parts` that can be used to sort the `inventory` array by part name instead of part number:

  ```c
  int compare_parts(const void *p, const void *q) {
      return strcmp(((struct part *) p)->name, ((struct part *) q)->name);
  }
  ```

**Other Uses of Function Pointers**

- Although function pointers are often used as arguments, that’s not all they’re good for.
- C treats pointers to functions just like pointers to data.
- They can be stored in variables or used as elements of an array or as members of a structure or union.
- It’s even possible for functions to return function pointers.

**Other Uses of Function Pointers**

- A variable that can store a pointer to a function with an `int` parameter and a return type of `void`:

  ```c
  void (*pf)(int);
  ```

  If `f` is such a function, we can make `pf` point to `f` in the following way:

  ```c
  pf = f;
  ```

  We can now call `f` by writing either

  ```c
  (*pf)(i);
  ```

  or

  ```c
  pf(i);
  ```

**Other Uses of Function Pointers**

- An array whose elements are function pointers:

  ```c
  void (*file_cmd[])(void) = {new_cmd, 
      open_cmd, 
      close_cmd, 
      close_all_cmd, 
      save_cmd, 
      save_as_cmd, 
      save_all_cmd, 
      print_cmd, 
      exit_cmd};
  ```

  We could get a similar effect with a `switch` statement, but using an array of function pointers provides more flexibility.

**Program: Tabulating the Trigonometric Functions**

- The `tabulate.c` program prints tables showing the values of the `cos`, `sin`, and `tan` functions.
- The program is built around a function named `tabulate` that, when passed a function pointer `f`, prints a table showing the values of `f`.
- `tabulate` uses the `ceil` function.
- When given an argument `x` of `double` type, `ceil` returns the smallest integer that’s greater than or equal to `x`. 

  ```c
  Program: Tabulating the Trigonometric Functions
  ```
Program: Tabulating the Trigonometric Functions

- A session with `tabulate.c`:

  Enter initial value: 0
  Enter final value: 0.5
  Enter increment: 0.1

<table>
<thead>
<tr>
<th>x</th>
<th>cos(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>0.10000</td>
<td>0.99500</td>
</tr>
<tr>
<td>0.20000</td>
<td>0.98007</td>
</tr>
<tr>
<td>0.30000</td>
<td>0.95534</td>
</tr>
<tr>
<td>0.40000</td>
<td>0.92106</td>
</tr>
<tr>
<td>0.50000</td>
<td>0.87758</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>sin(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>0.10000</td>
<td>0.09983</td>
</tr>
<tr>
<td>0.20000</td>
<td>0.19867</td>
</tr>
<tr>
<td>0.30000</td>
<td>0.29552</td>
</tr>
<tr>
<td>0.40000</td>
<td>0.38942</td>
</tr>
<tr>
<td>0.50000</td>
<td>0.47943</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>tan(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>0.10000</td>
<td>0.10033</td>
</tr>
<tr>
<td>0.20000</td>
<td>0.20271</td>
</tr>
<tr>
<td>0.30000</td>
<td>0.30934</td>
</tr>
<tr>
<td>0.40000</td>
<td>0.42279</td>
</tr>
<tr>
<td>0.50000</td>
<td>0.54630</td>
</tr>
</tbody>
</table>

---

`tabulate.c`

```c
/* Tabulates values of trigonometric functions */
#include <math.h>
#include <stdio.h>

void tabulate(double (*f)(double), double first, double last, double incr);

int main(void)
{
    double final, increment, initial;
    printf("Enter initial value: ");
    scanf("%lf", &initial);
    printf("Enter final value: ");
    scanf("%lf", &final);
    printf("Enter increment: ");
    scanf("%lf", &increment);

    printf("\n      x        cos(x)
   -------    -------\n");
    tabulate(cos, initial, final, increment);

    printf("\n      x        sin(x)
   -------    -------\n");
    tabulate(sin, initial, final, increment);

    printf("\n      x        tan(x)
   -------    -------\n");
    tabulate(tan, initial, final, increment);
    return 0;
}

void tabulate(double (*f)(double), double first, double last, double incr)
{
    double x;
    int i, num_intervals;
    num_intervals = ceil((last - first) / incr);
    for (i = 0; i <= num_intervals; i++)
    { x = first + i * incr;
        printf("%10.5f %10.5f\n", x, (*f)(x));
    }
```

---