More advanced C

Function Pointers,
C odds and ends
Today

- Function pointers
- More on string manipulation
- Changing pointers
Function Pointers

King, chapter 22
Pointers to Functions

• Since a pointer is just an address, we can have pointers to functions!

```c
int cube(int x) {
    return x*x*x;
}

int (*f)(int); /*Define a function pointer*/
f = cube;  /* Assign it to point to the memory location where f is stored */

/* Call the function that f points to */
printf ("%d\n", (*f)(5));
```
Pointers to Functions

• Since a pointer is just an address, we can have pointers to functions!

```c
int cube(int x) {
    return x*x*x;
}

int (*f)(int);
f = cube;

/*Call the function that f points to */
printf("%d\n", (*f)(5));
```
Example

```c
void map(int *nums, int size, int (*f)(int)) {
    int i;
    for(i = 0; i < size; i++)
        nums[i] = (*f)(nums[i]);
}

int square(int x) { return x*x; }

int div2(int x) { return x/2; }

int main() {
    int a[8] = {2, 4, 6, 8, 10, 12, 14, 16};
    int i;
    map(a, 8, square);
    map(a, 8, div2);

    for(i = 0; i < 8; i++) {
        printf("a[%d] = %d
", i, a[i]);
    }
    return 0;
}
```
Example: qsort

- Example – `qsort` generic quick sort routine:

```c
void qsort(void * base, size_t nmemb, size_t size,
           int (* compar)(const void *, const void *) );
```

- Can use any comparison function
- Just give a pointer to a function like `strcmp`
Example: qsort

• qsort generic quick sort routine:

```c
void qsort(void * base, size_t nmemb, size_t size,
           int (* compar)(const void *, const void *) );
```

/* qsort example */
int array[] = { 23, 99, 114, -5, 0, 44, 83 };
int compare_ints (const void * a, const void * b) {
    return ( *(int*)a - *(int*)b );
}

int main () {
    int i;
    qsort(array, 7, sizeof(int), compare_ints);
    for (i = 0; i < 7; i++)
        printf("%d ", array[i]);
    return 0;
}
```

Why do we do this?
Example: qsort

- **qsort** generic quick sort routine:

```c
void qsort(void * base, size_t nmemb, size_t size, 
    int (* compar)(const void *, const void *) );
```

/* qsort example – strings! Let's sort the commandline args */

```c
int cmp_strings(const void *p1, const void *p2) {
    return strcmp(* (const char **) p1, * (const char **) p2);
}
```

```c
int main (int argc, char **argv) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <string>...
", argv[0]);
        exit(1);
    }
    qsort(&argv[1], argc - 1, sizeof(char *), cmpstringp);
    for (int i = 1; i < argc; i++)
        printf("%s ", argv[i]);
    return 0;
}
```

What does this mean?
Example: bsearch

• Another example: bsearch() for binary search

```c
void *bsearch(const void *key, const void *base,
               size_t nmemb, size_t size,
               int (*compar)(const void *, const void *));
```

• Same idea, we provide the comparison function

• Homework:
  – Check the man pages – useful example!
  – Read the example and run it
  – Modify it to search by month number instead of month name!
Unfinished business:
More details on strings and pointers
Traversing strings using pointers

- Implement strlen without using [] notation:

```c
int mystrlen(const char* string) {
    int len;

    for(len = 0; *string != '\0'; len++, string++) ;

    return len;
}
```
Another C string example

• The next few slides show an example of how to implement a string comparison function.

• Of course, you should be using the C library strcmp or strncmp to compare strings.
int main(int argc, char **argv)
{
    char *result;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s string1 string2\n", argv[0]);
        exit(1);
    }
    switch (mystrcmp(argv[1], argv[2])) {
        case -1: result = "less than"; break;
        case 0: result = "equal to"; break;
        case 1: result = "greater than"; break;
        default: result = "causing a problem comparing to"; break;
    }
    printf("%s is %s %s\n", argv[1], result, argv[2]);
    return 0;
}
int mystrcmp (const char *a, const char *b) {
    while (*a && *b && *a == *b) {
        a++; b++;
    }
    return (*a - *b);
}
mystrcmp.c (return -1, 0, or 1)

```c
int mystrcmp (const char *a, const char *b) {
    while (*a && *b && *a == *b) {
        a++; b++;
    }
    if (*a < *b)
        return -1;
    else if (*a > *b)
        return 1;
    else /* a same as b */
        return 0;
}
```

returns sign of

\[ *a - *b \]
Splitting a string into tokens

• Consider this command
  – `cp -r srcfolder destfolder`

• Separators: characters that delimit the tokens
  – Can be anything we want: whitespaces, tabs, commas, etc.

• Break it down into tokens
  ⇒ 4 tokens: “cp”, “-r”, “srcfolder”, “destfolder”
  – Think of argv!

• We want to break down a line into tokens, considering a specific type of separators!
Splitting a string into tokens

• Predefined ways:
  – strtok() function
  – Check the man pages for examples
  – Not thread-safe though! Check strtok_r()
Splitting a string into tokens

• Implement our own - two ways:
  – 1. Destructive (the initial string is modified)
  – 2. Non-destructive (the initial string is left intact)

• Destructive:
  – String: “cp –r srcdir destdir\0”
  – Separator characters: space (‘ ‘), tab (‘\t’), newline (‘\n’)

• Non-destructive
  – same idea, but do not alter the separator characters
  – extract each token instead and copy to a separate location in memory
Destructive tokenize

```c
void parse_string(char *str, char **tokens) {

    while (*str != '\0') {
        /* Replace all whitespaces with '\0' */
        while (*str == ' ' || *str == '	' || *str == '
')
            *str++ = '\0';

        /* Next token points starts at this position in the string */
        *tokens++ = str;

        /* Ignore non-whitespaces, until next whitespace delimiter*/
        while (*str != '\0' && *str != ' ' &&
                   *str != '\t' && *str != '\n')
            str++;
    }

    *tokens = '\0';
}
```
void parse_string(char *str, char **tokens) {
    int token_number = 0;
    while (*str != '\0') {
        /* Ignore whitespaces */
        while (*str == ' ' || *str == '	' || *str == '
') str++;
        char *start = str;
        /* Ignore non-whitespaces, until next whitespace delimiter*/
        while (*str != '\0' && *str != ' ' &&
               *str != '	' && *str != '
')
            str++;
        /* str now points to the end of the token => extract it */
        tokens[token_number] = malloc(str-start+1);
        memcpy(tokens[token_number], start, str-start);
        tokens[token_number][str-start] = '\0';
        token_number++;
    }
}
Changing pointers
int i = 19;
int *p;
int *q;
*p = i; /*error*/
q = &i /* valid */

<table>
<thead>
<tr>
<th>Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>p</td>
</tr>
<tr>
<td>q</td>
</tr>
</tbody>
</table>

printf("%d", *q); => 19

Important:
&q = 0x80494e0 (where q is stored)
q = 0x80494e0 (where q points to)
*q = 19 (the value pointed to by q)
Remember pass-by-value

- Passing a pointer to a function creates a copy of the pointer (a copy of the address where it points to)
  => Once the function exits, the int value is changed.

```c
void increment (int *x) {
    int * p = malloc (sizeof(int));
    *p = 10; // assign a value in the location where p points to
    *x = *p; // modify the value in the address where x points
}
```

```c
int main () {
    int *a = malloc( sizeof(int) );   *a = 5;
    increment(a);
}
```

What do we know?
- ‘x’ is an address
- ‘x’ is a copy of ‘a’, points to same memory location
  => Changing the value at that location => *a changes

*a will be 10 now*
Changing the pointer

• What if we want to change the pointer?

```c
void increment (int *x) {
    int * p = malloc (sizeof(int));
    *p = 10; // assign a value in the location where p points to
    x = p; // now make x point to the same addr where p points
}

int main () {
    int *a = malloc( sizeof(int) );
    *a = 5;
    increment(a);

    *a will be still 5!!
    ....
```

Does a’s address change?
- ‘x’ is just a copy of the address passed to the func.
- once the function exits, the copy is gone!

=> Modifying ‘x’ does NOT affect ‘a’!
What if we need to change the pointer? How?

2 ways: a) return the new pointer

```c
int* increment (int *x) {
    int * p = malloc (sizeof(int));
    *p = 13; // assign a value in the location where p points to
    x = p; // now make x point to the same addr where p points
    return x;
}
```

```c
int main () {
    int *a = malloc( sizeof(int) );   *a = 5;
    a = increment(a);
```

- ‘x’ is just a copy of the address passed to the func.
- once the function exits, the copy is returned!
- the return value is assigned to ‘a’ => now ‘a’ points to where the copy used to!

*a will be 13 now!
a now points to a diff addr

Limitations?

What about modifying multiple pointers in a function?
Changing the pointer

• Option b) Pass the address of the pointer (a pointer to a pointer, basically)

```c
void increment (int **x) {
    int * p = malloc (sizeof(int));
    *p = 13; // assign a value in the location where p points to
    *x = p; // make *x point to the same addr where p points
    // no return, void
}

int main () {
    int *a = malloc( sizeof(int) );
    *a = 5;
    increment(&a);
}
```

- the `increment(&a)` call passes the address where the pointer is stored, aka “the address of the address where value 5 is stored”!

=> ‘x’ is a copy of the address of ‘a’; by changing *x, we also alter where ‘a’ points to!

*a will be 13 now!
*a now points to a diff addr

27