CSC 357 Lecture Notes Week 2, Part 2

Details of Pointers, Arrays, and Structs
I. Pointers and arrays (K&R chapter 5).

A. A *pointer* in C is a memory address.

B. An *array* in C is a block of memory, to which a pointer can point.

C. Hence, pointers and arrays are closely related.
II. Pointers and addresses (K&R Section 5.1).

A. Memory in C is laid out as consecutively numbered cells.

1. Most typically, smallest one `char`, or "byte".

2. Cells are typically grouped into bigger segments, often referred to as "words".
B. Memory viewed as byte-addressable words:
computer memory addresses

0x0

0x209b4
0x209b8
0x209bc
0x209c0
0x209c4

...
C. Variables in C occupy some place in memory, and hence have an address.

1. C provides ’&’ operator to get the address of (i.e., a pointer to) any variable.

2. C also provides pointer dereferencing operator ’*’, to access the value that a pointer points to.

3. The ’*’ symbol is also used to declare a variable as a pointer.
D. Here are examples from pp. 94 and 95 of K&R:

```c
int x = 1, y = 2, z[10];
int *ip;  /* ip is a pointer to int */
int *iq;  /* iq is another pointer to int */
```
ip = &x;         /* ip now points to x */
y = *ip;          /* y is now 1 */
*ip = 0;          /* x is now 0 */
ip = &z[0];      /* ip now points to z[0] */
While these examples are rather artificial, they illustrate the fundamentals of ‘&’ and ‘*’.
E. The size of memory segment to which a pointer refers is constrained by its type declaration.

1. E.g., an \texttt{int} pointer points to a 4-byte word (assuming that’s how big an \texttt{int} is).

2. A \texttt{char} pointer points to a single byte.

3. These constraints impact the way address arithmetic is performed, as we’ll see shortly.
III. Pointers and function args (K&R Section 5.2).

A. Function arguments are passed by value.

B. *Call-by-reference* parameter passing can be achieved using ‘*’ and ‘&’.

C. Consider the versions of the *swap* function on pages 95 and 96 of K&R:
/ * WRONG, in that swap(a,b) does not swap vars a and b */

```c
void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

`versus`
/* CORRECT, in that swap(&a,&b) does swap a and b */
void swap(int* x, int* y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

D. See the picture on Page 96 of K&R.
IV. Pointers, arrays, addr arith (K&R Section 5.3).

A. There is a strong relationship between pointers and arrays in C; specifically:

1. Given the declarations

   ```c
   int a[10]
   int *pa;
   ```
the following assignments are legal

\[
\begin{align*}
\text{pa} &= \text{a}; \\
\text{pa} &= \&\text{a}[0];
\end{align*}
\]

and have exactly the same effect, which is that the pointer \( \text{pa} \) is assigned to point to the zeroth element of array the \( \text{a} \).
2. Given the preceding decls of \( a \) and \( pa \), the following equivalences hold for all \( 0 \leq i < 10 \):

\[
\begin{align*}
a[i] &= * (pa + i) \\
\text{pa}[i] &= * (a + i) \\
\&a[i] &= pa + i
\end{align*}
\]
3. Used with formal parameters in a function def, \( t[\ ] \) and \( t^* \) denote the same type.

B. The following picture illustrates the pointer/array relationships just described:
pa

pa+1

pa+2

pa+9

pa+10

an invalid address

a[0]

a[1]

a[9]

a[10]

an invalid array reference
V. Memory allocation (K&R Section 5.4).

A. The ‘&’ operator is of limited practical utility for building dynamically linked data structures.

B. As illustrated in Part 1 of this week’s lecture notes, programmers need to allocate new blocks of memory for such data structures.

C. Section 5.4 of K&R talks about the implementation of a simplistic alloc function.
D. In practice, C programmers use the library-supplied `malloc`, as well as derivatives `calloc` and `realloc`.

E. The signature of `malloc` is the following:

```c
void* malloc(size_t size);
```
1. The type `size_t` is an `int` or `long`; the `size` parameter is the number of bytes to be allocated.

2. `void*` is the type of a generic pointer; in practice, the `void*` return value from `malloc` is always cast to a more specific type of pointer.
F. Here are typical examples of malloc:

/* Allocate memory for a 100-char string. */
char* some_string = (char*) malloc(100);

/* Allocate memory for an integer array ... */
int* a = (int*) malloc(array_size);

/* Allocate memory for a structured data value. */
typedef struct {int x; char y; char z[20];} SomeStruct;
SomeStruct* s = (SomeStruct*) malloc(sizeof(SomeStruct))

G. The last of these examples is so frequently used, that a macro like `new` can be very handy.

1. The definition of `new` is:
   ```c
   #define new(t) (t*) malloc(sizeof(t))
   ```

2. It is used, for example, like this:
   ```c
   SomeStruct* s = new(SomeStruct);
   ```
H. You should read the man page for `malloc` and related library functions (`man malloc`).
VI. **Char ptrs and functions (K&R Section 5.5).**

A. C has double-quoted string constants like "Hello world\n", used as a function arg to `printf`.

B. String constants can be used to initialize `char*` and `char[]` string variables, as in

```c
char amessage[] = "now is the time";
char* pmessage = "now is the time";
```
1. As explained on Page 104 of K&R, `amessage` and `pmessage` are not equivalent definitions.

2. `amessage` is an array of `strlen("now is the time")+1` elements, whose chars can be changed, but `amessage` as a whole cannot be reassigned.

3. `pmessage` is a pointer to a block of characters, changing the characters of which is `undefined`, but `pmessage` as a whole can be reassigned.
4. The following lines of code illustrate the differences between the two string variables:

```c
/* Next two lines are OK */
amessage[2] = 't';
pmessage = "another message";

/* Next three lines are NOT OK */
amessage = "another message";
pmessage[2] = 't';
free(pmessage);
```
C. There are a number of lib functions that allow strings to be operated on at the character level, including: `strcpy`, `strcat`, and `strcmp`.

1. K&R discusses implementations of these on pages 105-107.

2. You should read the man page descriptions of these and related string-processing functions (`man string(3C) on falcon/hornet`).
VII. More on pointers and arrays
(K&R Sections 5.6 - 5.10, 5.12).

A. You should read and understand these sections of K&R.

B. You can skip Section 5.10 for now.