LECTURE 17: POINTERS / ARRAYS (K&R §§ 5.6, 5.10)

POINTER REVIEW

```
int a[ ] ={1,3,5,7,9,11,13,15,17,19};
int *pa = &a[4];
int *pb = &a[1];
```

What is the value of: *(a + 2)?	same as a[2]
What is the value of: pa – pb?	3
What is the value of: pb[1]?	same as a[2]
What is the effect of: *pa += 5?	a[4] += 5
What is the effect of: *(pa += 2)?	pa = &a[6] value is a[6]
What is the effect of: *(a += 2)?	illegal. array name is a constant.
What is the value of: pa[3]? (after above assignment pa = &a[6])	19

VALID POINTER ARITHMETIC

Set one pointer to the value of another pointer of the same type. If they are of different types, you need to cast.	pa = pb;
Add or subtract a pointer and an integer or an integer variable	pa + 3; pa – 5; /* i is an int */ pa + i;
Subtract two pointers to members of same array. (Note: result is an integer.)	pa – pb;
Compare two pointers to members of same array.	if(pa <= pb)

Assign a pointer to zero/NULL (defined in stdio, guaranteed not to point to a memory location).	pa = NULL;
	/* same as */
	pa = 0;

Compare a pointer to NULL.	if(pa != NULL)
	/* but NOT */
	if(pa > NULL)

Note: A NULL pointer does not point to anything. (When used as a return value, it indicates failure of a function that is defined to return a pointer.)	
All other pointer arithmetic is invalid.	

K&R, page 103: It is not legal to	add two pointers,
	or to multiply or divide or shift or mask them,
	or to add float or double to them,
	or even, except for void *, to assign a pointer of one type to a pointer of another type without a cast.

VALID / INVALID POINTER ARITHMETIC EXAMPLES

```
int a[ ] ={1,3,5,7,9,11,13,15,17,19};
int *pa = &a[4];
int *pb = &a[1];
char s[ ] = "Hello, world!";
char *cp = &s[4];
```

	valid?
cp = cp – 3;	YES
pa = cp; (careful: int is 4 bytes, char is 1, possible alignment problem)	NO
pa = pa + pb;	NO
pa = pa + (pa – pb);	YES
s[4] = (cp < pa)? 'a': 'b';	NO
cp = NULL;	YES

POINTER ARRAYS

Recall that if we define:

```
  char a[10];
```

we are setting aside space in memory for the elements of array a.

a can be treated as a pointer — we can write:

```
  *a
```

or

```
  *(a + 5)
```

Now think about the declaration:

```
  char *a[10];
```

This array contains pointers to char variables or strings.

```
**a;  /* first char in string pointed to by a[0] */
*(*(a+5)+2); /* 3rd char in string pointed to by a[5] */
```

POINTERS TO POINTERS

pointers to pointers demo:

```
#include <stdio.h>

int main(void) {
  /* array of 4 digits and '/0' */
  char a[] = "1234";
  /* array of 4 letters and '/0' */
  char b[] = "abcd";
  /* array of pointers (garbage values) */
  char *ptr[5];
  /* pointer to a pointer */
  char **ptr2ptr;
  /* point first pointer in ptr array at array a */
  ptr[0] = a;
  /* point second pointer in ptr array at array b */
  ptr[1] = b;
  /* point pointer-to-pointer at array ptr */
  ptr2ptr = ptr;
  /**************print statements**************/
  /* print memory address of array a */
  printf("\nmemory address of a[]: %p\n", (void *)&a);
  /* print memory address of array b */
  printf("memory address of b[]: %p\n", (void *)&b);
  /* print value of ptr[0] */
  printf("\nvalue of ptr[0]: %p\n", (void *)ptr[0]);
  printf("contents of ptr[0]: %s\n", ptr[0]);
  /* print memory address of array a */
  printf("value of ptr[1]: %p\n", (void *)ptr[1]);
  printf("contents of ptr[1]: %s\n", ptr[1]);
  /* print ptr (addresses of a and b and then junk) */
  printf("\nlvalues held in pointer array ptr[]:\n");
  int i = 0;
  for(; i < 5; i++)
    printf("ptr[%d]: %p\n", i, (void *)ptr[i]);
  /* print memory address of ptr2ptr */
  printf("\nmemory address of ptr2ptr: %p\n",
         (void *)&ptr2ptr);
  /* print memory address of ptr array */
  printf("memory address of ptr[]: %p\n", (void *)&ptr);
  /* pointer arithmetic with ptr2ptr */
  printf("\ndereferencing ptr2ptr (print the pointer):\n");
  for(i = 0; i < 2; i++)
    printf("**ptr2ptr + %d: %p\n",
           i, (void *)*(ptr2ptr + i));
  /* pointer arithmetic with ptr2ptr */
  printf("(now the string the pointer points to):\n");
  for(i = 0; i < 2; i++)
    printf("*ptr2ptr + %d: %s\n", i, *(ptr2ptr + i));
}
```

output:

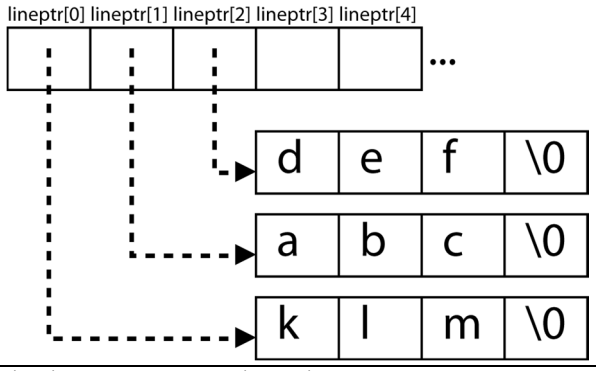
```
memory address of a[]: 0x7ffe1bb0f380
memory address of b[]: 0x7ffe1bb0f390

value of ptr[0]: 0x7ffe1bb0f380
contents of ptr[0]: 1234
value of ptr[1]: 0x7ffe1bb0f390
contents of ptr[1]: abcd

lvalues held in pointer array ptr[]:
ptr[0]: 0x7ffe1bb0f380
ptr[1]: 0x7ffe1bb0f390
ptr[2]: 0x1
ptr[3]: 0x40080d
ptr[4]: 0x7ffe1bb0f3a0

memory address of ptr2ptr: 0x7ffe1bb0f348
memory address of ptr[]: 0x7ffe1bb0f350

dereferencing ptr2ptr (print the pointer):
**ptr2ptr + 0: 0x7ffe1bb0f380
**ptr2ptr + 1: 0x7ffe1bb0f390
(now the string the pointer points to):
*ptr2ptr + 0: 1234
*ptr2ptr + 1: abcd
```

HOW TO USE AN ARRAY OF POINTERS

K&R gives an example (page 108):
To put the lines in a document in alphabetical order:
-read in the sequence of lines
-place them in blocks of memory (e.g., malloc)
-build an array of pointers to the blocks
-sort by moving pointers, not strings

Example of pointers to unsorted char strings:

```
char *lineptr[MAXLINES];
```

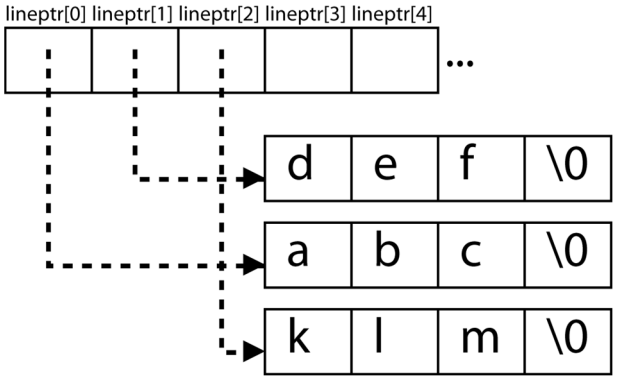


To initialize the array with fixed values:

```
  char a[] = "klm";
  char b[] = "abc";
  char c[] = "def";
  lineptr[0] = a;    /* or = &a[0];  */
  lineptr[1] = b;    /* or = &b[0];  */
  lineptr[2] = c;    /* or = &c[0];  */
```

Example of pointers to sorted char strings:

```
char *lineptr[MAXLINES];
```

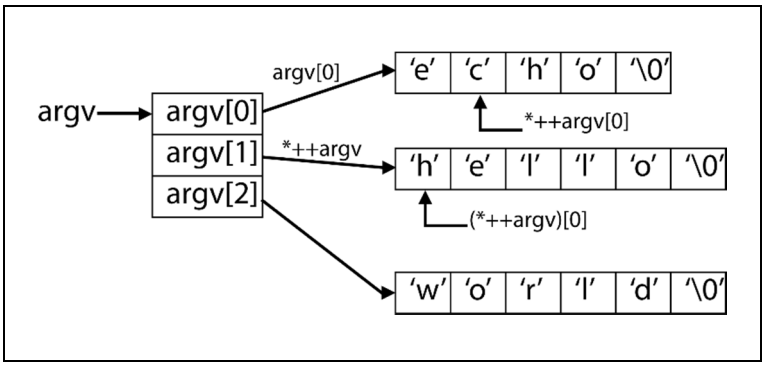


We can then output the lines in pointer order (easy way):

```
void writelines(char* lineptr[], int nlines) {
  int i = 0;
  while(i < nlines)
    printf("%s\n", lineptr[i++];
}
```

(efficient way):

```
void writelines(char** lineptr, int nlines) {
  while(nlines-- > 0)
    printf("%s\n", *lineptr++);
}
```



COMMAND-LINE ARGUMENTS

We can write our main function to be called with arguments.
We must declare them in main's function header to use them:

```
  int main(int argc, char *argv[])
```

argc	argument count the number of command-line arguments the program was invoked with.
argv	argument vector a pointer to an array of character strings that contain the arguments, one per string. argv[1] is the first optional argument argv[argc - 1] is the last optional argument the standard requires that argv[argc] be a null pointer.

Element argv[0] always points to the command name types by the user to invoke main (i.e., the name of the exectuable).

This means argc is always at least 1.
(It is 1 if there are no arguments.)

If there are other arguments:	argc will be 3 (the number of strings in argv[])
e.g., the program was compiled as echo, and the user typed: echo hello, world	argv[0] points to the beginning address of "echo" argv[1] points to the beginning address of "hello," argv[2] points to the beginning address of "world"

The program can print back the arguments typed in by the user following the echo command:

```
/* int main (int argc, char **argv)
     equivalent to */
int main (int argc, char *argv[]) {
  while (--argc > 0)
    printf("%s%s", *++argv, (argc > 1) ? " " : "");
  printf("\n");
  return 0;
}
```

K&R, page 117 has a good example of using command-line arguments. The program is called find and the authors have added the ability to provide an argument and some optional flags.
  find pattern: print the lines where pattern is found
  -x : print all lines except those containing pattern
  -n : precede each printed line by its line number

We want to be able to use the options as follows:
  find [-x] [-n] pattern OR find [-xn] pattern

The authors set up the following loop to do so:

```
/* loop through argv entries that begin with '-' */
while (--argc > 0 && (*++argv)[0] == '-')
  /* match the next letter(s) to 'x' or 'n' */
  while (c = *++argv[0])
    switch (c) {
      case 'x' : except = 1;
                 break;
      case 'n' : number = 1;
                 break;
      default  : printf("illegal option %c\n", c);
                 break;
    }
```

Note that *++argv is a pointer to a string
  (*++argv)[0] is its first character
  **++argv is an alternate form for the same

(*++argv)[0]	*++argv[0]
Increments pointer to argv[] array.	Dereference argv and obtain the pointer to the string
Dereference the pointer and get the resulting pointer to the string.	Increment the resulting pointer and it points to the second position of the string
Dereference the pointer to the string and obtain the ascii value of the first character in the string	Dereference the pointer to get the ascii value 'c'