Pointers and Arrays

- Declarations of automatic variables:

```
int x = 1, y = 2, z[10];
int *ip;        /* ip is a pointer to an int */
ip = &x;       /* ip is a pointer to int x   */
```

- Read “int *ip” right to left

  Variable ip is a pointer (*) to a variable of type int

Pointers and Memory

- Memory Address, Contents, and Variable Names

0xFF1054	0x00	0x00	0x00	0x01	x
0xFF1050	0x00	0x00	0x00	0x02	y
0xFF104C	0x??	0x??	0x??	0x??	z[9]
…		…			…
0xFF1028	0x??	0x??	0x??	0x??	z[0]
0xFF1024	0x00	0xFF	0x10	0x54	ip

Operators (& and *)

- Operator '&' → value of &x is "address of x"

```
ip = &x;             /* ip now points to x */
```

- Operator '*' → de-references a pointer (indirection)

```
y = *ip;             /* set y = x (the int at address ip) */
*ip = 0;             /* set x to 0                        */
ip = &z[0];          /* set ip to address of z[0]         */
*ip = 3;             /* set z[0] to 3                     */
*ip = *ip + 10;      /* set z[0] to 13                    */
```

- Note: & and * are unary operators - See K&R pg 53

Pointer Operations

- More pointer operation examples:

```
        *ip + 1;     /* add 1 to the int pointed to by ip                 */
        *ip += 1;    /* adds one to the int pointed to by ip              */
same    ++*ip;       /* pre increments int pointed to by ip               */
        ++(*ip);     /* same as above, binds right to left                */
        *ip++        /* point to int at pointer ip, post increment ip*/
                     /* binds right to left as *(ip++)                    */
        (*ip)++;     /* post increments int pointed to by ip              */
                     /* need ( ) - otherwise binds as *(ip++)             */
```

Incrementing Pointers

- A pointer is a number corresponding to the address of the byte used to store the variable

- When you increment a pointer, the address is incremented by the number of bytes used to store that type of variable

- For example, a char pointer cp declared as:

```
char *cp;
cp++;        /* byte address is incremented by 1 */
```

Incrementing Pointers

- For example, an int pointer ip declared as:

```
int *ip;
ip++;    /* byte address is incremented by 4 */
```

- The int pointer is not thought of as being incremented by 4 - that's hidden from the C programmer - it's said to be incremented by the size of the data type that it points at

Pointers as Function Arguments

* DOESN'T WORK *

```
 swap (i, j);
 …
void swap (int a, int b)
{
  int dummy;
  dummy = a;
  a = b;
  b = dummy;
}
```

* POINTER VERSION *

```
 swap (&i, &j);
 …
void swap (int *pa, int *pb)
{
  int dummy;
  dummy = *pa;
  *pa = *pb;
  *pb = dummy;
}
```

Declaration and Initialization

- Example of Declaration and Initialization:

  ```
  int a[10]
  int  *pa = &a[0];    /* initialize pa to point to a[0] */
  ```

- When we are initializing in the declaration, the * acts as part of the type for variable pa

- When we are initializing pa, we are setting pa (not *pa) equal to the address after = sign

- For normal assignment, we use: pa = &a[0];

Pointers and Arrays

- C treats an array name (without a subscript value) and a pointer in THE SAME WAY

- We can write:

  pa = &a[0];

- OR we can write the equivalent:

  pa = a;

- Array name "a" acts as specially initialized pointer pointing to element 0 of the array

Pointers and Arrays

- Array a is an unchanging (constant) pointer
- a = a+1; not possible (like writing 7 = 7+1)
- **defining** an array allocates the required space for contents of all array elements!
- **defining** a pointer allocates memory for the pointer but not for the data that the pointer points to!

Pointers and Arrays

- Given the way incrementing a pointer works, it's useful for accessing successive elements of an array that it points to:

    *pa means same as a[0]

    *(pa + 1) means the same as a[1]

    *(pa + m) means the same as a[m]

Pointers and Arrays

- Consider the example:

  int i, a[ ] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
  int *pa = &a[3];
  What is the value of *(pa + 3) ? (12)
  What is the value of *pa + 3 ? (9)
  What happens when i = *pa++ evaluated? (pa=&a[4])
  What is the value of i? (6)
  What happens when i = ++*pa evaluated? (++a[4])
  What is the value of i? (9)

Pointers and Arrays

- An array name can be used in an expression the same way that a pointer can be in an expression (its actual value cannot be changed permanently)

  a + m   is the same as &a[m]

  if pa = a, *(a + m) is the same as *(pa + m)

  *(pa + m) can be written as pa[m]

Examples – strlen ( )

- We now discuss how “real” C programmers deal with strings in functions

- We can call strlen( ) with arguments that are an array or a pointer to an array of type char:

  strlen (arrayname)

  strlen (ptr)

  strlen("hello, world")

Examples – strlen

- Here is a variant of the way we did strlen ( )

```
int strlen(char s[ ])
{
    int n;
    for (n = 0; s[n]; n++)  /*sum s+n, use as ptr */
      ;                     /* for test, and increment n */
    return n;
}
```

Examples – strlen

- "Real" C programmers use pointers in cases like this:

```
int strlen(char *s)   /* s is just a copy of pointer */
{                     /* so no real change to string */
      char *cp;
      for (cp = s; *s ; s++)  /* no sum, just incr. pointer */
            ;                 /* more efficient */
    return s - cp;   /* difference of pointers has int value*/
}
```

Examples – strlen

- Simplest form yet (See K&R, pg 39)

```
int strlen (char *s)
{
    char *p = s;
    while ( *p++)
            ;
    return p - s - 1;
}
```

Examples – strcpy

```
void strcpy ( char *s, char *t)   /* copy t to s */
{
    while ( *s++ = *t++ )  /* stops when *t = '\0' */
            ;         /* look at page 105 examples  */
}
```

- Note: there is an assignment statement inside the while ( ) (not just comparison) and the copy stops AFTER the assignment to *s of the final value 0 from *t

Examples - strcmp

```
int strcmp ( char * s, char * t)   /* space after * is OK */
{
   for (  ; *s == *t;  s++, t++)
         if ( *s == '\0')
                   return 0;        /*have compared entire string
                                      and found no mismatch */
   return *s - *t;             /*on the first mismatch, return */
}        /* (*s - *t is < 0 if *s < *t and is > 0 if *s > *t) */
```

Review Pointers.

```
int a[ ] ={1,3,5,7,9,11,13,15,17,19};
int *pa = &a[4],*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, can't modify an array name such as a
What is the value of: pa[3]?	Same as a[9]

Valid Pointer Arithmetic

- Set one pointer to the value of another pointer <u>of the same type</u>. 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
pa + i                /* i has a type int */
```

- Subtract two pointers to members of same array:

```
pa – pb
```

Note: Result is an <u>integer</u>

- Compare two pointers to members of same array:

```
if (pa <= pb)
```

Valid Pointer Arithmetic

- Assign a pointer to zero (called NULL in stdio)

  pa = NULL;   **same as**   pa = 0;
- Compare a pointer to zero (called NULL in stdio)

  If (pa != NULL);   **BUT NOT**   if (pa > NULL)
- Note: a NULL pointer doesn't point to anything (When used as a return value, it indicates failure of a function that is defined to return a pointer)
- <u>All other pointer arithmetic is invalid</u>.
  - KR p.103: it is not legal to add two pointers, or to multiply, divide or shift or mask them, or to add float or double to them, or except for void * to assign a pointer of one type to a pointer of another type without a cast.

Valid Pointer Arithmetic

- If we add new declarations to ones on slide #2,
  char s[ ] = "Hello, world!", *cp = &s[4];

- Which assignments below are valid?:

cp = cp - 3;	YES
pa = cp;	NO:
(Possible alignment problem, int’s on 4 byte boundaries)	
pa = pa + pb;	NO: no adding of pointers
pa = pa + (pa-pb);	YES: (pa-pb) is an integer
s[4] = (cp < pa)? 'a': 'b';	NO: not members of same type
cp = NULL;	YES

Pointer Arrays, K&R 5.6

- Recall that if we define

    char a[10];

- We are setting aside space in memory for the elements of array a, but a can be treated as a pointer. We can write *a or *(a + 5).

- Now think about the declaration:

    char *a[10];

Pointers to Pointers

- What does the array a contain now?

  Pointers to char variables or strings!

- Though hard to think about, we can write:

```
**a     /* First char in string pointed to by a[0]  */
*(*(a + 5) + 2)
        /* Third char in string pointed to by a[5] */
```

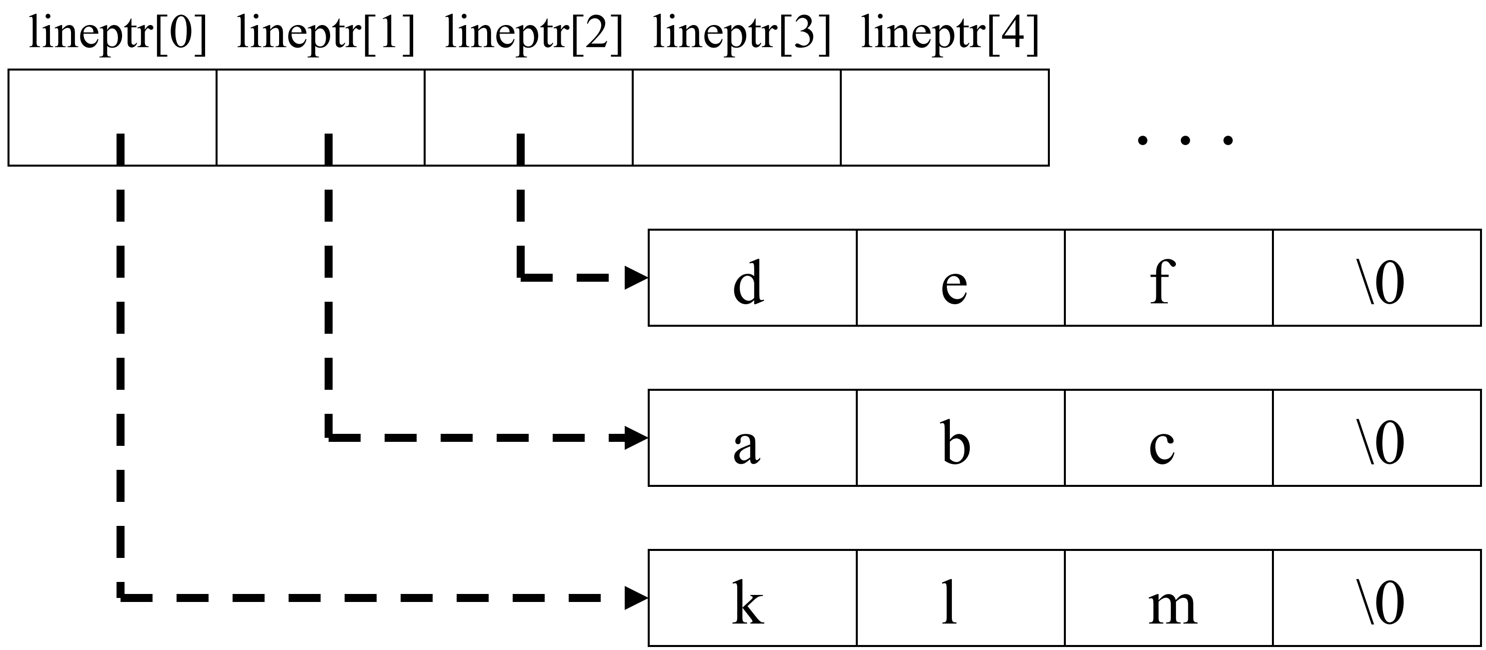
Pointers to Pointers

- Now what is the use of keeping an array of pointers to char strings?
- K&R gives an example on p.108:
  - Reading in a sequence of lines
  - Placing them in blocks of memory (e.g. malloc)
  - Building an array of pointers to the blocks
  - Sorting by moving pointers – not strings

Pointers to Pointers

- Example of pointers to unsorted char strings

char *lineptr[MAXLINES];



Pointers to Pointers

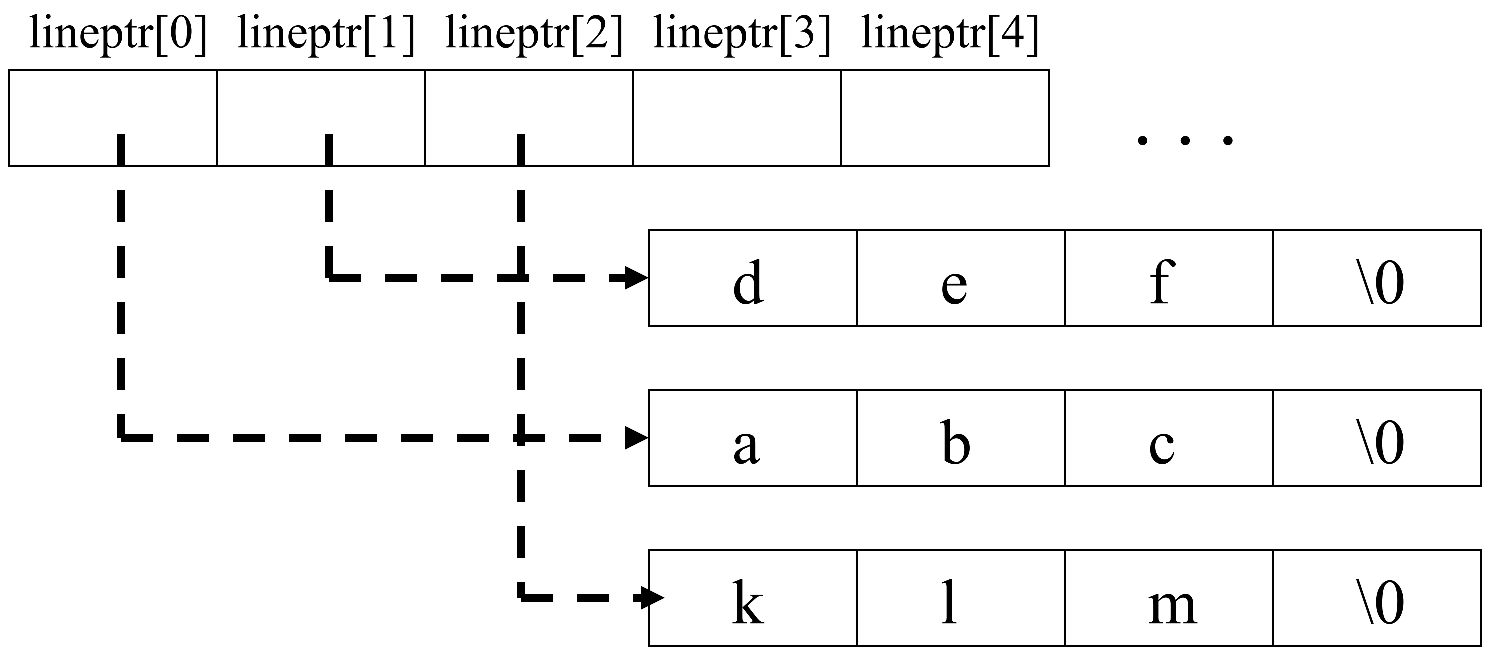
- To initialize the array with fixed values

```
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];  */
```

Pointers to Pointers

- Examples of pointers to sorted char strings

char *lineptr[MAXLINES];



Pointers to Pointers

- Write out lines in pointer order (easy way)

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

Pointers to Pointers

- Write out lines in pointer order (efficiently)

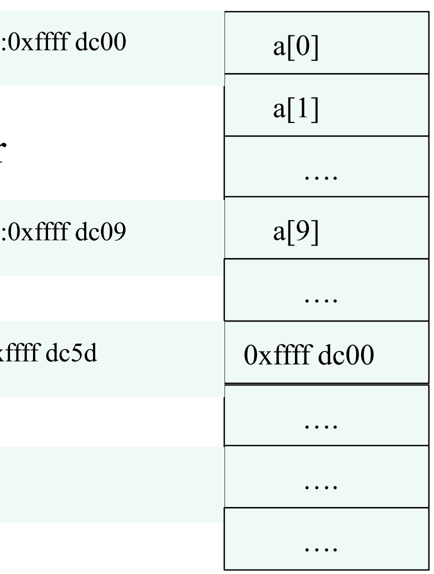
```
void writelines(char * * lineptr, int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Review of Pointers

/* demo of pointer */
char a[10];
char * p = &a[0];

It is illegal to do:
a = a+1 or
&a[0] = &a[0] +1 or
a++

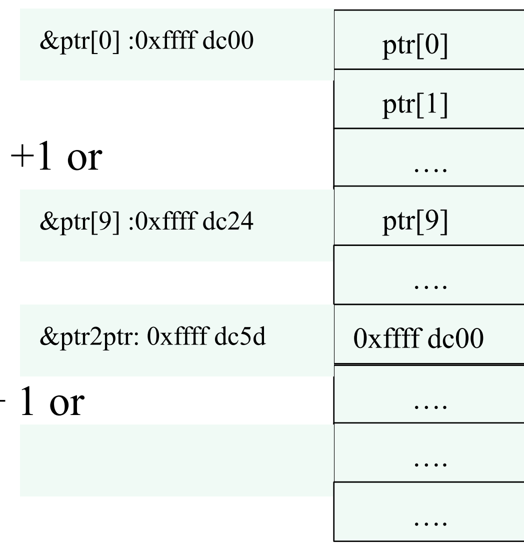
It is legal to do:
p = p+1 or
p++



/* demo of pointer array */
char *ptr[10];
char ** ptr2ptr = &ptr[0];

It is illegal to do:
ptr = ptr+1 or
&ptr[0] = &ptr[0] +1 or
ptr++

It is legal to do:
ptr2ptr = ptr2ptr + 1 or
ptr2ptr++



Command-line Arguments, K&R 5.10

- The main( ) function can be called with arguments

- Must declare them to use them

  main (int argc, char *argv[ ])

- The value of argc is the number of char strings in the array argv[ ] which is an array of ptrs to the command line tokens separated by white space

- Element argv[0] always points to the command name typed in by the user to invoke main

- If there are no other arguments, argc = 1

Command-line Arguments

- If there are other arguments:
  - For example, if the program was compiled as echo, and the user typed

    echo hello, world

- argc will be 3 (the number of strings in argv[])
- 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”

Command-line Arguments

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

```
int main (int argc, char *argv[ ])
{        /* envision argc = 3, *argv[0]="echo", …*/
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? "  " : "");
    printf("\n");
    return 0;
}
```

malloc( ) and free( )

- To get a pointer p to a block of memory that is n characters in length, program calls

```
p = malloc(n);
```

- When it is finished with that memory, the program returns it by calling

```
free(p);
```

- Sounds simple, huh?

- It is NOT so simple!

malloc( ) and free( )

- malloc returns a pointer (void *) that points to a memory block of n bytes

- If you need a pointer to n of a specific type, you must request a memory block in size of the type and cast pointer returned by malloc

```
int *p;
p = (int *) malloc(n * sizeof(int));
```

malloc and free

- If it can not provide the requested memory, malloc returns a NULL pointer value
- If you dereference a NULL pointer to access memory → System Crash!!
- Always check to be sure that the pointer returned by malloc is NOT equal to NULL
- If pointer is NULL, code must take appropriate recovery action to handle lack of memory

malloc and free

- Call to free does not clear the program's pointer to the memory block, so it is now a "stale" pointer
- If program uses pointer after free( ) by accessing or setting memory via pointer, it could overwrite data owned by another program → System Crash!
- If program calls free again with the same pointer, it releases memory possibly owned by a different program now → System Crash!
- SHOULD set pointer to NULL after calling free( )

malloc and free

- However, if you set the pointer to a memory block to NULL before calling free, you have caused the system to lose the memory forever
- This is called a memory leak!!
- If it happens enough times → System Crash!!
- MUST not clear or overwrite a pointer to a memory block before calling free!!

malloc( ) and free( )

- Memory model for malloc( ) and free( )

  Before call to malloc( ) and after call to free( ):

allocbuf:

In use	Free	In use	Free

After call to malloc( ) and before call to free( ):

allocbuf:

In use	Free	In use	In use	Free

malloc( ) and free( )

- Fragmentation with malloc( ) and free( )

  Before call to malloc( ) for a large memory block:

  allocbuf:

In use	Free	In use	Free	In use	Free	In use

  malloc can NOT provide a large contiguous block of memory - even though there is theoretically sufficient free memory! Memory is fragmented!!

malloc and free

- The malloc memory fragmentation problem is not easy to solve!!
- Not possible to “defragment” malloc memory as you would do for a disk
- On a disk, the pointers to memory are in the disk file allocation table and can be changed
- With malloc, programs are holding pointers to memory they own - so can’t be changed!!

Pointers to Functions, K&R 5.11

- Function prototype with pointer to function

  ```
  void qsort ( …  , int (*comp) (void *, void *));
  ```

- Function call passing a pointer to function

  ```
  qsort( … , (int (*) (void *, void *)) strcmp);
  ```

  This is a cast to function pointer of strcmp

- Within qsort(), function is called via a pointer

  ```
  if ((*comp) (v[i], v[left]) < 0) …
  ```

Pointers to Functions

- Initialize a pointer to a function

```
/* function pointer *fooptr = cast of foo to func ptr */
int (*fooptr) (int *, int*) = (int (*) (int *, int *)) foo;
```

- Call the function foo via the pointer to it

```
(*fooptr) (to, from);
```