Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (pass pointer by value) (5.2)
- Pointer arithmetic +- ++ -- (5.4)
- Pointers and arrays (5.3)
  - Stored consecutively
  - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
  - Array name contains address of 1st element a = &a[0]
  - Pointer arithmetic on array (extension) p1-p2 p1<>!= p2
  - Array as function argument – "decay"
  - **Pass sub_array**
- Array of pointers (5.6-5.9)
- Command line arguments (5.10)
- Memory allocation (extra)
- Pointer to structures (6.4)
- Pointer to functions

fact1 *fact2* *fact3* *fact4* *fact5*

last

Today

Passing Sub-arrays to Functions

LET'S RECAP...

- It is possible to pass part of an array to a function, by passing a pointer to the beginning of the sub-array.

```
char arr[20] = "hi world";
char * p = arr;  // &arr[0]
strlen(&arr[0]);
strlen(arr);
strlen(p);
```

Functions receive address 92

8

Pointer level

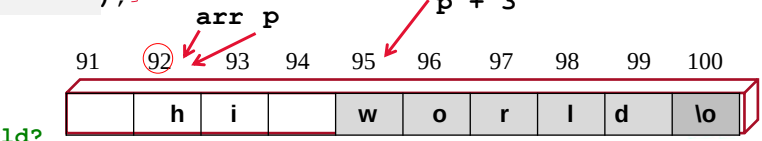
```
printf("%s", p); // &arr[0]
```

```
//length of world
strlen (      );
strlen (      );
strlen (      );
```

Functions receive address 95

5

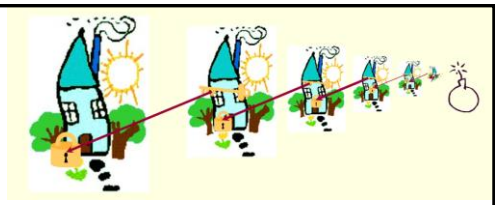
```
&arr[3]
arr + 3
p + 3
```

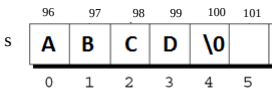



```
print world?
printf("%s", p+3);?   // world   arr+3
printf("%s", p+5);?   // rld     strlen(p+5)?
```

sub-array

Passing Subarrays to Functions -- Recursion






```
length("ABCD")
= 1 + length("BCD")
= 1 + ( 1 + length("CD"))
= 1 + ( 1 + ( 1 + length("D")))
= 1 + ( 1 + ( 1 + (1+ (1+length(""))))
= 1 + ( 1 + ( 1 + (1+ (1+0) ))) = 4
```

```
int main(){
    char s[] = "ABCD";
    int len = length(s); //pass 96
    printf("%d",len); // 4
}

int length(char * c){
    if (*c == '\0')
      return 0;
    else
      return 1 + length( );
}
```

97 98 99 100

46

Array Arguments (Summary)

"decay"

- The fact that an array argument is passed by a pointer (its starting address) has some important consequences.

- Consequence 1:
  - Due to 'pass by value', when an ordinary variable is passed to a function, its value is copied; any changes to the corresponding parameter don't affect the variable.

  - In contrast, by passing array by pointer, argument array can be modified

```
void processArr(chars[]) // no &
strcpy (message, "hello");   // no &
scanf ("%s", message);  // no &
```

49

YORK UNIVERSITÉ UNIVERSITY

49

Pointers and arrays (Summary revisit)

- Consequence 2:
  - The time required to pass an array to a function doesn't depend on the size of the array. There's no penalty for passing a large array, since no copy of the array is made.
- Consequence 3:
  - An array parameter can be declared as a pointer if desired.

```
strlen (char * s)
processArr(char *s)
```

- Consequence 4:
  - A function with an array parameter can be passed an array "slice" — substring

```
strlen (&a[6])
strlen (a + 6)
strlen (p + 6) // assume type* p = a
```

50

YORK UNIVERSITÉ UNIVERSITY

"Disadvantages"?

General array as function argument

- Pass an array/string by only the address/pointer of the first element
  - `strlen("Hello");`

"decay"

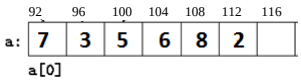
- You need to take care of where the array ends, the function does not know if it is an array or just a pointer to a char or int
- Two possible approaches:
  1. Special token/sentinel/terminator at the end (case of "string" `'\0'`)
  2. Pass the length as additional parameter

Function: `findMax(int [])` `countSum(int *)`

Caller: `int a[20]={..}; findMax(a), countSum(a);`

51

```
int main(){
    int a [] = {7,3,5,6,8,2};

    int max = findMax(a);
    …
}


/* find max in the int array */
int findMax (int arr[]){ // (int * arr)

    int len = sizeof(arr)/sizeof(int);  //  8/4=2

    while ( i <  len ){
        …
```



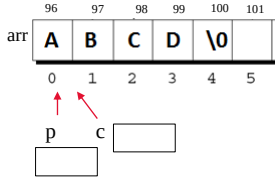
sizeof does not work in function

```
lab5E.c:66:28: warning: 'sizeof (arr)' will return the size of the pointer, not the array itself
      [-Wsizeof-pointer-div]
    int size = sizeof(arr)/sizeof(int);
```

52 Some nice compiler (MAC. not lab gcc ☹)

sizeof is not a function. It is an operator

```
int main(){
    char arr [] = "ABCD";
    char * p = arr;

    strlen(arr);
    strlen(p);

    sizeof arr;
    sizeof p;

    aFunction(arr);
    …
}

int aFunction (char c[]){ // (char * c)

    strlen(c);
    sizeof(c);
    …
```



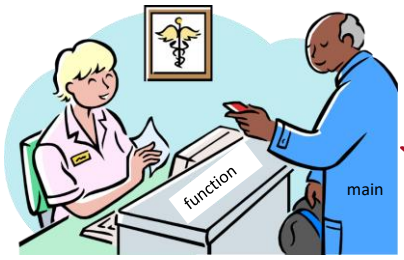
same. Pass 96

Not same!

For length, sizeof does not work on pointer and in function

53

strlen(char *) **arrayLen(int *, int n)**



Mr. Main:
Hi, Mrs binding function, I have some manuscripts, stored in lockers (memory), and I need you to bind them into a book. Could I bring the manuscripts to you?

Ms function:
Hi, Mr Main, here is how we work:
First, we don't take your original manuscript (not pass by reference). We always photocopy things (call by value), and work on copies.
Second, we only photocopy one paper a time(a single value)

Mr. Main:
Then, is there a way to have my original papers bound by you?

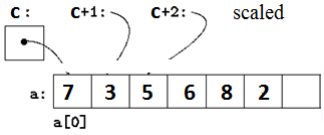
Ms function:
Write down the locker number (starting address) on a paper, bring that paper to us (pass pointer/address) we photocopy the paper (still pass by value),
Then, based on the locker number on the copy, we go to your locker, fetch your original manuscripts there and bind them!

Ms function:
Well, then you also need to tell us where to stop fetching.
Either 1) tell us how many lockers to fetch, or, 2) put a special token in the last locker

Mr. Main:
My manuscripts are in multiple (consecutive) lockers

54

```
int main(){
    int i, arr[20];  count=0;
    while ( scanf("%d", &i) != EOF){
        *(arr + count) = i;
        count++;
     }

     finaMax(arr, count);
}
/* find max in the int array. */
int findMax (int *c, int leng){
   int max = *c;
   int i=1;
   while ( i < leng )
       ...
   return max;
 }
```



55

55

Function processing general arrays

Description
The C library function **qsort** sorts an array.

Declaration
void **qsort** (void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))

Parameters
- **base** – This is the pointer to the first element of the array to be sorted.
- **nitems** – This is the number of elements in the array pointed by base.
- **size** – This is the size in bytes of each element in the array.
- **compar** – This is the function that compares two elements.

Description
The C library function **bsearch** searches an array of **nitems** objects

Declaration
void * **bsearch** (const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const v
const void *))

Parameters
- **key** – This is the pointer to the object that serves as key for the search, type-casted as a void*
- **base** – This is the pointer to the first object of the array where the search is performed, type-casted as a void*.
- **nitems** – This is the number of elements in the array pointed by base.
- **size** – This is the size in bytes of each element in the array.
- **compar** – This is the function that compares two elements.

For your information

57

Java avoids the hassle


```
public static void main(String[]args)
{
    int arr [] = {17,3,5,19,8,2};
    int a = findMax(arr);
    …
}

/* find max in the int array */
public static int findMax (int c[]){
    int max = c[0]; i=1;
    while ( i <  c.length ) {
        ……

    }
    return max;
```

Array object

arr	
value	17 3 5 19 8 2
length	6
……	

Java also pass starting address (call-by-value)

58

For your information

YORK UNIVERSITÉ UNIVERSITY

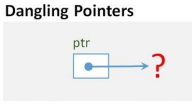
58

Problems with pointers

```
int *ptr;          /* I'm a pointer to an int */
ptr= &a            /* I got the address of a */
*ptr = 5;          /* set contents of the pointee a */
```

```
int *ptr;          /* I'm a pointer to an int */
*ptr = 5;          /* set contents of the pointee to 5 */
```

- `ptr` is uninitialized. "points to nothing". "dangling"
  Has some random value `0x7fff033798b0`
  - may be your OS!



- dereferencing an uninitialized pointer? Undefined behavior!

- Always make `ptr` point to sth! How?

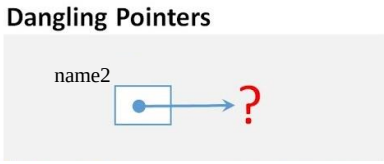
```
1) int a; ptr =&a;    int arr[20]; ptr=&arr[0];
2) ptr = ptr2   /* indirect. assuming ptr2 is
3) ptr = malloc (.....) /* later today? *
```

60

60

Problems with pointers, another scenario

```
char name[20];
char *name2;
int age; float rate;
```




```
printf("Enter name, name2, age, rate: ");
scanf("%s %s %d %f",name,name2,age, rate);

while( strcmp(name, "xxx") )
{    ……
}
```

segmentation fault
core dump

segmentation fault
core dump

61

61

Whenever you need to set a pointer's pointee

e.g.,

- `*ptr = var;`
- `scanf("%s", ptr);`
- `strcpy(ptr, "hello");`
- `fgets(ptr, 10, STDIN);`
- ……
- `*ptrArr[2] = var;  // pointer array`

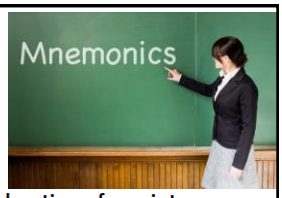
Ask yourself: Have you done one of the following

1. `ptr = &var.  /* direct */`
   `arr[20]; ptr=&arr[0];`
2. `ptr = ptr2  /* indirect, assuming ptr2 is good */`
3. `ptr = (..)malloc(....)  /* later */`

62

Stopped here last time

62



As mentioned in the textbook and class, the declaration of a pointer related variable is intended as a ***mnemonic*** (means 'it helps you memorize things'). For example, declaration `int *ptr;` can be interpreted as "expression `*ptr` is an `int`" -- thus `ptr` is an integer pointer.

Following this rule, what is the type that `argv` is declared to be?

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

```
int * a[]

int a[]
int ** a[]
```

64

64

Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (pass pointer by value) (5.2)
- Pointer arithmetic +- ++ -- (5.4)
- Pointers and arrays (5.3)
  - Stored consecutively
  - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
  - Array name contains address of 1st element a = &a[0]
  - Pointer arithmetic on array (extension) p1-p2 p1<>!= p2
  - Array as function argument – "decay"
  - Pass sub_array
- **Array of pointers (5.6)**
- Pointer arrays vs. two dimensional arrays (5.9)
- Command line argument (5.10)
- Memory allocation (extra)
- Pointer to structures (6.4)
- Pointer to functions

YORK UNIVERSITÉ UNIVERSITY

65

Pointers K&R Ch 5

- **Pointer arrays (5.6)**
  - Declaration, initialization, accessing via element pointers
    - Array of pointers to scalar type
    - Array of pointers to strings
  - Pointer to the pointer arrays (what type is it?)
    - Array of pointers to scalar type
    - Array of pointers to strings
  - Passing pointer arrays to functions (what is it decayed to?)
    - Array of pointers to scalar type
    - Array of pointers to strings
  - Pointer array vs. 2D array

YORK UNIVERSITÉ UNIVERSITY

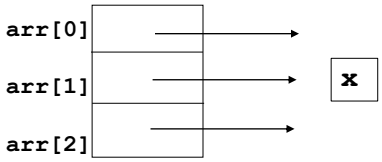
66

Array of Pointers (5.6)

- Pointers are variables
  - Can be arrayed like others (int, char, double) …

```
int a[3]
int ** a[3]
```

```
int * arr[3]; // array of 3 pointers to integer
```



- `arr[i]` is an integer pointer `int *`

```
int x;
arr[1] = &x;
```

`*arr[1] = 4`

YORK UNIVERSITÉ UNIVERSITY

67

67

Precedence

Operator Type	Operator	
Primary Expression Operators	() [] . ->	
Unary Operators	* & + - ! ~ ++ -- (typecast) sizeof	
Binary Operators	* / %	arithmetic
	+ -	arithmetic
	>> <<	bitwise
	< > <= >=	relational
	== !=	relational
	&	bitwise
	^	bitwise
	|	bitwise
	&&	logical
	||	logical
Ternary Operator	?:	
Assignment Operators	= += -= *= /= %= >>= <<= &= ^= |=	
Comma	,	


```
int * arr[3]
/* array of 3
integer pointers */
```

```
char * arr[5]
/* array of 5 char
pointers */
```

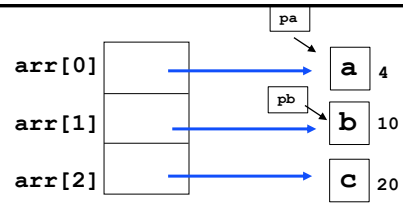
```
No () needed
```

```
char (*arr)[5]
/* ??? */
```

68

Array of pointers to scalar types

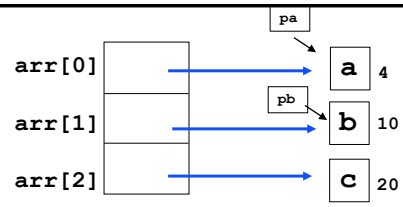
```
main(){
  int a,b,c, *pa, *pb;
  a=4; b=10;c=20;
  pa=&a, pb=&b;

  int * arr[3]; // an array of 3 (uninitialized) int pointers
  arr[0]= pa;  arr[1]= pb;  arr[2]= &c;  //different ways
  arr[0]= &a;  arr[1]= &b;
```

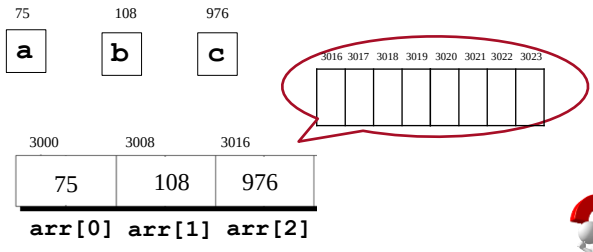


69

Array of pointers to scalar types

```
main(){
  int a,b,c, *pa, *pb;
  a=4; b=10;c=20;
  pa=&a, pb=&b;

  int * arr[3]; // an array of 3 (uninitialized) int pointers
  arr[0]= pa;  arr[1]= pb;  arr[2]= &c;  //different ways
  arr[0]= &a;  arr[1]= &b;
```





Each element is a pointer, size usually 8 bytes (regardless of the type)

Access a,b,c via arr

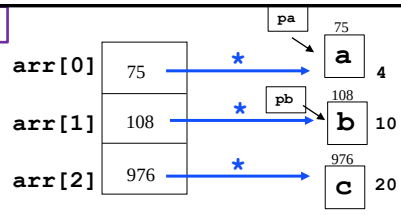
70

```
printf("%p %p\n", arr[0], arr[1]); // 75 108
```

70

```
main(){
  int a,b,c, *pa, *pb;
  a=4; b=10;c=20;
  pa=&a, pb=&b;

  int * arr[3]; // an array of 3 (uninitialized) int pointers
  arr[0]= pa;   arr[1]= pb;   arr[2]= &c;

  printf("%p %p\n", arr[0], arr[1]); // 75 108
  printf("%d\n",          ;  // arr[0] is a pointer to a
  printf("%d\n",          ;  //
  printf("%d\n",          ); //

  ?  = 100; // set b to 100
```

Array of pointers to scalar types



Operator
() [] . ->
* & + - !~ ++ -- (typecast) sizeof

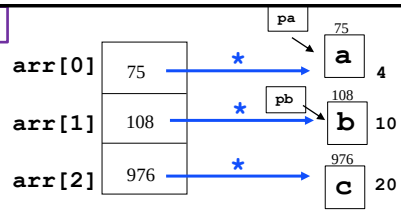
Recall:

```
int a=10;    char arr[]="apple";
int pA = &a;  char * pArr  = arr;
printf("%d %d", a, *pA);     // pointee level
printf("%s %s", arr, pArr); // pointer level
```

71

```
main(){
  int a,b,c, *pa, *pb;
  a=4; b=10;c=20;
  pa=&a, pb=&b;

  int * arr[3]; // an array of 3 (uninitialized) int pointers
  arr[0]= pa;   arr[1]= pb;   arr[2]= &c;

  printf("%p %p\n", arr[0], arr[1]); // 75 108
  printf("%d\n", *arr[0]);     // 4    **(arr+0)
  printf("%d\n", *arr[1]);     // 10   **(arr+1)
  printf("%d\n", *(arr[2]));   // 20   **(arr+2)

  *arr[1] = 100;  // alias of b.   Set b to 100

  for (i=0; i<3, i++)
    printf("%d ", *arr[i]);  // **(arr+i)   4 100 20
}
```

Array of pointers to scalar types



Pointee level

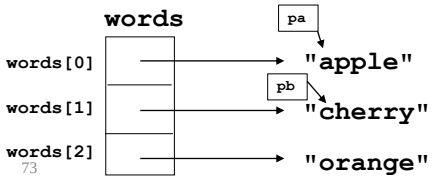
72

Array of Pointers (5.6)

- Common use: array of char pointers (strings)

```
char a[] = "apple";    char * pa = a; // &a[0]
char b[] = "cherry";   char * pb = b; // &b[0]
char c[] = "orange";
char * words[3];
words[0]= pa; words[1]= pb;
words[2] =
```

- **words** is an array of pointers to char (**char ***)
- Each element of **words** (**words[0],words[1],words[2]**) contains address of a char (which may be the start of a string)



73

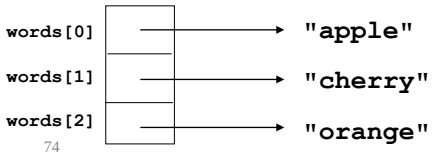
73

Arrays of Pointers (5.6)

- Common use: array of char pointers (strings)

```
char * words[]={"apple", "cherry", "orange"};

char words[4][5]={"apple", "cherry", "orange"}; //another
```

- **words** is an array of pointers to char (**char ***)
- Each element of **words** (**words[0],words[1],words[2]**) contains address of a char (which may be the start of a string)



higher level view

74

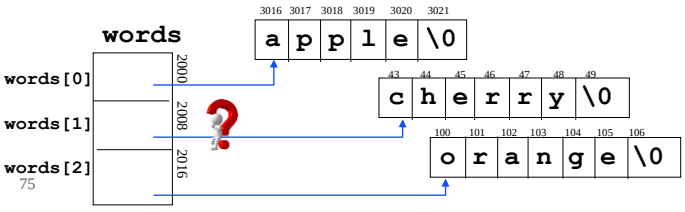
74

Arrays of Pointers (5.6)

- Common use: array of char pointers (strings)

```
char * words[]={"apple", "cherry", "orange"};

char words[4][5]={"apple", "cherry", "orange"}; //another
```

- **words** is an array of pointers to char (**char ***)
- Each element of **words** (**words[0],words[1],words[2]**) contains address of a char (which may be the start of a string)



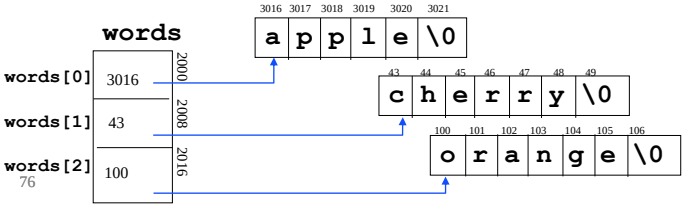
lower level view

Arrays of Pointers (5.6)

- Common use: array of char pointers (strings)

```
char * words[]={"apple", "cherry", "orange"};

char words[4][5]={"apple", "cherry", "orange"}; //another
```

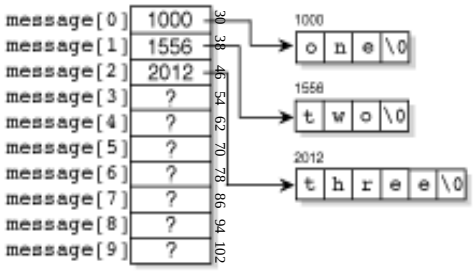
- **words** is an array of pointers to char (**char ***)
- Each element of **words** (**words[0],words[1],words[2]**) contains address of a char (which may be the start of a string)

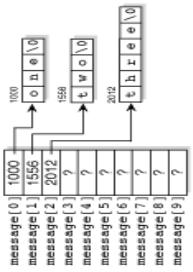


lower level view

Another example, initialization

- `char *message[10] = {"one", "two", "three"};`



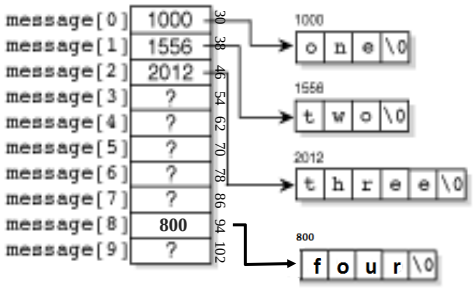


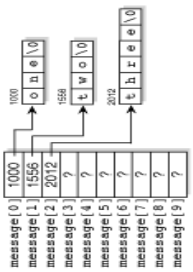
```
char arr[] = "four";
message[8] = arr;  // &arr[0]
```

77

Another example, initialization

- `char *message[10] = {"one", "two", "three"};`



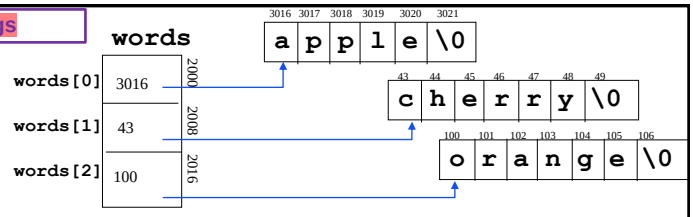


```
char arr[] = "four";
message[8] = arr;
```

Access “two” via message?

78

Array of pointers to strings




```
char * words[]={"apple", "cherry", "banana"};
printf("%p %p\n", words[0], words[1]);   3016  43
printf("%s\n",          ); // apple    words[0] is the pointer
printf("%s\n",          ); // cherry   *words[0] is ?   'a'
printf("%s\n",          ); // orange   *words[1] is ?   'c'

for (i=0; i<3, i++)
  printf("%d ",strlen(          ));   // 5  6  6
}
```

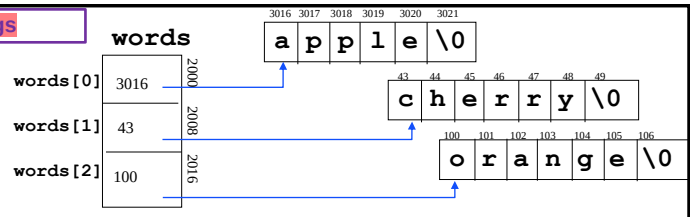
Recall:

```
int a=10;    char arr[]="apple";
int pA = &a;   char * pArr  = arr;
printf("%d %d", a, *pA);      // pointee level
printf("%s %s", arr, pArr); // pointer level
```

79

79

Array of pointers to strings




```
char * words[]={"apple", "cherry", "banana"};
printf("%p %p\n", words[0], words[1]);   //  3016  43
printf("%s\n", words[0]); // apple    words[0] is the pointer
printf("%s\n", words[1]); // cherry   *(words+1)
printf("%s\n", words[2]); // orange   *(words+2)

                                       *words[1] is ?   'c'
for (i=0; i<3, i++)
  printf("%d ",strlen( words[i]));// *(words+i) 5 6 6
}
```

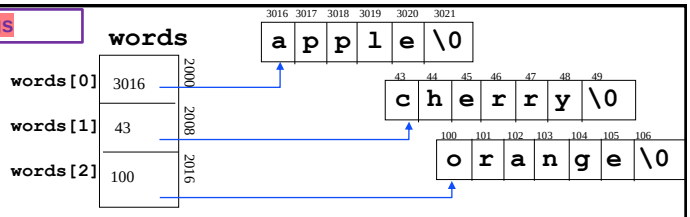
Recall:

```
int a=10;    char arr[]="apple";
int pA = &a;   char * pArr  = arr;
printf("%d %d", a, *pA);      // pointee level
printf("%s %s", arr, pArr); // pointer level
```

80

80

Array of pointers to strings




```
char * words[]={"apple", "cherry", "banana"};
printf("%p %p\n", words[0], words[1]);  //  3016  43

printf("%s\n", words[0]); // apple    words[0] is the pointer
printf("%s\n", words[1]); // cherry   *(words+1)
printf("%s\n", words[2]); // orange   *(words+2)

printf("%p\n", words[1]+5  );
printf("%p\n", words[2]+3  );
printf("%p\n", *(words+2)+3);

printf("%s\n", words[2]+3);
printf("%s\n", *(words+2)+3);

printf("% \n", *(words[1]+5) );
printf("% n", *(words[2]+3) );
printf("% \n", *(*(words+2)+3) );
```

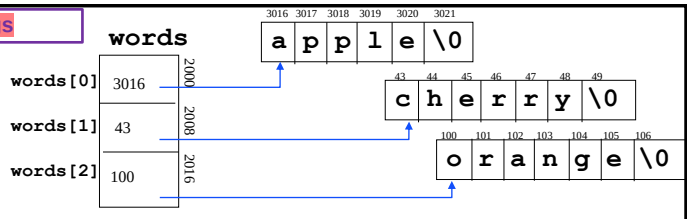
Recall:

```
p + i ==  &words[i]
*(p+i) == words[i]
```

82

82

Array of pointers to strings




```
char * words[]={"apple", "cherry", "banana"};
printf("%p %p\n", words[0], words[1]);  //  3016  43

printf("%s\n", words[0]); // apple    words[0] is the pointer
printf("%s\n", words[1]); // cherry   *(words+1)
printf("%s\n", words[2]); // orange   *(words+2)

printf("%p\n", words[1]+5  );  //  48
printf("%p\n", words[2]+3  );  // 103
printf("%p\n", *(words+2)+3);  // 103

printf("%s\n", words[2]+3);      // nge
printf("%s\n", *(words+2)+3);    // nge

printf("%c\n", *(words[1]+5) ); // y
printf("%c\n", *(words[2]+3) ); // n
printf("%c\n", *(*(words+2)+3) ); // n
}
```

Recall:

```
p + i ==  &words[i]
*(p+i) == words[i]
```

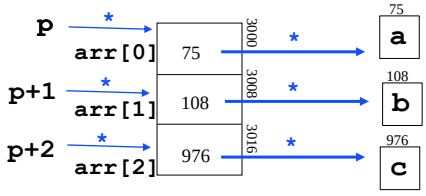
83

83

Pointers K&R Ch 5

- **Pointers arrays (5.6)**
  - Declaration, initialization, accessing via element pointers
    - Array of pointers to scalar type `printf("%d", *arr[2])` `**(arr+2)`
    - Array of pointers to strings `printf("%s", words[2])` `*(words+2)`
  - Pointer to the pointer arrays (what type is it?)
    - Array of pointers to scalar type
    - Array of pointers to strings
  - Passing pointer arrays to functions (what is it decayed to?)
    - Array of pointers to scalar type
    - Array of pointers to strings
  - Pointer array vs. 2D array

Array of pointers to scalar types



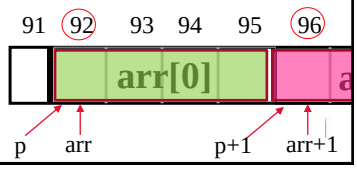

```
main(){
  int a,b,c, *pa, *pb;
  a=4; b=10;c=20;
  pa=&a, pb=&b;

  int * arr[3];
  arr[0]= pa;   arr[1]= pb;  arr[2]= &c;

  int ? p = arr; // p = &arr[0] == 1000
```

Hint: arr[0] is a pointer (to int)

Recall:

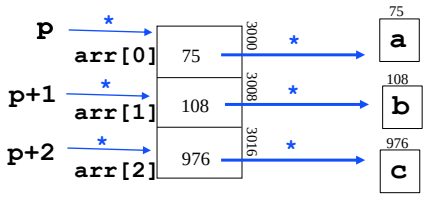
```
int arr[] ={3,5,7,10};

int * pA = arr; // = &arr[0];
```

arr[0] is an int



90

Array of pointers to scalar types




```
main(){
   int a,b,c, *pa, *pb;
   a=4; b=10;c=20;
   pa=&a, pb=&b;

   int * arr[3];
   arr[0]= pa;   arr[1]= pb;  arr[2]= &c;

   int **  p = arr; // p = &arr[0] == 3000

                                          *p==arr[0]
   printf("%p %p %p\n", &arr[0], p,   *p );      // 3000 3000 75
   printf("%p %p %p\n", &arr[1], p+1, *(p+1)); // 3008 3008 108
   printf("%p %p %p\n", &arr[2], p+2, *(p+2)); // 3016 3016 976

}
```

91

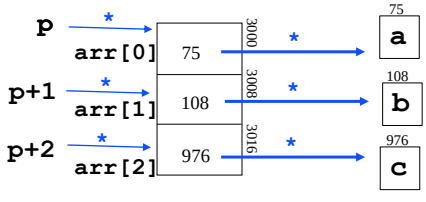
Access a,b,c via p 

Recall:

```
p + i ==  &arr[i]
*(p+i) == arr[i]
```

91

Array of pointers to scalar types




```
main(){
   int a,b,c, *pa, *pb;
   a=4; b=10;c=20;
   pa=&a, pb=&b;

   int * arr[3];
   arr[0]= pa;   arr[1]= pb;  arr[2]= &c;

   int **  p = arr; // p = &arr[0] == 3000

   printf("%d\n",      ); // 4   *arr[0] "pointee level"
   printf("%d\n",      ); // 10  *arr[1]
   printf("%d\n",      ); // 20  *arr[2]

   for (i=0; i<3, i++)
      printf("%d\n",    );
```

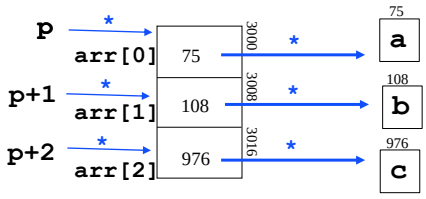
92

Recall:

```
p + i ==  &arr[i]
*(p+i) == arr[i]
```

92

Array of pointers to scalar types




```
main(){
  int a,b,c, *pa, *pb;
  a=4; b=10;c=20;
  pa=&a, pb=&b;

  int * arr[3];
  arr[0]= pa;   arr[1]= pb;  arr[2]= &c;

  int **  p = arr; // p = &arr[0] == 1000

  printf("%d\n", **p);          // 4   *arr[0]   ** arr
  printf("%d\n", **(p+1));      // 10  *arr[1]   **(arr+1)
  printf("%d\n", *(*(p+2)) );   // 20  *arr[2]

  for (i=0; i<3, i++)
    printf("%d\n", **(p+i));
```

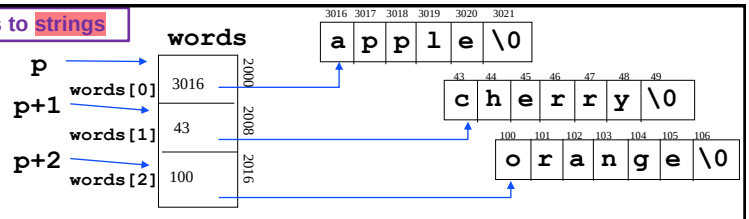
Recall:

```
p + i ==  &arr[i]
*(p+i) == arr[i]
```

93

93

Array of pointers to strings




```
char * words[]={"apple", "cherry", "orange"};

char ? p = words; // p = &words[0] == 2000
```

Hint: words[0] is a pointer (to char)

Recall:

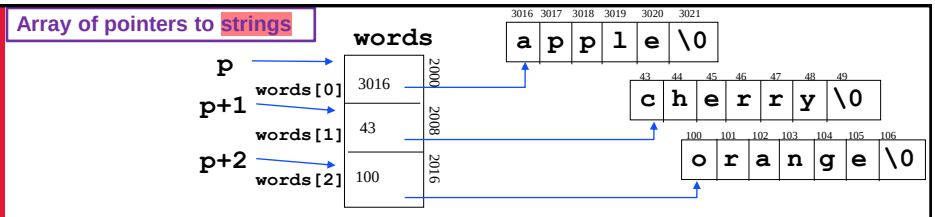
```
char arr[] = "apple";

char * pA = arr; // &arr[0];
```

words[0] is a char

94

94

Array of pointers to strings




```
char * words[]={"apple", "cherry", "orange"};

char **  p = words; // p = &words[0] == 2000

printf("%p %p %p\n", &words[0], p,   *p );     // 2000 2000  3016
printf("%p %p %p\n", &words[1], p+1, *(p+1)); // 2008 2008  43
printf("%p %p %p\n", &words[2], p+2, *(p+2)); // 2016 2016 100
```

Recall:

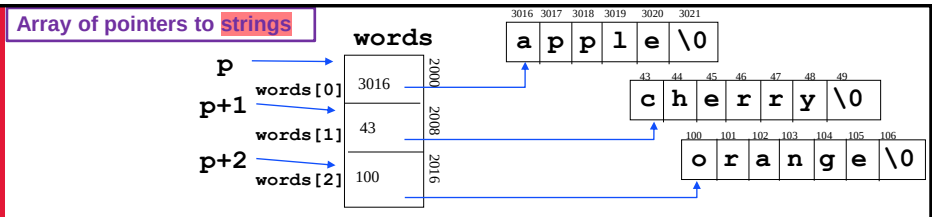
```
p + i ==  &words[i]
*(p+i) == words[i]
```

```
}
```

Access `apple`, `orange` via `p`

95

Array of pointers to strings



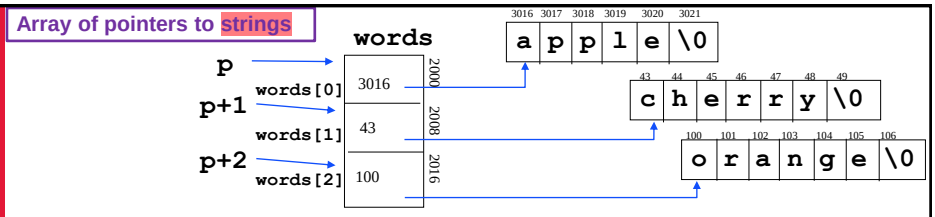

```
char * words[]={"apple", "cherry", "orange"};
char **  p = words; // p = &words[0] == 2000

printf("%s\n",    ); // apple   words[0]
printf("%s\n",    ); // cherry  words[1]
printf("%s\n", ?  ); // orange  words[2]

for (i=0; i<3, i++)
  printf("%d\n", strlen(        ) ); // 5 6 6
}
```

Recall:

```
p + i ==  &words[i]
*(p+i) == words[i]
```


96

Array of pointers to strings




```
char * words[]={"apple", "cherry", "banana"};
char **  p = words; // p = &words[0] == 2000

printf("%s\n", *p  );     // apple  words[0]
printf("%s\n", *(p+1) ); // cherry words[1]
printf("%s\n", *(p+2) ); // banana words[2]
for (i=0; i<3, i++)
  printf("%d\n", strlen( *(p+i) ) ); // 5 6 6

printf("%p\n",   words[1]+5  );  //  48
printf("%p\n",   *(words+1)+5 ); //  48
printf("%p\n",   *(p+1)  +5  );  //  48

printf("%c\n", *(words[1]+5) );  //  y
printf("%c\n", *(*(words+1)+5) ); //  y
printf("%c\n", *(*(p+1)+5)   );   //  y
```

Hardest today

Recall:

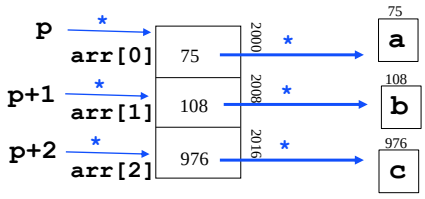
```
p + i ==  &words[i]
*(p+i) == words[i]
```

((p+1)+5) + 1 ??

Pointers K&R Ch 5

- **Pointers arrays (5.6)**
  - Declaration, initialization, accessing via element pointers
    - Array of pointers to scalar type `printf("%d", *arr[2])` `**(arr+2)`
    - Array of pointers to strings `printf("%s", words[2])` `*(words+2)`
  - Pointer to the pointer arrays (what type is it?)
    - Array of pointers to scalar type `int **` `printf("%d",**(p+2))`
    - Array of pointers to strings `char **` `printf("%s",*(p+2))`
  - Passing pointer arrays to functions (what is it decayed to?)
    - Array of pointers to scalar type
    - Array of pointers to strings
  - Pointer array vs. 2D array

Passing an array of pointers to functions



Array of pointers to scalar types

```
main(){
 int * arr[]= ….
 printf("%d", *arr[1]); // 4

 print_message( arr, 3);
}

void print_message(int *p[], int n){
 int count;
 for (count=0; count<n; count++)
   printf("%d ", *p[count]);
}
```

Expect an array of int *

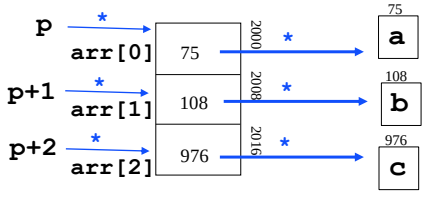
Needed to provide !!!

// compiler: **(p+count)

Pointee level

100

Passing an array of pointers to functions



Array of pointers to scalar types

```
main(){
 int * arr[]= ….
 printf("%d", *arr[1]); // 4

 print_message( arr, 3);
}

void print_message(          , int n){
 int count;
 for (count=0; count<n; count++)
   printf("%d ", **(p + count));
}
```

"decay"?

Pass address of 1st element -- &pointer

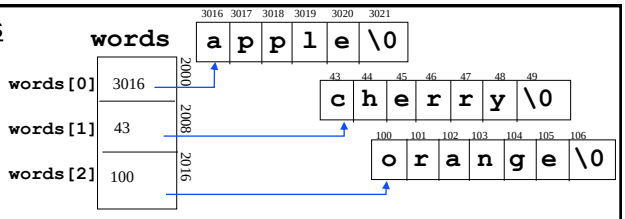
Needed to provide !!!

Pointee level

101

Passing an array of pointers to functions

Array of pointers to strings




```
main(){
 char * words[]={"apple", "cherry", "orange"};
 printf("%s", words[1]); // cherry   *words[1]

 print_message( words, 3);
}

void print_message(char *p[],  int n){
 int count;
 for (count=0; count<n; count++)
   printf("%s ", p[count]);
}
```

Expect an array of char *

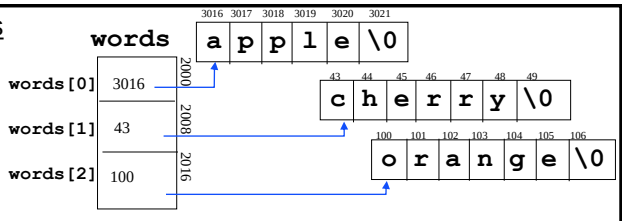
Needed to provide !!!

// compiler: *(p+count)

Pointer level

Passing an array of pointers to functions

Array of pointers to strings




```
main(){
 char * words[]={"apple", "cherry", "orange"};
 printf("%s", words[1]);  // cherry

 print_message( words, 3);
}

void print_message(          , int n){
 int count;
 for (count=0; count<n; count++)
   printf("%s ", *(p + count));
}
```

"decay"?

Pass address of 1st element -- &pointer

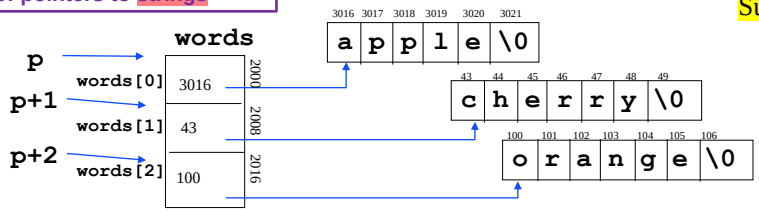
Pointer level

Pointers K&R Ch 5

- **Pointers arrays (5.6)**
  - Declaration, initialization, accessing via element pointers
    - Array of pointers to scalar type `printf("%d", *arr[2])` `**(arr+2)`
    - Array of pointers to strings `printf("%s", words[2])` `*(words+2)`
  - Pointer to the pointer arrays (what type is it?)
    - Array of pointers to scalar type `int **` `printf("%d",**(p+2))`
    - Array of pointers to strings `char **` `printf("%s",*(p+2))`
  - Passing pointer arrays to functions (what is it decayed to?)
    - Array of pointers to scalar type `int **`
    - Array of pointers to strings `char **`
  - Pointer array vs. 2D array

Array of pointers to strings

Summary




```
char **  p = words;

printf("%s\n", words[2]);    // orange   (100)
printf("%s\n", *(words+2)); // orange
printf("%s\n", *(p+2) );

    printf("%s\n", words[2]+3 );    // nge   (103)
    printf("%s\n", *(words+2)+3 ); // nge
    printf("%s\n",  *(p+2)+3 ); // nge

      printf("%c\n", *(words[2]+3) );  // n
      printf("%c\n", *(*(words+2)+3)); // n
      printf("%c\n", *(*(p+2)+3)); // n
```

Pointers K&R Ch 5

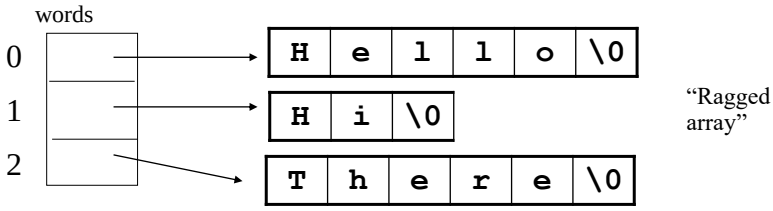
- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (pass pointer by value) (5.2)
- Pointer arithmetic +- ++ -- (5.4)
- Pointers and arrays (5.3)
  - Stored consecutively
  - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
  - Array name contains address of 1st element a = &a[0]
  - Pointer arithmetic on array (extension) p1-p2 p1<>!= p2
  - Array as function argument – "decay"
  - Pass sub_array
- Array of pointers (5.6)
- Pointer arrays vs. two dimensional arrays (5.9)
- Command line argument (5.10)
- Memory allocation (extra)
- Pointer to structures (6.4)
- Pointer to functions

YORK UNIVERSITÉ UNIVERSITY

Array of pointers to strings

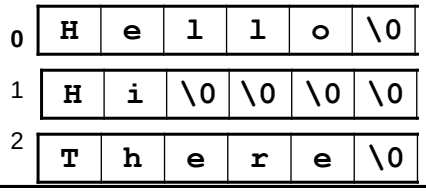
Advantage of Pointer Arrays (vs. 2D array)

```
char * words[]={"Hello", "Hi", "there"};
```



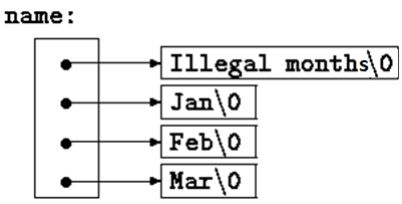
Both store table of strings. What is the difference?

```
char words[3][6] = {"Hello", "Hi", "There"};
```



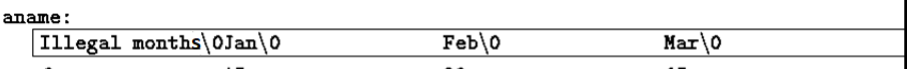
Advantage of Pointer Arrays (vs. 2D array) example 2

```
char *name[]={"Illegal months, "Jan", "Feb", "Mar"};
```



"Ragged array"

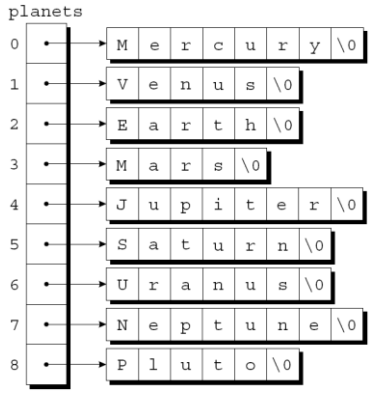
```
char aname[][15]={"Illegal months","Jan","Feb","Mar"};
```



108

Advantage of Pointer Arrays (vs. 2D array) example 3

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```



	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	?	?
2	E	a	r	t	h	\0	?	?
3	M	a	r	s	\0	?	?	?
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	?
6	U	r	a	n	u	s	\0	?
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	?	?

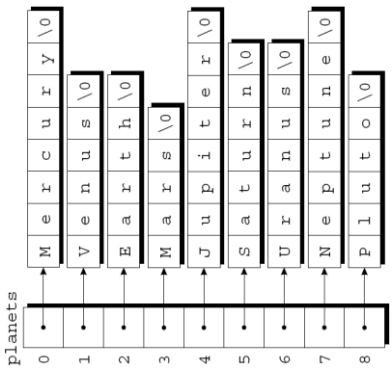
```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

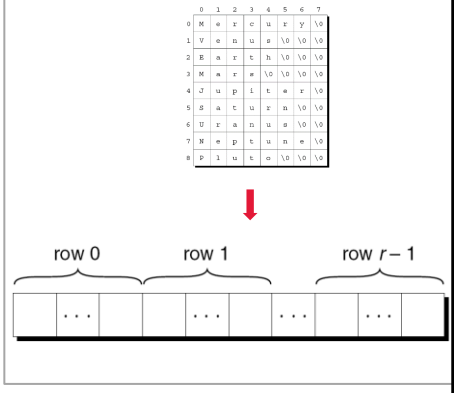
109

109

Advantage of Pointer Arrays (vs. 2D array) example 3

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```






```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

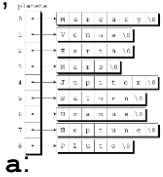
110

110

Advantage of Pointer Arrays (vs. 2D array)

```
int a[10][20];
int *b[10];
```

- **a**: 200 int-sized locations have been set aside.
  - Total size: 10*20*4
- **b**: only 10 pointers are allocated (and not initialized); initialization must be done explicitly.
  - Total size: 10*8 + size of all pointees



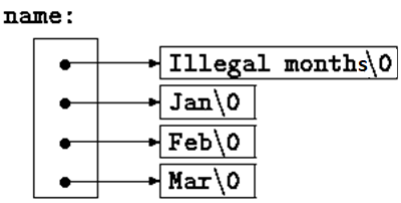
- Potential advantage of pointer array **b** vs. 2D array **a**:
  1. the rows of the array may be of different lengths (potentially saving space).
  2. Another advantage? Swap rows!

111

111

Advantage of Pointer Arrays (vs. 2D array)

```
char *name[] = {"Illegal  months", "Jan","Feb","Mar"};
```



How to swap "rows"?

sizeof name: 4*8 = 32 total memory size 4*8+15+4+4+4=59

```
char aname[][15]={"Illegal months","Jan","Feb","Mar"};
```

aname:

Illegal months\0	Jan\0	Feb\0	Mar\0
0	15	30	45

112 sizeof aname: 4*15 = 60

How to swap rows?

112

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	?	?
2	E	a	r	t	h	\0	?	?
3	M	a	r	s	\0	\0	\0	?
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	?
6	U	r	a	n	u	s	\0	?
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	?	?

```
for(i=0; i<9; i++)
  printf("%s",arr[i]);
```

```
"Mercury"
"Venus"
"Earth"
"Mars"
...
```

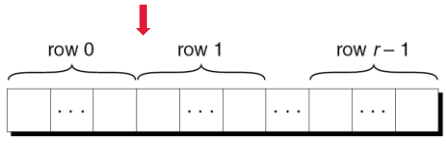
```
"Venus"
"Mercury"
"Earth"
"Mars"
..."
```

Sort 

113

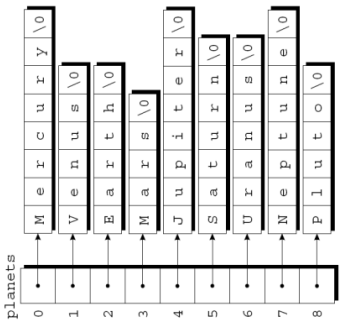
113

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```



How to swap 1st 2nd row?

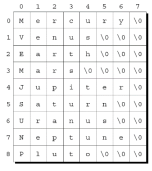
```
char tmp[8];
tmp = planets[0]
planets[0] = planets[1]
planets[1] = tmp;
```

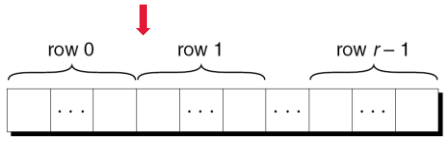



```
char *planets[] =
{"Mercury", "Venus", "Earth",
   "Mars", "Jupiter", "Saturn",
  "Uranus", "Neptune", "Pluto"};
```



```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```



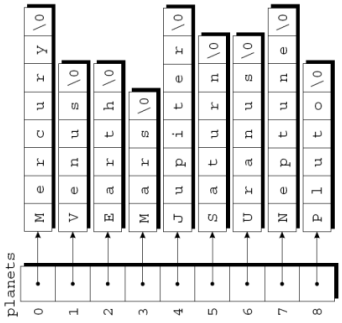


How to swap 1st 2nd row?

~~`char tmp[8];`~~
~~`tmp = planets[0]`~~
~~`planets[0] = planets[1]`~~
~~`planets[1] = tmp;`~~

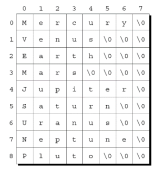
```
strcpy(tmp,planets[0]);
strcpy(planets[0],planets[1]);
strcpy(planets[1],tmp);
```

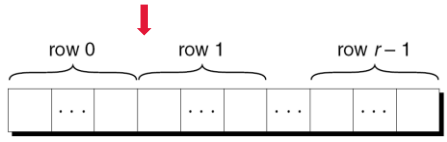
O(n)




```
char *planets[] =
{"Mercury", "Venus", "Earth",
   "Mars", "Jupiter", "Saturn",
  "Uranus", "Neptune", "Pluto"};
```

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```



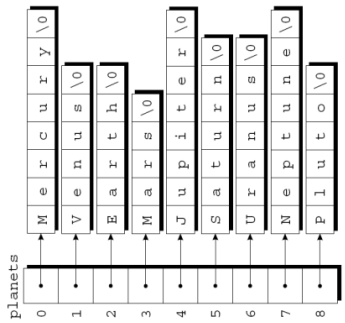


How to swap 1st 2nd row?

~~`char tmp[8];`~~
~~`tmp = planets[0]`~~
~~`planets[0] = planets[1]`~~
~~`planets[1] = tmp;`~~

```
for(i=0;i<8;i++){ //copy char one by one
  char tmp = planets[0][i];
  planets[1][i] = planets[0][i];
  planets[0][i] = tmp;
}
```

O(n)




```
char *planets[] =
{"Mercury", "Venus", "Earth",
   "Mars", "Jupiter", "Saturn",
  "Uranus", "Neptune", "Pluto"};
```