EECS2031 Software Tools

SU 2019

June 24, 2019 Lecture 7

1

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)
- Command line argument (5.10)
- Pointer arrays and two dimensional arrays (5.9)
- Memory allocation (extra)
- Pointer to structures (6.4)
- Pointer to functions

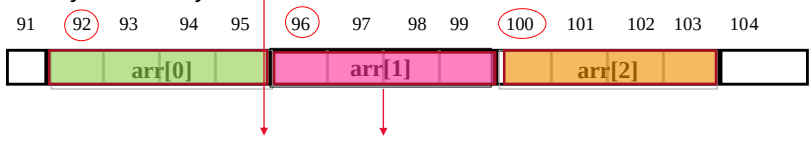
Last Three lectures

YORK UNIVERSITÉ UNIVERSITY

2

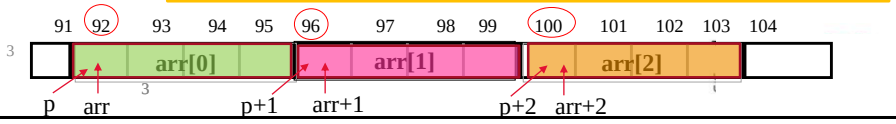
Summary

- Pointer arithmetic: If p points to an integer of 4 bytes, p + n advances by 4*n bytes: p + 1 = 96 + 1*4 = 100 p + 2 = 96 + 2*4 = 104
- Array in memory:



- Suppose `p` points to array element `k`, then `p+1` points to `k+1` (next) element. `p + i` points to `arr[k+i]`.
  - `p = &arr[k]:  p + i == &arr[k+i]  → *(p+i) == arr[k+i]`
  - `k=0: p=&arr[0]:  p + i == &arr[i]  → *(p+i) == arr[i]`
- **Array name contains pointer to 1st element** `arr==&arr[0]`
  - `arr==&arr[0]:  arr+i == &arr[i]  → *(arr+i) == arr[i]`

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



3

Summary

`arr` can be used as a pointer

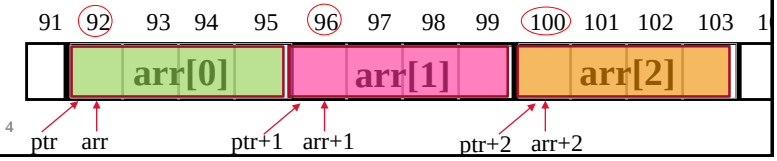
```
int arr[3]; int * p;
ptr = arr;   /*   ptr = &arr[0]   */
```

`arr+i == &arr[i]`
`ptr+i == &arr[i]`

`arr[i]`
`*(ptr + i)`
`*(arr + i)`
`ptr[i];`

equivalent

`Compiler convert arr[2] to *(arr+2)`



4

2.6 (2 pt) Given the following ANSI C program, and assume that the first element of `arr` is stored in memory address 1000; What is the output of the program? (Recall that %p is used to print the content/value of the pointer, which is the address of its pointee. Here we assume it prints in decimal)

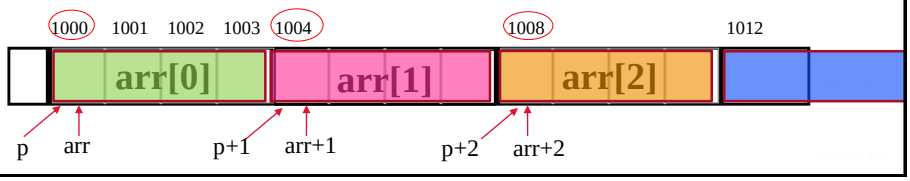
```
int main() {
 int i, arr[] = {1, 3, 5, 7, 9, 11};
 int *p = arr;
 printf("%p  %p\n", arr, p);
 for (i=0; i<3; i++)
   printf("%p  %p\n", arr+i, p+i);
 }
```

1000 1000

1000 1000

1004 1004

1008 1008



5

2.7 (5.5 pt) Given the following ANSI C program, and assume that the first element of `arr` is stored in memory address 2000; What is the output of each printf( ) statement?

```
int main() {
 int i, arr[] = {17, 3, 58, 10, 126, 22, 43, 456};
 int *p = arr;    int *q;
 printf("%p  %d\n", arr+3, *(arr+3) );
 printf("%p  %d\n", p+4, *(p+4) );
 p++;
 printf("%p  %d\n", p+3, *(p+3));
 q = arr + 5;
 printf("%p  %d\n", q, *q);
 printf ("%d-%d-%d\n", q-p, p
 (*p)++;   *(q+1) -= 10;
 for (i=0; i< sizeof(arr)/sizeof(int);i++)
   printf("%d ", *(arr+i));
}
```

2012 10

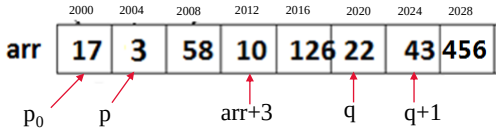
2016 126

2016 126

2020 22

4 – 1 – 1

17, 4, 58, 10, 126, 22, 33, 456



6

6

Simple rule: **arr** can not be changed

```
char arr[] = "hello";  int i;
char * p;
p = arr;        // p=&arr[0]

arr = p;        /*invalid*/
arr = &i;       /*invalid*/      p = arr+2;      /*valid*/
arr = arr +1;   /*invalid*/      *(arr + 1)=5;   /*valid*/
arr++;          /*invalid*/      c = *(arr+2);   /*valid*/

p++;            /*valid*/
p = &i;         /*valid.  now points to others*/
```

```
strlen(arr);  /*valid 5*/      sizeof arr ?   6
strlen(p);    /*valid 5 */     sizeof p ?     8
```

same Not same!

7

7

arr can be used as a pointer, but different from **ptr**

2.5 (2 pt) Given the following declaration statements in ANSI C,

```
int arr[]={1,2,3,4,5,6};  int *ptr = &arr[0];   int i;
```

Specify if each line of (independent) statement below is valid or invalid

`*(arr + 2)=0;`	valid	invalid
`ptr = &i;`	valid	invalid
`i = ptr - arr;`	valid	invalid
`i = ptr + arr;`	valid	invalid
`arr +=2;`	valid	invalid
`arr = &i;`	valid	invalid
`ptr ++;`	valid	invalid
`ptr = ptr*3;`	valid	invalid

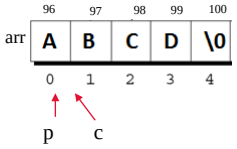
8

8

sizeof

- is not a function. It is an operator
- **`sizeof (x)`** or **`sizeof x`**
- **`sizeof (int)`**

Operator	
() [] . ->	
* & + - ! ~ ++ -- (typecast) sizeof	
* / %	arithmetic
+ -	arithmetic
>> <<	bitwise




```
int main(){
   char arr [] = {'A','B','C','D'};
   char * p = arr;

   strlen(arr);   // 4
   strlen(p);     // 4

   sizeof arr;   // 1*5=5   memory size of the array
   sizeof p;     // 8   size of pointer variable p

   ...
} 9
```

same. Pass 96

not same

9

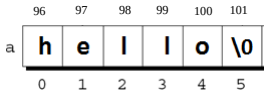
- Some interesting facts so far
  - p + n is scaled "for int * p, p+1 is p+4"
  - p1 - p2 is scaled `(116-96)/4 = 5`
  - Array name contains address of its first element a == &a[0]
- Why designed this way?
  - Facilitate Passing Array to functions!
  - We will see how.
- we will also look into, under call-by-value,
  - how array can be passed to function
  - how does **`strcpy(arr, arr2), strcat(arr,arr2)`** etc modify argument array

10

YORK UNIVERSITÉ UNIVERSITY

10

Arrays Passed to a Function



- The name/identifier of the array passed is actually a pointer/address to its first element. arr == &arr[0];

```
char a[20] = "Hello";
strlen(a); /* strlen(&a[0]). 96 is passed */
```

- The call to this function does not copy the whole array itself, just a *address* *(starting address -- a single value)* to it.

- Thus, function expecting a char array can be declared as either

```
strlen(char s[]);
```

or

```
strlen(char * s);
```

Actual prototype man 3 strlen

String-related library functions

Defined in standard library, prototype **<string.h>**

- `unsigned int strlen(char *)`
- `strcpy (char * toStr, char * fromStr)`
- `strcat(char * s1, char * s2)`
- `int strcmp(char * s1, char * s2)`

Defined in standard library, prototype **<stdlib.h>**

- `int atoi(char *)`
- `long atol(char *)`
- `double atof(char *)`

String-related library functions

Basic I/O functions <stdio.h>

- **int printf (char *format, arg1, ….);**
  - Formats and prints arguments on standard output (screen or > outputFile)
  - **printf("This is a test %d \n", x)**
- **int scanf (char *format, arg1, ….);**
  - Formatted input from standard input (keyboard or < inputFile)
  - **scanf("%x %d", &x, &y)**

- **int sprintf (char * str, char *format, arg1,…..);**
  - Formats and prints arguments to str
  - **sprintf(str, "This is a test %d \n", x)**
- **int sscanf (char * str, char *format, arg1, ….);**
  - Formatted input from str
  - **sscanf(str, "%x %d", &x, &y)** // tokenize string str

YORK UNIVERSITÉ UNIVERSITY

13

String-related library functions

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

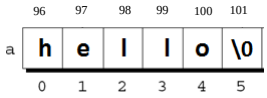
Declaration
void * bsearch (const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const v
const void *))

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

14

Arrays Passed to a Function



- Thus, function expecting a char array can be declared as either

```
strlen(char s[]);
```

or

```
strlen(char * s);
```

Actual prototype man 3 strlen

- The call to this function does not copy the whole array itself, just a *address* *(starting address -- a single value)* to it.

```
char a[20] = "Hello";
char * ps = a;
strlen(a); /* strlen(&a[0]). 96 is passed */
strlen(ps);
```

"decay"

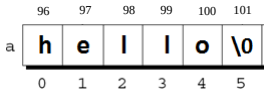
Pass by value: 96 is passed and copied to s

```
s = a = &a[0] // s is a local pointer variable
s = ps = a = &a[0] // in function strlen()
```

15

15

Arrays Passed to a Function



- Arrays passed to a function are passed by starting address.

- The name/identifier of the array passed is treated as a pointer to its first element. arr = &arr[0];

"decay"

By passing an array by a pointer (its starting address)

1. Array can be passed (efficiently)
   - a single value (e.g, address 96, no matter how long the array is)
2. Argument array can be modified
   - no & needed

```
strcpy(arr, "hello");
scanf("%s %d %f %c", arr, &age, &rate, &c);
sscanf (table[i], "%s %d %f %c", name,&age,&rate,&c)
```

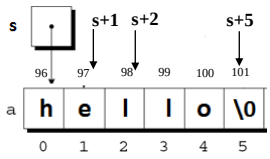
16

16

Examples using prior knowledge

Computing String Lengths -- Access argument array

```
int strlen(char *s) //s = arr == &arr[0] 96 passed by value
{                   // access arr
   int n=0;
   while ( *(s+n) != '\0')
   {
     n++;
   }
   return n;
}
```



compiler

```
int strlen(char s[])
{
  int n=0;
  while (s[n] != '\0')
  {
    n++;
  }
  return n;
}
```

```
char * ptr = arr;
strlen(arr);   /* s==arr==&arr[0]. arr 'decayed' to 96 */
strlen(ptr);   /* s== ptr == arr == &arr[0] */
```

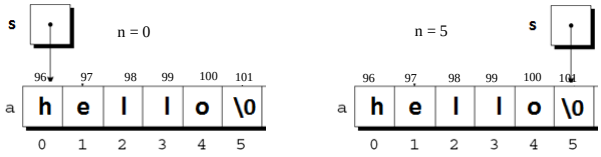
Function receives a single address value.
Does not know/care if it an array or not

Examples using prior knowledge

Computing String Lengths -- another version

```
/* move the pointer s */
int strlen(char *s)    /* s = arr == &arr[0] 96 passed */
{
   int n =0;
   while (*s != '\0'){
     n++;
     s++;    // move s (by 1),jumping to next element of arr
   }
   return n;
}
```

Function receives a single address value.
Does not know/care if it an array or not

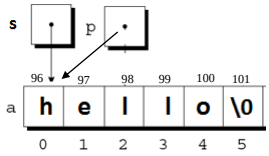



```
char * p = arr;
strlen(arr);  /*  s==arr==&arr[0] */
strlen(ptr);  /*  s== prt == arr == &arr[0] */
```

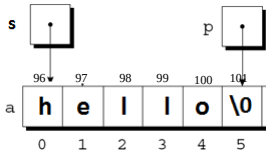
Examples using prior knowledge

Computing String Lengths -- 'cool' version A

```
/* strlen: return length of string s */
int strlen(char *s)
{
    char *p = s;
    while ( *p != '\0')
       p++;
    return p - s; // how far apart? (101-96)/1=5
}
```



Don't need n, n++, potentially faster




```
char * p = arr;
strlen(arr);
strlen(ptr);
```

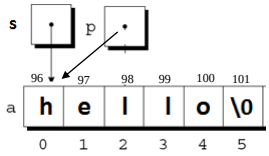
YORK UNIVERSITÉ UNIVERSITY

19

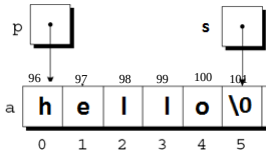
Examples using prior knowledge

Computing String Lengths -- 'cool' version B

```
/* strlen: return length of string s */
int strlen(char *s)
{
    char *p = s;
    while ( *s != '\0')
       s++;
    return s - p; // or abs(p-s)  how far apart? (101-96)/1=5
}
```



Don't need n, n++, potentially faster




```
char * p = arr;
strlen(arr);
strlen(ptr);
```

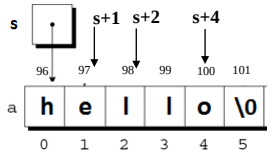
YORK UNIVERSITÉ UNIVERSITY

20

Access argument array – another example (lab5)

```
int isPalindrome (char *s)
{
    int length = strlen(s);
    int i;
    for (i = 0; i < length; i++){
      if (*(s+i) != *(s+length-1-i))
          return 0;
    }
    return 1;
 }
```

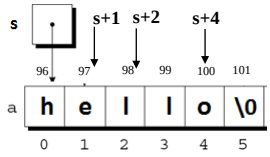
flaw ?



i		length-1-i
[0]	vs.	[4]
[1]	vs.	[3]
[2]	vs.	[2]
[3]	vs.	[1]
[4]	vs.	[0]

Access argument array – another example (lab5)

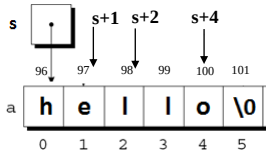
```
int isPalindrome (char *s)
{
    int length = strlen(s);
    int i;
    for (i = 0; i < length/2; i++){
      if (*(s+i) != *(s+length-1-i))
          return 0;
    }
    return 1;
 }
```



i		length-1-i
[0]	vs.	[4]
[1]	vs.	[3]

Access argument array – another example (lab5)

```
int isPalindrome (char *s)
{
    int length = strlen(s);
    int i=0; int j = length-1;
    while ( i < j)
    {
       if (*(s+i) != *(s+j))
          return 0;
       i++;
       j--;
    }
    return 1;
 }
```

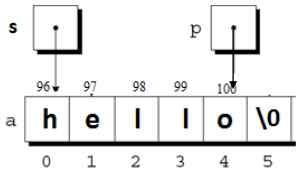


i j
[0] vs. [4]
[1] vs. [3]

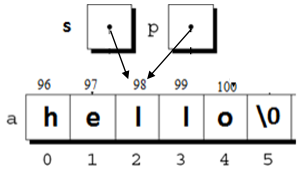
Another version

Access argument array – another example (lab5)

```
int isPalindrome (char *s)
{
    int length = strlen(s);
    int * p = s+length-1;
    while ( s < p)
    {
       if (*s != *p)
          return 0;
       s++;  // move right
       p--;  // move left
    }
    return 1;
 }
```



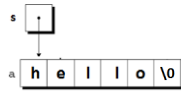
s p
[0] vs. [4]
[1] vs. [3]



'cool' version

Examples using prior knowledge

Modify argument arrays




```
int processArr(char *s) // s=arr == &arr[0] call by value
{ int i=0;
  int len = strlen(s)
  while ( i < len){
       *(s+i)='X';
    i++;
  }
}
```

compiler

```
void processArr(char s[])
{  int i=0;
   int len=strlen(s);
   while ( i< len)
   { s[i] = 'X';
     i++;
   }       // s[i++]='X';
 }
```

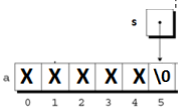
```
int processArr(char *s)
{
  int i=0;
  while ( *(s+i) != '\0')
  {  *(s+i)='X';
     i++;
  }
```

No strlen()

Traverse just once

```
int processArr(char *s)
{
   while ( *s != '\0' ){
     *s = 'X';
     s++;
  }
}
```

Move s, no i



25

25

Pointers K&R Ch 5

- Basics: Declaration and assignment
- Pointer to Pointer
- Pointer and functions (pass pointer by value)
- Pointer arithmetic +- ++ --

- **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)
  - Array as function argument – “decay”
  - Pass sub_array

YORK UNIVERSITÉ UNIVERSITY

26

Passing Sub-arrays to Functions

- 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(p);
strlen(arr);
```

functions receive address 92

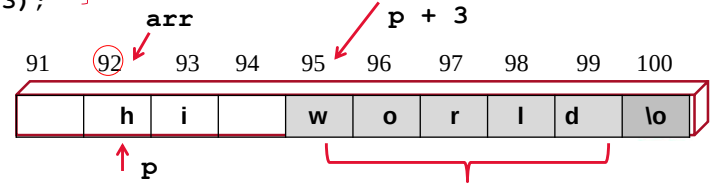
8

```
strlen (&arr[3]);
strlen (arr + 3);
strlen (p + 3);
```

functions receive address 95

5

&arr[3]
arr + 3



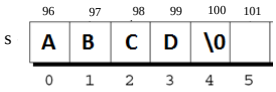
27

```
printf("%s",pa + 3);?   // world
```

sub-array

27

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(c + 1);
}
```

97 98 99 100

28

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 &
```

Pointers and arrays (Summary)

- 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)
```

"Disadvantages"?

Array Arguments (Summary)

"decay"

- An array argument is passed by a pointer (its starting address). Thus the called function just receives an single address, has no view of the whole array
  - efficient

- Then how does the function know where to stop?
- For char [], rely on `'\0'`, but what about general arrays?

```
// find the max value in the array
int findMax (int c[]){ // (int * c)

}
```

YORK UNIVERSITÉ UNIVERSITY

General array as function argument

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

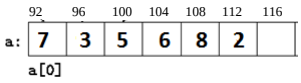
"decay"

- 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: `arrayLen(int *)` `arraySum(int *)`

Caller: `int a[20]; arrLen(a); arraySum(a);`

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


   int a = findMax(arr);
   …
}

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

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

   while ( i <  len ){
      …
```



Sizeof does not work in function

33

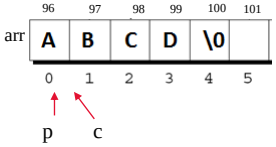
33

sizeof is not a function. It is an operator

```
int main(){
   char arr [] = {'A','B','C','D'};
   char * p = arr;

   strlen(arr);  // 4
   strlen(p);    // 4

   sizeof arr;   // 1*5=5
   sizeof p;     // 8;

   aFunction(arr);
   …
}

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

   strlen(c);   // 4
   sizeof(c);   // 8
   …
```

same. Pass 96

Not same

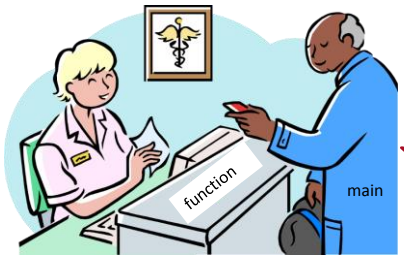


For length, sizeof does not work on pointer and in function

34

34

`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

35

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

   a = findMax(arr, len);
   …
}

/* find max in the int array. */
int findMax (int c[], int leng){
   int max = c[0];
   int i=1;
   while ( i <  leng )
      if (c[i]> max)
        max = c[i];
      i++;
   return max;
 }
```

36

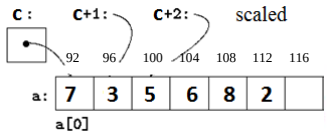
YORK UNIVERSITÉ UNIVERSITY

36

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

/* Pointer version */
int findMax (int *c, int leng){
    int max = *c; // c[0]
    int i=1;
    while ( i <  leng )
        if (*(c+i)> max) // c+4
          max = *(c+i);
        i++;
    return max;
 }
```

int c[] is converted to int * c by compiler

c[i] is converted to * (c+i) by compiler



37

37

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

38

Java avoids the hassle

```
public static void main(String[]args)
{
   int arr [] = {7,3,5,6,8,2};
   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 ) {
      if (c[i] > max)
        max = c[i]
      i++;
   }
   return max;
```

Array object

arr	
value	7 3 5 6 8 2
length	6
……	

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

39

For your information

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. Has some random value
- Points to somewhere **unknown**, may be your OS!

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

```
1) int a; ptr =&a;      int arr[20]; ptr=arr;
2) ptr = ptr2    assuming ptr2 is good
3) ptr = malloc (.....)     later ....
```

40

Problems with pointers, another scenario

```
char name[20];
char *name2;
int age; double wage;

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

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

segmentation fault
core dump

segmentation fault
core dump

41

Whenever you need to set its "pointee"

e.g.,

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

Ask yourself: Have you done one of the following

1. `ptr = &var.  /* direct */`
   `a[20]; ptr=a;`
2. `ptr = ptr2  /* indirect, assume ptr2 is good */`
3. `ptr = (..)malloc(....)  /* later today and next lecture */`

42

2.8 (2 pt) Consider the following ANSI C code fragment.

```
int main(){
  char msg[] = "Hello the World";
  char * p = msg;
  processArr (msg);    /* line 4   */
}
void processArr (char [] arr){……}
```

1) **When line 4 is executed, array `msg` itself is copied to function `processArr`, due to "call-by value"**

   True **False**

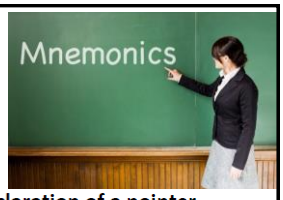
2) **Which of the following could be a valid function call to `processArr` in line 4?**

   a. `proceeeArr(&msg[2]);`
   b. `processArr(p);`
   c. `processArr(p+3);`
   d. `processArr(msg+3);`
   e. None of the above
   f. All of the above

3) **For function `processArr` which expects a char array as its argument, its argument can be declared as a pointer to char. That is, `void processArr (char * arr){……}`**

   True **False**

43



2.9 (0.5 pt Bonus) 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[])
{……}
```

(We will talk about this next week in class.)

44

YORK UNIVERSITÉ UNIVERSITY

44

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

Today and next week

YORK UNIVERSITÉ UNIVERSITY

45

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

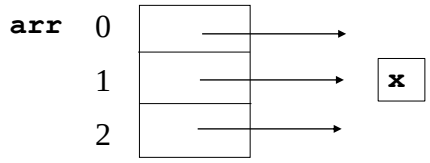
YORK UNIVERSITÉ UNIVERSITY

46

Array of Pointers (5.6)

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

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



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

```
int x;
arr[1] = &x;
*arr[1] = 4;    alias of var var=4
```


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	,	

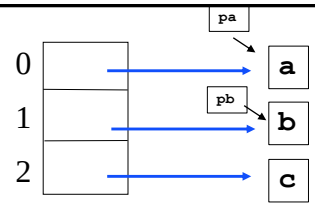

```
int * arr[3]
/* array of 3
integer pointers */
```

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

```
No () needed
```

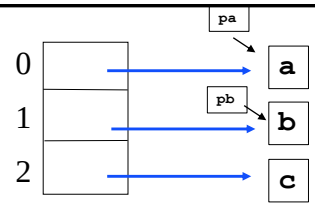
```
char (*arr)[5]
/* ??? */
```

```
#include<stdio.h>
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;
```



49

```
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("%d\n", *arr[0]);  // arr[0] is a pointer to a
  printf("%d\n", *arr[1]);
  printf("%d\n", *(arr[2]));

  ? = 100; // alias of b.   b=100
```

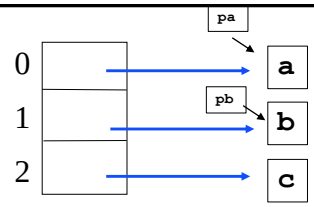


Recall:

```
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
```

50

```
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("%d\n", *arr[0]); // 4  arr[i] is pointer
  printf("%d\n", *arr[1]); // 10 *(arr[i]) is an int
  printf("%d\n", *(arr[2])); // 20  mnemonic

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

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



Pointee level

51

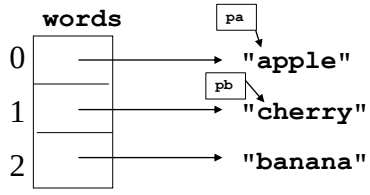
51

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;
char c[] = "banana";
char * words[3];
words[0]= pa; words[1]= pb;
words[2] = c; // array name c used as pointer
```

- **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)



52

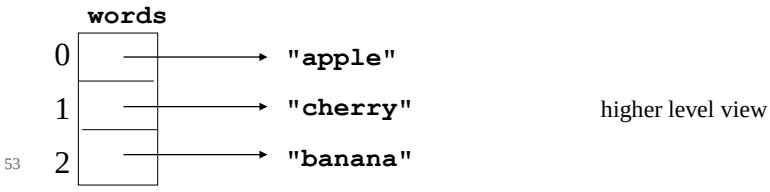
52

Arrays of Pointers (5.6)

- Common use: array of char pointers (strings)

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

char words[4][5]={"apple", "cherry", "banana"}; //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

53

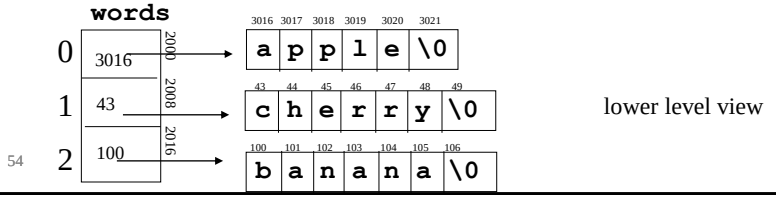
53

Arrays of Pointers (5.6)

- Common use: array of char pointers (strings)

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

char words[4][5]={"apple", "cherry", "banana"}; //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)



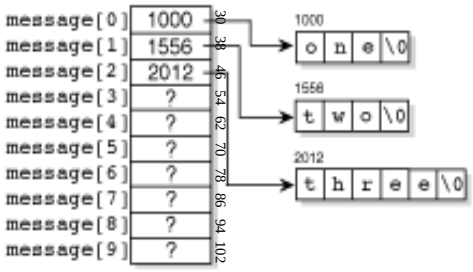
lower level view

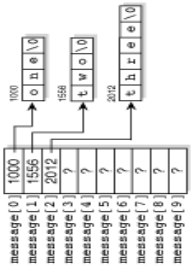
54

54

Another example, initialization

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



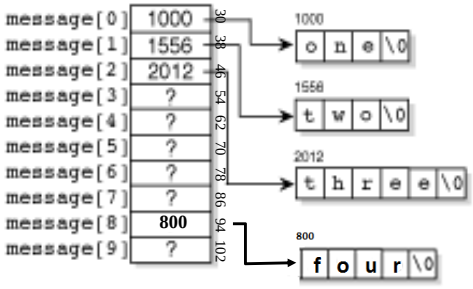


55

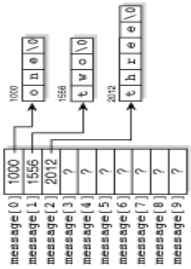
55

Another example, initialization

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




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

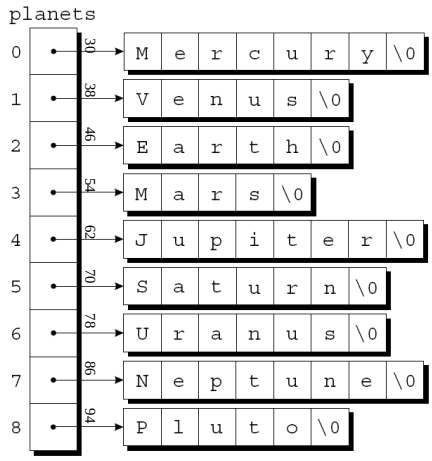


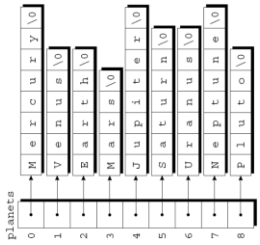
56

56

One more example

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

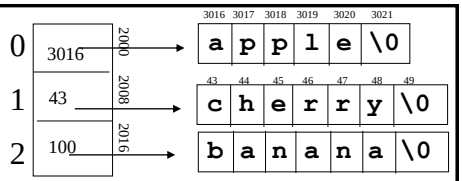




57

57

```
#include<stdio.h>
main(){

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

  printf("%s\n",          ); // apple
  printf("%s\n",          ); // cherry
  printf("%s\n",          ); // banana

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



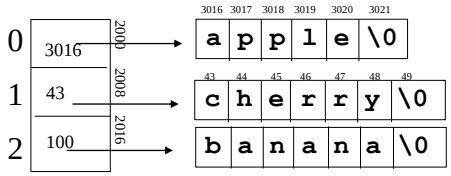
Recall:

```
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
```

58

58

```
#include<stdio.h>
main(){

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

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

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



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
```

59

59

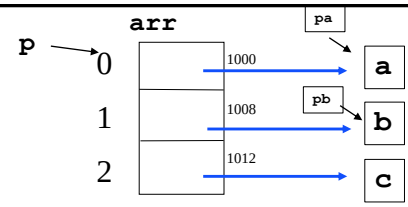
Pointers K&R Ch 5

- **Pointers 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

60

```
#include<stdio.h>

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
```

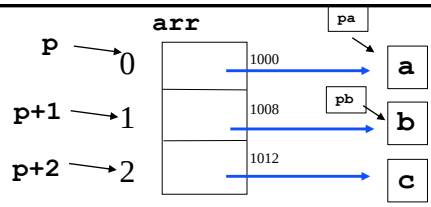


Recall:

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

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

61

```
#include<stdio.h>

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", ? ); // 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", ? );
```



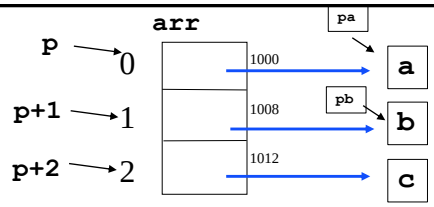
Recall:

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

62

62

```
#include<stdio.h>

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

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

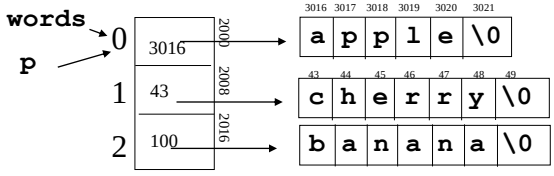


Recall:

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



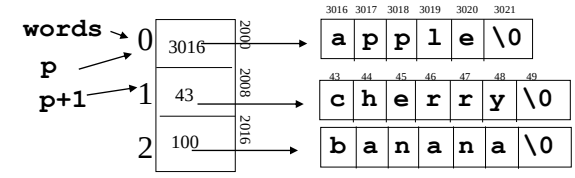
```
#include<stdio.h>

main(){
  char * words[]={"apple", "cherry", "banana"};

  char * p = words; // p = &words[0] == 3000
```



Recall:

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

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

```
}
```




```
#include<stdio.h>

main(){
  char * words[]={"apple", "cherry", "banana"};

  char ** p = words; // p = &words[0] == 3000

  printf("%p %s\n", p,      );     // 2000 apple  words[0]
  printf("%p %s\n", p+1,    ); // 2008 cherry words[1]
  printf("%p %s\n", p+2,    ); // 2016 banana words[2]

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

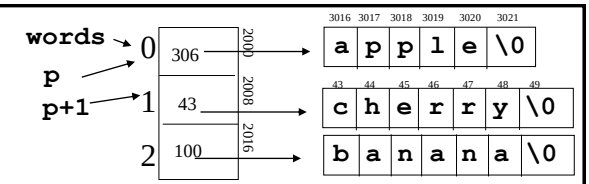
Recall:

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

Hardest today

65 `printf("%c\n", *(*(p+1)+5 ));  ?`

65




```
#include<stdio.h>

main(){
  char * words[]={"apple", "cherry", "banana"};

  char ** p = words; // p = &words[0] == 3000

  printf("%p %s\n", p,  *p  );     // 2000 apple  words[0]
  printf("%p %s\n", p+1,*(p+1)); // 2008 cherry words[1]
  printf("%p %s\n", p+2, *(p+2)); // 2016 banana words[2]

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

Recall:

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

Hardest today

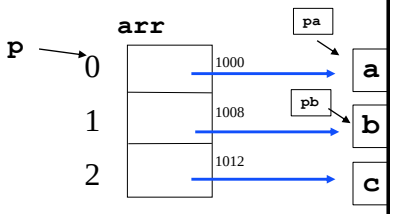
66 `printf("%c\n", *(*(p+1)+5 )); //words[1][5]  y`

66

Pointers K&R Ch 5

- **Pointers 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

Passing an array of pointers to functions




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

 print_message( words, 3);
}

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

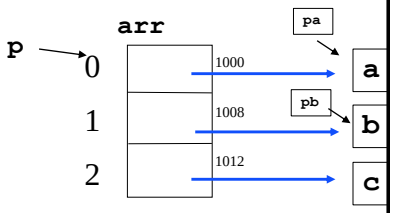
Expect an array of int *

Needed to provide !!!

// compiler: **(p+count)

Pointee level

Passing an array of pointers to functions




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

 print_message( words, 3);
}
```

"decay"?

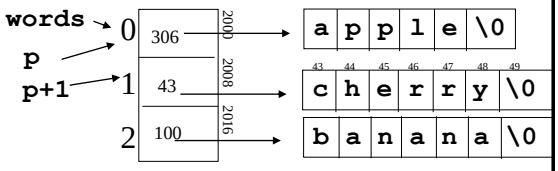
Pass address of 1st element -- &pointer

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

Needed to provide !!!

Pointee level

Passing an array of pointers to functions




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

 print_message( words, 3);
}
```

Expect an array of char *

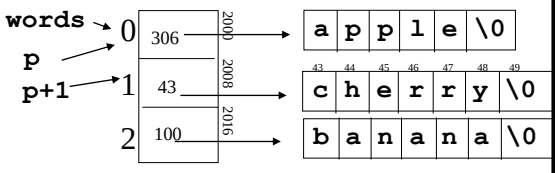
```
void print_message(char *p[], int n){
 int count;
 for (count=0; count<n; count++)
   printf("%s ", p[count]);
}
```

Needed to provide !!!

// compiler: *(p+count)

Pointer level

Passing an array of pointers to functions




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

 print_message(words, 3);
}
```

"decay"?

Pass address of 1st element -- &pointer

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

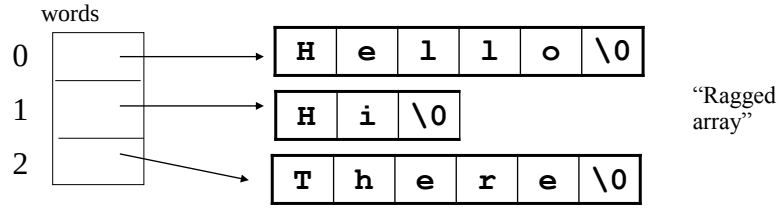
Pointer level

Pointers K&R Ch 5

- **Pointers 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

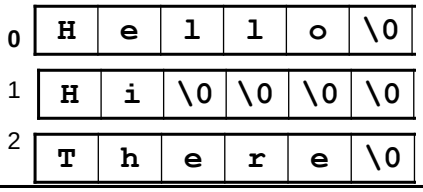
Advantage of Pointer Arrays (vs. 2D array)

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



What is the difference?

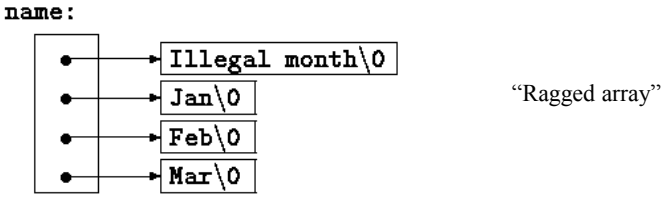
```
char words[3][6] = {"Hello", "Hi", "There"};
```



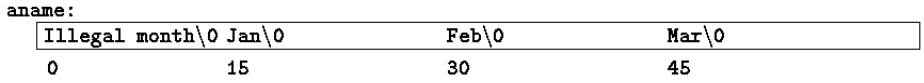
73

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

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




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

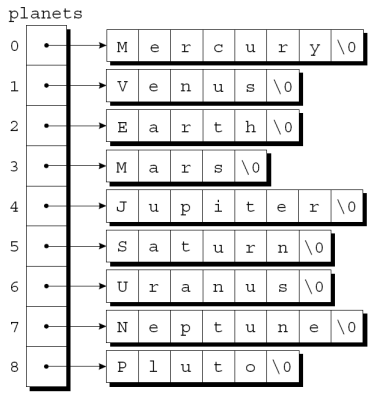


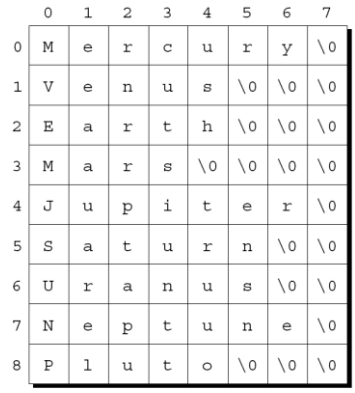
74

74

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

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





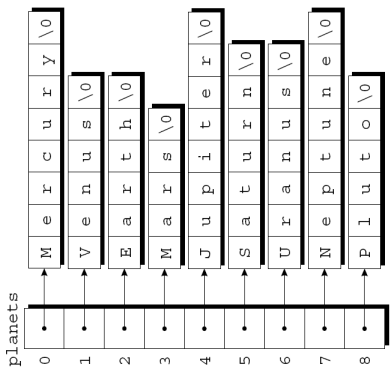

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

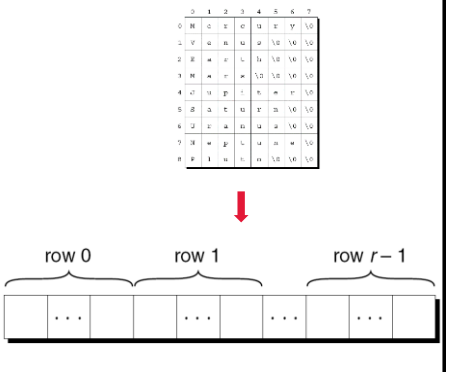
75

75

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"};
```

76

76

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

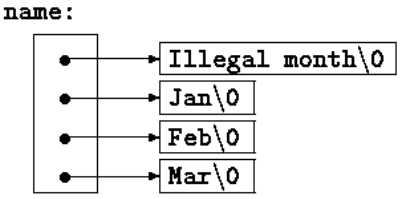
- Advantage of pointer array **b** vs. 2D array **a**:
  1. the rows of the array may be of different lengths (potentially saving space).
  2. Another advantage? **Swap rows!**

77

77

Advantage of Pointer Arrays (vs. 2D array)

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



How to swap ?

sizeof name: 4*8 = 32 total memory size 4*8+15+4+4+4=63

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

aname:

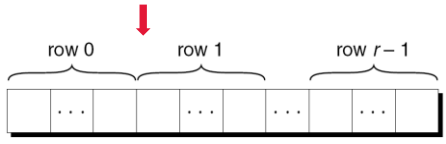
Illegal month\0 Jan\0	Feb\0	Mar\0
0 15	30	45

How to swap?

78 sizeof aname: 4*15*1 = 60

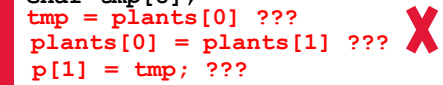
78

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

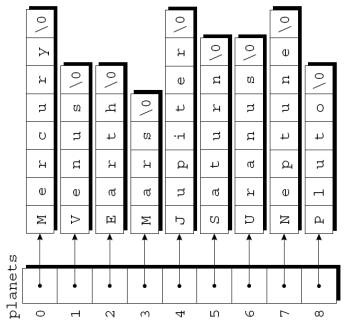


How to swap?

```
char tmp[8];
tmp = plants[0] ???
plants[0] = plants[1] ???
p[1] = tmp; ???
```



```
strcpy(tmp,p[0]);  // copy chars
strcpy(p[0],p[1]); // one by one
strcpy(p[1],tmp);
```

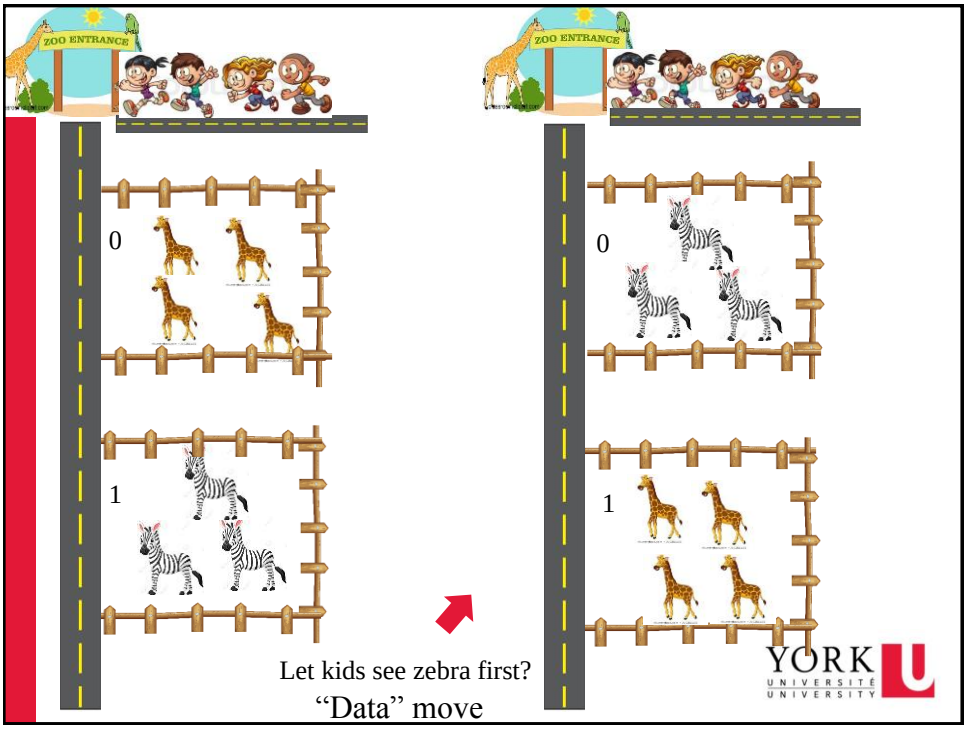


How to swap?

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

79

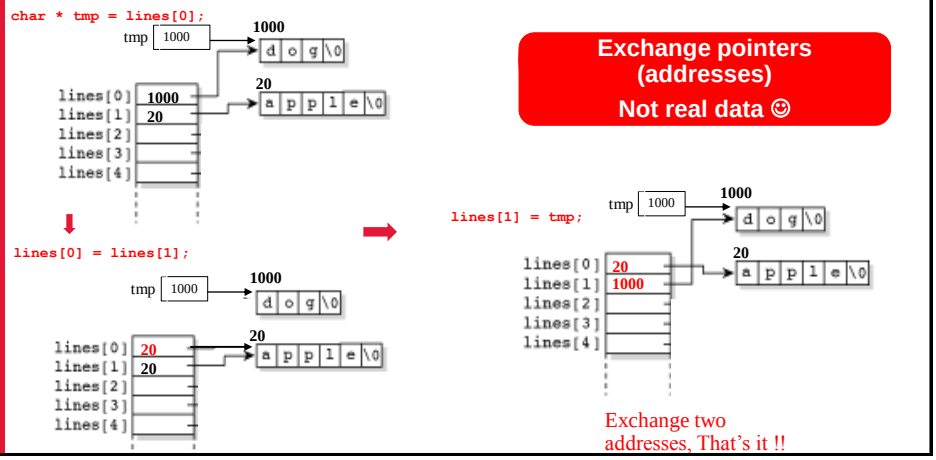
79



80

Efficient manipulation of strings

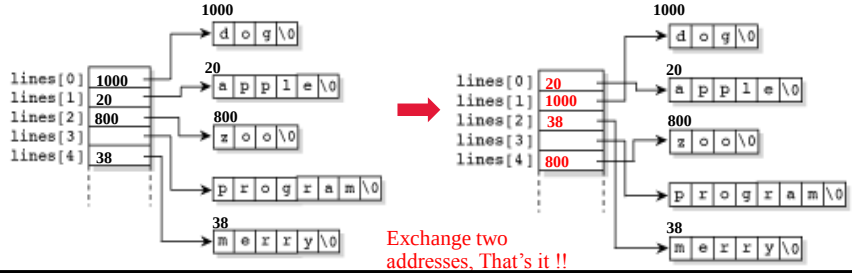
```
char *lines[]={"dog", "apple", "zoo", "program", "merry"};
// [0] vs [1]
char * tmp = lines[0]; // tmp gets 1000, pointing to "dog"
lines[0] = lines[1]; // [0] gets 20, pointing to "apple"
lines[1] = tmp;   // [1] gets 1000, pointing to "dog"
```



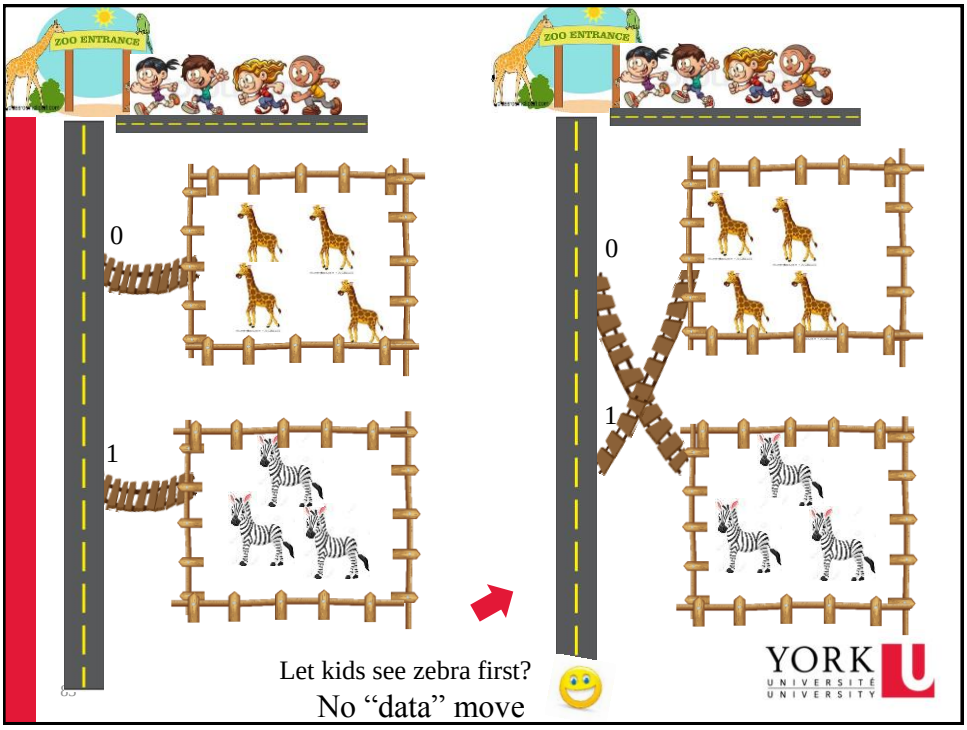
81

Efficient manipulation of strings

```
char *lines[]={"dog", "apple", "zoo", "program", "merry"};
// [0] vs [1]
char * tmp = lines[0]; // tmp gets 1000, pointing to "dog"
lines[0] = lines[1]; // [0] gets 20, pointing to "apple"
lines[1] = tmp;   // [1] gets 1000, pointing to "dog"

// [2] vs.[4]
tmp = lines[2];  // tmp points to "zoo"
lines[2] = lines[4];  // [2] points to "merry"
lines[4] = tmp;  // [4] points to "zoo"
```



82



83

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 argument (5.10)**
- Memory allocation (extra)
- Pointer to structures (6.4)
- Pointer to functions

Today and next week

YORK UNIVERSITÉ UNIVERSITY

84

2.9 (+0.5 pt Bonus question) 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`", which implies that `ptr` is an integer pointer.

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

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

YORK UNIVERSITÉ UNIVERSITY


```
public static void main(String[] args)
```

Command-Line Arguments (5.10) (Program arguments)

- Up to now, we defines main as **`int main()`**
- Usually it is defined as

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

  - **`argc`** is the number of arguments (including program name)
  - **`argv`** is an array containing the arguments.

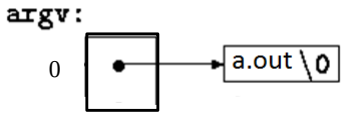
  - **`argv[0]`** is a pointer to a string with the program name. So **`argc`** is at least 1. (Java?)
  - **`Optional arguments:`** **`argv[1] ~~ argv[argc-1]`**

YORK UNIVERSITÉ UNIVERSITY

Command-line arguments (program arguments)

- red 421 % a.out
  argv[0]: a.out

  argc: 1

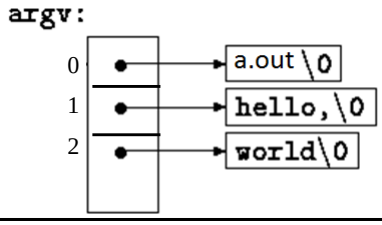


1 arg

- red 421 % a.out hello, world
  argv[0]: a.out
  argv[1]: hello,
  argv[2]: world

  argc: 3

3 args



87

```
public static void main(String[] args)
```

Different from Java

- indigo 421 % a.out we are program arguments
  argv[0]: a.out 0 1 2 3 4
  argv[1]: we
  argv[2]: are
  argv[3]: program
  argv[4]: arguments

  argc: 5

5 args

- indigo 422 % java Prog we are program arguments
  args[0]: we 0 1 2 3
  args[1]: are
  args[2]: program
  args[3]: arguments

  args.length: 4

4 args

YORK UNIVERSITÉ UNIVERSITY

88

Command-Line Arguments (cont.)

file.c

```
main( int argc, char *argv[] ) {
   int i;
   printf("Number of arg: %d\n", argc );
   for(i=0; i<argc; i++ )
    printf("argv[%d]: %s\n",i, argv[i] );
}                                    *(argv+i)  // ccmpiler
```

```
% gcc file.c
% a.out
Number of arg: 1
argv[0]: a.out
```

```
% gcc file.c –o xyz
% xyz how are you
Number of arg: 4
argv[0]: xyz
argv[1]: how
argv[2]: are
argv[3]: you
```

```
% a.out how "are you"
Number of arg: 3
argv[0]: a.out
argv[1]: how
argv[2]: are you
```


Command-Line Arguments (cont.)

```
main( int argc, char *argv[] ) {
   int i;
   printf("Number of arg: %d\n", argc );
   char ** p = argv; // &argv[0]
   for(i=0; i<argc; i++ )
     printf("argv[%d]: %s\n", i, *(p+i) );
}
```

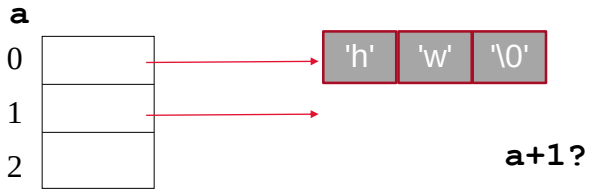
```
% gcc file.c
% a.out
Number of arg: 1
argv[0]: a.out
```

```
% gcc file.c –o xyz
% xyz how are you
Number of arg: 4
argv[0]: xyz
argv[1]: how
argv[2]: are
argv[3]: you
```

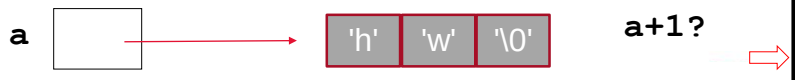
```
% a.out how "are you"
Number of arg: 3
argv[0]: a.out
argv[1]: how
argv[2]: are you
```

Array of points vs. pointers to whole Array

```
char *a[3]; /* array of 3 pointers  */
```




```
char (*a)[3]; /* pointer to a 3 char array */
```



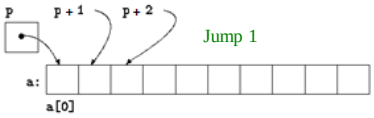
91

For your information

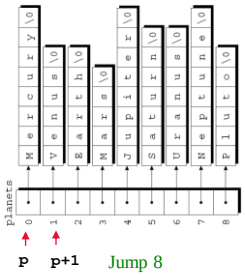
91

Summary of

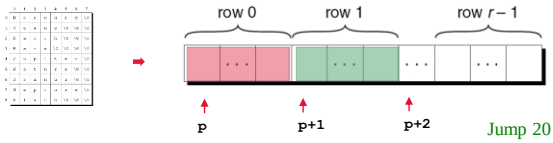
"decay"



- char a [20] ➡➡ char * p
- char *a [20] ➡➡ char ** p



- char a [ ][20]? ➡➡ char (*p)[20]



92

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 argument (5.10)
- Memory allocation (extra)
- Pointer to structures (6.4)
- Pointer to functions

Today

YORK UNIVERSITÉ UNIVERSITY

- Lab 6 this week.
- No lab meeting this Wednesday

YORK UNIVERSITÉ UNIVERSITY

quiz

```
char a[] = "hello";
char * p = a;
sizeof a?   6
sizeof p?   8
strlen(a)?  5
strlen(p)?  5

char a[][10]={"hello", "the", "beautiful", "world"};
char * b[4];
sizeof a?   40
sizeof b?   32
```

(4*8=32)