EECS2031 Software Tools

SU 2019

Jul 8, 2019 Lecture 8

1

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". Has some random value
  - 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 good */ _ _
3) ptr = malloc (.....) /* later today */
```

2

2

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

3

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 today */`

4

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)

last lecture

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

YORK UNIVERSITÉ UNIVERSITY

5

Pointers K&R Ch 5

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

YORK UNIVERSITÉ UNIVERSITY

6

Precedence

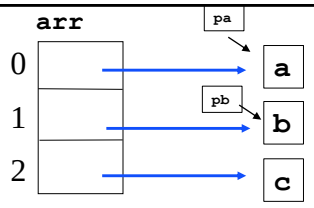
Operator Type	Operator	
Primary Expression Operators	() [] . ->	
Unary Operators	* & + - !~ ++ -- (typecast) sizeof	
Binary Operators	* / %	arithmetic
	+ -	arithmetic
	>> <<	bitwise
	< > <= >=	relational
	== !=	relational
	&	bitwise
	^	bitwise
	|	bitwise
	&&	logical
	||	logical
Ternary Operator	?:	
Assignment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

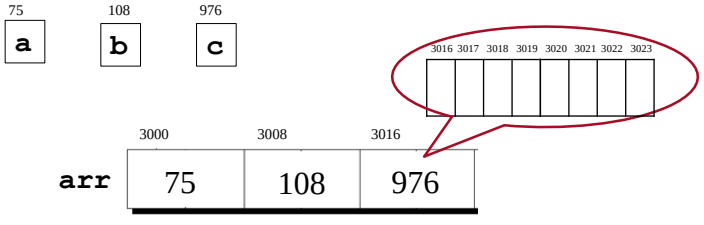

```
int * arr[3]
/* array of 3
integer pointers */

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

No () needed

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

7

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;
```



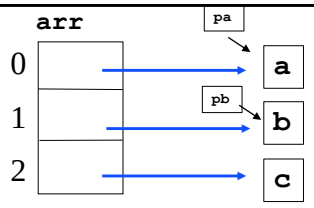


Each element is a pointer, size usually 8 bytes (regardless of the type)

8

8

Array of pointers to scalar types



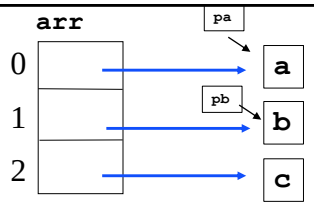

```
main(){
  int a=4, b=10,c=20;
  int *pa=&a; int *pb=&b;

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

  printf("%d\n",          : // 4
  printf("%d\n",          : // 10
  printf("%d\n",          ); // 20

  ?  = 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
```

9

Array of pointers to scalar types




```
main(){
  int 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
}
```

10

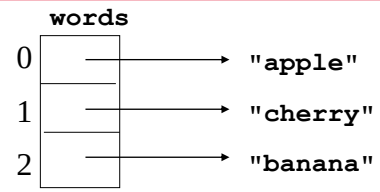
Pointee level

Array of pointers to strings

- 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

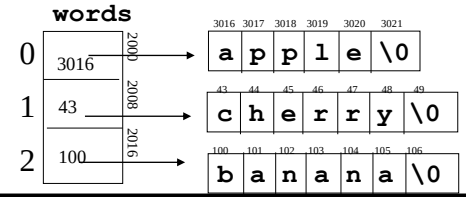
11

Array of pointers to strings

- 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)



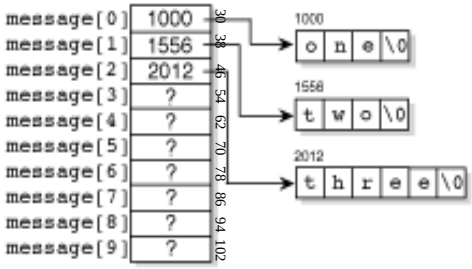
lower level view

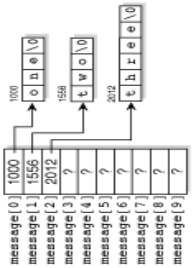
12

Array of pointers to strings

Another example, initialization

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

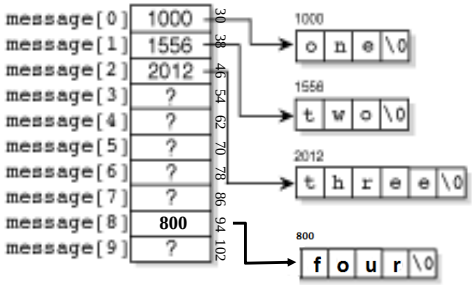




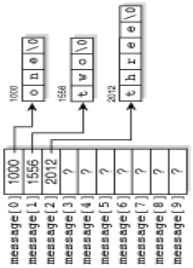
Array of pointers to strings

Another example, initialization

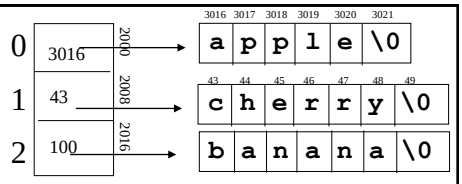
- `char *message[10] = {"one", "two", "three"};`




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



Array of pointers to strings




```
#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
```

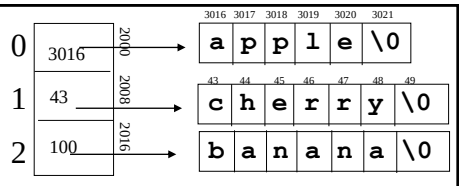
Recall:

```
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
```

15

15

Array of pointers to strings




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

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

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

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

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
```

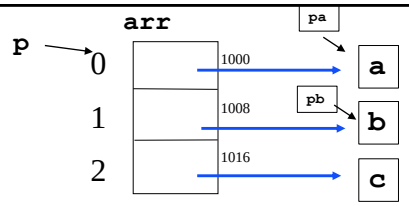
16

16

Pointers K&R Ch 5

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

Array of pointers to scalar types

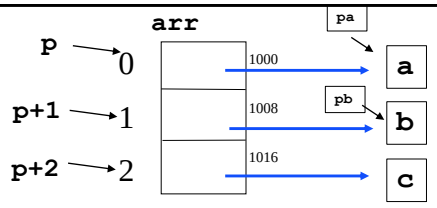



```
main(){
  int a,b,c, *pa, *pb;
  a=4; b=10;c=20;
  pa=&a, pb=&b;

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

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

Recall:

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

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

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

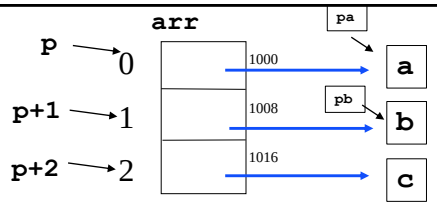
Recall:

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

19

19

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] "pointee level"
  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));
```

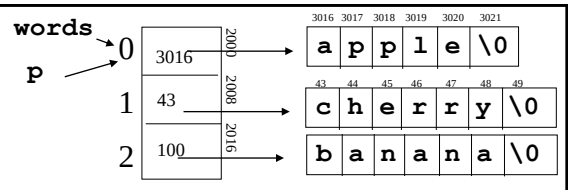
Recall:

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

20

20

Array of pointers to strings



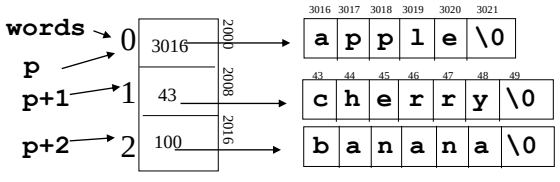

```
main(){
  char * words[]={"apple", "cherry", "banana"};

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

Recall:

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

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

```
}
```


Array of pointers to strings




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

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

  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
}
```

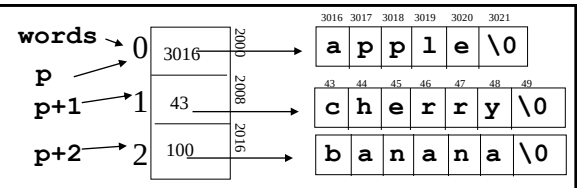
Recall:

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

Hardest today


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

Array of pointers to strings




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

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

  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

```
    printf("%c\n", *(*(p+1)+5 )); //words[1][5]  y
    printf("%c\n", **p);  //words[0][0]  a
```

23

23

Pointers K&R Ch 5

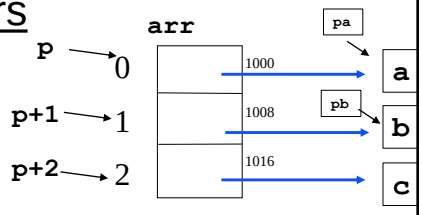
- **Array of pointers (5.6)**
  - Declaration, initialization, accessing pointees via element pointers
    - Array of pointers to scalar types (e.g., int)
    - Array of pointers to strings
  - Pointer to pointer arrays (what type is it?)
    - Array of pointers to scalar types
    - Array of pointers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

YORK UNIVERSITÉ UNIVERSITY

24

Passing an array of pointers to functions

Array of pointers to scalar types




```
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 *

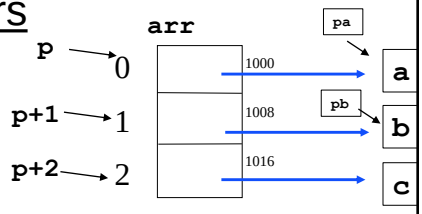
Needed to provide !!!

// compiler: **(p+count)

Pointee level

Passing an array of pointers to functions

Array of pointers to scalar types




```
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));
}
```

"decay"?

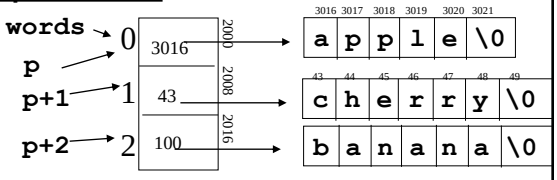
Pass address of 1st element -- &pointer

Needed to provide !!!

Pointee level

Passing an array of pointers to functions

Array of pointers to strings




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

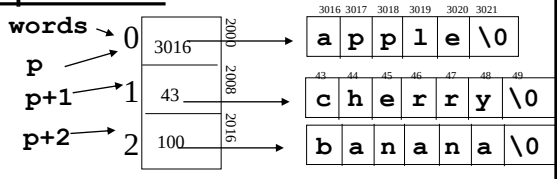
Needed to provide !!!

// compiler: *(p+count)

Pointer level

Passing an array of pointers to functions

Array of pointers to strings




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

 print_message(words, 3);
}

void print_message(char** p, 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

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

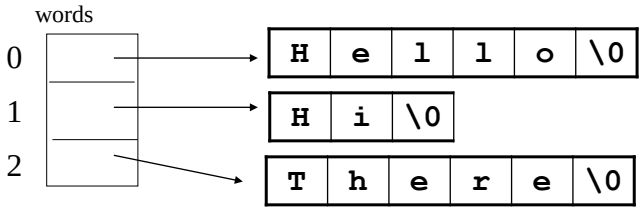
YORK UNIVERSITÉ UNIVERSITY

29

Array of pointers to strings

Advantage of Pointer Arrays (vs. 2D array)

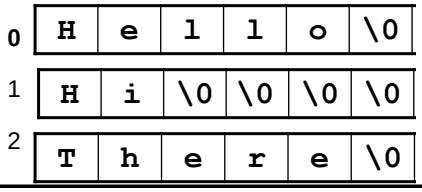
```
char * words[]={"Hello", "Hi", "there"};
```



"Ragged array"

What is the difference?

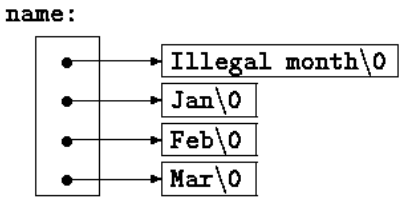
```
char words[3][6] = {"Hello", "Hi", "There"};
```



30

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

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



"Ragged array"

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

aname:

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

31

31

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 + sizeof 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!**

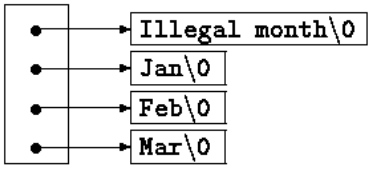
32

32

Advantage of Pointer Arrays (vs. 2D array)

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

name:



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"}
```

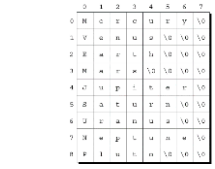
aname:

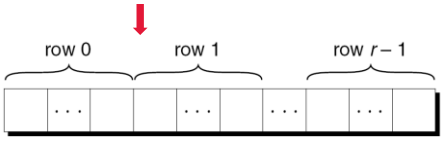
Illegal month\0 Jan\0	Feb\0	Mar\0
0 15	30	45

How to swap?

33 sizeof aname: 4*15*1 = 60


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





How to swap?

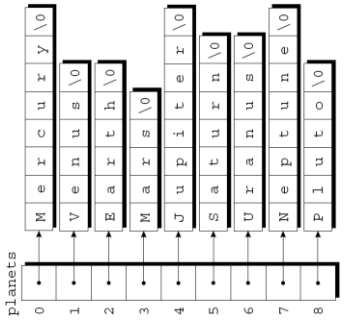
```
char tmp[8];
tmp = planets[0] ???
planets[0] = planets[1] ???
planets[1] = tmp; ???
```

✗

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

34

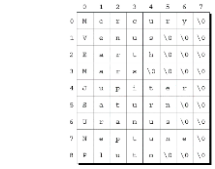
O(n)

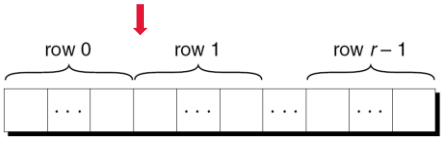


How to swap? ➡

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

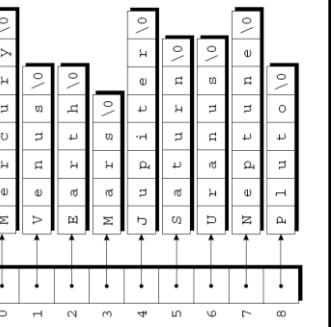
```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```





How to swap?

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



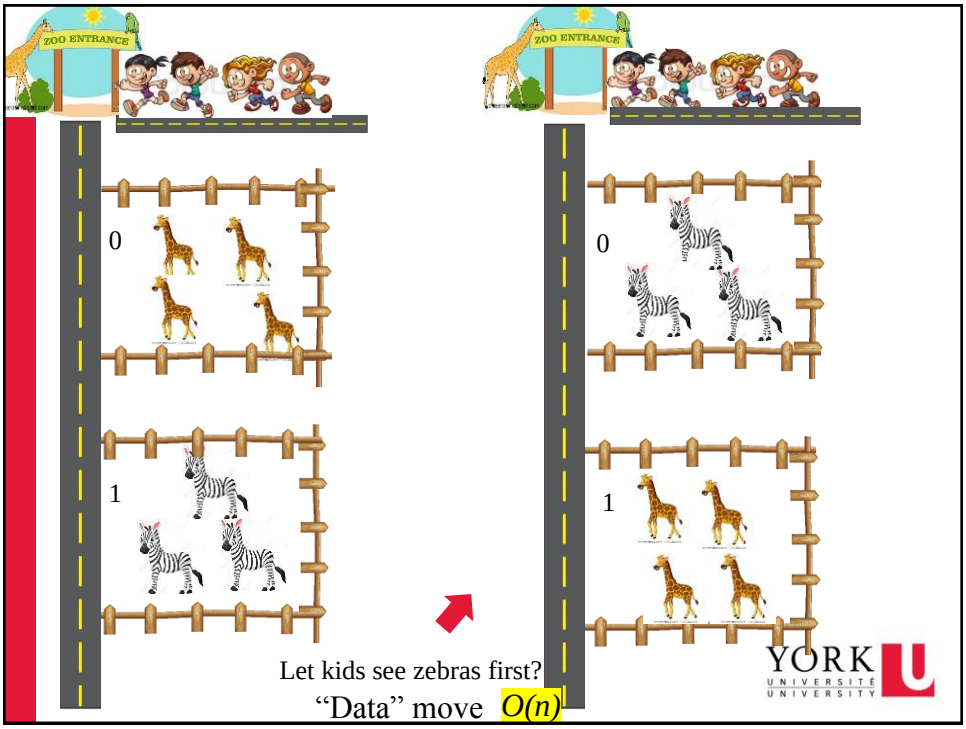
How to swap?

```
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"};
```

35



Let kids see zebras first?

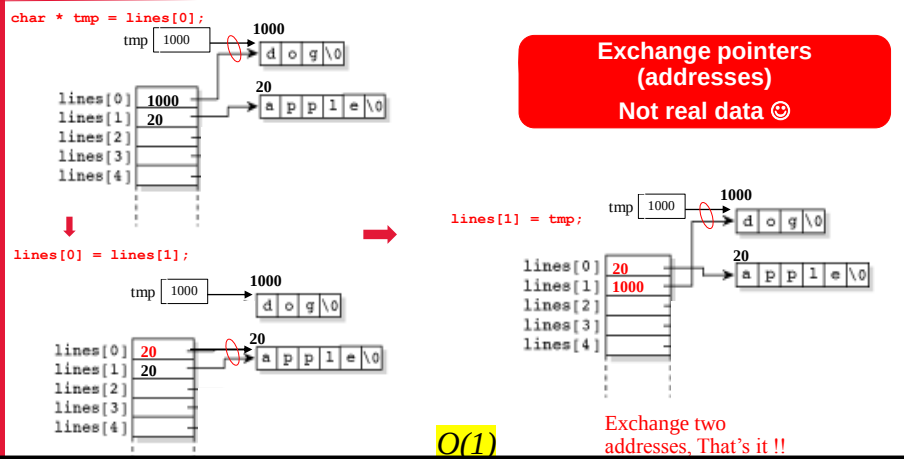
"Data" move *O(n)*

YORK UNIVERSITÉ UNIVERSITY

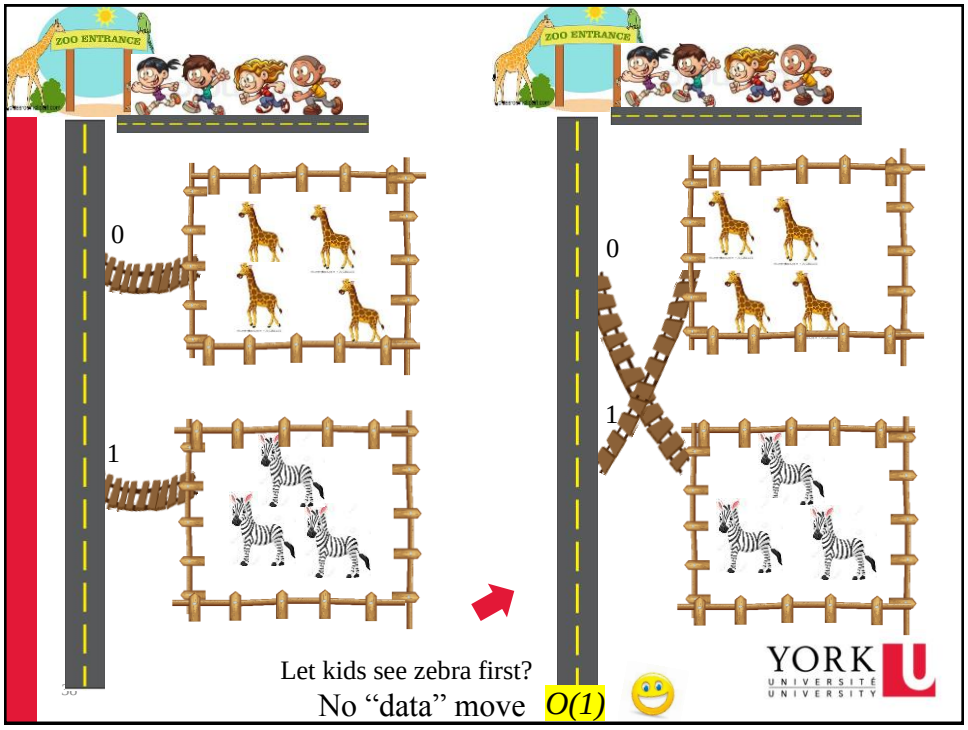
36

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"
```



37



38

Pointers K&R Ch 5

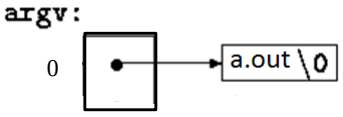
- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (pass pointer by value) (5.2)
- Pointer arithmetic +- ++ -- (5.4)
- Pointers and arrays (5.3)
  - Stored consecutively
  - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
  - Array name contains address 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

Last lecture

Command-line arguments (program arguments)

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

  argc: 1

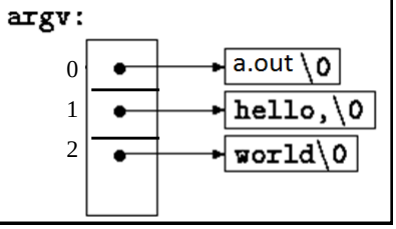


1 arg

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

  argc: 3

3 args



```
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

  5 args

  argc: 5

- indigo 422 % java Prog we are program arguments

  args[0]: we 0 1 2 3
  args[1]: are
  args[2]: program
  args[3]: arguments

  4 args

  args.length: 4

YORK UNIVERSITÉ UNIVERSITY

41

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
```

42

42

Command-Line Arguments (cont.)

file.c

```
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
```

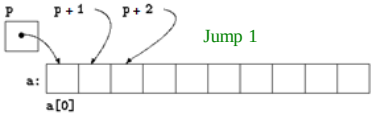
```
% gcc file.c -o xyz
% xyz how are you
Number of arg: 4
argv[0]: xyz
argv[1]: how
argv[2]: are
argv[3]: you
```

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

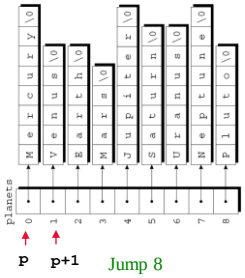
43

Summary of

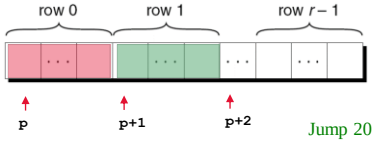
"decay"



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



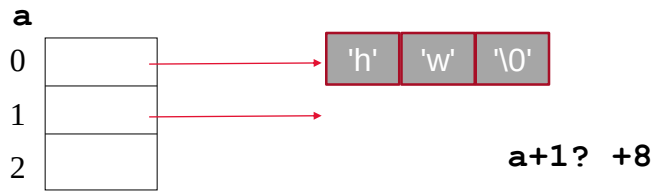
- char a [ ][20]? ➡➡ char (*p)[20]



44

Array of points vs. pointers to whole Array

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



a+1? +8

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

a+1? +3

45

For your information

45

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

46

Dynamic memory allocation scenario / motivation 1

- When we define an array, we allocate memory for it

```
int arr[20];
```

sets aside space for 20 ints (80 bytes)

- This space is allocated at compile-time (i.e. when the program is compiled)

```
char arr[20];
int arr[20][30];    20*30*4 bytes
```

47

47

Dynamic memory allocation scenario / motivation 1

- What if we do not know how large our array should be?
- In other words, we need to be able to allocate memory at run-time (i.e. while the program is running)

- How?

```
int n;
scanf("%d", &n);
int my_array[n];    /* but not allowed in ANSI-C */
```

```
gcc -ansi -pedantic varArray.c
gcc -ansi -pedantic-errors varArray.c
   ISO C90 forbids variable length array 'my_array'
```

48

48

- Fortunately, C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution.
- Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

- The `<stdlib.h>` header declares three memory allocation functions:
  - `malloc` Allocates a block of memory but doesn't initialize it.
  - `calloc` Allocates a block of memory and clears it.
  - `realloc` Resizes a previously allocated block of memory.
- These functions return a value of type `void *` (a "generic" pointer).
  - function has no idea what type of data to store in the block.

49

Common library functions [Appendix of K+R]

<stdio.h>

```
printf()
scanf()
getchar()
putchar()

sscanf()
sprintf()

gets()  puts()
fgets() fputs()

fprintf()
fscanf()
```

<string.h>

```
strlen(s)
strcpy(s,s)
strcat(s,s)
strcmp(s,s)
```

<math.h>

```
sin() cos()
exp()
log()
pow()
sqrt()
ceil()
floor()
```

<stdlib.h>

```
double atof(s)
int    atoi(s)
long   atol(s)
void    rand()
void   system()
void   exit()
int    abs(int)

void* malloc()
void* calloc()
void* realloc()
void  free()
```

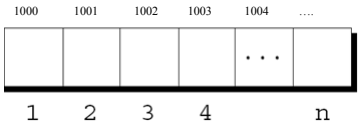
<ctype.h>

```
int islower(int)
int isupper(int)
int isdigit(int)
int isxdigit(int)
int isalpha(int)

int tolower(int)
int toupper(int)
```

<assert.h>

```
assert()
```

malloc()



- **"stdlib.h"** defines:

```
void * malloc (int n);
```

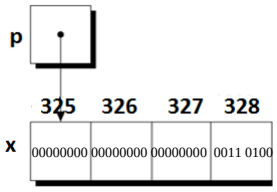
- allocates memory at run-time
- returns a void pointer to the memory that has at least n bytes available (just allocated for you).
  - Address of first byte e.g., 1000
  - Can be casted to any type

malloc()

```
#include <stdlib.h>

int main() {
  int *p;   // uninitialized, not point to anywhere

  *p = 52;
  printf("%d\n", *p);
}
```

```
segmentation fault
core dump
```

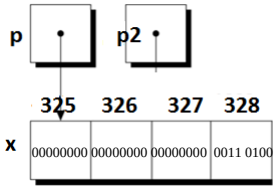
malloc()




```
#include <stdlib.h>

int main() {
  int *p, x;
  p = &x;
  *p = 52;  // x=52
  printf("%d\n", *p);
}
```

fix1

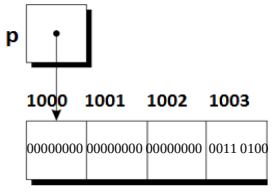
malloc()




```
#include <stdlib.h>

int main() {
  int *p, x;
  int *p2 = &x;  p = p2;
  *p = 52;  // x=52
  printf("%d\n", *p);
}
```

fix2

malloc()




```
#include <stdlib.h>

int main() {
  int *p;
  p = (int *) malloc(4);
  *p = 52;
  printf("%d\n", *p);
}
```

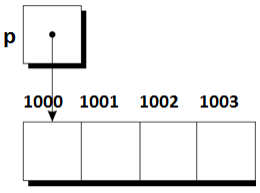

fix3

- Note: type conversion (cast) on result of malloc
  `p = malloc(4);` also works. Will convert

55

55

sizeof



- A better approach to ensure portability

```
int *p;
p = (int *) malloc( sizeof(int) );
*p = 52;
```

56

56

NULL

- Not always successful
- malloc() returns **NULL** when it cannot fulfill the request, i.e., memory allocation fails (e.g. no enough space)

```
int *p;
p = (int *)malloc(100000);// malloc returns NULL
p = (int *)malloc(-10);   // malloc returns NULL
```

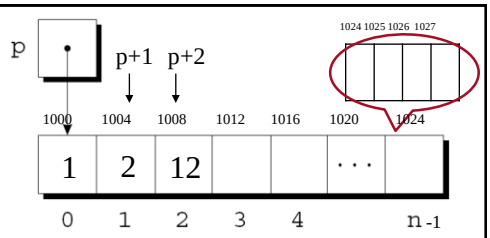

NULL

- <stdlib.h> <stdio.h> <string.h> …defines macro
  NULL a special pointer constant with value 0
- 0 (zero) is never a valid address
- **NULL** == "0 as a pointer" == "points to nothing"
  - `int * p;  // p == NULL`

```
p = malloc(1000000);
if (p == NULL) {
   exit(0) /* allocation failed; take appropriate action
}
else …
```

```
if ( (p = malloc(1000000)) == NULL) {
   exit(0) /* allocation failed; take appropriate action
}else ….
```

malloc()



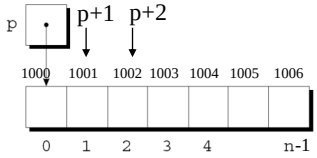
4n bytes allocated.
n=7 28 bytes 1000~1027 allocated

```
#include <stdlib.h>

int main() {
  int n;
  printf("Size of array: ");
  scanf("%d", &n);

  int * p = (int *)malloc(n * sizeof(int));
  if (p == NULL)
     exit(0);

  *p = 1;          // p[0] = 1
  *(p+1) = 2;      // p+1 = 1004  p[1]= 2
  *(p+2) = 12;     // p+2 = 1008  p[2] = 12
}                    pointer arithmetic!!!
```

59

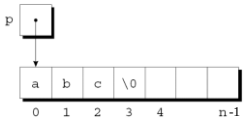
59

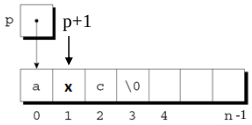
malloc()



n bytes allocated.
n=7 7 bytes 1000~1006 allocated

```
#include <stdlib.h>

int main() {
  int n;
  printf("Size of array: ");
  scanf("%d", &n);

  char * p = (char *)malloc(n * sizeof(char));
  if (p == NULL)
     exit(0);

  strcpy(p, "abc");

  *(p+1) = 'x';

}
```





60

60

calloc()

- What if we want to allocate arrays of **n** element?

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

alternatively,

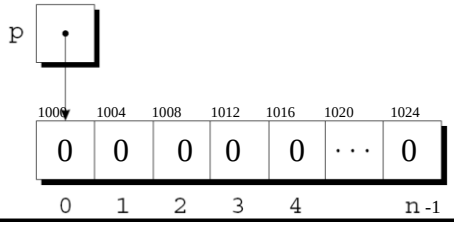
```
void * calloc(int n, int size);
```

- **calloc** allocates an array of **n** elements where each element has size **size**
- e.g.

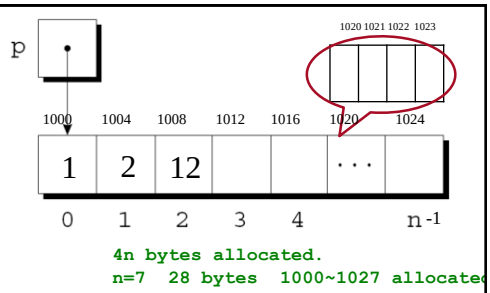
```
int *p;
p = (int *)calloc(6, sizeof(int));
```


calloc() vs. malloc()

- **calloc(x , y)** is pretty much the same as **malloc(x * y)**
- except
  - **malloc** does not initialize memory
  - **calloc** initializes memory content to 0 (zero)



malloc()




```
#include <stdlib.h>

int main() {
  int n;
  printf("Size of array: ");
  scanf("%d", &n);

//int * p = (int *)malloc(n * sizeof(int));
  int * p = (int *)calloc(n , sizeof(int));
  if (p == NULL) exit(0);

  *p = 1;          // p[0] = 1
  *(p+1) = 2;      // p+1 = 1004  p[1]= 2
  *(p+2) = 12;     // P+2 = 1008  p[2] = 12;
```

4n bytes allocated.
n=7 28 bytes 1000~1027 allocate

63

63

free()

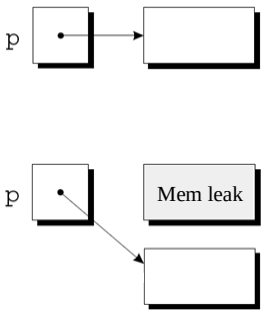
- memory allocation functions `malloc, calloc` obtain memory blocks from a storage pool known as the ***heap,*** where storage is persistent until the programmer explicitly requests that it be deallocated
- A block of memory that's no longer accessible to a program is said to be ***garbage.***
  - A program that leaves garbage behind has a ***memory leak.***
- Some languages (e.g., Java) provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't.

64

64

Memory Leaks

```
int *p;
p = (int *) malloc( 20 );
...
p = &i; //now point to sth else
```



- The first memory block is lost “forever” (until program terminates).
- MAY cause problems (exhaust memory).

65

65

Memory Leaks

- What happens if some memory is heap allocated, but never deallocated?
- A program which forgets to deallocate a block is said to have a "memory leak" which may or may not be a serious problem. The result will be that the heap gradually fill up as there continue to be allocation requests, but no deallocation requests to return blocks for re-use.
- For a program which runs, computes something, and exits immediately, memory leaks are not usually a concern. Such a "one shot" program could omit all of its deallocation requests and still mostly work.
- Memory leaks are more of a problem for a program which runs for an indeterminate amount of time. In that case, the memory leaks can gradually fill the heap until allocation requests cannot be satisfied, and the program stops working or crashes.

66

For your information

66

free()

- Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

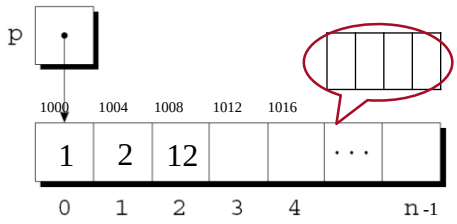
```
void free(void *ptr);
```

- "frees" memory we previously allocated, tells the system we no longer need this memory and that it can be reused
- address in "**ptr**" must have been returned from either **malloc**, **calloc** or **realloc**.

```
p = malloc(…);
…
free(p);
```

67

malloc()




```
#include <stdlib.h>

int main() {
   int n;
   scanf("%d", &n);

  //int * p = (int *)malloc(n * sizeof(int));
  int * p = (int *)calloc(n , sizeof(int));
  if (p == NULL)
     exit(0);
  *p = 1;          // p[0] = 1
  *(p+1) = 2;      // p+1 = 1004  p[1]= 2
  *(p+2) = 12;     // p+2 = 1008  p[2] = 12;
  ……
  free(p);
  p = ……
```

68

realloc()

```
char *ptr;
ptr = malloc(20);
…
ptr = realloc(ptr,50);
```

- resize a dynamically allocated array.

```
void *realloc(void *ptr, int size);
```

- **ptr** must point to a memory block obtained by a previous call of **malloc**, **calloc**, or **realloc**.
  - **ptr** is NULL, a new block is allocated
- **size** represents the new size of the block, which may be larger or smaller than the original size.

- **realloc(NULL, n)** behaves like **malloc(n)**.
- **realloc (ptr, 0)** behaves like **free(prt),** as it frees the memory block.

For your information

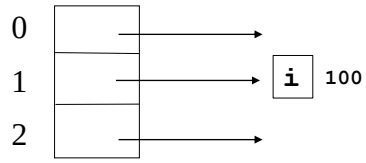
YORK UNIVERSITÉ UNIVERSITY

More on memory allocation

- We know the syntax
- But when to use it ?????
  - When need to allocate at run time, of course
  - What else?
- Another feature of malloc -- request for heap space!

YORK UNIVERSITÉ UNIVERSITY

```c
#include <stdio.h>

void setArr (int);

int * arr[10]; // global, array of 10 int pointers

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

   setArr(1);


   printf("arr [%d] = %d\n", 1, *arr[1]);

   return 0;
}

/* set arr[index], which is a pointer,
to point to an integer of value 100 */
void setArr (int index){

   int i = 100;
   arr[index] = &i;
}
```

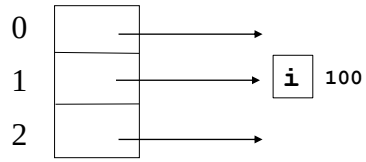


What is wrong here??

71

71

```c
#include <stdio.h>

void setArr (int);

int * arr[10]; // global, array of 10 int pointers

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

   setArr(1);


   printf("arr [%d] = %d\n", 1, *arr[1]);

   return 0;
}

/* set arr[index], which is a pointer,
to point to an integer of value 100 */
void setArr (int index){

   int i = 100;
   arr[index] = &i;
}
```



i is local variable, lifetime is block/function -- i is in stack, where it is deallocated when function exits !!!

72

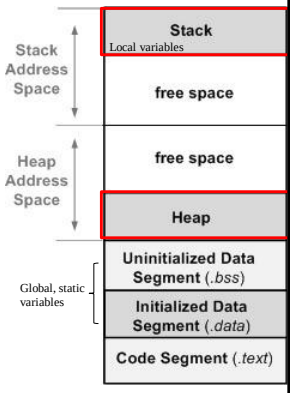
72

When to use malloc ?

- When you need to allocate memory in run time, of course
- When you need memory space throughout the program execution
  1. `ptr = &i.  /* direct */`
     - `i needs to have persistent lifetime`
     - `if i is a local variable in function?` ✖
  2. `ptr = ptr2  /* indirect 1*/`
     - `ptr2 needs to point to persistent address`
     - `if ptr2 points to a local variable?` ✖
  3. `ptr = (..)malloc(....)` — correct choice !
- local variable `i` is in stack. Not in heap.

73

Stack vs. heap



- Local (stack) memory, automatic
  - Allocated on function call, and deallocated automatically when function exits
- Dynamic (heap) memory
  - The heap is an area of memory available to allocate areas ("blocks") of memory for the program.
  - Not deallocated when function exits

What we need!

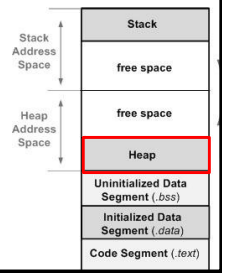
How to allocate in heap then?

Stack vs. heap

- Local (stack) memory, automatic
  - Allocated on function call, and deallocated automatically when function exits
- Dynamic heap memory
  - The heap is an area of memory available to allocate areas ("blocks") of memory for the program.
  - Not deallocated when function exits.

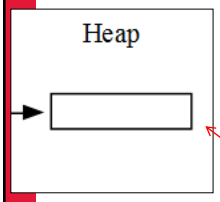
What we need!

- **Request a heap memory:**
  - malloc() / calloc() / realloc() in C
  - new in C++ and Java
    - Student s = new Student();
- **Deallocate from heap memory:**
  - free() in C,
  - delete in C++
  - garbage collection in Java



Correct implementation

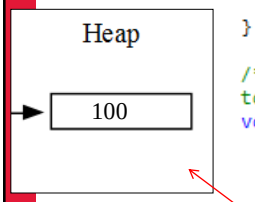
```
#include <stdio.h>

void setArr (int);

int * arr[10]; // global, array of 10 int pointers

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

   setArr(1);


   printf("arr [%d] = %d\n", 1, *arr[1]);      // 100

   return 0;
}

/* set arr[index], which is a pointer,
to point to an integer of value 100 */
void setArr (int index){

   arr[index] = (int *) malloc(sizeof (int)); // malloc(4)


}
```



Correct implementation

```c
#include <stdio.h>

void setArr (int);

int * arr[10]; // global, array of 10 int pointers

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

   setArr(1);


   printf("arr [%d] = %d\n", 1, *arr[1]);      // 100

   return 0;
}

/* set arr[index], which is a pointer,
to point to an integer of value 100 */
void setArr (int index){

   arr[index] = (int *) malloc(sizeof (int)); // malloc(4)

   *arr[index] = 100;
}
```

or `int i=100;  *(arr[index])=i;`



77

77

```c
#include <stdio.h>

int * arr[10]; // array of 10 int pointers, global variable

int main(int argc, char *argv[])
{
     int i;

     int a=0, b=100, c=200,d=300,e=400;
     arr[0] = &a;
     arr[1] = &b;
     arr[2] = &c;
     arr[3] = &d;
     arr[4] = &e;

     for(i=0; i<5;i++)
         printf("arr[%d] -*-> %d\n", i, *arr[i]);  /*  0, 100, 200, 300, 400 */

     return 0;
}
```

This program works.
a,b,c,d,e are local variables, in stack, but not deallocated before program terminates

78

78

- Pointers (Ch5)
  - 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)

today

- **Structures (Ch6)**
  - **Pointer to structures (6.4)**

YORK UNIVERSITÉ UNIVERSITY

EECS2031 – Software Tools

C - Structures, Unions, Enums & Typedef (K+R Ch.6)

YORK UNIVERSITÉ UNIVERSITY

Structures

- A collection of one or more variables grouped under a single name for easy manipulation
- The variables can be of different types
  - Primitive data types, arrays, pointers and other structure
- Encapsulate data

```
int x;
int y;
```

```
float speed;
int directionX;
int directionY;
```

- Only contains data (no functions).

Structures

- Basics: Declaration and assignment
- Structures and functions
- Pointer to structures
- Arrays of structures
- Self-referential structures (e.g., linked list, binary trees)

Structures

```
struct {
  float width;
  float height;
} chair;
```

```
struct {
   float width;
   float height
  }     is the type     //   int a;
```

chair is variable name.

83

Structure Names

- Give a name (tag) to a struct, so we can reuse it:

```
struct shape {
  float width;
  float height;
};
```

`struct shape` is a valid type

```
struct shape chair, chair2;  /* int i, j */
struct shape table;
```

```
shape table;
```

✘

84

Structures
access members, initialization, operations (. = &)

- use the "." operator to access members of a struct

```
chair.width = 10;
table.height = chair.width + 2;
```

Operator Type	Operator		Associativity
Primary Expression Operators	() [] . ->		left-to-right
Unary Operators	* & + - !~ ++ -- (typecast) sizeof		right-to-left
Binary Operators	* / %	arithmetic	left-to-right
	+ -	arithmetic	
	>> <<	bitwise	
	< > <= >=	relational	
	== !=	relational	

Structures
access members, initialization, operations (. = &)

```
struct shape {
    float width;
    float height;
};
struct shape chair = {2,4}; // approach 1
```

width height

```
struct shape chair;
chair.width = 2;
chair.height = 4;
```

approach 2

```
struct myshape {
  int data;
  float arr[3];
};
struct myshape s2 = {2, {1.5, 2.5}}; //approach 1
(s2.arr)[2] = 3.3;     // approach 2   set directly
```

associativity

Size of struct not necessarily the sum of its elements. Use sizeof()

Structures
access members, initialization, operations (. = &)

- use the "." operator to access members of a struct

```
chair.width = 10;
table.height = chair.width + 2;
```

- can also use assignment with struct variables (same type)

```
chair2 = chair; /* valid. But diff from Java! */
                /* copy members value  */
```

- can take address as well

```
&chair
```

Recall: Array cannot assign
arr2 = arr1

No `==` `!=` …

Structures
access members, initialization, operations (. = &)

```
struct shape chair = {2,4};
```

width height

```
struct shape chair2 = chair; // copy members values only
                             // different from Java
```

chair2.width = chair.width
chair2.height = chair.height

```
printf("%d %d", chair.width, chair2.width);
                 2           2
printf("%d %d", chair.height,chair2.height);
                 4           4

chair2.width = 20; // does not affect chair

printf("%d %d", chair.width, chair2.width);
                 2           20
```


? What if an element is a pointer ?

Precedence

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

```
scanf("%f",
    &chair2.width )
    &(chair2.width)
```

```
s2.arr[2] =3
```

No () needed

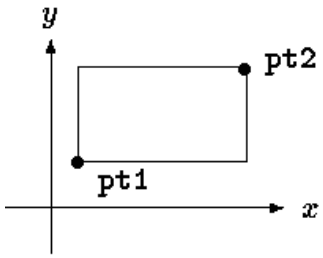
```
(* ptr).width
```

later

89

Nested Structures

```
struct point {
  int x;
  int y; };

struct rect {
  struct point pt1;
  struct point pt2;
};

struct rect screen;
screen.pt1.x = 1;
screen.pt2.x = 8;
(screen.pt2).y = 7;
```



Associativity left to right

Structures vs. Arrays (so far)

- Both are aggregate (non-scalar) types in C -- type of data that can be referenced as a single entity, and yet consists of more than one piece of data.
- Both cannot be compared using == !=

- Array: elements are of same type
  Structure: elements can be of different type

- Array: element accessed by [index/position] `arr[1] = 3;`
  Structure: element accessed by .name `chair.width = 4`

- Array: cannot assign as a whole `arr2 = arr1` ✖
  Structure: can assign/copy as a whole `chair2 = chair1`
  Diff from Java

- Array: size is the sum of size of elements
  Structure: size not necessarily the sum of size of elements

Structures

- Basics: Declaration and assignment
- Structures and functions
- Pointer to structures
- Arrays of structures
- Self-referential structures (e.g., linked list, binary trees)

Structure and functions
-- Structures as arguments

- You can pass structures as arguments to functions

```
float get_area(struct shape d) // shape as argument
{
   return d.width * d.height;
}
```

- This is call-by-value -- a copy of the struct is made
  - **d** is a copy of the actual parameter (copy member values)
  - No starting address, no “decay”

93

93

Structure and functions
--Structures as arguments

- You can pass structures as arguments to functions

```
void do_sth(struct shape d)
{
   d.width  += 100;
   d.height += 200;
}
main()  {
  struct shape  s = {1,2};
  do_sth(s) /*  s  is not modified  */
}
```

call-by-value
`d = s` // copy members
d.width = s.width
d.height = s.height

- This is call-by-value - a copy of the struct is made
  - Function cannot change the passed struct

94

94

structure and functions

-- structures as Return Values

- structs can be used as return values for functions as well

```
struct shape make_dim(int width, int height)
{
  struct shape d;    // in stack
  d.width = width;
  d.height = height;
  return d;
}
main(){
  struct shape myShape = make_dim(3,4);
}
              // myShape = d;
              Copy members, d is gone (deallocated) afterwards
```

95

95

Structures

- Basics: Declaration and assignment
- Structures and functions
- Pointer to structures
- Arrays of structures
- Self-referential structures (e.g., linked list, binary trees)

96

YORK UNIVERSITÉ UNIVERSITY

96

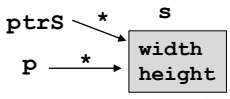
structure and functions

-- Structure Pointers

- call-by-value is inefficient for large structures: not decayed
  - use pointers (explicitly) !!!
- This also allows to change the passing struct

```
main(){
  struct shape s = {1,3};
  struct shape * ptrS = &s; // pointer to struct shape
  float f = get_area(ptrS); // float f = get_area(&s);
}
float get_area(struct shape *p)
{
   return
}
```

Expect a pointer to struct shape



97

97

structure and functions

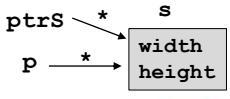
-- Structure Pointers

- call-by-value is inefficient for large structures: not decayed
  - use pointers (explicitly) !!!
- This also allows to change the passing struct

```
main(){
  struct shape s = {1,3};
  struct shape * ptrS = &s; // pointer to struct shape
  float f = get_area(ptrS); // float f = get_area(&s);
}
float get_area(struct shape *p)
{
   return (*p).width   *   (*p).height;
}
```

Expect a pointer to struct shape

Assess member via pointer



98

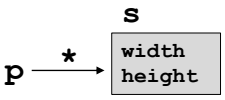
98

structure and functions

-- Structure Pointers

- call-by-value is inefficient for large structures: not decayed
  - use pointers!!!
- This also allows to change the passing struct

```
do_sth(&s);
```




```
void do_sth(struct shape * p)
{
    (*p).width  += 100;
    (*p).height += 200;
}
```

Pointee s is modified !

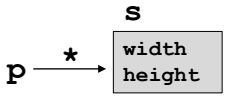
- This is call-by-value --- but address
  - Function can change the passed struct

structure and functions

-- Structure Pointers

Operator Type	Operator
Primary Expression Operators	() [] . ->
Unary Operators	* & + - (typecast)

```
void do_sth(struct shape *p){
    (*p).width += 100;
}
```



- Beware when accessing members a structure via its pointer

  `* p.width` --- incorrect
- Operator . takes higher precedence over operator *

  `(*p).width` --- correct

- Accessing member of a structure via its pointer is so common that it has its own operator

  `p -> width`

structure and functions

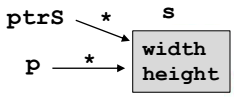
-- Structure Pointers

```
(*p).width
 p -> width
```

} Equivalent

```
main(){
  struct shape s = {1,3};
  struct shape * ptrS = &s;
  do_sth (ptrS); // or do_sth (&s);
}

void do_sth(struct shape *p)
{
    p -> width  += 100;
    p -> height += 200;
}
```

Expect a pointer to struct shape



101

101

Precedence and Associativity p53

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	,	

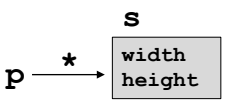
```
x -> data = 2;
x -> data += 2;
```

() never needed!

102

structure and functions

-- Structure Pointers




```
void do_sth(struct shape *p){
    p -> width  += 100;  // (*p).width += 100;
    p -> height += 200;  // (*p).height+= 200;
}
```

. works with structures, accessing members

-> works with structure pointers, accessing members

```
struct shape{
  int width; int height;
} s, *p;
```

`s.width;` valid `s -> width;` invalid

`p.width;` invalid `p -> width;` valid

Pointers to Structures: Shorthand

- `(*pp).x` can be written as `pp -> x`

```
struct rect r, *rp = &r;
    r.pt1.x = 1;

(*rp).pt1.x = 1;      access pt1.x

rp -> pt1.x = 1;
```

```
struct point {
  int x;
  int y; };

struct rect {
  struct point pt1;
  struct point pt2;
};
```

YORK UNIVERSITÉ UNIVERSITY

Pointer to structures -- malloc/calloc

```
struct shape * ptable;  // pointer to struct shape

ptable = malloc (sizeof(struct shape));

ptable -> width = 1.0;  // (* ptable).width  = 1.0
ptable -> height = 5.0; // (* ptable).height = 5.0
```

or

```
ptable =(struct shape *) malloc (sizeof(struct shape));
```

105

When to use? Few slides later

YORK UNIVERSITÉ UNIVERSITY

105

Structures vs. Arrays (so far)

- Both are aggregate (non-scalar) types in C -- type of data that can be referenced as a single entity, and yet consists of more than one piece of data.
- Both cannot be compared using == !=

- Array: elements are of same type
  Structure: elements can be of different type
- Array: element accessed by [index/position] `arr[1] = 3;`
  Structure: element accessed by .name `chair.width = 4`
- Array: cannot assign as a whole `arr2 = arr1` ✖
  Structure: can assign/copy as a whole `chair2 = chair1`
  Diff from Java
- Array: size is the sum of size of elements
  Structure: size not necessarily the sum of size of elements
- Array: decay to pointer when passed to function, can modify
  Structure: need '&' to modify (like scalar types int, char, float etc)

106

106

Structures

- Basics: Declaration and assignment
- Structures and functions
- Pointer to structures
- Arrays of structures
- Self-referential structures (e.g., linked list, binary trees)

Arrays of structures -- declaration

- Structures can be arrayed same as the other variables

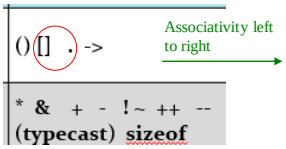
```
struct shape {
   float width;
   float height;
};
```

array of 10 struct

```
struct shape chairs[10];   // int arr[10]
```

Array of structures -- Initialization

```
struct shape chairs[] = {
                        {1.4, 2.0},
                        {0.3, 1.0},
                        {2.3, 2.0} };

struct shape chairs[10]; //chairs[n] is a struc.
chairs[0].height = 1.4;
(chairs[0]).width =  2.0;
.....
float x = chairs[3].height;
```




```
struct shape * chairsA[10];
```

what is chairsA

109

109

Structures

- Basics: Declaration and assignment
- Structures and functions
- Pointer to structures
- Arrays of structures
- Self-referential structures (e.g., linked list, binary trees)

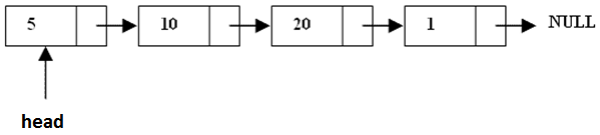
110

YORK UNIVERSITÉ UNIVERSITY

110

Self-referential structures

- Linked list, trees
- Linked list
  - alternative to Array
  - more flexible than array – can easily insert, delete
  - lost the *O(1)* access in Array, as not stored sequentially. Have to follow the link. Farther ones cost more than closer ones
- Simplest example: a linked list of int's



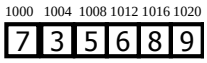
111

111

List

- Array based list

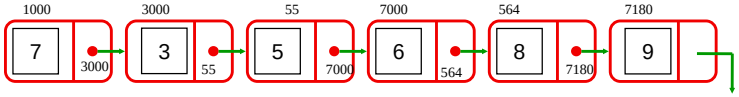
arr[i] = *(arr+i)



O(1) access

arr[3]? Content at 1000+3*4

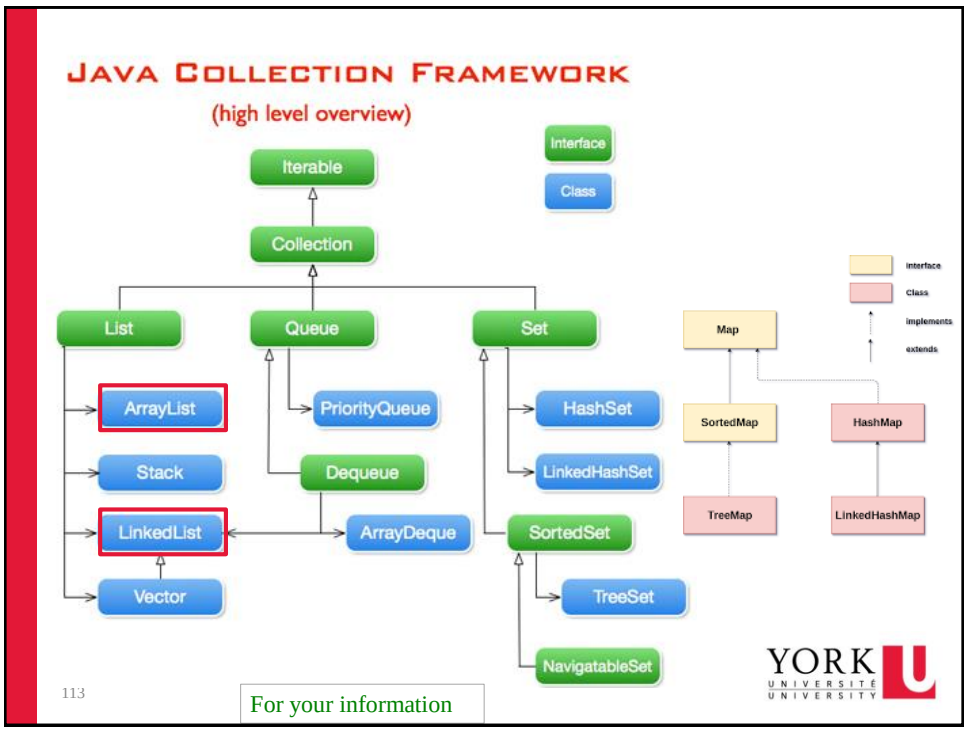
- Linked-based list



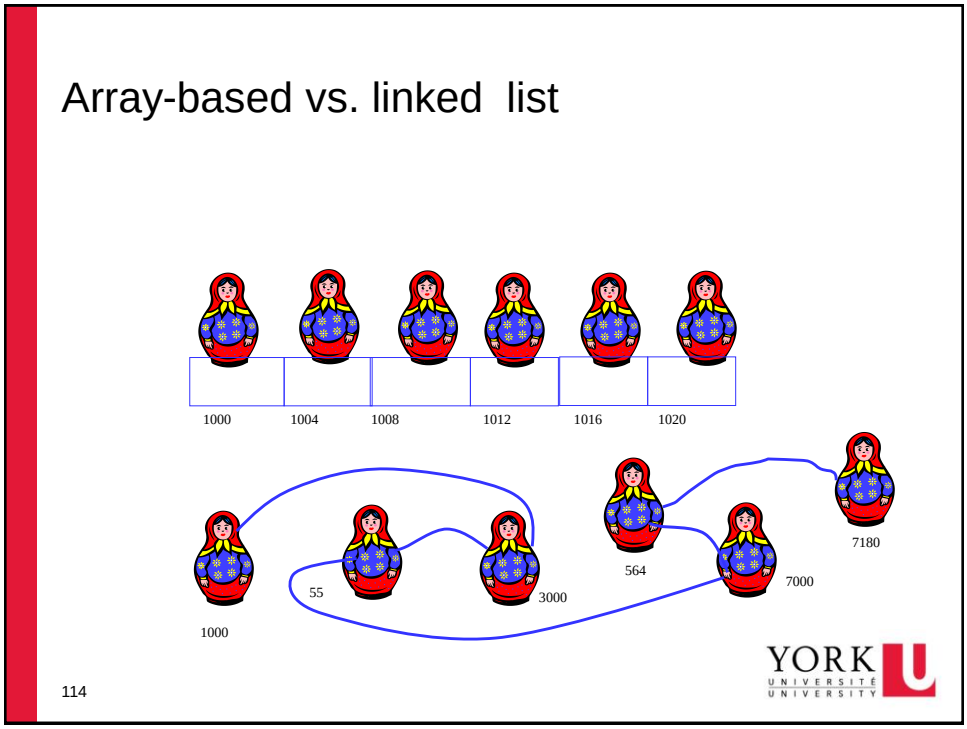
O(n) access

get(3)?

112

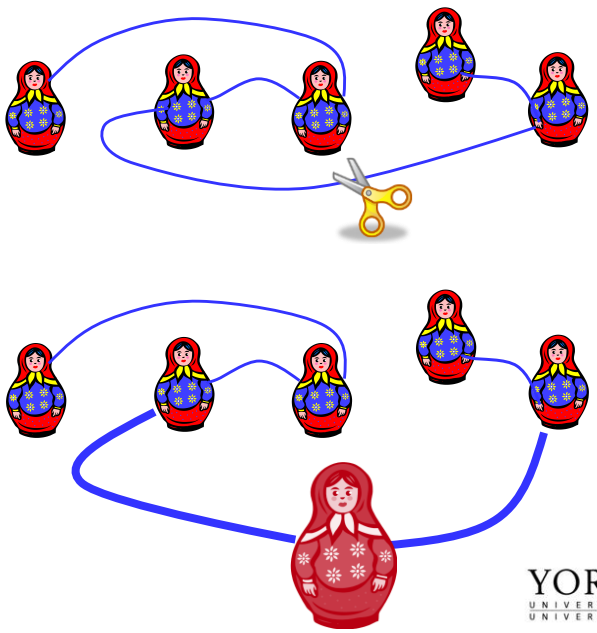

JAVA COLLECTION FRAMEWORK
(high level overview)
Interface
Class
Iterable
Collection
List
Queue
Set
ArrayList
Stack
LinkedList
Vector
PriorityQueue
Dequeue
ArrayDeque
HashSet
LinkedHashSet
SortedSet
TreeSet
NavigatableSet
Map
SortedMap
HashMap
TreeMap
LinkedHashMap
interface
Class
implements
extends
113
For your information
YORK UNIVERSITÉ UNIVERSITY

113


Array-based vs. linked list
1000
1004
1008
1012
1016
1020
1000
55
3000
564
7000
7180
114
YORK UNIVERSITÉ UNIVERSITY

114

Insertion



115

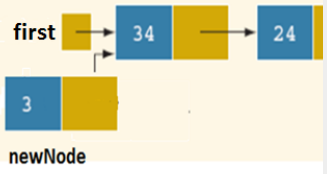
115

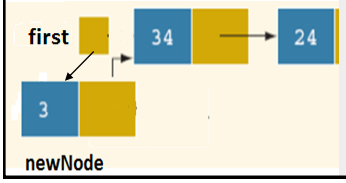
```
class Node {
    public int data1;
    public double data2;
    public Node nextLink;

    //Link constructor
    public Node(int d1, double d2) {
            data1 = d1;
            data2 = d2;
    }
```

How to implement in Java?

```
class LinkList {
    private Node first;

    //LinkList constructor
    public LinkList() {
            first = null;
    }

    //Returns true if list is empty
    public boolean isEmpty() {
            return first == null;
    }

    //Inserts a new Link at the first of the list
    public void insert(int d1, double d2) {
            Node newN = new Node(d1, d2);
            newN.nextLink = first;
            first = newN;
    }
```

Order matters!



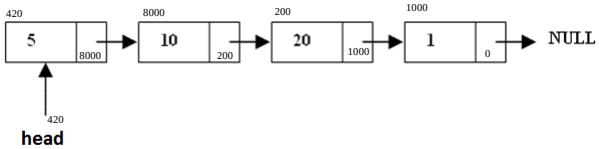


116

Self referential structures in C

- Simplest example: a linked list of integers

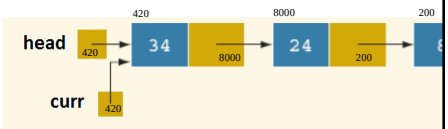
```
struct node {
  int data;
  struct node *next; //pointer to struct node
};

struct node * head;  // a pointer to first node
```

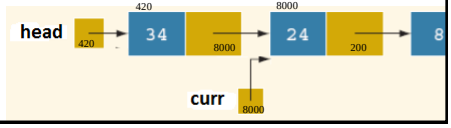


traverse the list example 1

```
struct node * head;  // assume global

//whether the list contains a node with data 'dat'
int has_value(int dat)
{
  struct node * curr;     // a local pointer

  /* traverse the list */
  curr = head;
  while (curr != NULL){
    if ( curr -> data == dat )
        return 1; // find it!
    curr = curr -> next;  // curr = (*curr).next  8000
  }  //pointer assignment
  return 0;
}
```

(*curr).data

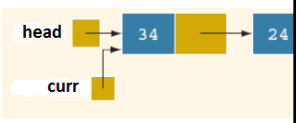


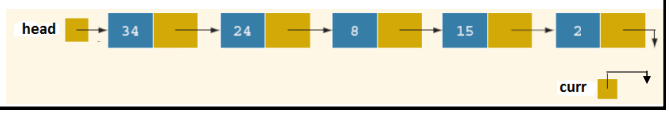


traverse the list example 1

for loop

```
struct node * head;

int has_value(int dat)
{
  struct node * curr;       // a local pointer

 /* traverse the list */
  for(curr = head; curr!=NULL; curr=curr -> next)
  {   if (curr -> data == dat)
        return 1;    /* find it!  */
  }
  return 0;
}
```



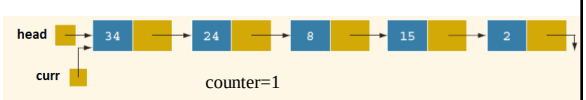


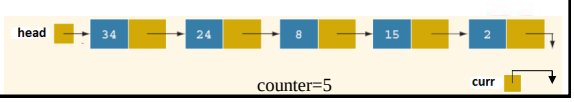
119

119

traverse the list example 2

```
struct node * head;

// # of node in the list
int len()
{
  struct node * curr = head;   // a local pointer
  int counter = 0;
  /* traverse the list */
  while(curr != NULL){
      counter ++;
      curr=curr -> next; // curr = (*curr).next
  }
  return counter;
}
```





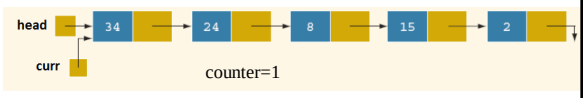
120

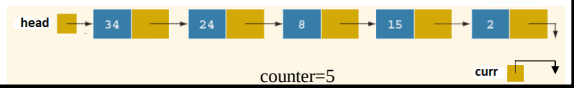
120

traverse the list example 2

for loop

```
struct node * head;

int len()
{
  struct node * curr;   // a local pointer
  int counter = 0;
 /* traverse the list */
  for(curr = head; curr!=NULL; curr=curr -> next)
     counter ++;
  return counter;
}
```





121

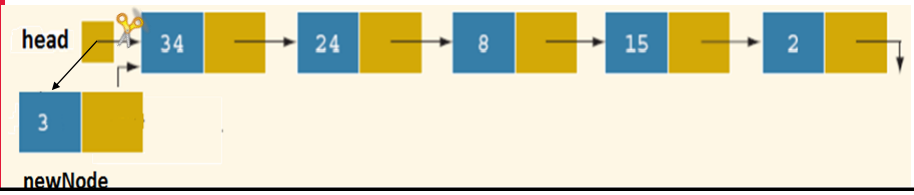
121

Insert into the list example1

```
public void insert(int d1, double d2)
        Node newN = new Node(d1, d2);
        newN.nextLink = first;
        first = newN;
```

```
struct node * head;

void  insert_begining(int dat)
{
   struct node newNode;

   newNode.data = dat;
   newNode.next = head;

   head = &newNode;
}
```

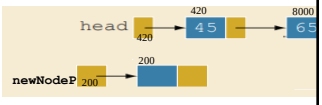
newNode
is in stack!



122

Insert into the list example1

```
struct node * head;

void  insert_begining(int dat)
{
   struct node * newNodeP;
   newNodeP = malloc(sizeof(struct node));
```



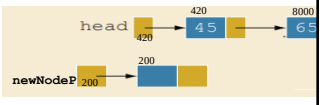
request space in heap !!!

```
   .
   .
```

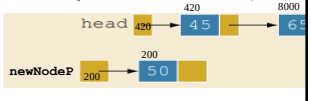
123

Insert into the list example1

```
struct node * head;

void  insert_begining(int dat)
{
   struct node * newNodeP;
   newNodeP = malloc(sizeof(struct node));

   newNodeP -> data = dat;// (*newNodeP).data = dat;
```



request space in heap !!!



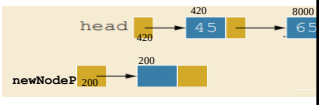

```
   .
   .
```

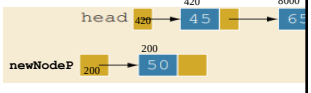
124

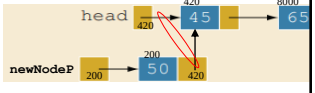
Insert into the list example1

```
struct node * head;

void  insert_begining(int dat)
{
  struct node * newNodeP;
  newNodeP = malloc(sizeof(struct node));

  newNodeP -> data = dat;// (*newNodeP).data = dat;


  newNodeP -> next = head;//(*newNodeP).next = head
```

request space in heap !!!







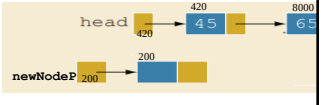
125

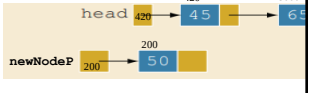
Insert into the list example1

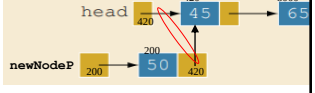
```
struct node * head;

void  insert_begining(int dat)
{
  struct node * newNodeP;
  newNodeP = malloc(sizeof(struct node));

  newNodeP -> data = dat;// (*newNodeP).data = dat;


  newNodeP -> next = head;//(*newNodeP).next = head


  head = newNodeP;
}
```

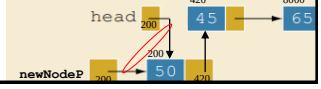
126

request space in heap !!!



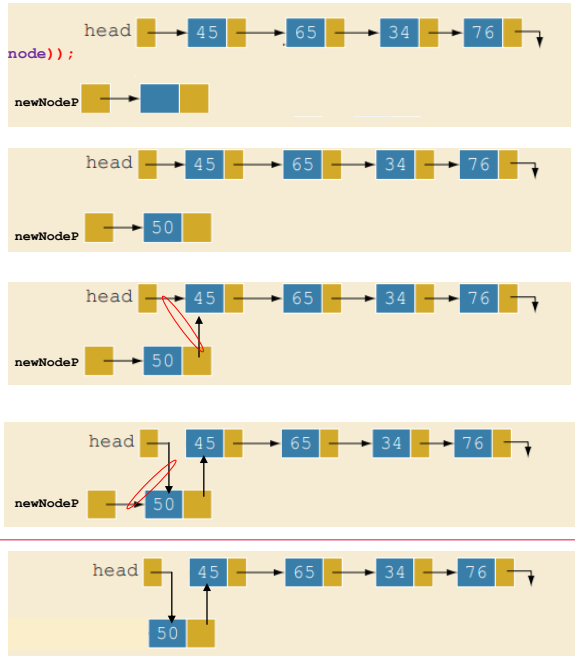






126

```
void  insert_begining(50)

struct node * newNodeP;
newNodeP = malloc(sizeof(struct node));

newNodeP -> data = dat;

newNodeP -> next = head;

Order matters!

head = newNodeP;

After function returns
```

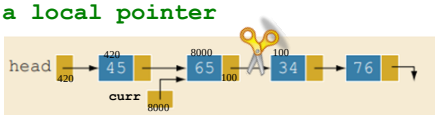


127

127

Insert into the list example2

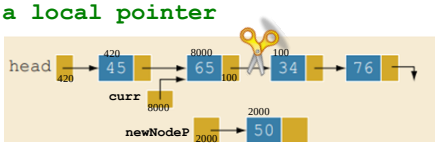
```
insertAfter(1,50);

struct node * head;
// insert a new node with data 'dat' after the node of position 'index'
int insertAfter(int index, int dat) // assume list is not empty
{
   struct node * curr = head;  // a local pointer
   int i;

  /* traverse the list */
  for(i = 0; i<index; i++)
    curr = curr -> next;

  /* insert after curr */
```



128

Insert into the list example2

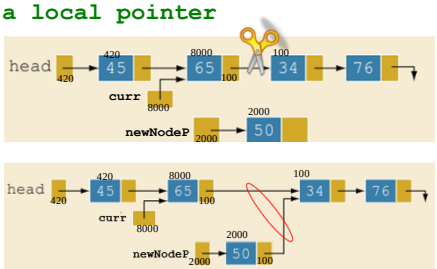
```
insertAfter(1,50);
struct node * head;
// insert a new node with data 'dat' after the node of position 'index'
int insertAfter(int index, int dat) // assume list is not empty
{
   struct node * curr = head;  // a local pointer
   int i;

  /* traverse the list */
  for(i = 0; i<index; i++)
    curr = curr -> next;

  /* insert after curr */
  struct node * newNodeP = malloc(sizeof(struct node));
  newNodeP -> data = dat;  //   (*newNodeP).data = dat;
```



129

Insert into the list example2

```
insertAfter(1,50);
struct node * head;
// insert a new node with data 'dat' after the node of position 'index'
int insertAfter(int index, int dat) // assume list is not empty
{
   struct node * curr = head;  // a local pointer
   int i;

  /* traverse the list */
  for(i = 0; i<index; i++)
    curr = curr -> next;

  /* insert after curr */
  struct node * newNodeP = malloc(sizeof(struct node));
  newNodeP -> data = dat;  //   (*newNodeP).data = dat;

  newNodeP -> next = curr -> next; //(*newNodeP).next=(*curr).next;
```

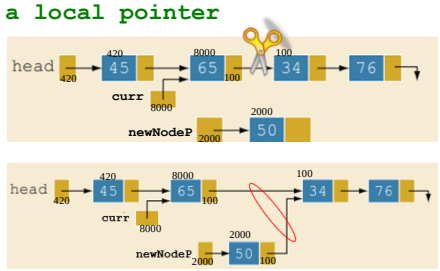


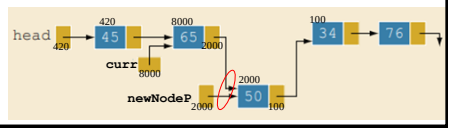
130

Insert into the list example2

`insertAfter(1,50);`

```
struct node * head;
// insert a new node with data 'dat' after the node of position 'index'
int insertAfter(int index, int dat) // assume list is not empty
{
   struct node * curr = head;  // a local pointer
   int i;

  /* traverse the list */
  for(i = 0; i<index; i++)
    curr = curr -> next;

  /* insert after curr */
  struct node * newNodeP = malloc(sizeof(struct node));
  newNodeP -> data = dat;  //   (*newNodeP).data = dat;

  newNodeP -> next = curr -> next; //(*newNodeP).next=(*curr).next;
  curr -> next = newNodeP;

} // if list empty, need to
   change head
```

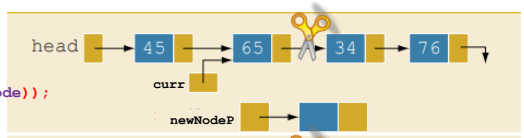




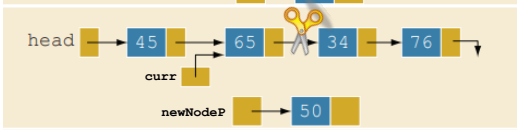
131

int insertAfter(1, 50)

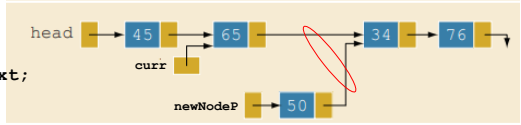
```
struct node * newNodeP;
newNodeP = malloc(sizeof(struct node));
```




```
newNodeP -> data = dat;
```

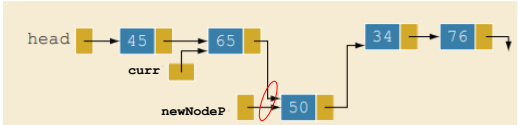



```
newNodeP -> next = curr -> next;
```

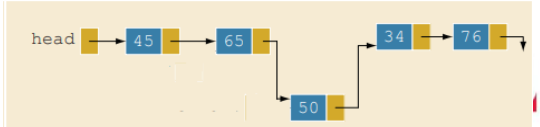


Order matters!

```
curr -> next = newNodeP;
```



After function returns



132

132

EECS2031 - Software Tools

C - Input/Output (K+R Ch. 7)

skipped

YORK UNIVERSITÉ UNIVERSITY

133

EECS2031 - Software Tools

C - System Calls (K+R Ch. 8)

skipped

YORK UNIVERSITÉ UNIVERSITY

134

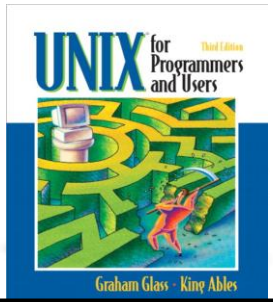
Topics that we did not get to cover
-- might be useful in your future studies

- const
- Union, enum, typedef
- Pointer to whole arrays, int (* arr) [] [][] decayed to
- Pointer to functions
- Stream IO Ch7 e.g., read/write disk files
- System calls Ch 8 (fork, pipe … read, write)
  - You will deal with them if you take EECS3221 Operating Systems.
- Others
  - Make file **make**
  - gdb and testing

YORK UNIVERSITÉ UNIVERSITY

135

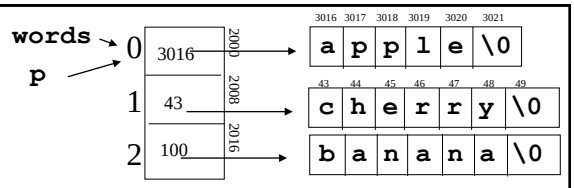
- That's all for C for this course
- Now we have to start a new book, a new programming language …..
- Let's do it now!



136

136

In class quiz




```
#include<stdio.h>

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

  printf("%p %p\n", words, p);     //     ?    2000     2000
  printf("%p %p\n", words+1, p+1);  //   ?    2008     2008
  printf("%p %p\n", words+2, p+2);  //   ?    2016     2016
  printf("%p %p\n", *(words+1),*(p+1) ); //?  bonus 43 43
  printf("%p\n", *(p+1)+5); //? bonus  48
}
  sizeof words?    3*8 = 24
  sizeof p?        8
  total memory allocated after statement
 char * words[]={"apple", "cherry", "banana"};
 (for words and pointees)?    24+6+7+7=44
```

137