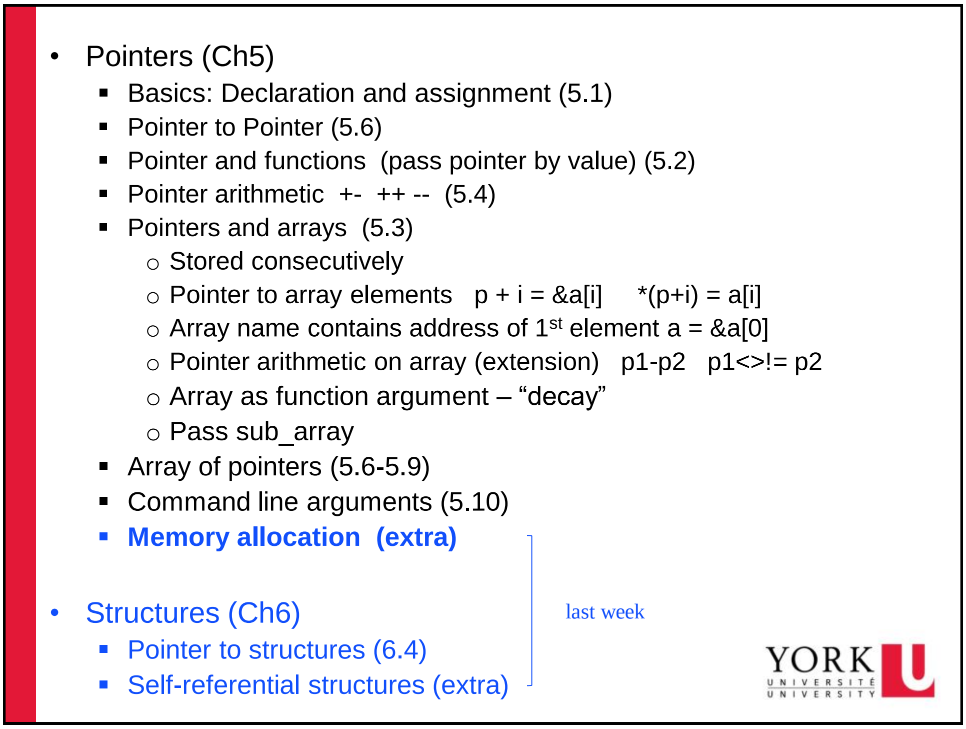
EECS2031 Software Tools

SU 2019

Jul 15, 2019 Lecture 9

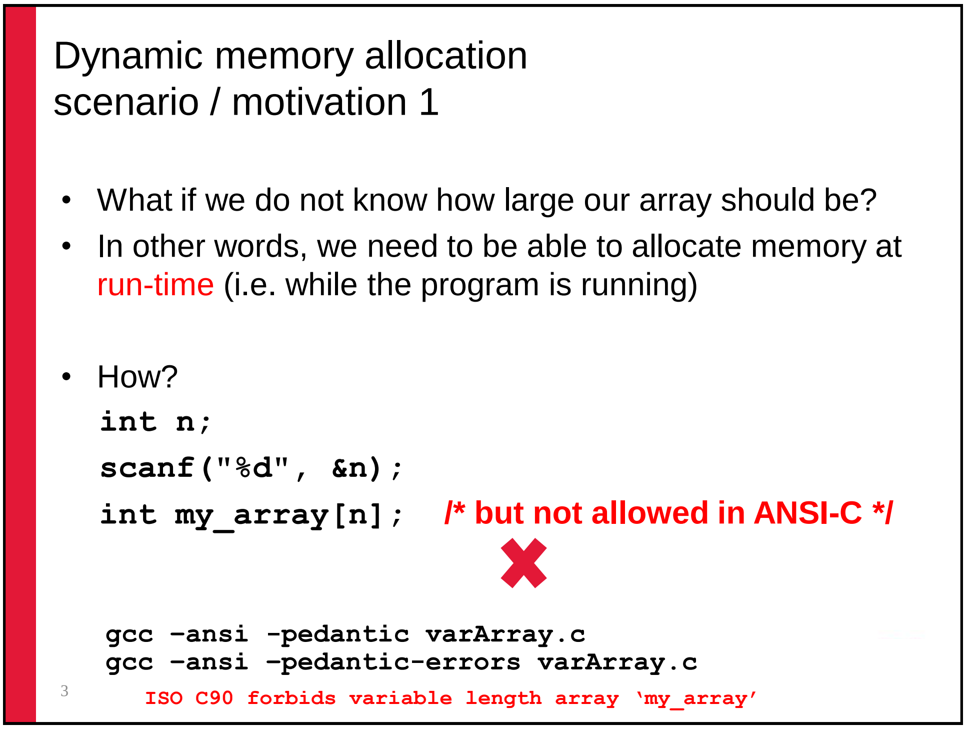
1



- 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)**
- Structures (Ch6)
  - Pointer to structures (6.4)
  - Self-referential structures (extra)

last week

2



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?

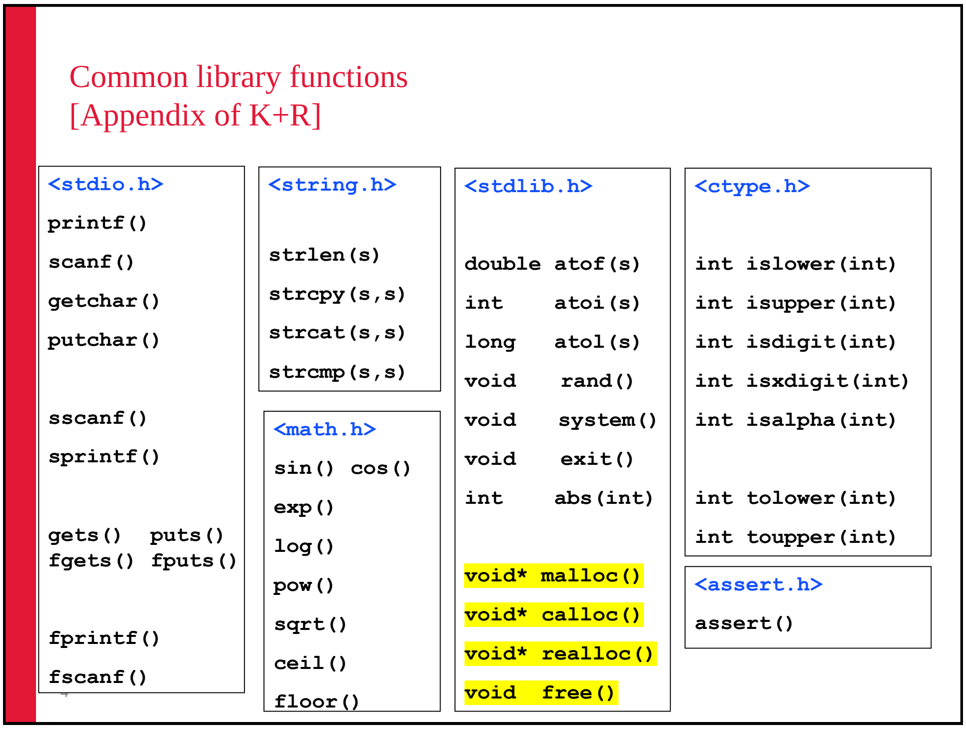
```
int n;
scanf("%d", &n);
int my_array[n];    /* but not allowed in ANSI-C */
```

✖

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

3

3



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()
```

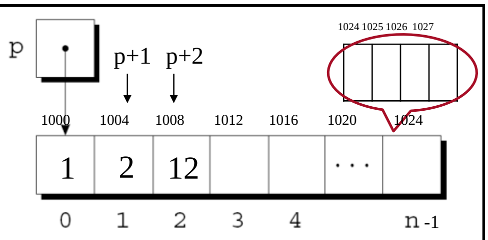
```
<stdlib.h>

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()
```

4

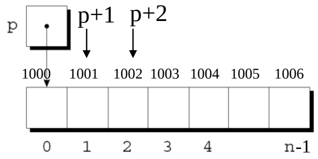
malloc()



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

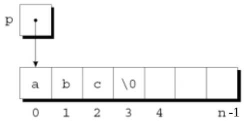
```
#include <stdlib.h>

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

  p = (int *)malloc(n * sizeof(int)); //or
  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
}                     pointer arithmetic!!!
• free (p);
```

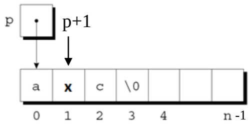
malloc()



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

```
#include <stdlib.h>

int main() {
  int n; char *p;
  printf("Size of array: ");
  scanf("%d", &n);
  p = (char *)malloc(n * sizeof(char)); //or
  p = (char *)calloc(n , sizeof(char));

  if (p == NULL)
     exit(0);

  strcpy(p, "abc");

  *(p+1) = 'x';

}
```

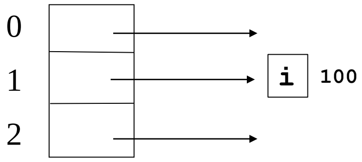




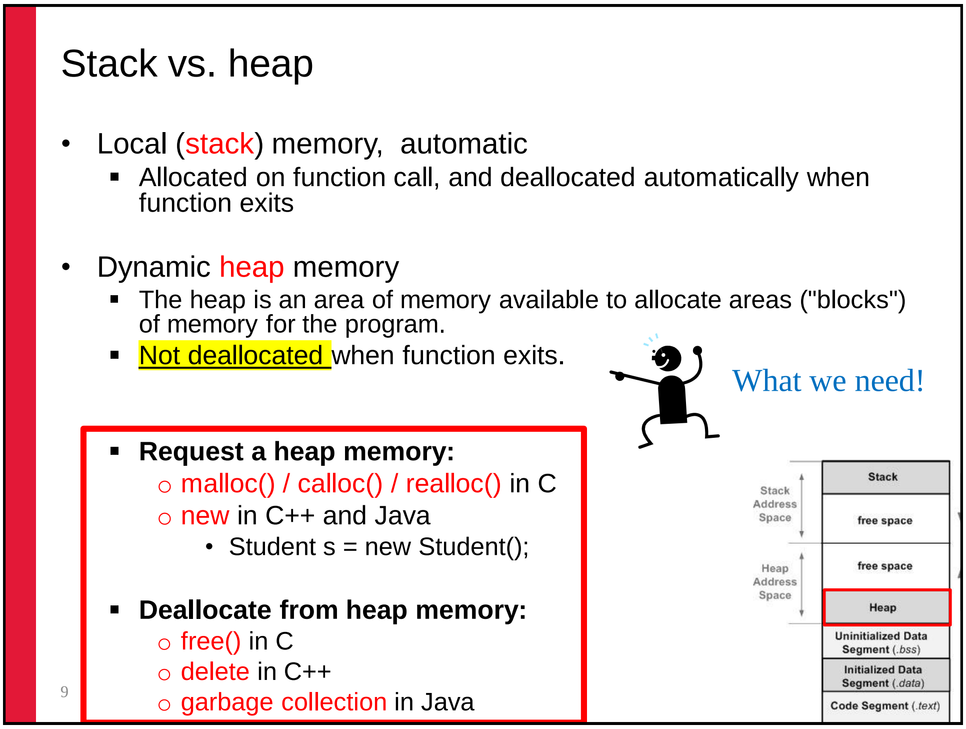
More on memory allocation

- 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!

```
#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 !!!



Stack vs. heap

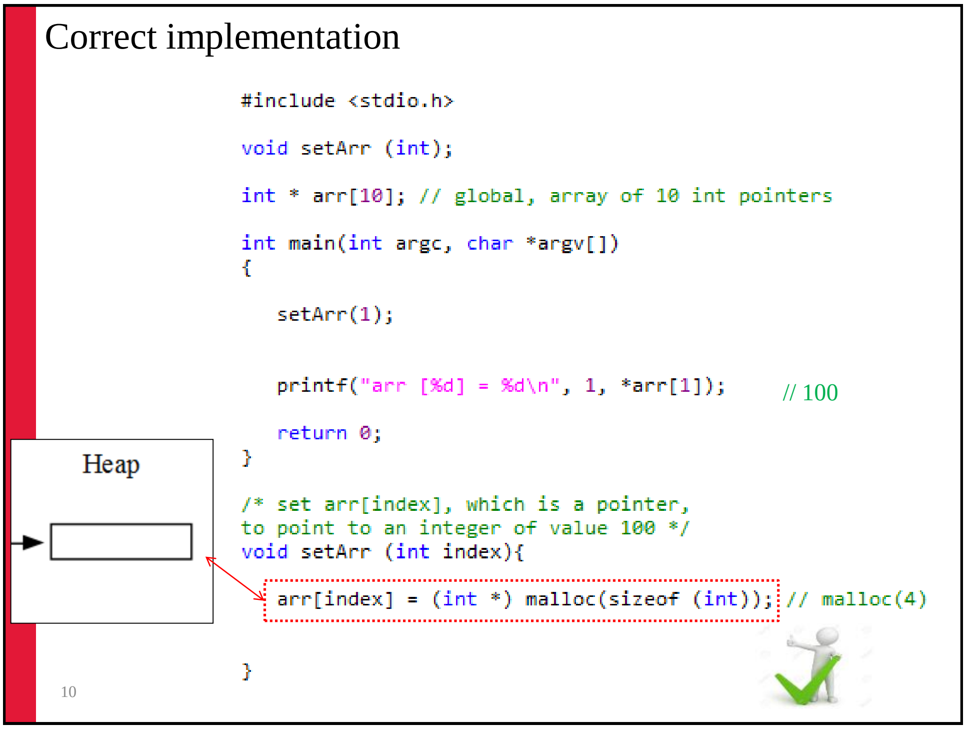
- Local (stack) memory, automatic
  - Allocated on function call, and deallocated automatically when function exits

- 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();

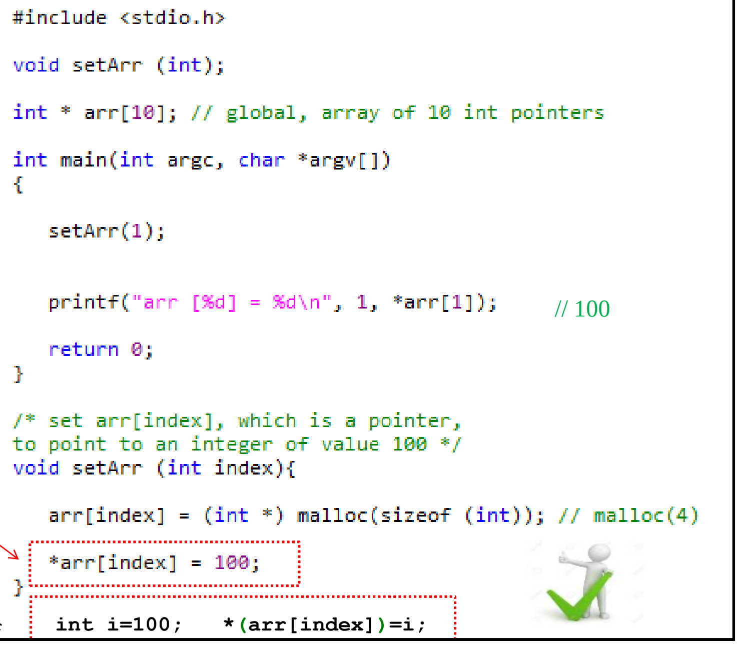
- **Deallocate from heap memory:**
  - free() in C
  - delete in C++
  - garbage collection in Java



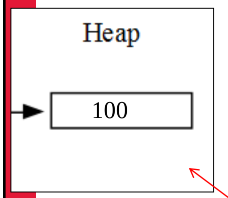
Correct implementation

```
#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



```
#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;
}
```



11

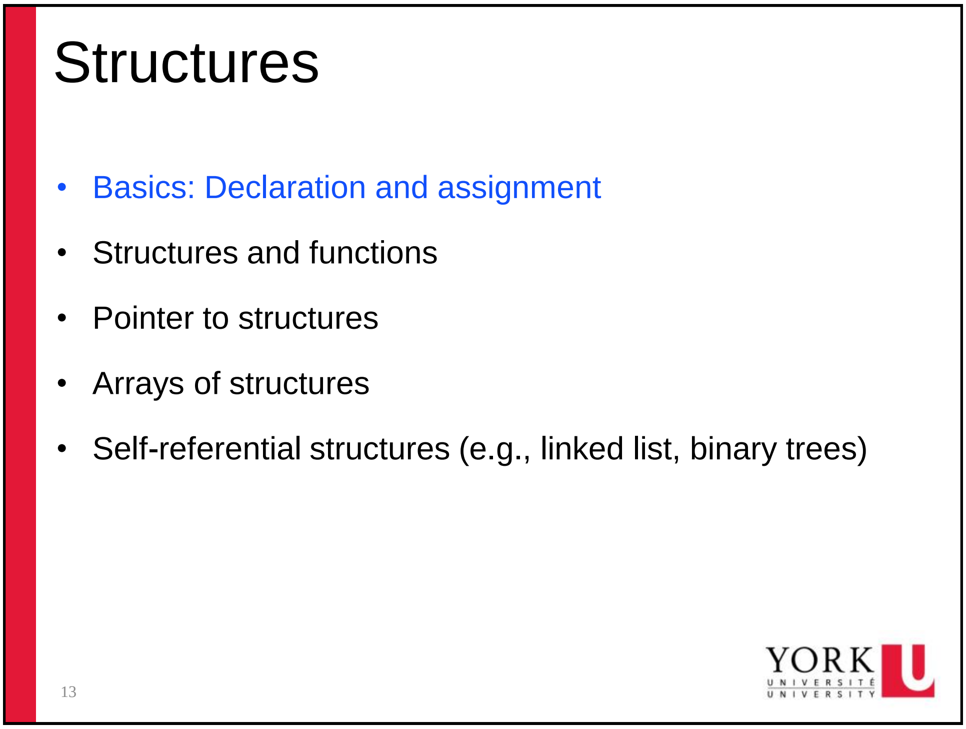
or

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

- 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)

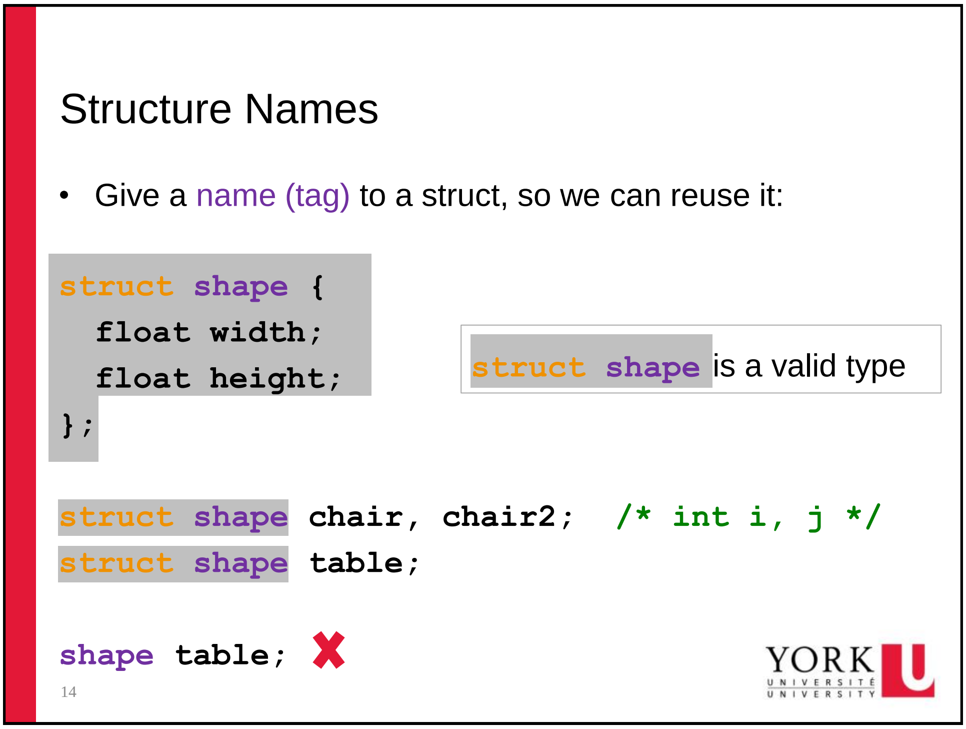
- **Structures (Ch6)**
  - **Pointer to structures (6.4)**
  - **Self-referential structures (extra)**

last week



Structures

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



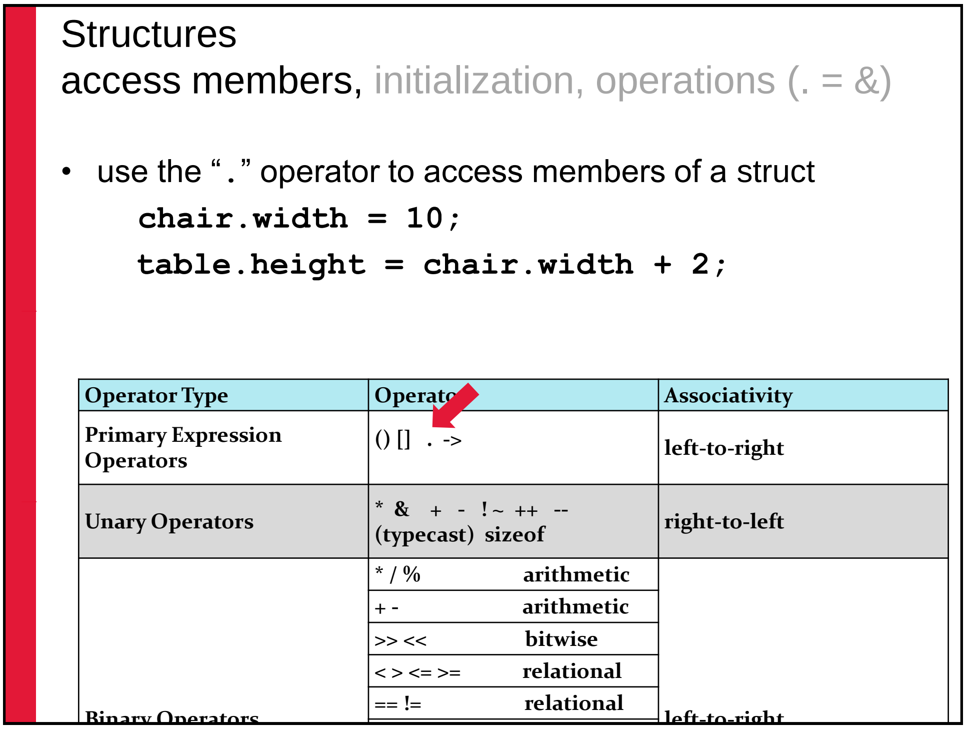
Structure Names

- 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;  ✘
```

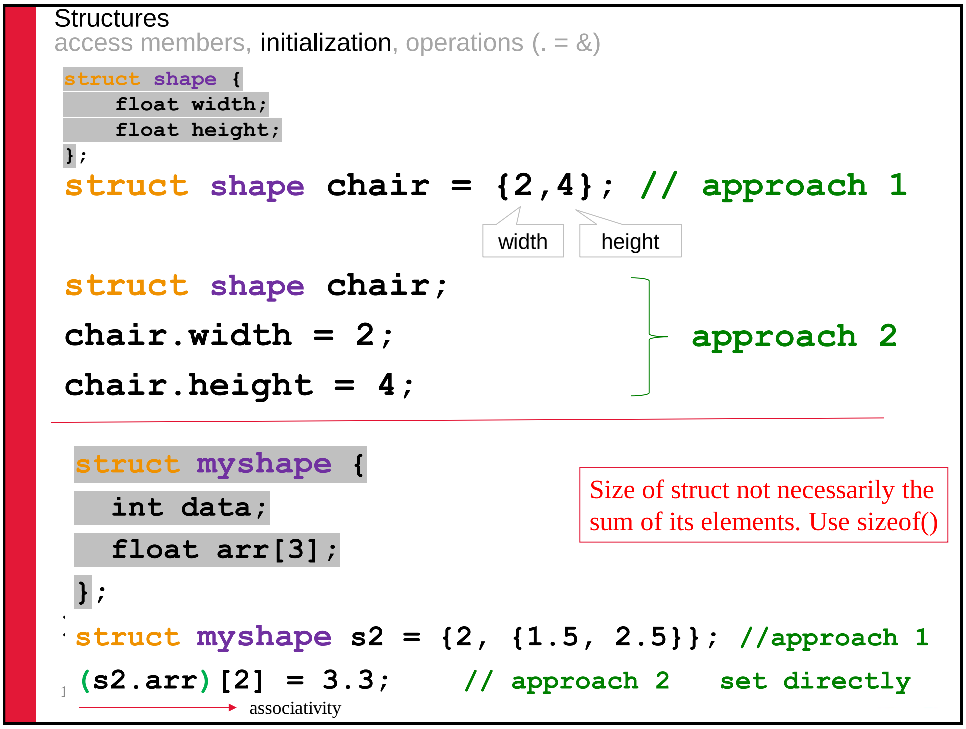


Structures
access members, initialization, operations (. = &)

- 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 + - arithmetic >> << bitwise < > <= >= relational == != relational	left-to-right



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

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

width
height

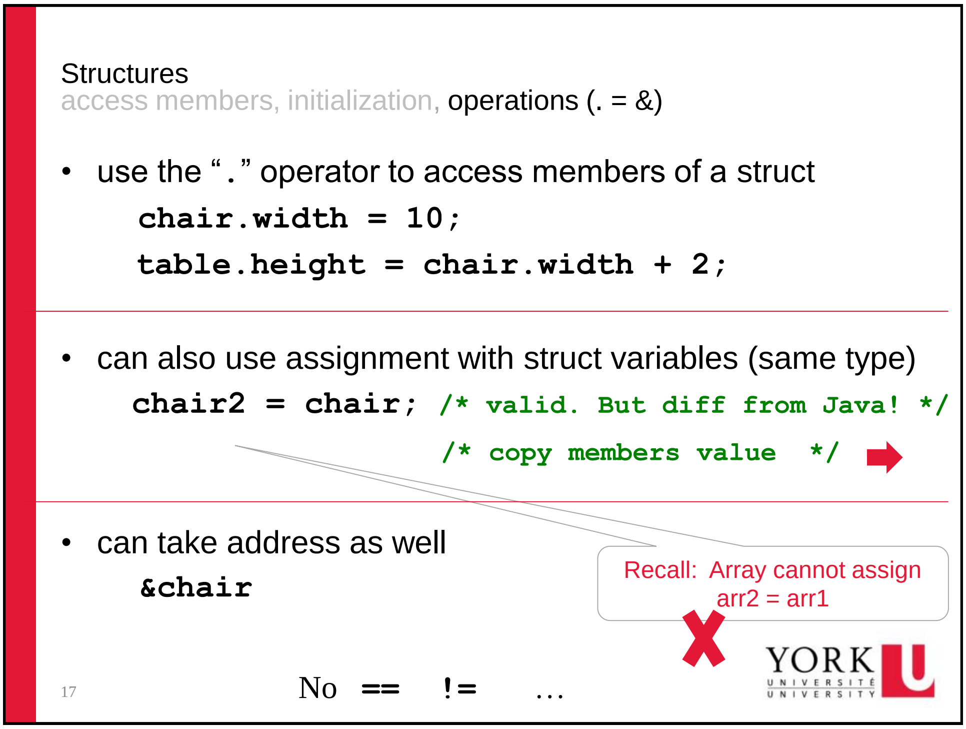
```
struct shape chair;
chair.width = 2;
chair.height = 4;
```

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)

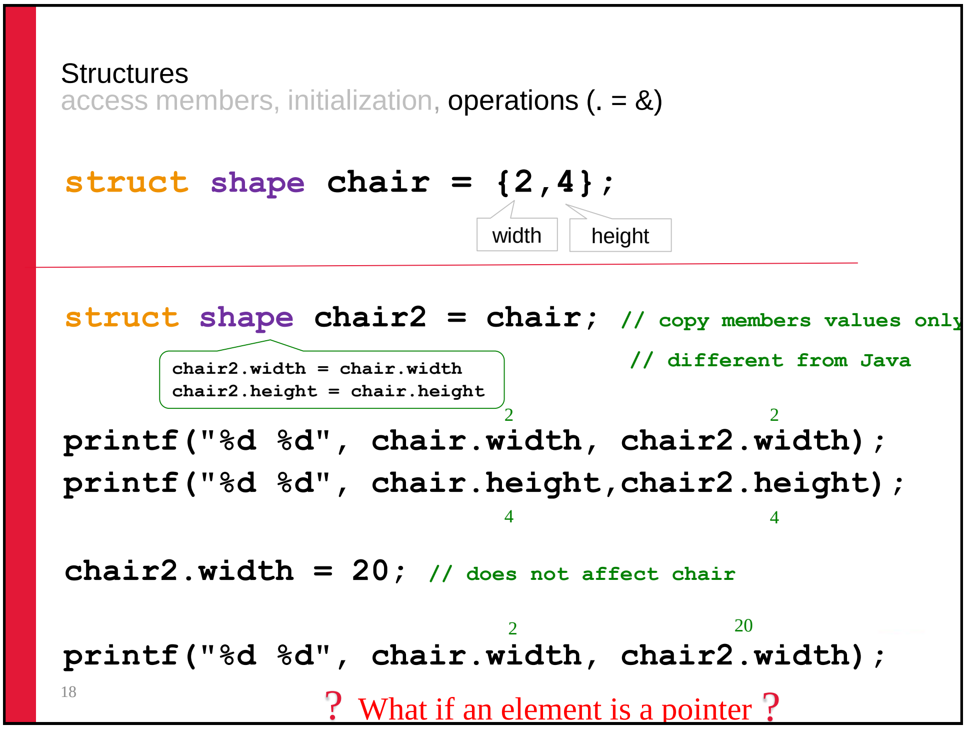
```
chair2 = chair; /* valid. But diff from Java! */
                /* copy members value  */
```

- 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
```

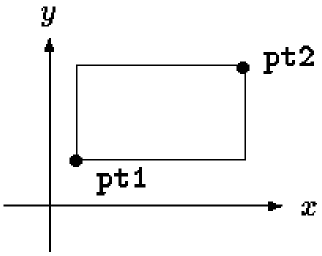
```
chair2.width = chair.width
chair2.height = chair.height
```

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

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

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

? What if an element is a pointer ?

Nested Structures

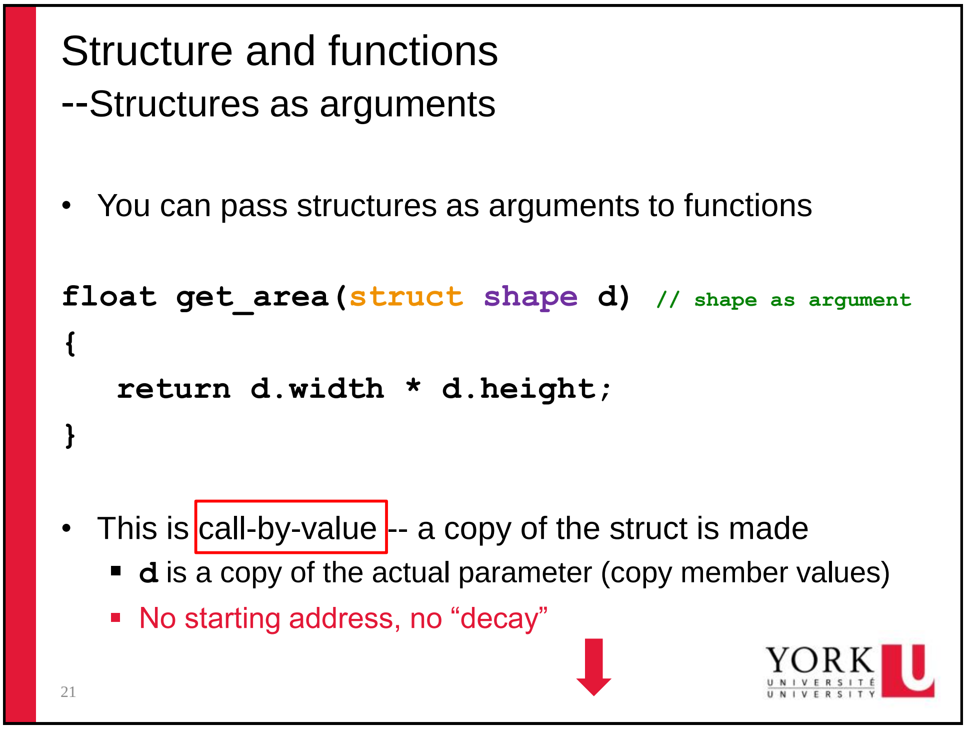
```
struct point {
  int x;
  int y; };

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

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

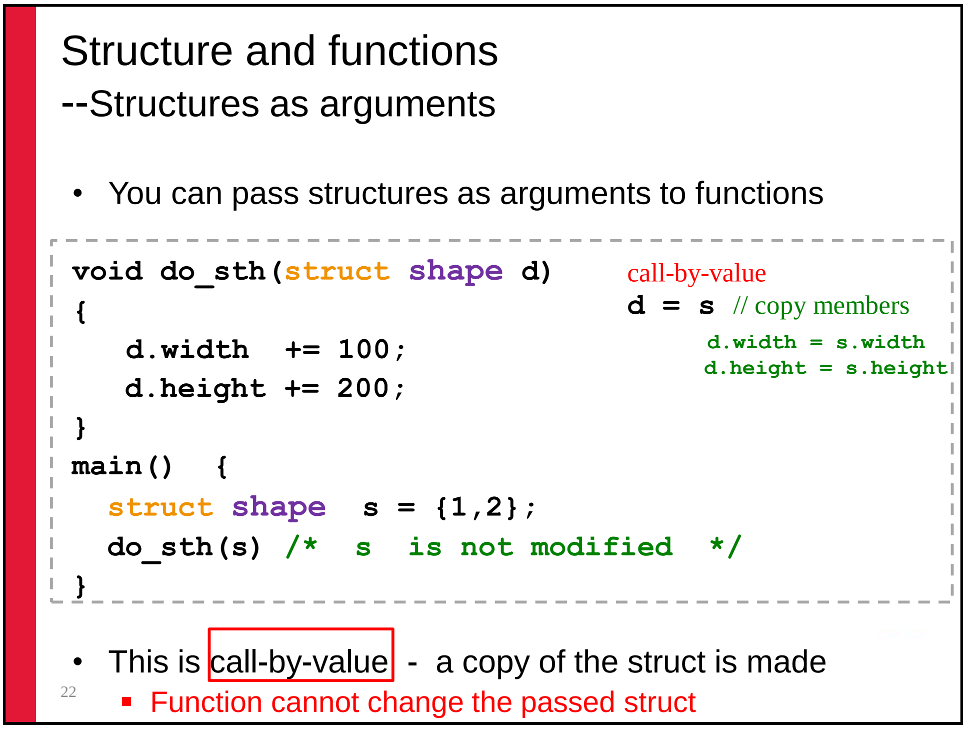


Structure and functions
--Structures as arguments

- 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”

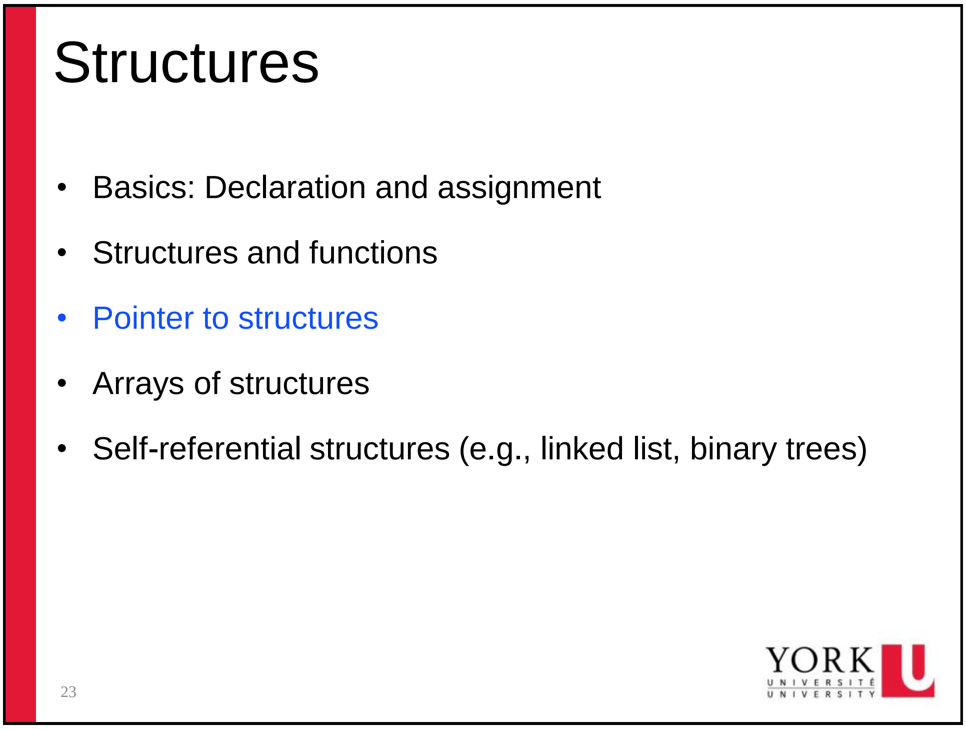


Structure and functions
--Structures as arguments

- You can pass structures as arguments to functions

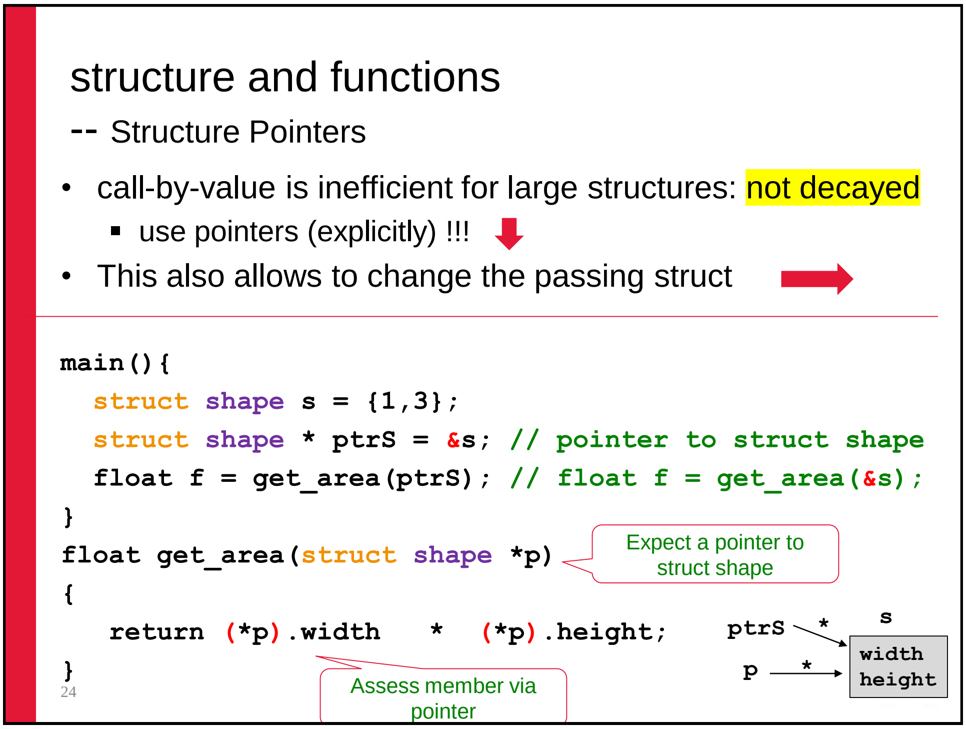
```
void do_sth(struct shape d)        call-by-value
{                                  d = s  // copy members
   d.width  += 100;                      d.width  = s.width
   d.height += 200;                      d.height = s.height
}
main()  {
  struct shape  s = {1,2};
  do_sth(s) /*  s  is not modified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

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



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

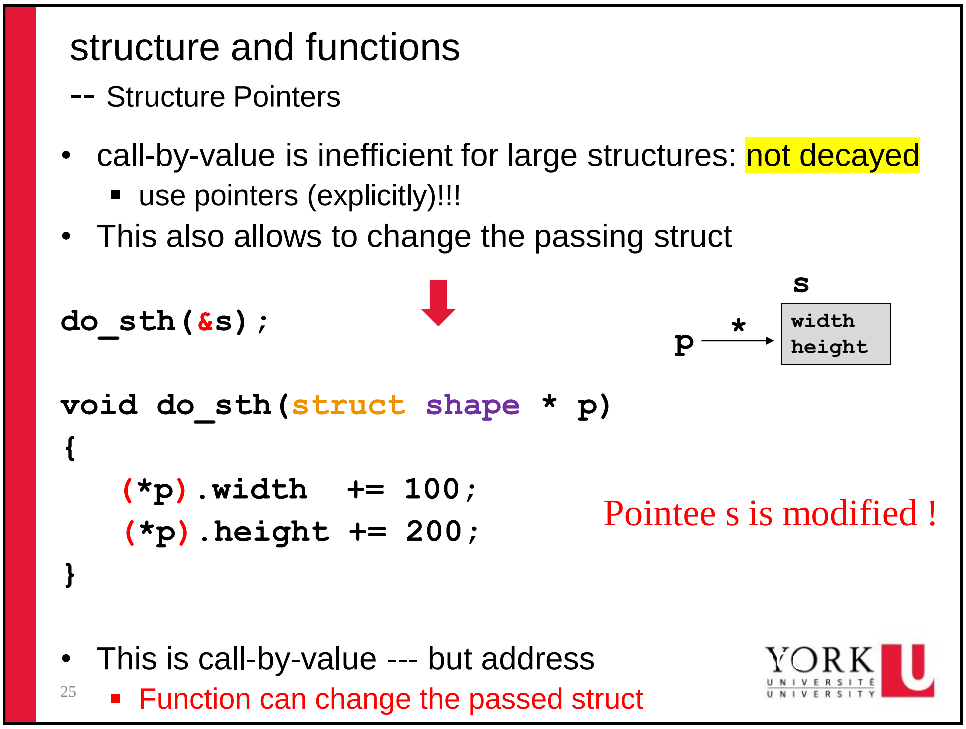
-- Structure Pointers

- 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



structure and functions

-- Structure Pointers

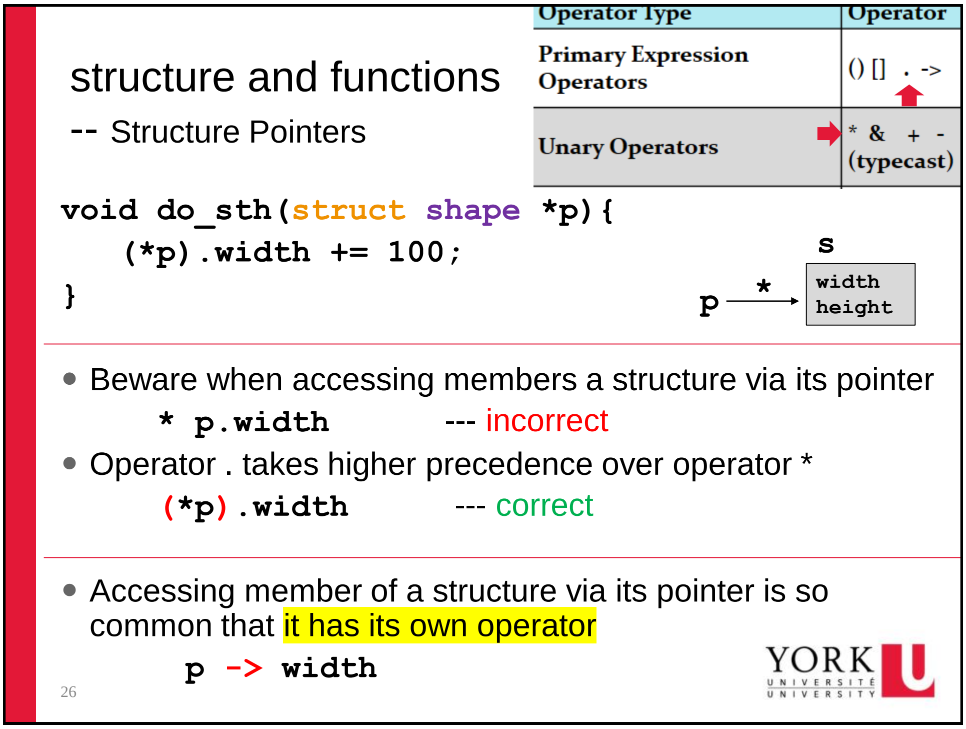
- call-by-value is inefficient for large structures: not decayed
  - use pointers (explicitly)!!!
- This also allows to change the passing struct

```
do_sth(&s);

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

s
p
*
width
height

Pointee s is modified !

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



structure and functions

-- Structure Pointers

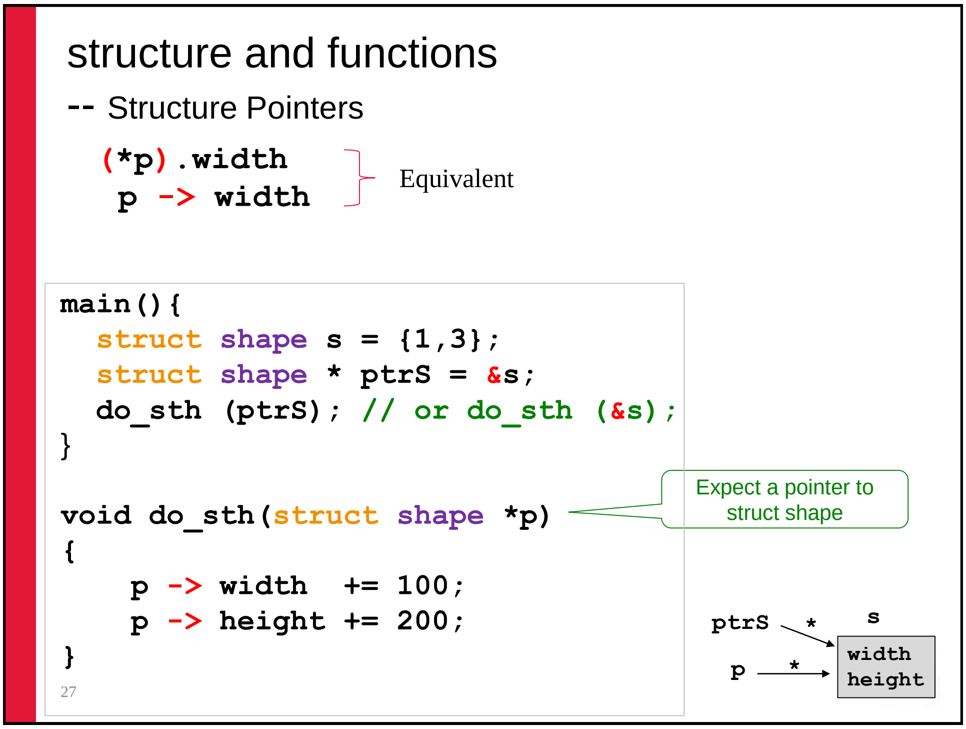
Operator Type	Operator
Primary Expression Operators	() [] . ->
Unary Operators	* & + - (typecast)

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

s
p
*
width
height

- 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

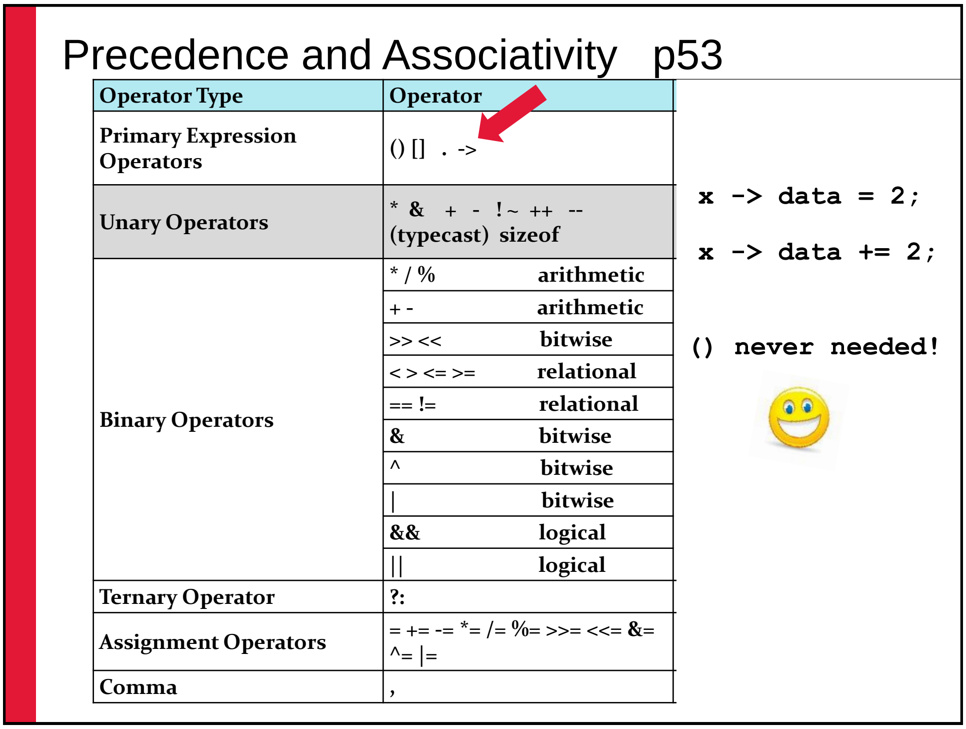
```
main(){
  struct shape s = {1,3};
  struct shape * ptrS = &s;
  do_sth (ptrS); // or do_sth (&s);
}

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

27

Expect a pointer to struct shape

ptrS
*
s
p
*
width
height

27



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!`

28

Pointer to structures -- malloc/calloc

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

ptable = malloc (sizeof(struct shape));

// set struct members
ptable -> width = 1.0;   // (* ptable).width  = 1.0
ptable -> height = 5.0;  // (* ptable).height = 5.0


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

When to use? Few slides later

Structures vs. Arrays

- 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 == != ✖

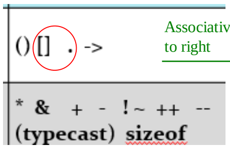
- Array: elements are of same type
  Structure: elements can be of different type
- Array: element accessed by [index/position] `arr[1] = 3;`
  Structure: element accessed by .name `chair.width = 4`
- Array: cannot assign as a whole `arr2 = arr1` ✖
  Structure: can assign/copy as a whole `chair2 = chair1`
  Different 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)

Structures

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

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;
```



() [] . ->

* & + - ! ~ ++ --
(typecast) sizeof

Associativity left to right

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

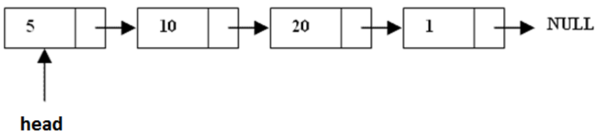
what is chairsA

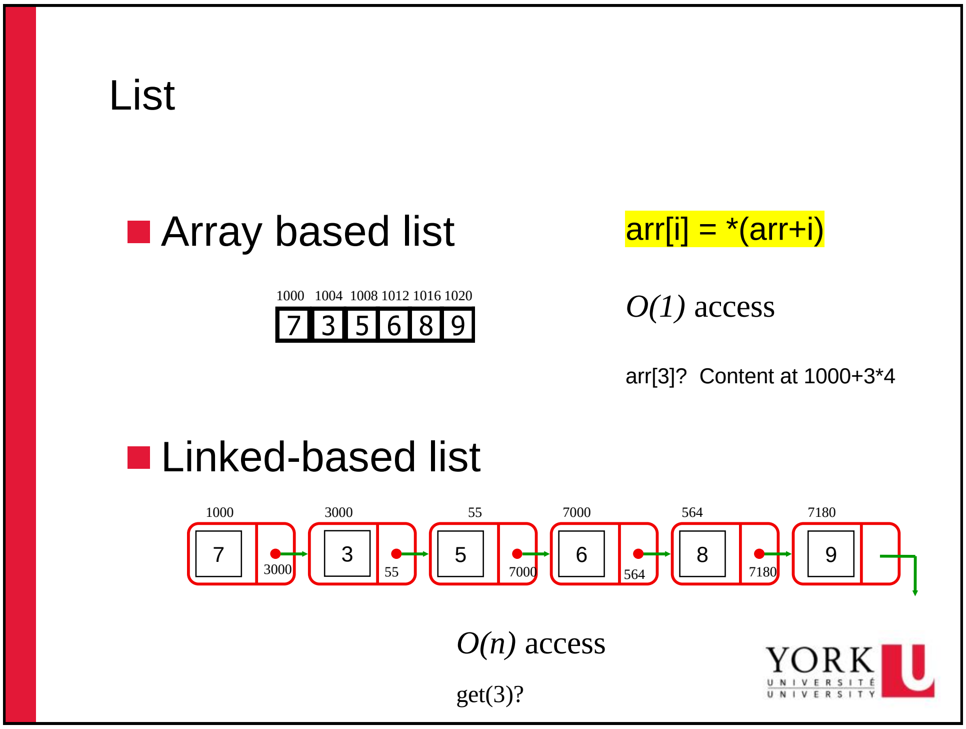
Structures

- Basics: Declaration and assignment
- Structures and functions
- Pointer to structures
- Arrays of structures
- Self-referential structures (last topic in C)
  - Structure + pointer to structure + malloc/calloc
  - e.g., linked list, binary trees

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





List

- Array based list

1000	1004	1008	1012	1016	1020
7	3	5	6	8	9

arr[i] = *(arr+i)

O(1) access

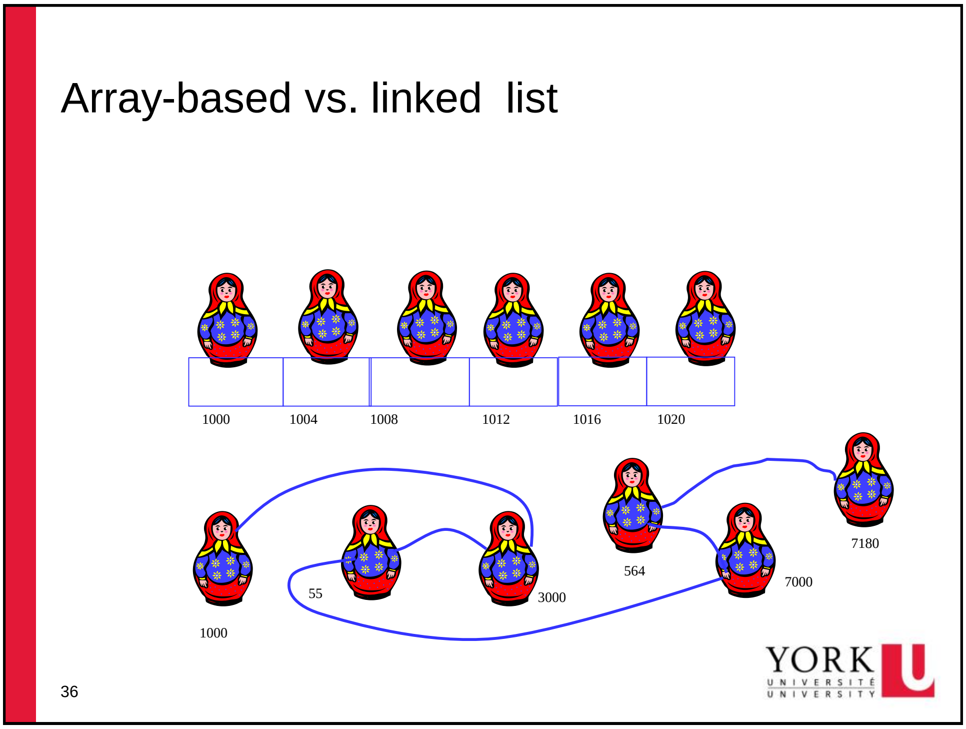
arr[3]? Content at 1000+3*4

- Linked-based list

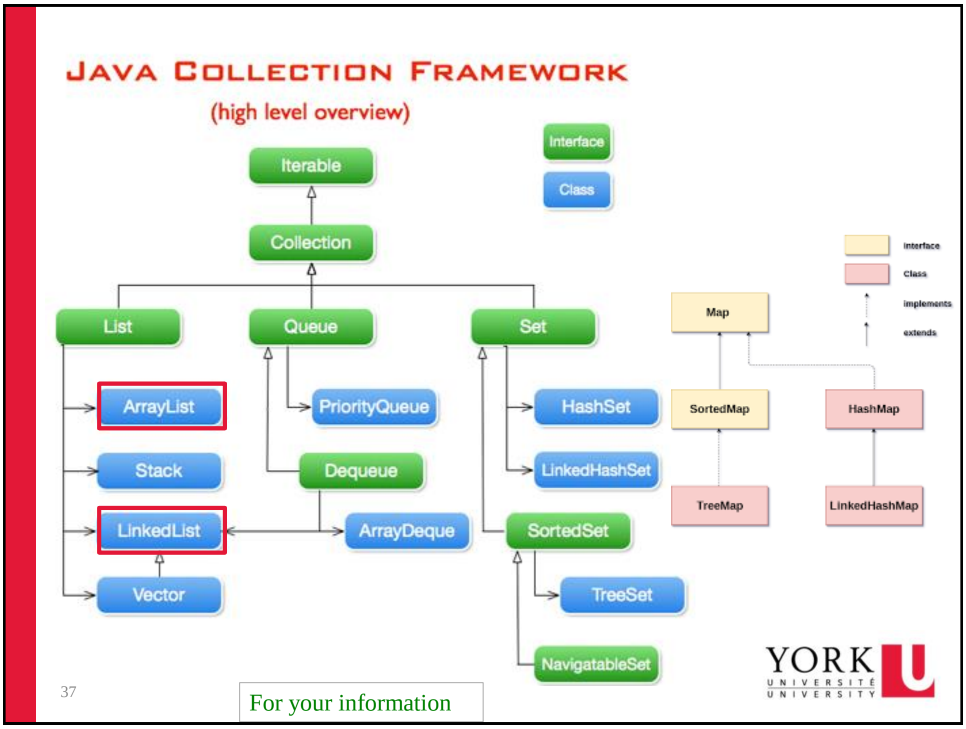
1000	3000	55	7000	564	7180
7	3	5	6	8	9
3000	55	7000	564	7180	

O(n) access

get(3)?



Array-based vs. linked list



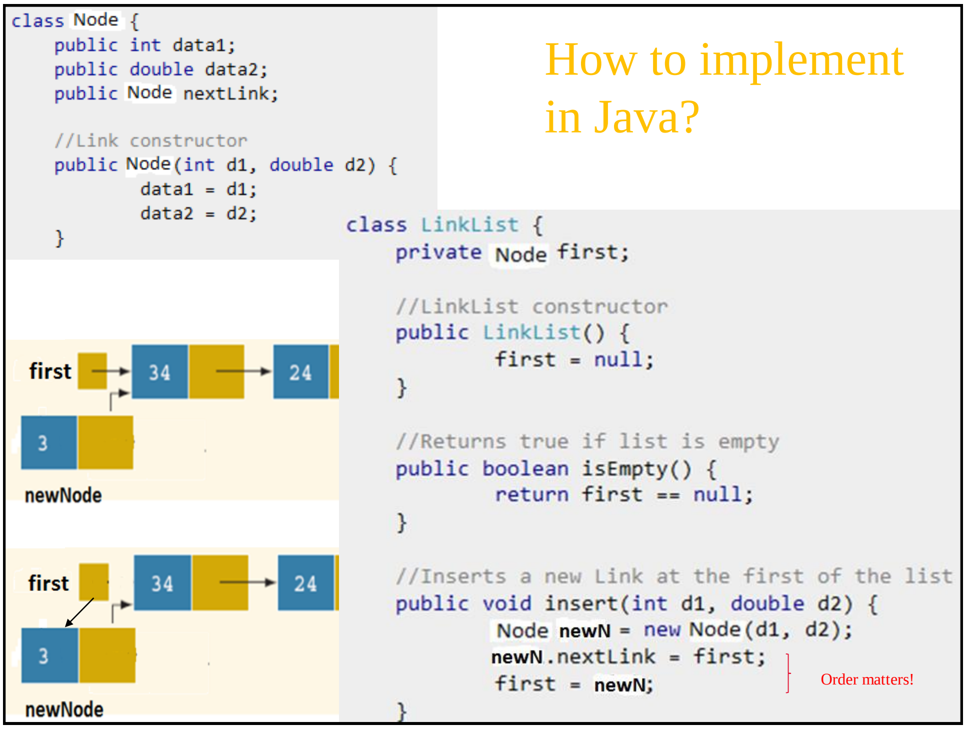
Java Collection Framework

(high level overview)

37

For your information

37



```
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;
    }
```

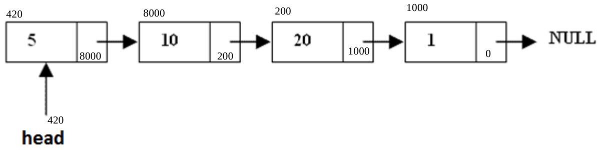
Order matters!

38

Self referential structures in C

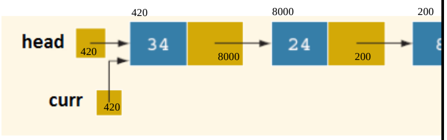
- Simplest example: a linked list of integers

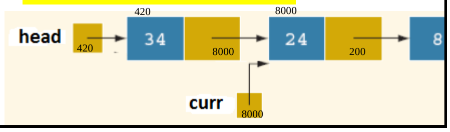
```
struct node {
  int data;
  struct node *next; //pointer to struct node
};

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 ) //(*curr).data == dat
         return 1; // find it!
     curr = curr -> next;  // curr = (*curr).next  8000
   }  //pointer assignment
   return 0;
}
```

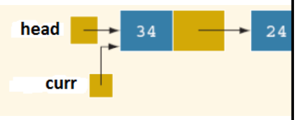


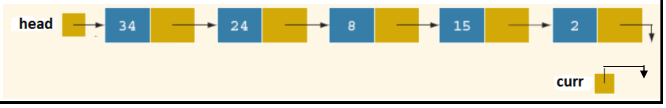


traverse the list example 1

for loop

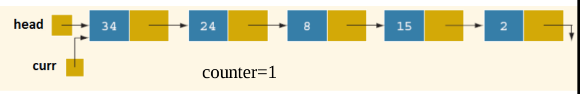
```
struct node * head;

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;
}
```



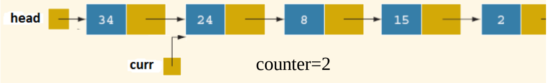


traverse the list example 2

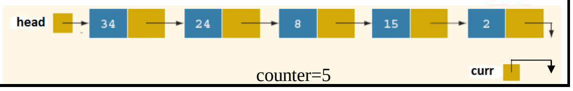
```
struct node * head;

// # 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;
}
```



counter=1



counter=2

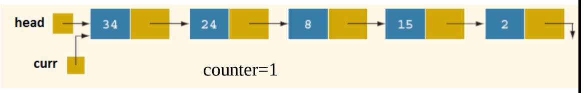


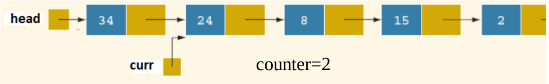
counter=5

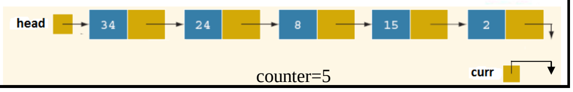
traverse the list example 2

for loop

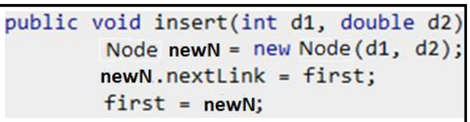
```
struct node * head;

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;
}
```







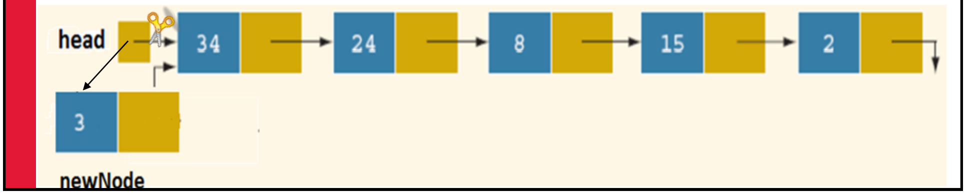
Insert into the list example1

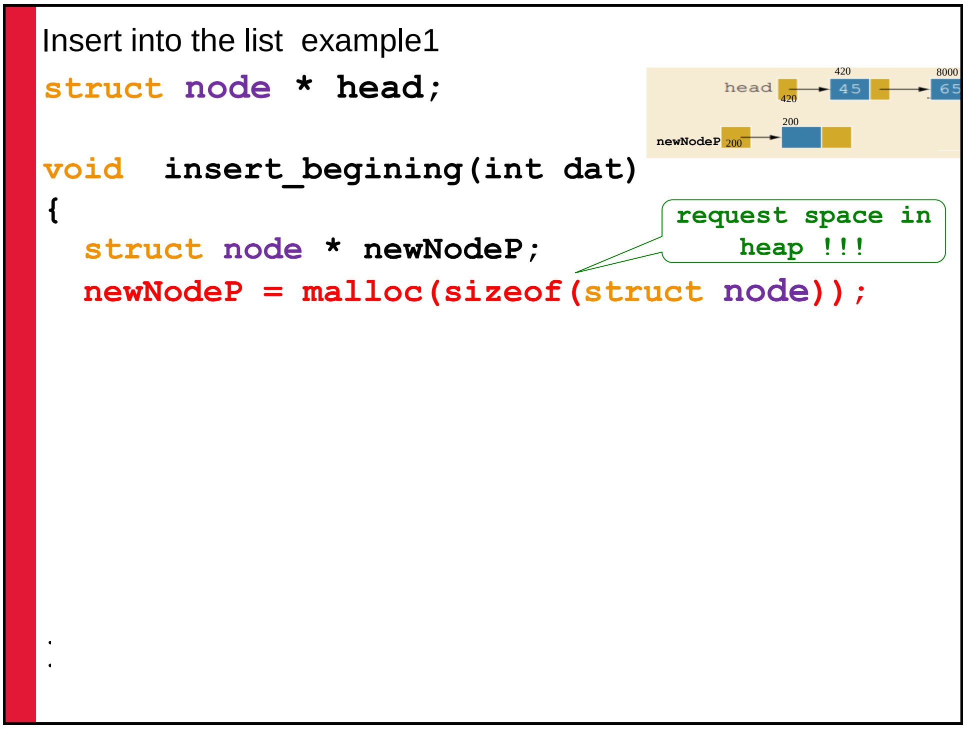


```
public void insert(int d1, double d2)
        Node newN = new Node(d1, d2);
        newN.nextLink = first;
        first = newN;
```

```
struct node * head;

void  insert_begining(int dat)
{
    struct node newNode;
     newNode.data = dat;

    newNode.next = head;

    head = &newNode;
}
```

newNode
is in stack!





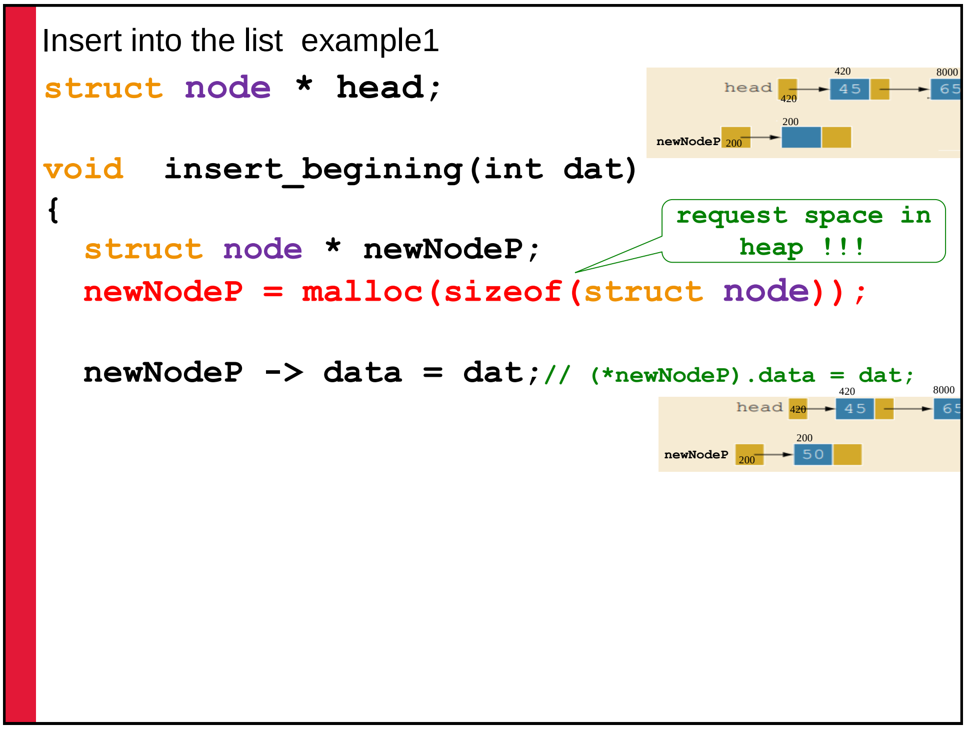
Insert into the list example1

```
struct node * head;

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

request space in heap !!!

.
.

45



Insert into the list example1

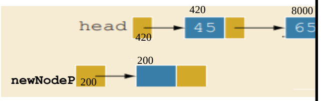
```
struct node * head;

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

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

request space in heap !!!

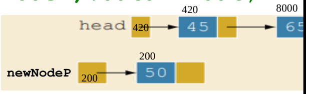
46

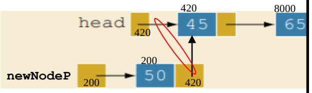
Insert into the list example1

```
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
```



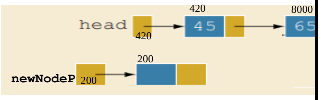
request space in heap !!!



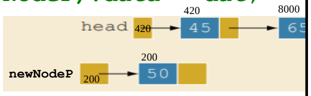


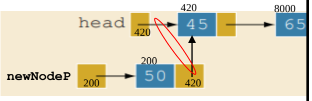
Insert into the list example1

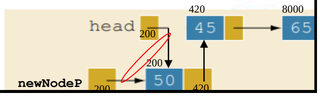
```
struct node * head;

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;
}
```

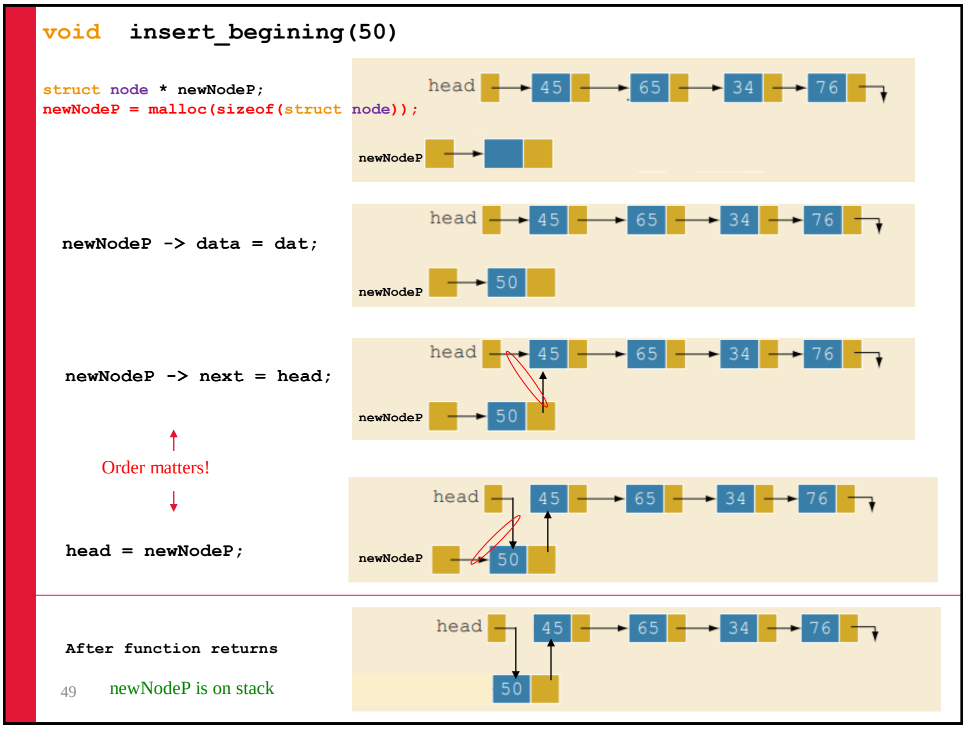


request space in heap !!!









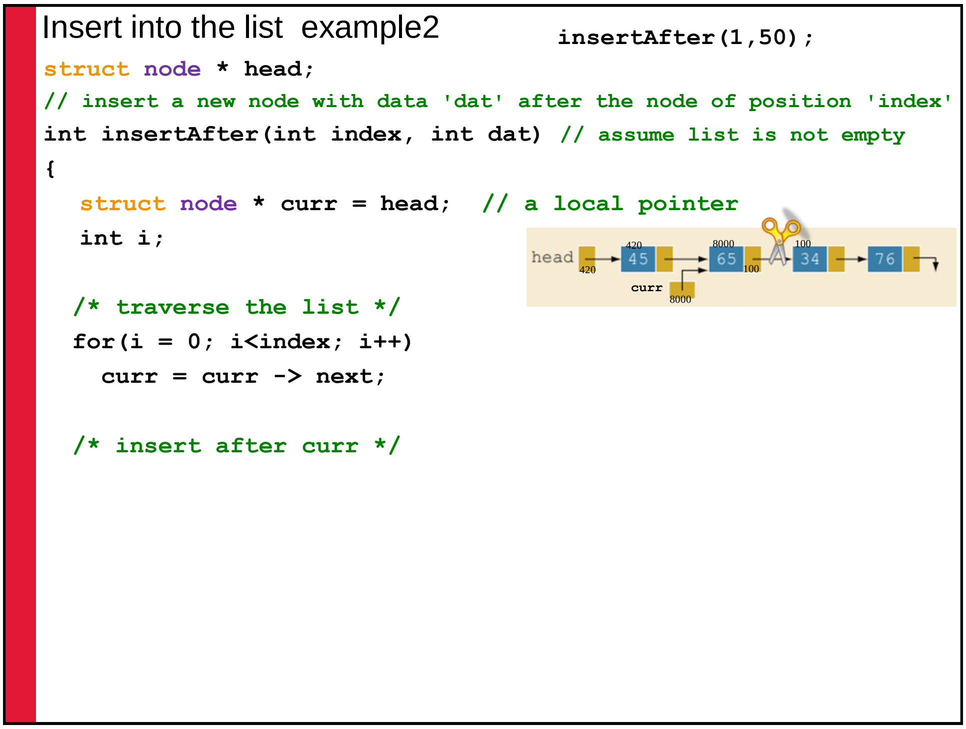
```
void  insert_begining(50)

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

  newNodeP -> data = dat;

  newNodeP -> next = head;
```

Order matters!

```
  head = newNodeP;
```

```
  After function returns
```

49 newNodeP is on stack



Insert into the list example2

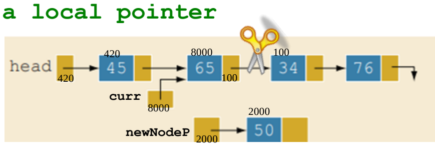
```
insertAfter(1,50);

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 */
```

head
420
45
420
8000
65
100
100
34
76
curr
8000

Insert into the list example2

```
insertAfter(1,50);
```

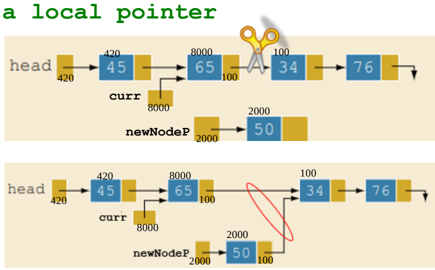
```
struct node * head;
// insert a new node with data 'dat' after the node of position 'index'
int insertAfter(int index, int dat) // assume list is not empty
{
    struct node * curr = head;  // a local pointer
    int i;

   /* 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;
```



Insert into the list example2

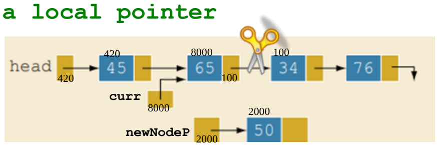
```
insertAfter(1,50);
```

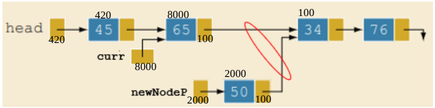
```
struct node * head;
// insert a new node with data 'dat' after the node of position 'index’
int insertAfter(int index, int dat) // assume list is not empty
{
    struct node * curr = head;  // a local pointer
    int i;

   /* 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;
```

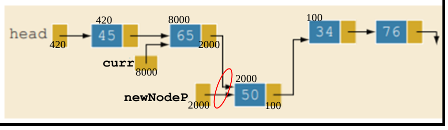


Insert into the list example2

```
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
```

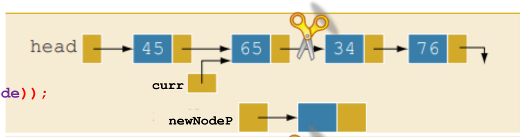




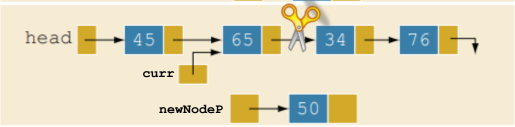


```
int insertAfter(1, 50)
```

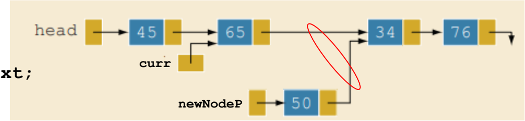
```
struct node * newNodeP;
newNodeP = malloc(sizeof(struct node));
```



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

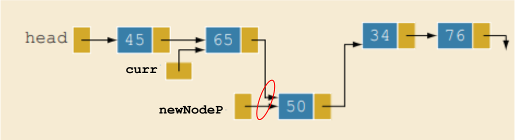


```
newNodeP -> next = curr -> next;
```

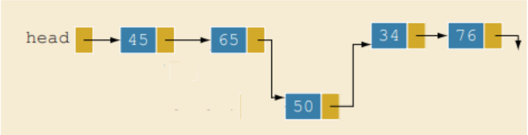


Order matters!

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



After function returns



curr, newNodeP are on stack

EECS2031 - Software Tools

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

skipped

EECS2031 - Software Tools

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

skipped

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

- const
- Union, enum, typedef
- Library functions, e.g., memset(), strtok()
- 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

- 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!