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

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

main(){
  struct shape s = {1,3};
  float f = get_area(s);
}

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

```
// shape as argument
        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
  - **d** is a copy of the actual parameter (copy member values)
  - No starting address, no "decay" not efficient for large struct

Structure and functions
-- structures as arguments

- You can pass structures as arguments to functions

```
main()  {
  struct shape  s = {1,2};
  do_sth(s)
}

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

```
/* s.width?  s.height?  */
```

call-by-value

```
d = s  // copy members
    d.width = s.width
    d.height = s.height
```

143

Structure and functions
-- structures as arguments

- You can pass structures as arguments to functions

```
main()  {
  struct shape  s = {1,2};
  do_sth(s) /*  s  is not modified  */
}

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

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

144

144

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
```

145

145

Structure and functions

-- structures as arguments

- You can pass structures as arguments to functions

```
main()  {
   struct shape  s = {1,2};
   s = do_sth(s) /*  s  is modified  */
}

struct shape do_sth(struct shape d)
{
    d.width  += 100;
    d.height += 200;
    return d;
}
```

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

146

146

structure and functions

-- structures as Return Values another example

```
struct ints {
   float int1;
   float int2;
};
struct ints sum_diff(int a, int b)
{
   struct ints resu;
   resu.int1 = a+b;
   resu.int2 = a-b;
   return resu;
}
main(){
   struct resu res = sum_diff(10,4);
   int sum = res.int1;
   int diff = res.int2;
}
```

147

147

Structures

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

148

148

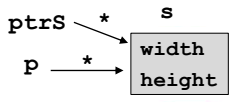
structure and functions

-- 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)
{  struct shape tmp = *p;
   return tmp.width   *  tmp.height;
}
```

Expect a pointer to struct shape



149

( )

149

structure and functions

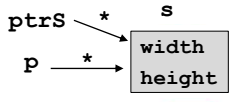
-- Structure Pointers

- 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

Assess member via pointer



150

150

structure and functions

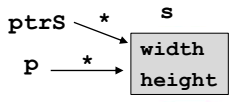
-- Structure Pointers

- 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

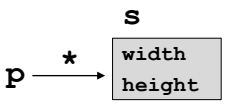


structure and functions

-- Structure Pointers

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

```
main(){
  struct shape  s = {1,2};
  do_sth(&s);
}
void do_sth(struct shape * p)
{
   (*p).width  += 100;
   (*p).height += 200;
}
```



Pointee s is modified !

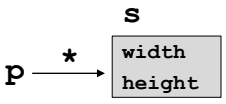
- This is call-by-value --- but address
  - Function can change the passed struct

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` 
- Operator . takes higher precedence over operator *

  `(*p).width` 

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

  `p -> width`

153

structure and functions

-- Structure Pointers

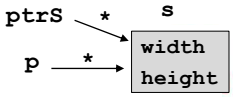
`(*p).width`
`p -> width`
} Equivalent

Either way in your work

```
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;
}
```

154

No () needed



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	,	

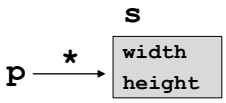
```
x -> data = 2;
x -> data += 2;
```

() never needed!

155

structure and functions

-- Structure Pointers

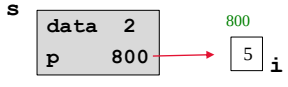



```
void do_sth(struct shape *p){
   p -> width  += 100;  // (*p).width  += 100;
   p -> height += 200;  // (*p).height += 200;
}
```

. works with structures, accessing members

-> works with structure pointers, accessing members

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




```
*(s2.p) = 20;
```

```
s.width;
p.width;
```

```
s -> width;
p -> width;
```

156

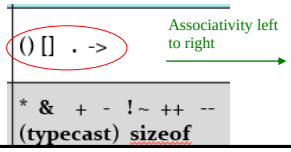
156

Pointers to Structures: Shorthand

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

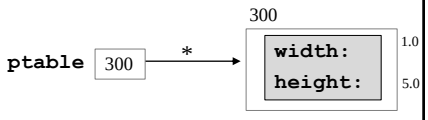
```
struct rect r, *rp = &r;
    r.pt1.x = 1;

(*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;
};
```



157

Pointer to structures -- malloc/calloc



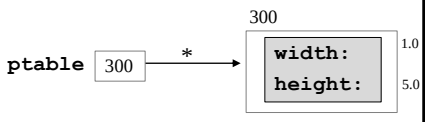

```
struct shape * ptable;   // pointer to struct shape

ptable = malloc (sizeof(struct shape));

// set member value one by one, directly
ptable -> width = 1.0;     // (* ptable).width = 1.0
ptable -> height = 5.0;    // (* ptable).height = 5.0
                    Other way?

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

159

When to use? A good question. Few slides later

159

Pointer to structures -- malloc/calloc

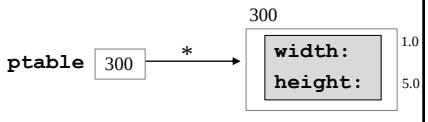



```
struct shape s = {1.0, 5.0};
struct shape * ptable;  // pointer to struct shape

ptable = malloc (sizeof(struct shape));

// set member value by copying s field, directly
ptable -> width = s.width;    // (* ptable).width = s.w
ptable -> height = s.height;  // (* ptable).height =s.h
                  Other way?

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

160

When to use? A good question. Few slides later

Pointer to structures -- malloc/calloc




```
struct shape s = {1.0, 5.0};
struct shape * ptable;  // pointer to struct shape

ptable = malloc (sizeof(struct shape));

// set member value by copying s directly
* ptable = s;    // (* ptable).width = s.width …

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

161

When to use? A good question. Few slides later

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)

Structures

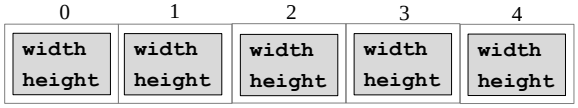
- Basics: Declaration and assignment
- Structures and functions
- Pointer to structures
- Arrays of structures
- Self-referential structures (e.g., linked list, binary trees)

Arrays of structures -- declaration

- Structures can be arrayed same as the other variables

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

struct shape chairs[5];    // recall: int arr[5]
```

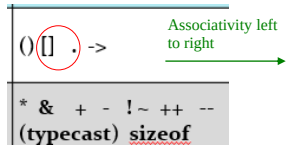
array of 5 (uninitialized) struct

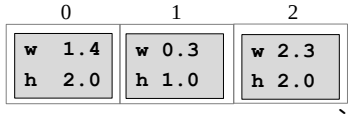


Not NULL

Array of structures -- Initialization

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

struct shape chairs[3]; //chairs[n] is a struc.
chairs[0].height = 1.4;
(chairs[0]).width =  2.0;
.....
float x = chairs[2].height;
```





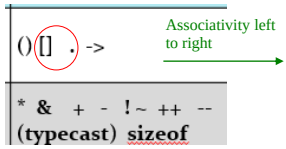

```
chairs.height[2] =1.2; ?
```

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

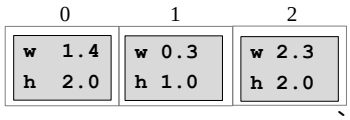
what is chairsA

Array of structures -- Initialization

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

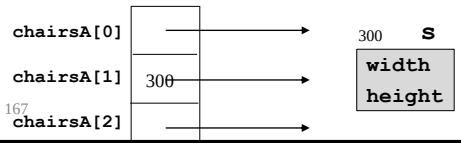



```
struct shape chairs[3]; //chairs[n] is a struc.
chairs[0].height = 1.4;
(chairs[0]).width =  2.0;
.....
float x = chairs[2].height;
```




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

what is chairsA

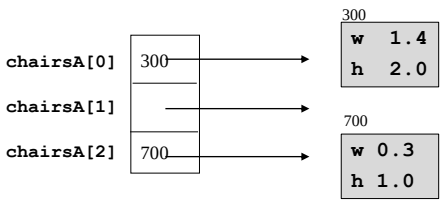



```
chairsA[1] = &s;
```


Array of structures -- Initialization

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

what is chairsA




```
struct shape x = {1.4, 2.0};
chairsA[0] = &x;
struct shape y = {0.3, 1.0};
chairsA[2] = &y;     // change 0.3 to 1.4?
```

Dynamic

```
chairsA[0]= malloc(sizeof(struct shape))
chairsA[0] ?   (*chairsA[0]).width = 1.4;
               chairsA[0] -> height = 2.0;
```

typedef

For creating new data type names

```
typedef int Length;
Length len, maxlen;    // int len, maxlen;
Length *lengths[];     // int* lengths[];

typedef char* String;
String p, lineptr[MAXLINES]; // char* p, lineptr[MAXLINES];
p = (String) malloc(100);    // p=(char *) malloc(100);
int strcmp(String, String); //int strcmp(char*, char*);
```


For your information

YORK UNIVERSITÉ UNIVERSITY

typedef with struct

- We can define a new type and use it later

```
typedef struct {
   int x, y;
   float z;
} mynewtype;

mynewtype a, b, c, x;
```

- Now, **mynewtype** is a type in C just like **int** or **float**.

For your information

YORK UNIVERSITÉ UNIVERSITY

`typedef` with `struct`

- 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;   /* like int i, j */
struct shape table;
```

```
shape table;
```

172

YORK UNIVERSITÉ UNIVERSITY

172

`typedef` with `struct`

- We can define a new type and use it later

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

`shape` is a valid type

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

```
shape table;
```

173

For your information

YORK UNIVERSITÉ UNIVERSITY

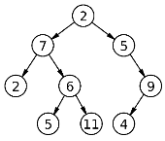
173

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

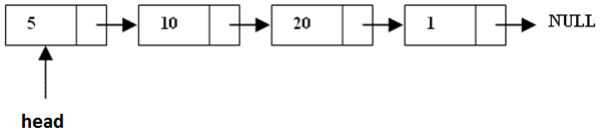
Self-referential structures

- Linked list, trees

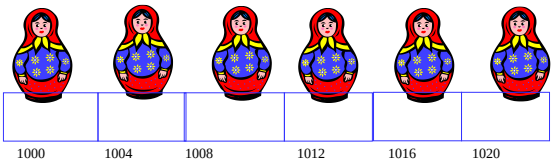


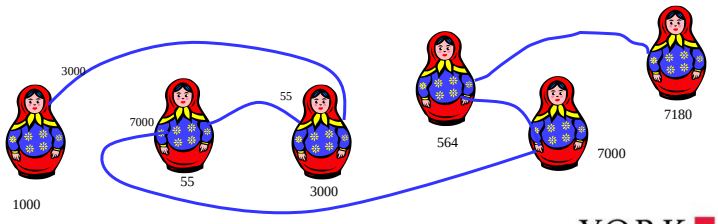
- Linked list
  - alternative to Array
  - data not stored sequentially
    - more flexible than array – can easily insert, delete
    - lost the *O(1)* access in Array, Have to follow the link. Farther ones cost more than closer ones

- Simplest example: a linked list of int's



Array-based vs. linked list





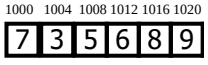
Linked by Reference (Java), pointer (c) to next

Array-based vs. linked list

fact4

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

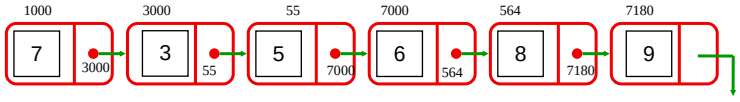
- Array based list



arr[3]? Content at address 1000+3*4
arr[300]?

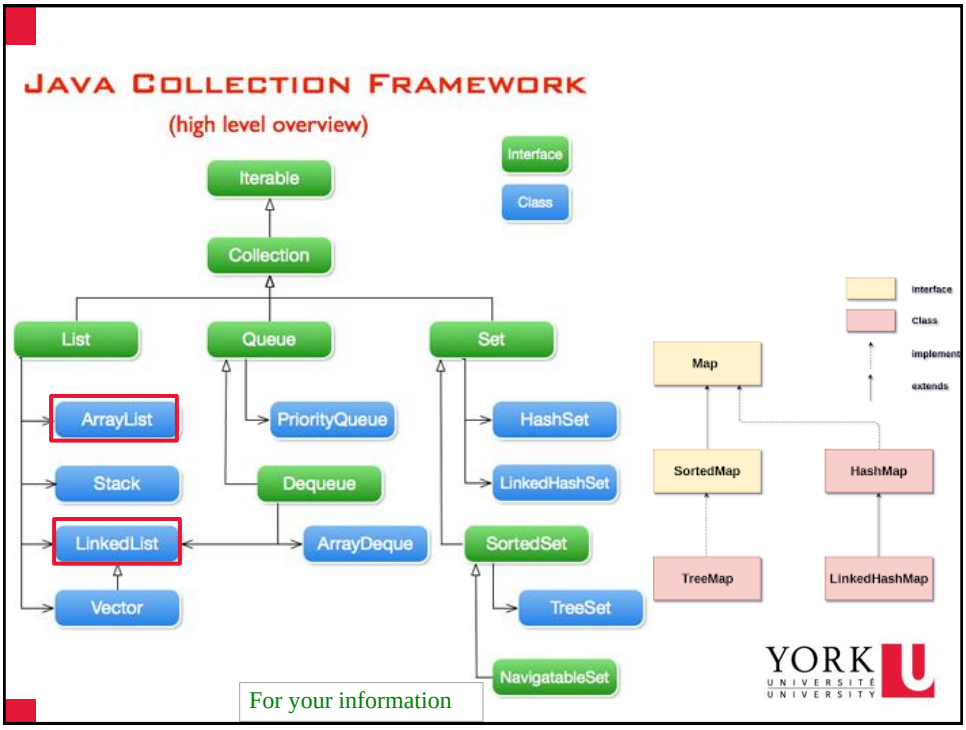
O(1) access

- Linked-based list



O(n) access

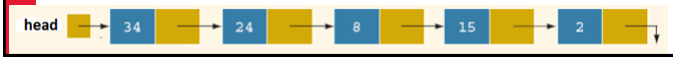
get(3)?
get(300)?



180

```
class Node {

    int data;
    Node next;

    // Node constructor
    public Node(int dataValue)
    {
        this.data = dataValue;
        this.next = null;
    }
```

How to implement in Java?



O(n)
Order matters!

181

```
class Node {

    int data;
    Node next;

    // Node constructor
    public Node(int dataValue)
    {
        this.data = dataValue;
        this.next = null;
    }
```

```
public class MyLinkedList {
    private Node head;  // instance variable.

    // Default constructor
    public MyLinkedList() {
        this.head=null;
    }

    public int len() {
        Node curr = this.head;
        int len = 0;
        while (curr != null)
        {
            curr = curr.next;
            len++;
        }
        return len;
    }

     public int get(int index)
    {
        Node curr = this.head;
        for (int i = 0; i < index; i++) {
            curr = curr.next;
        }
        return curr.data;
    }
```

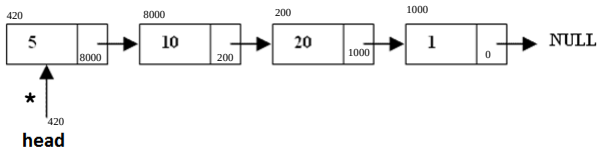
O(n)

182

Self referential structures in C

- Simplest example: a linked list of integers

```
struct node {
  int data; // can be any type
  struct node *next; //pointer to struct node
};

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



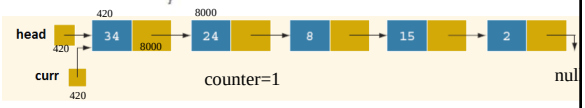
183

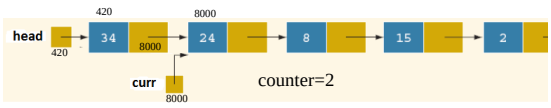
traverse the list example 1

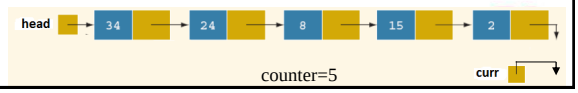
```
public int len() {
    Node curr = this.head;
    int len = 0;
    while (curr != null)
    {
        curr = curr.getNext();
        len++;
    }
    return len;
}
```

Java

```
struct node * head;
 // # of node in the list

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







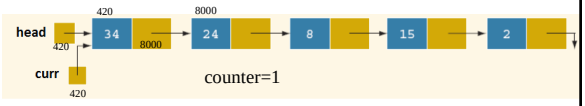
185

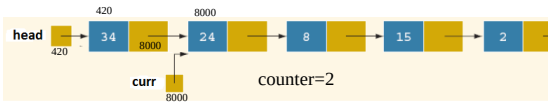
185

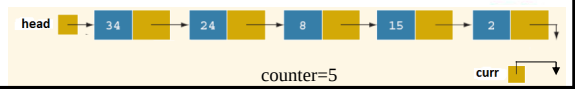
traverse the list example 1

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++;                    // curr = (*curr).next
   return counter;
}
```





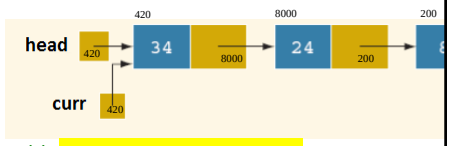


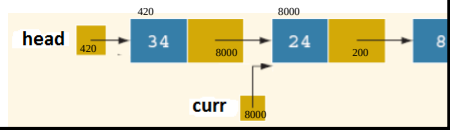
186

186

traverse the list example 2

```
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;
}
```



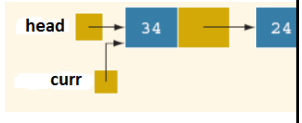


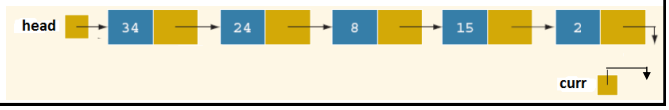
187

traverse the list example 2

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;
}
```

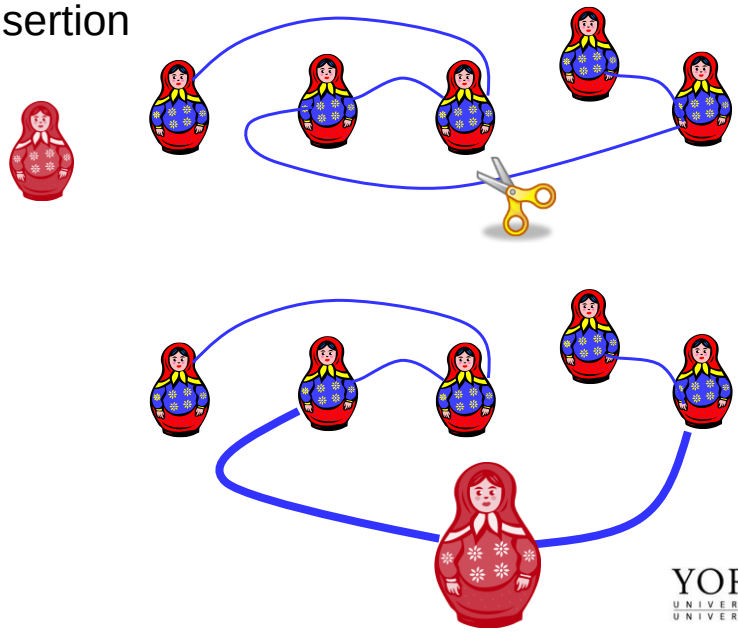




188

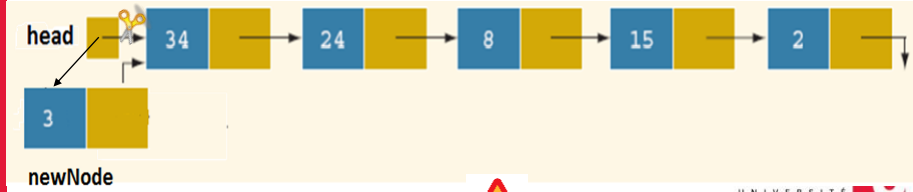
188

Insertion



Insert at beginning

```
// appends the specified element to the begining of this list.
// the list may or may not be empty (but the code is same)
public void insertBegining(int data) {
    Node newNode = new Node(data);

    // Order matters.
    newNode.next = this.head;    // will be null if list is empty
    head = newNode;
}
```



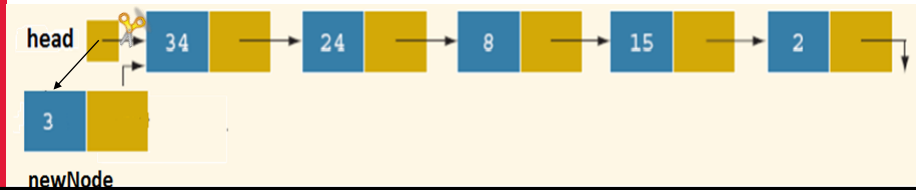
Order matters

Insert into the list example1

```
struct node * head;

void  insert_begining(int dat)
{
    struct node newNode;

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

    head = &newNode;
}
```

```
public void insertBegining(int data)
    Node newNode = new Node(data);

    // Order matters.
    newNode.next = this.head;    //
    head = newNode;
}
```

Working?

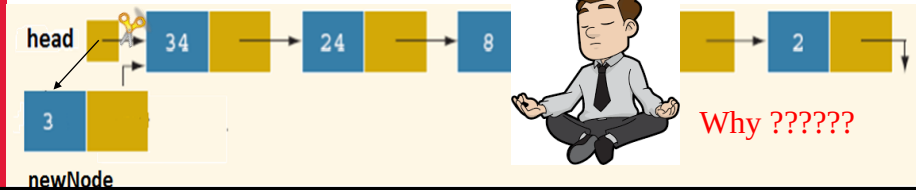


191

Insert into the list example1

```
struct node * head;

void  insert_begining(int dat)
{
    struct node newNode;

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

    head = &newNode;
}
```

```
public void insertBegining(int data)
    Node newNode = new Node(data);

    // Order matters.
    newNode.next = this.head;    //
    head = newNode;
}
```



Why ??????

192

Think about setArr of array of pointer to struct

```c
#include <stdio.h>
#include <stdlib.h>

struct shape {
    float height;
    float width;
};


void setArr (int, float, float);

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

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

  setArr(0, 40, 50);
  setArr(1, 4000, 5000);
  printf("arr[%d] -*->%f  %f \n",0, arr[0] -> width, (*arr[0]).height);
  printf("arr[%d] -*->%f  %f \n",1, arr[1] -> width, (*arr[1]).height);

  return 0;
}

/* set arr[index], which is a pointer, to point to a struct of value h, w */
void setArr (int index, float h, float w){
     struct shape newS = {h, w};
     arr[index] = &newS;
}
```

5000.000000 4000.00000
0.000000 0.000000

Working?

YORK UNIVERSITÉ UNIVERSITY

193

Think about setArr of array of pointer to struct

```c
#include <stdio.h>
#include <stdlib.h>

struct shape {
    float height;
    float width;
};


void setArr (int, float, float);

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

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

  setArr(0, 40, 50);
  setArr(1, 4000, 5000);
  printf("arr[%d] -*->%f  %f \n",0, arr[0] -> width, (*arr[0]).height);
  printf("arr[%d] -*->%f  %f \n",1, arr[1] -> width, (*arr[1]).height);

 return 0;
}

/* set arr[index], which is a pointer, to point to an integer of some value */
void setArr (int index, float h, float w){
     /*struct shape newS = {h, w};
     arr[index] = &newS;*/

     struct shape *newS = malloc(sizeof(struct shape));
     newS -> width = w;
     newS -> height = h;
     arr[index] = newS;

     // or directly
     arr[index] = malloc(sizeof(struct shape));
     arr[index] ->width = w;
     arr[index] ->height = h;
```

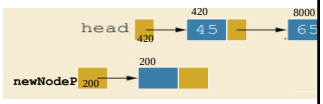
50.000000 40.000000
4000.00000 5000.000000

YORK UNIVERSITÉ UNIVERSITY

194

Insert into the list example1

```
struct node * head;

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

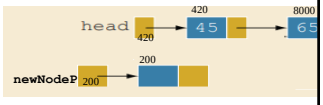


request space in heap !!!

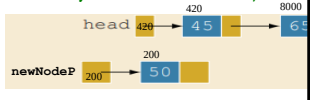
195

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 !!!

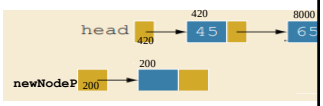


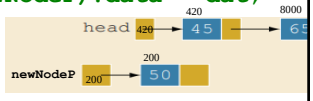
196

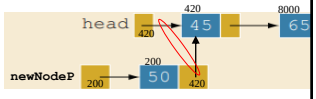
Insert into the list example1

```
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 !!!







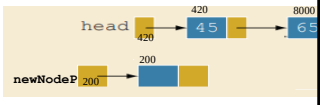
197

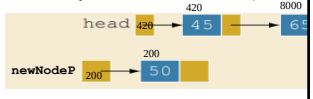
Insert into the list example1

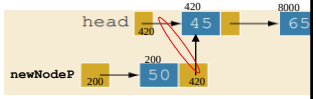
```
struct node * head;

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;
}
```

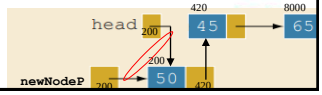
198

request space in heap !!!



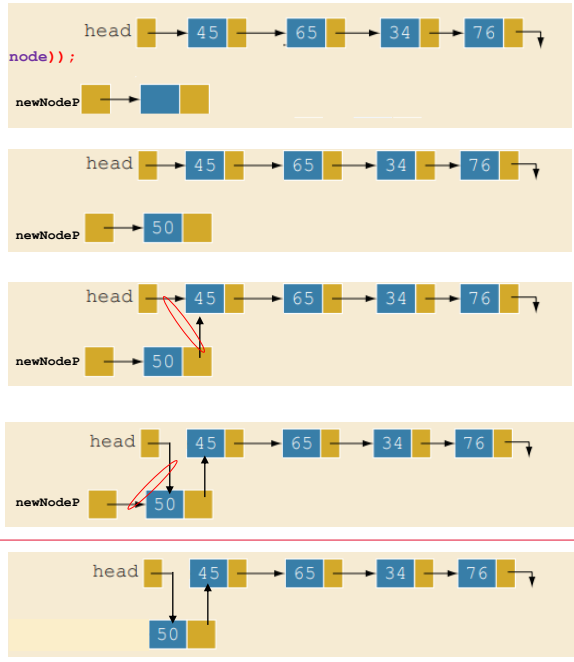






198

```
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
```

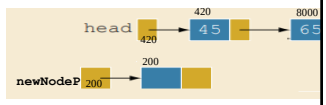


199 newNodeP is on stack

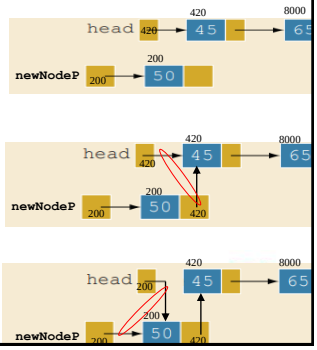
199

Insert into the list example1

```
struct node * head;

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

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

  *newNodeP = s;


  head = newNodeP;
}
```



Working



200

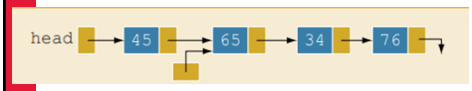
200

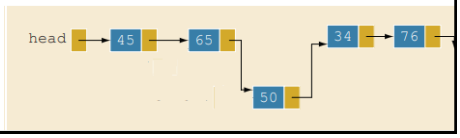
Insert into the list example2

insertAfter(1,50);

```
// inserts the specified element after the specified position in this list
// assume list is not empty.  index is valid [0 - len()-1]
public void insertAfter(int index, int data) {
    Node curr = this.head;

    // crawl to the requested index
    for (int i = 0; i < index; i++) {
        curr = curr.next;
    }

    // insert after curr, and before curr.next;  order matters!!!
    Node newNode = new Node(data);

    // set the new node's next-node reference to curr node's next-node refer
    newNode.next = curr.next;

    // now set curr node's next-node reference to the new node
    curr.next = newNode;
}
```

Order matters!



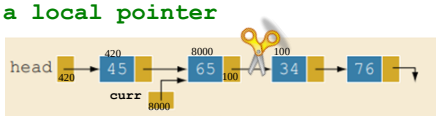


201

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 */
```

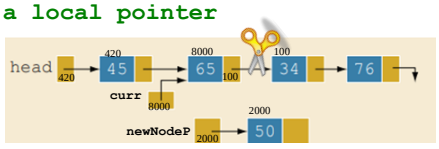


202

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;
```

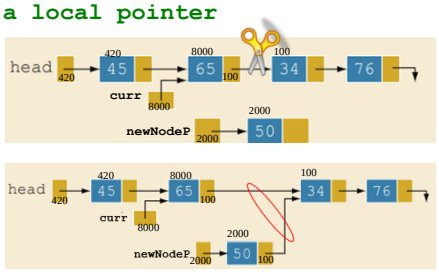


203

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;
```

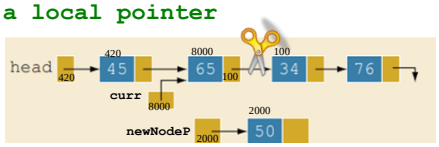


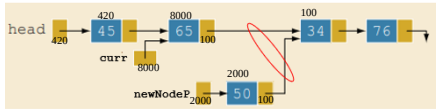
204

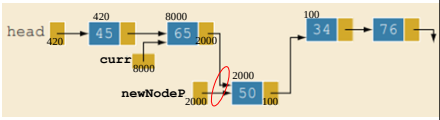
Insert into the list example2

```
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
```



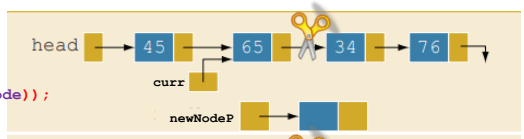




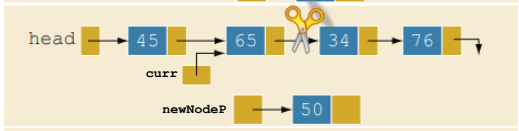
205

int insertAfter(1, 50)

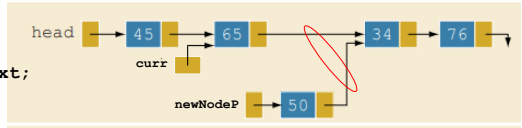
```
struct node * newNodeP;
newNodeP = malloc(sizeof(struct node));
```




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

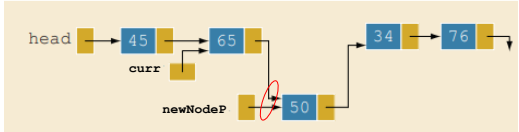



```
newNodeP -> next = curr -> next;
```

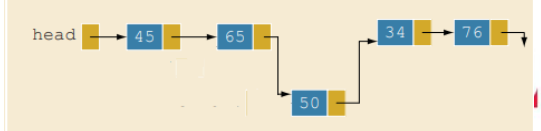


Order matters!

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




```
After function returns
```



206 curr, newNodeP are on stack

206

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


```
<stdio.h>
printf()
scanf()
getchar()
putchar()

sscanf()
sprintf()

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

fprintf()
fscanf()
fopen()
```

EECS2031 - Software Tools

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

skipped

EECS2031 - Software Tools

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

skipped

YORK UNIVERSITÉ UNIVERSITY

209

Topics that we did not get to cover (yet), -- may be useful in your future studies, research and career

- ~~Pre-processing~~
- const
- Union, enum, typedef
- Library functions, e.g., **`memset()`**, **`strtok()`**
- ~~Point array decayed to~~
- Pointer to whole arrays, **`int(* arr)[]`** **`[][]`** decayed to
- Pointer to functions
- Stream IO Ch7 and read/write disk files fopen(…) fread() PE2
- 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
  - Version control

YORK UNIVERSITÉ UNIVERSITY

210

- That's all for C for now

- Now we have to start a new book, a new programming language .....

700 pages