EECS2031M Software Tools

W 2020

Mar 17, 2020 Lecture 19

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

YORK UNIVERSITÉ UNIVERSITY

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
```

3 ISO C90 forbids variable length array 'my_array'

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)
```

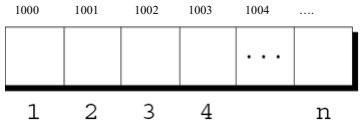
<assert.h>

```
assert()
```

4

4

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

5

malloc()

Dangling Pointers


```
#include <stdlib.h>

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

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

```
segmentation fault
core dump
```


6

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`

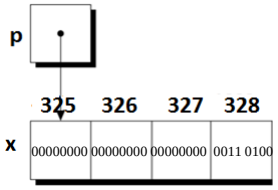
Ask yourself: Have you done one of the following

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

7

7

malloc()



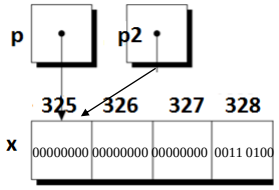

```
#include <stdlib.h>

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

fix1

8

8

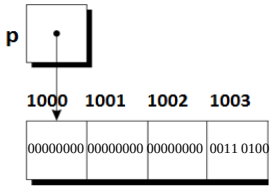
malloc()




```
#include <stdlib.h>

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

fix2

malloc()

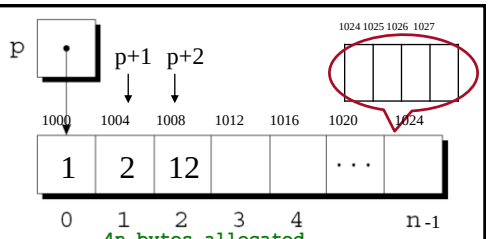



```
#include <stdlib.h>

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

malloc()



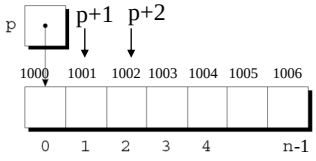
4n bytes allocated.
n=7 28 bytes 1000~1027 allocated

```
#include <stdlib.h>

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

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

  *p = 1;          // store 1 at address 1000 (1000~1003)
  *(p+1) = 2;      // p+1 = 1004  store 2 at address 1004
  *(p+2) = 12;     // p+2 = 1008  store 12 at address 1008
}                      pointer arithmetic!!!
  free (p);
```

11

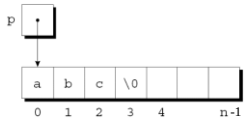
11

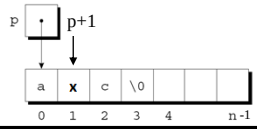
malloc()



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

```
#include <stdlib.h>

int main() {
  int n; char *p;
  printf("length 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';

  printf("%s", p); // axc
  printf("%d", strlen(p)); // 3
```





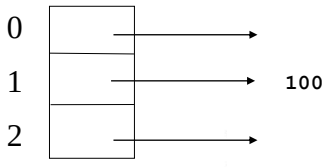
12

More on memory allocation

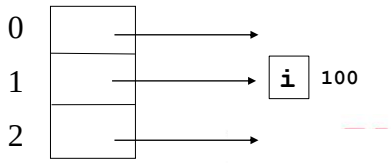
- We know the syntax
- But when to use it ?????
  - When need to allocate at run time, of course
  - What else?


```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){

   *arr[index] = 100;

}
```

What is wrong here??



```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;
}
```

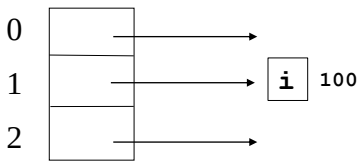


What is wrong here??

15

15

```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;
}
```

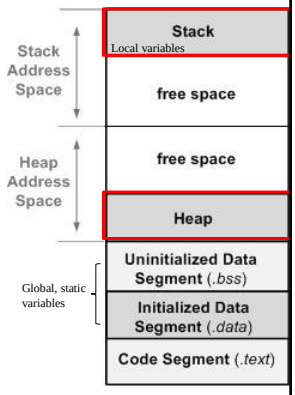


i is local variable, lifetime is block/function -- i is in stack, where it is deallocated when function exits !!!

16

16

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!

17

How to allocate in heap then?

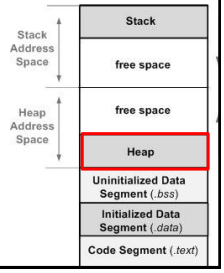
17

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



18

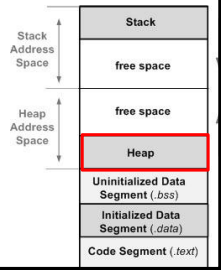
18

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();



19

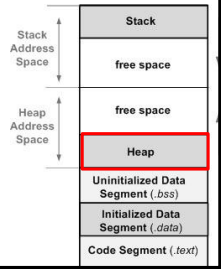
19

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:**



20

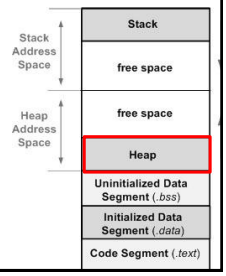
20

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.

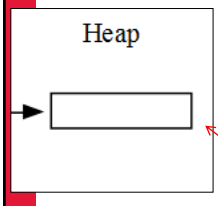
What we need!

- **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)


}
```

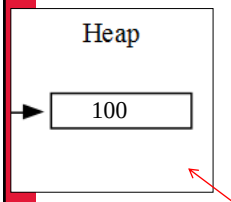


Correct implementation

```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

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



23

23

```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

24

24

- 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)

today

EECS2031 – Software Tools

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

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
  }

struct {
   float width;
   float height;
  table;
```

is the type `// like int a;`

`chair` is variable name.

Need to repeat

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;   /* like 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	left-to-right
	+ - arithmetic	
	>> << bitwise	
	< > <= >= relational	
	== != relational	

31

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];
};
```

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

32

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. copy members value */
                /* Different from Java! */
```

- 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,  chair.height);
printf("%d %d", chair2.width, chair2.height);

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

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

2 4 2 4 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

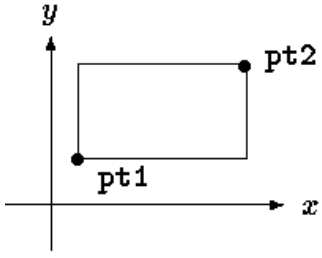
```
(* ptr).width
```

later

35

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
  use **`sizeof`**

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

```
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"

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

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
```

41

41

Structures

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

42

YORK UNIVERSITÉ UNIVERSITY

42

structure and functions

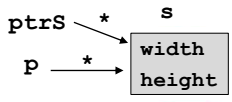
-- Structure Pointers

- 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



43

43

structure and functions

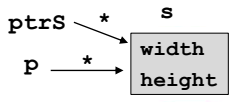
-- Structure Pointers

- 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



44

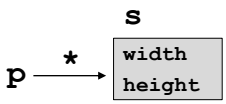
44

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

45

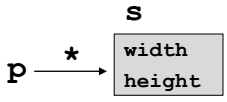
45

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`

46

46

structure and functions

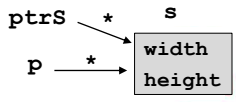
-- Structure Pointers

```
(*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;
}
```

47



47

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	,	

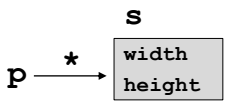
```
x -> data = 2;

x -> data += 2;
```

() never needed!

48

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`

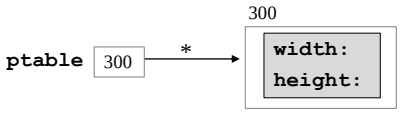
```
struct rect r, *rp = &r;
    r.pt1.x = 1;

(*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;
};
```

() [] . ->	Associativity left to right
* & + - !~ ++ -- (typecast) sizeof	

YORK UNIVERSITÉ UNIVERSITY

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
```

or

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

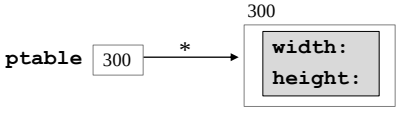
51

When to use? Few slides later

YORK UNIVERSITÉ UNIVERSITY

51

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

or

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

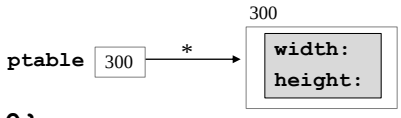
52

When to use? Few slides later

YORK UNIVERSITÉ UNIVERSITY

52

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;


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

53

When to use? Few slides later

YORK UNIVERSITÉ UNIVERSITY

53

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)

54

54

Structures

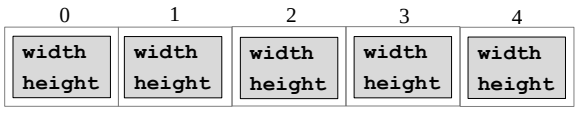
- Basics: Declaration and assignment
- Structures and functions
- Pointer to structures
- Arrays of structures
- Self-referential structures (e.g., linked list, binary trees)

Arrays of structures -- declaration

- Structures can be arrayed same as the other variables

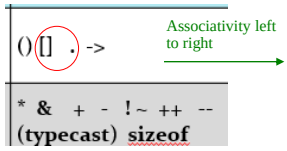
```
struct shape {
   float width;
   float height;
};

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

array of 5 struct



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];
```

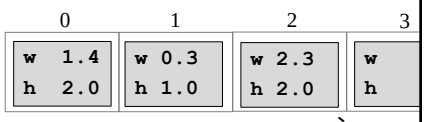
what is chairsA

57

57

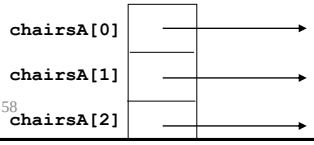
Array of structures -- Initialization

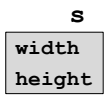
```
struct shape chairs[] = {
                        {1.4, 2.0},
                        {0.3, 1.0},
                        {2.3, 2.0} };

struct shape chairs[10]; //chairs[n] is a struc.
chairs[0].height = 1.4;
(chairs[0]).width =  2.0;
.....
float x = chairs[2].height;
```




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

what is chairsA






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

58

58

Structures

- Basics: Declaration and assignment
- Structures and functions
- Pointer to structures
- Arrays of structures

today

- Self-referential structures (last topic in C)
  - Structure + pointer to structure + malloc/calloc
  - e.g., linked list, binary trees

Next time