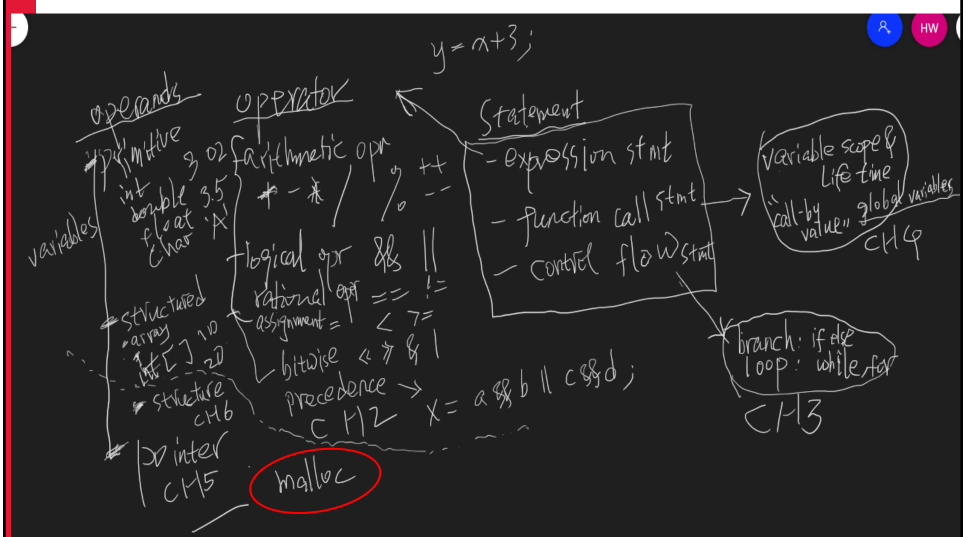
- Pointers (Ch5)
  - Basics: Declaration and assignment (5.1)
  - Pointer to Pointer (5.6)
  - Pointer and functions (pass pointer by value) (5.2)
  - Pointer arithmetic +- ++ -- (5.4)
  - Pointers and arrays (5.3)
    - Stored consecutively
    - Pointer to array elements p + i = &a[i] *(p+i) = a[i]
    - 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)

Previous lecture

today

YORK UNIVERSITÉ UNIVERSITY

79



YORK UNIVERSITÉ UNIVERSITY

80

80

Dynamic memory allocation scenario / motivation 1

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

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

sets aside space for 20 ints (80 bytes)

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

```
define SIZE 20

int arr[SIZE];          20*4 bytes
char arr[20][30];       20*30*1 bytes
int arr[] = {3,5,6};    3*4 bytes
char arr[] = "Hello"    6*1 bytes
```


Dynamic memory allocation scenario / motivation 1

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

- How?

```
int n;
printf("How many elements in int array? ");
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'
```

- Fortunately, C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution.

- Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

- The **<stdlib.h>** header declares three memory allocation functions:
  - **malloc** Allocates a block of memory but doesn't initialize it.
  - **calloc** Allocates a block of memory and clears it.
  - **realloc** Resizes a previously allocated block of memory.

- These functions return a value of type **void *** (a "generic" pointer).
  - function has no idea what type of data to store in the block.

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)
strtok(s,s)
```

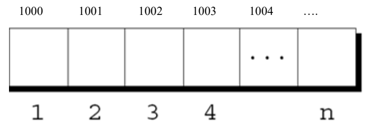
```
<math.h>
sin()  cos()
exp()
log()
pow()
sqrt()
ceil()
floor()
```

```
<stdlib.h>

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

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

```
<ctype.h>

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

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

```
<assert.h>
assert()
```

malloc()



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

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

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

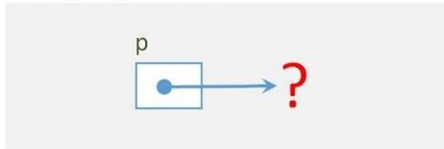
Summary of pointer operations

RECALL

p1 → x
p2 → x

- Legal:
  - assignment of pointers of the same type `p2 = p1`
  - adding or subtracting a pointer with an integer `p++ , p+2, p-2`
  - subtracting or comparing two pointers to members of the same array `p2- p1` `if (p1 < p2)` `while (p1 != p2)`
  - assigning or comparing to zero (NULL) (later) `p = NULL` `p==NULL`
- Illegal: `p1+p2; p1*p2; p*3`
  - add two pointers, multiply or divide two pointers, integers
  - add or subtract float or double to pointers `p + 1.23`
  - shift or mask pointer variables `p << 2` `p|3`
  - assign a pointer of one type to a pointer of another type (except for void *) without a cast used in OS course

Dangling Pointers



malloc()

```
#include <stdlib.h>

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

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

segmentation fault
core dump

YORK UNIVERSITÉ UNIVERSITY

Whenever you need to set a pointer's pointee

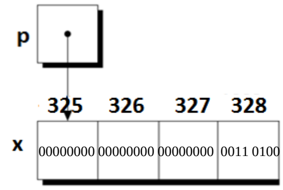
e.g.,

- `*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(....)   /* now */
```

malloc()



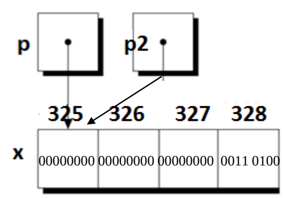

```
#include <stdlib.h>

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

fix1

89

89

malloc()



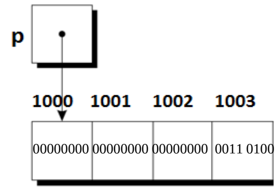

```
#include <stdlib.h>

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

fix2

90

90

malloc()




```c
#include <stdlib.h>

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

fix3

Improve?

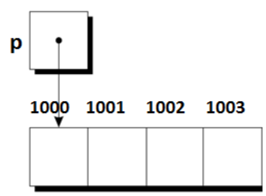
- Note: type conversion (cast) on result of malloc
  `p = malloc(4);` also works. Will convert

91

YORK UNIVERSITÉ UNIVERSITY

91

Improve1 sizeof



- A better approach to ensure portability

```c
int *p;
p = (int *) malloc(4);
        ⬇
p = (int *) malloc( sizeof(int) );
*p = 52;
```

92

Improve 2 NULL

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

```
int *p;
p = (int *)malloc(100000000);// malloc returns NULL
p = (int *)malloc(-10);      // malloc returns NULL
```


YORK UNIVERSITÉ UNIVERSITY

NULL

- <stdlib.h> <stdio.h> <string.h> …defines macro **NULL** a special pointer constant with value 0
- 0 (zero) is never a valid address
- **NULL** == "0 as a pointer" == "points to nothing"
  - `int * p;  // p == NULL?  Not really`
  - `p == 0 ?  // better use NULL   like EOF`

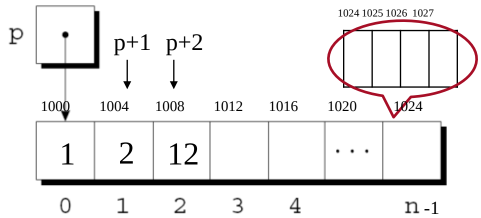
```
p = malloc(10000000);
if (p == NULL) {  // an "exception"
  exit(0) /* allocation failed; take appropriate action *
}
else …
```

⬇

```
if ( (p = malloc(10000000)) == NULL) {
  exit(0) /* allocation failed; take appropriate action *
}else ….
```


UNIVERSITY

malloc()

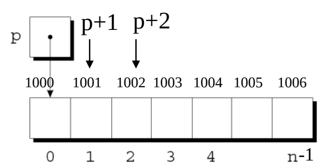



```
#include <stdlib.h>

int main() {
  int n;
  printf("How many elements in int array? ");
  scanf("%d", &n);

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

  // else
  *p = 1;          // p[0] = 1     second +1 +4?
  *(p+1) = 2;      // p+1 = 1004  p[1]= 2
  *(p+2) = 12;     // p+2 = 1008  p[2] = 12
}                     pointer arithmetic!!!
```

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

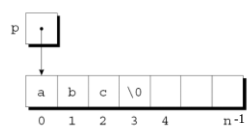
95

malloc()



n bytes allocated. Include for \0
n=7 7 bytes 1000~1006 allocated

```
#include <stdlib.h>

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

  char * p = (char *)malloc(n * sizeof(char)); //n+1?
  if (p == NULL)
     exit(0);

  strcpy(p, "abc");

  *(p+1) = 'x';

  printf("%s", p); // axc
  printf("%d", strlen(p)); /
} printf("%s", p+1);
```




```
*(p+0) = 'a';
*(p+1) = 'b';
*(p+2) = 'c';
*(p+3) = '\0'
```

96

calloc()

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

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

  alternatively,

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

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

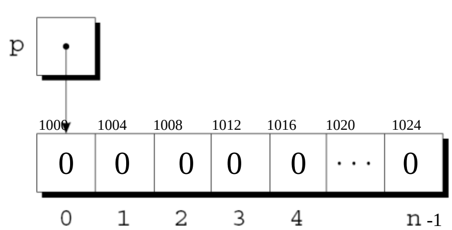
```
int *p;
p = (int *)calloc(6, sizeof(int));
```

98

98

calloc() vs. malloc()

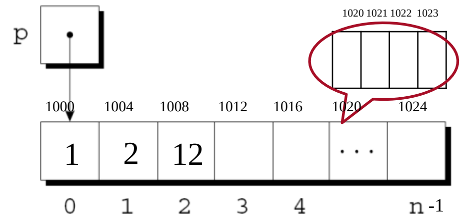
- **`calloc(x , y)`** is pretty much the same as **`malloc(x * y)`**
- except
  - **`malloc`** does not initialize memory
  - **`calloc`** initializes memory content to 0 (zero)



99

99

calloc() malloc()




```
#include <stdlib.h>

int main() {
  int n;
  printf("How many elements in int array? ");
  scanf("%d", &n);

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

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

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

100

100

free()

- memory allocation functions **malloc**, **calloc** obtain memory blocks from a storage pool known as the ***heap,*** where storage is persistent until the programmer explicitly requests that it be deallocated (or program terminates)

- A block of memory that's no longer accessible to a program is said to be ***garbage.***
  - A program that leaves garbage behind has a ***memory leak.***

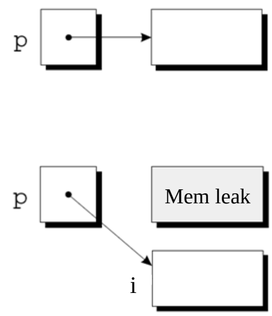
- Some languages (e.g., Java) provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't.

101

101

Memory Leaks

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



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

Memory Leaks

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

For your information

free()

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

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

- "frees" memory we previously allocated, tells the system we no longer need this memory and that it can be reused

- address in "**ptr**" must have been returned from either **malloc**, **calloc** or **realloc**.

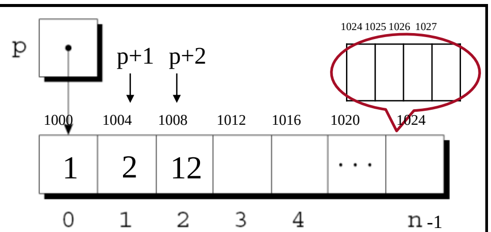
```
p = malloc(7*4);
…
free(p);
```

104

YORK UNIVERSITÉ UNIVERSITY

104

malloc() calloc()



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

```
#include <stdlib.h>

int main() {
  int n;  int *p;
  printf("How many elements in int 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);
  p=&i;
```

105

105

realloc()

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

- resize a dynamically allocated array.

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

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

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

106 For your information

YORK UNIVERSITÉ UNIVERSITY

106

More on memory allocation

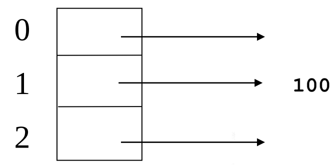
- We know the syntax
- But when to use it ?????
  - When need to allocate at run time, of course
  - What else?
- Another feature of malloc -- request for heap space!

107

YORK UNIVERSITÉ UNIVERSITY

107

```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;

}
```

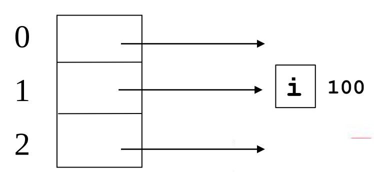


What is wrong here??

110

110

```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;
}
```



Compiles but may or may not work

What is wrong here??

111

111

```c
#include <stdio.h>

void setArr (int);

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

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

     setArr(0);
     setArr(1);

     printf("arr[%d] -*->%d\n", 0, *arr[0]);
     printf("arr[%d] -*->%d\n", 1, *arr[1]);
     return 0;
}

/* set arr[index], which is a pointer, to point to an integer of some value */

void setArr (int index){
     int i = index+100;
     arr[index] = &i;

}
```

```
red 396 % a.out
arr[0] -*-> 101
arr[1] -*-> 32706
red 397 % a.out
arr[0] -*-> 101
arr[1] -*-> 32712
red 398 % a.out
arr[0] -*-> 101
arr[1] -*-> 32737
red 399 %
```

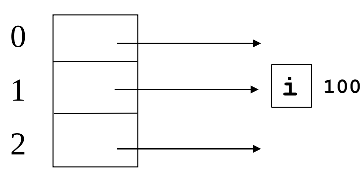
This will probably not work

What is wrong here??

112

112

```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;
}
```

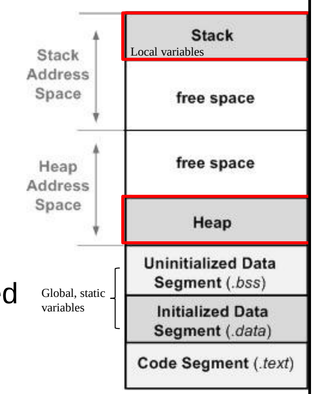


i is local variable, lifetime is block/function -- i is in stack, where it is deallocated when function exits !!!

113

113

Stack vs. Heap



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

What we need!

How to allocate in heap then?

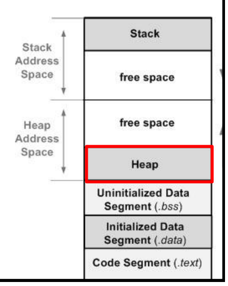
115

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



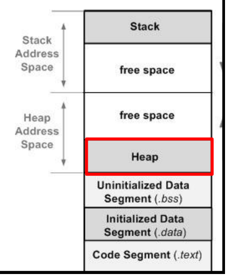
116

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();



117

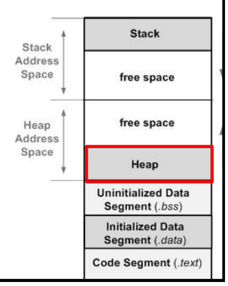
117

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:**



118

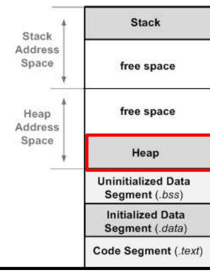
118

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



119

119

When to use malloc ?

- When you need to allocate memory in run time, of course
- When you need memory space throughout the program execution

```
1. ptr = &i.    /* direct */
   o i needs to have persistent lifetime
   o if i is a local variable in function?

2. ptr = ptr2   /* indirect 1*/
   o ptr2 needs to point to persistent address
   o if ptr2 points to a local variable?

3. ptr = (..)malloc(....)
```

correct choice !

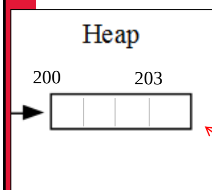
- local variable **i** is in stack. Not in heap.

120

120

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)

}
```



121

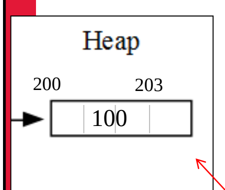
121

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;
```

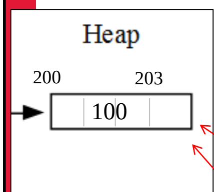


122

122

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){

   int *p = malloc(sizeof(int));
   *p = 100;
   arr[index] = p; // points to heap space
}
```



124

another way

124

```
#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 (but not practical).
a,b,c,d,e are local variables, in stack, but not deallocated before program main() terminates/returns

125

125

- 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)**

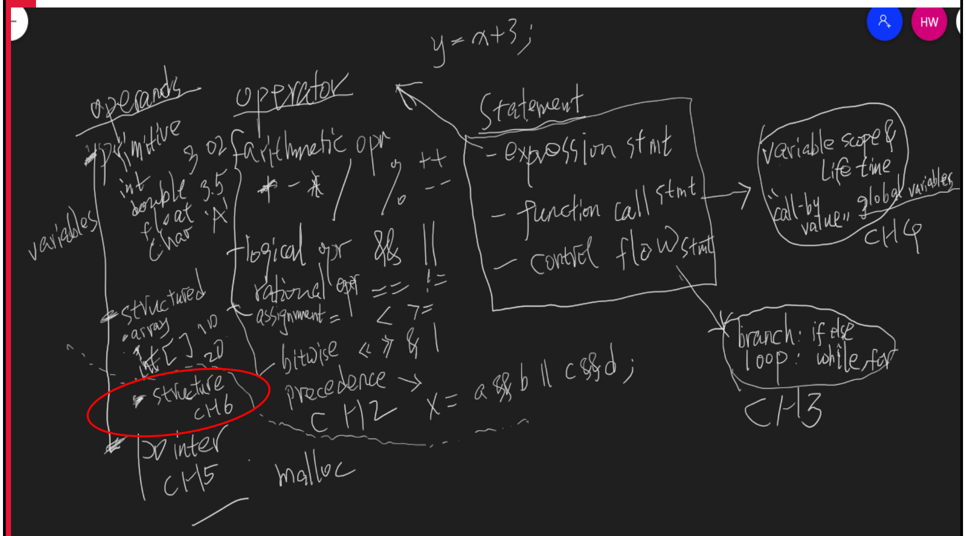
today

126

EECS2031 – Software Tools

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

127



128

YORK U UNIVERSITÉ UNIVERSITY

Structures

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

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

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

- Only contains data (no functions).

129

YORK U UNIVERSITÉ UNIVERSITY

Structures

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

130

130

Structures

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

```
- width
- height
```

is the type `// like int a;`

`// Student s;` Java

chair is variable name.

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

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

Need to repeat

131

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

```
shape table;
```

✗ typedef, later if have time

YORK UNIVERSITÉ UNIVERSITY

132

132

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

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

```
chair.width = 10;
float f = chair.height;
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	

133

Structures

access members, initialization, operations (. = &)

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

width height

chair

width	2
height	4

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

approach 2

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

```c
struct myshape {
  int data;
  float arr[3];
};
```

chair

data	2		
arr	1.5	2.5	

```c
struct myshape s = {2, {1.5, 2.5}}; //approach 1
(s.arr)[2] = 3.3;     // approach 2   set directly
```

associativity

134

Structures

access members, initialization, operations (. = &)

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

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

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

```c
chair2 = chair; /* valid. copy members value */
                /* Different from Java! */
```

- can take address as well

```c
&chair
```

Recall: Arrays cannot assign
arr2 = arr1

135 No == != > <

YORK UNIVERSITÉ UNIVERSITY

135

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

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

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

136

? What if an element is a pointer ? deep/shallow copy?

136

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

chair

width	2
height	4

chair2

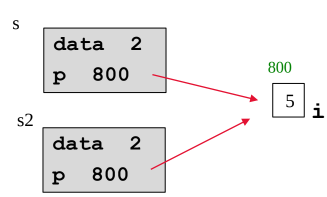
width	2
height	4

width height

```
struct shape chair = {2,4};
struct shape chair2 = chair; // copy members values
                             // different from Java
```

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

```
struct shape2 {
    float data;
    int * p;
};
```




```
int i= 5;
struct shape2 s ={2, &i};
struct shape2 s2 = s1; // s2.p = s.p
*(s2.p) = 20;

printf("%d %d", *(s.p), *(s2.p));   20 20
```

"shallow copy"

YORK U UNIVERSITÉ UNIVERSITY

137

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	,	

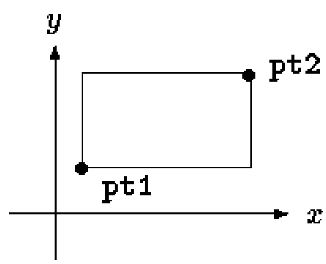
```
*s.p = 3;

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

s2.arr[2] = 3;

No () needed

(* ptr).width
```

later

138

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

YORK U UNIVERSITÉ UNIVERSITY

139

139

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`

140

140

Structures

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

141

141