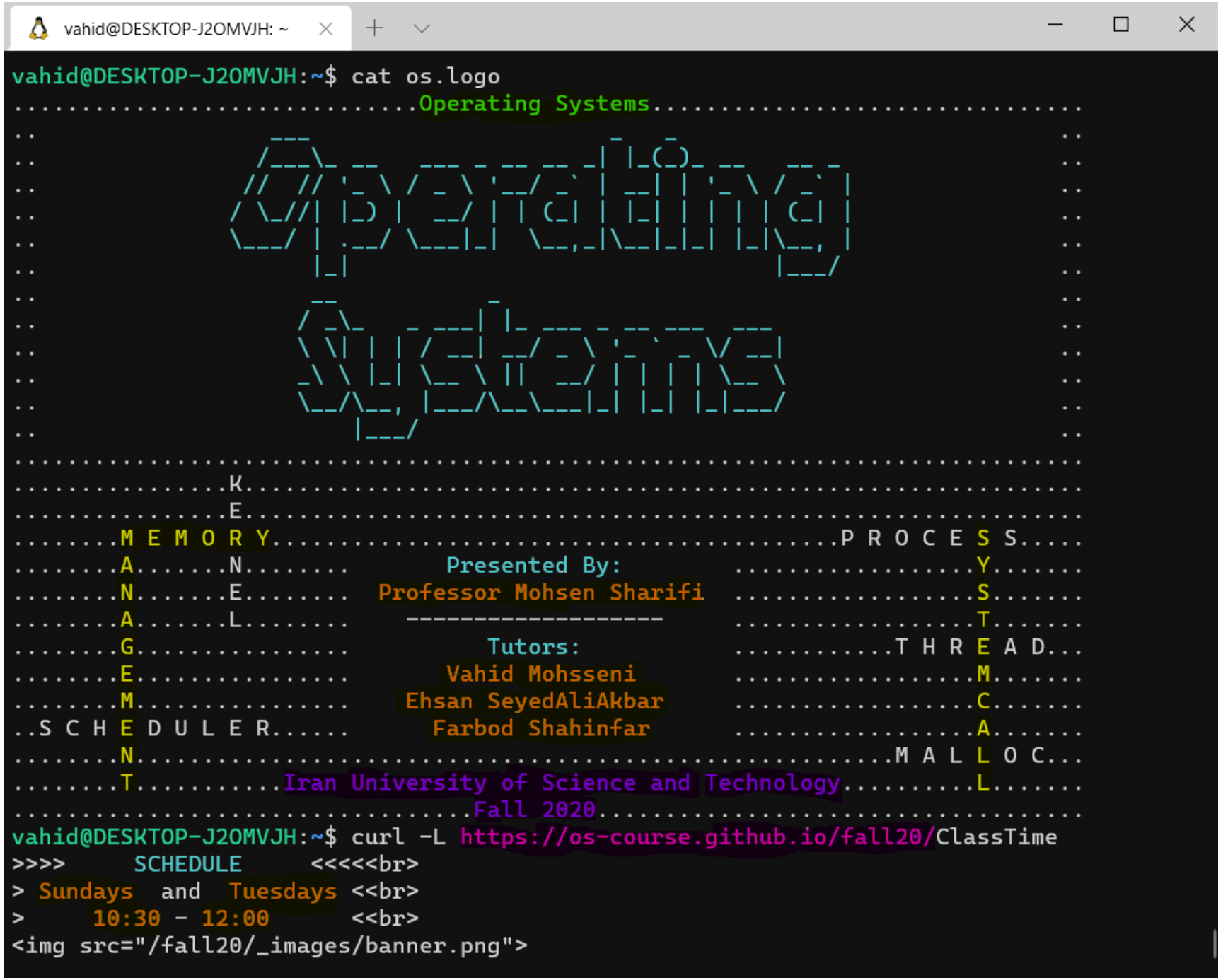
Operating Systems

Quick Introduction to C Programming Language

Farbod Shahinfar

Iran University of Science and Technology

Fall 2020


vahid@DESKTOP-J2OMVJH: ~
vahid@DESKTOP-J2OMVJH:~$ cat os.logo
Operating Systems
KERNEL
MEMORY MANAGEMENT
SCHEDULER
PROCESSYS
THREADS
MALLOC
Presented By:
Professor Mohsen Sharifi
Tutors:
Vahid Mohsseni
Ehsan SeyedAliAkbar
Farbod Shahinfar
Iran University of Science and Technology
Fall 2020
vahid@DESKTOP-J2OMVJH:~$ curl -L https://os-course.github.io/fall20/ClassTime
>>>> SCHEDULE <<<<
> Sundays and Tuesdays <
> 10:30 - 12:00 <
<img src="/fall20/_images/banner.png">

Agenda

- Brief History
- Structure of a C Program
- Data Types
- Array
- Operators
- Flow Control
- Functions
- Struct and Typedef
- Pointers
- Memory Allocation
- String Processing
- Pointer to Functions
- Header Files
- XV6 Shell

Brief History of C Programming Language

History

- The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system.
- The C programming language was created with the purpose of writing an operating system with a high level language.

History

- The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system.
- The C programming language was created with the purpose of writing an operating system with a high level language.

History

- C was influenced by B programming language.
- B programming language was the developed by Ken Thompson

Ken Thompson

Code written in B

```
main() {
    extrn putchar, n, v;
    auto i, c, col, a;

    i = col = 0;
    while(i<n)
        v[i++] = 1;
    while(col<2*n) {
        a = n+1 ;
        c = i = 0;
        while (i<n) {
            c =+ v[i] *10;
            v[i++]  = c%a;
            c =/ a--;
        }

        putchar(c+'0');
        if(!(++col%5))
            putchar(col%50?' ': '*n');
    }
    putchar('*n*n');
}
v[2000];
n 2000;
```

History: PDP-7

- Unix was developed for PDP-7 written in assembly by Ritchie and Thompson.

History: PDP-11

- Ritchie and Thompson decided to port UNIX on PDP-11
- UNIX for PDP-11 was also developed in assembly
- There was need for a programming language for developing utilities on the new platform

History: PDP-11

- Try to implement Fortran compiler
- Try to use BCPL which resulted in B
  - B was slow and not taking advantage of hardware capabilities.
- Dennis Ritchie created C (1972)

History

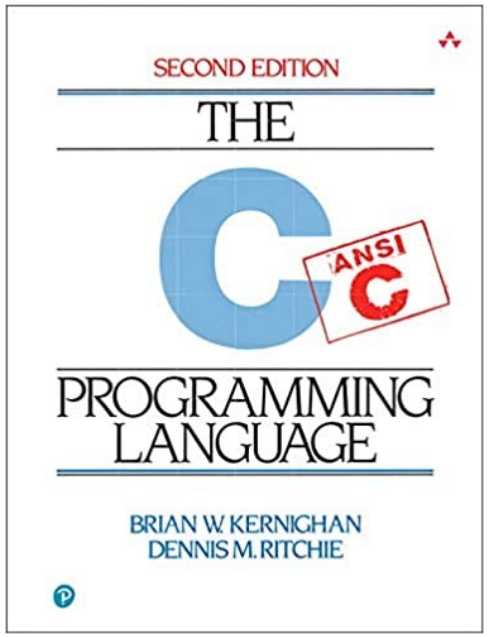
- Unix v2, had C compiler and related utility
- Unix v4 was re-implemented with C.
- Unix was one of the first operating system kernels to be implemented in a language other than assembly.

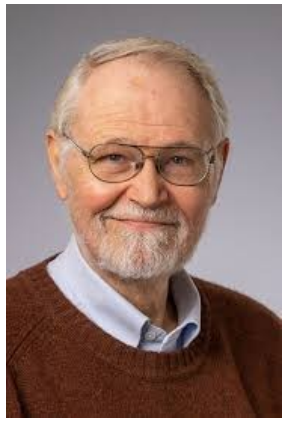
History: CD&K

Dennis Ritchie

1941 – 2011

Known for:
ALTRAN
B
BCPL
C
Multics
Unix





Brian W. Kernighan

1942 – present

Known for:
Unix
AWK
Kernighan–Lin algorithm
The C programming book

History

- During 1970 and 1980 versions of C was implemented for different types of computers so there was a need for a standard definition.
- Since then ANSI and then ISO have voted on different C standards including: C89, C99, C11, C18

Compiled or Interpreted

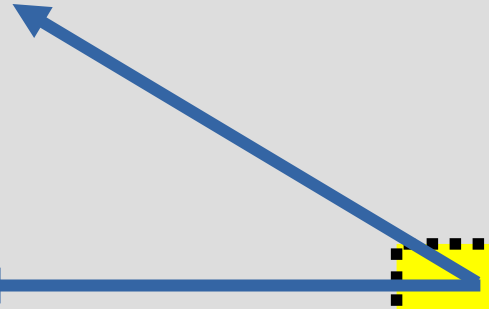
Structure of a C Program

Structure of a C Program

```c
/* adding standard input output header file to the
 * source code.
 * */
#include <stdio.h>
int main(int argc, char *argv[])
{
    // defining some variables
    int a;
    int b, c;
    a = 10;
    b = 20;
    c = a + b;

    // writing to stdout
    printf("hello world\n");
    printf("a + b = %d\n", c);

    return 0;
}
```

Structure of a C Program

```c
/* adding standard input output header file to the
 * source code.
 * */
#include <stdio.h>
int main(int argc, char *argv[])
{
    // defining some variables
    int a;
    int b, c;
    a = 10;
    b = 20;
    c = a + b;

    // writing to stdout
    printf("hello world\n");
    printf("a + b = %d\n", c);

    return 0;
}
```



Commenting, multiline and single line

Structure of a C Program

```
/* adding standard input output header file to the
 * source code.
 * */
#include <stdio.h>
int main(int argc, char *argv[])
{
    // defining some variables
    int a;
    int b, c;
    a = 10;
    b = 20;
    c = a + b;

    // writing to stdout
    printf("hello world\n");
    printf("a + b = %d\n", c);

    return 0;
}
```

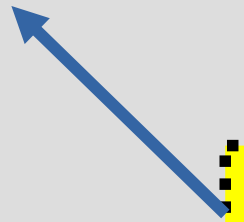
Adding header file to the source code.

#include <….> // search in the systems directories
#include "……" // can have relative path

(more on the topic of header file in future.)

Structure of a C Program

```c
/* adding standard input output header file to the
 * source code.
 * */
#include <stdio.h>
int main(int argc, char *argv[])
{
    // defining some variables
    int a;
    int b, c;
    a = 10;
    b = 20;
    c = a + b;

    // writing to stdout
    printf("hello world\n");
    printf("a + b = %d\n", c);

    return 0;
}
```



By convention the program starts from the main function.

The main function can have two variables
Int argc and char *argv[].

With help of these variable you can access the parameters passed to the program with they are called.

Structure of a C Program

```c
/* adding standard input output header file to the
 * source code.
 * */
#include <stdio.h>
int main(int argc, char *argv[])
{
    // defining some variables
    int a;
    int b, c;
    a = 10;
    b = 20;
    c = a + b;

    // writing to stdout
    printf("hello world\n");
    printf("a + b = %d\n", c);

    return 0;
}
```

A block of code is defined by { } in the C programming language.

This block determines the scope of function, variables and other statements.

Structure of a C Program

```c
/* adding standard input output header file to the
 * source code.
 * */
#include <stdio.h>
int main(int argc, char *argv[])
{
    // defining some variables
    int a;
    int b, c;
    a = 10;
    b = 20;
    c = a + b;

    // writing to stdout
    printf("hello world\n");
    printf("a + b = %d\n", c);

    return 0;
}
```

Variable definition

```
<type> <variable name> [,<variable name>];
```

Data Types

Data Types

Name	Size (bytes)
[unsigned] char	1
[unsigned] short	2
[unsigned] int	4
[unsigned] long	8
[unsigned] long long	8
[unsigned] float	4
[unsigned] double	8

- Size of data types may vary depending on compiler and its configurations.
- No boolean type but:

  #include <stdboo.h>

  defines bool, true, and false.

Data Types

- Notice, in <stdint.h> there are some useful type definitions.
  - int8_t, int16_t, int32_t, int64_t
  - uint8_t, uint16_t, uint32_t, uin64_t
  - Link: https://www.gnu.org/software/libc/manual/html_node/Integers.html

Data Types

Signed

bit: 3 2 1 0

[0][0][0][0] = 0
[0][0][0][1] = 1
[0][0][1][0] = 2
[0][0][1][1] = 3
[0][1][0][0] = 4
.
.
.
[0][1][1][1] = 7
[1][0][0][0] = -8
[1][0][0][1] = -7
[1][0][1][0] = -6
[1][0][1][1] = -5
[1][1][0][0] = -4
.
.
.

Unsigned

bit: 3 2 1 0

[0][0][0][0] = 0
[0][0][0][1] = 1
[0][0][1][0] = 2
[0][0][1][1] = 3
[0][1][0][0] = 4
.
.
.
[0][1][1][1] = 7
[1][0][0][0] = 8
[1][0][0][1] = 9
[1][0][1][0] = 10
[1][0][1][1] = 11
[1][1][0][0] = 12
.
.
.

Operators

Operators

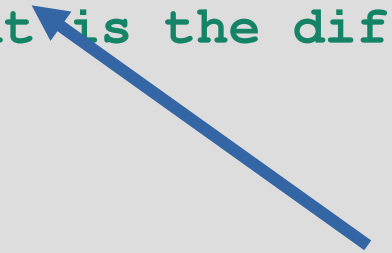
Type	Operators
Arithmetic	*, /, +, -, %, ++, --
Relational	==, !=, >, <, >=, <=
Logical	&&, ||, !
Bitwise	&, |, ^, ~, <<, >>
Assign	=, <arithmetic op>=, <bitwise op>=
Others	sizeof(), &, *, (condition) ? <value> : <value>

- Sizeof returns the number of bytes a data type requires.

Array

Array

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int values[] = {1,2,3,4};
    // int values[10] = {1,2,3,4}; // what is the difference?
    printf("values[2]: %d\n", values[2]);

    Int arr2d[4][5] = {
        {1,2,3,4},
        {5,6,7,8},
        {9,0,1,2}
    };
    return 0;
}
```



- You can access values in the array by using its index counting from 0.
- Initialization will start from index 0 and assigns values.
- Unspecified indexes are set to 0.

Array

- Sizeof an array variable evaluates to the amount of memory array has acquired.

Array: Declaring an Array

```
{
    int i, j, intArray[ 10 ], number;
    float floatArray[ 1000 ];
    int tableArray[ 3 ][ 5 ];  /* 3 rows by 5 columns */

    const int NROWS = 100;
    const int NCOLS = 200;
    float matrix[ NROWS ][ NCOLS ];
}
```

Array: Initializing Array

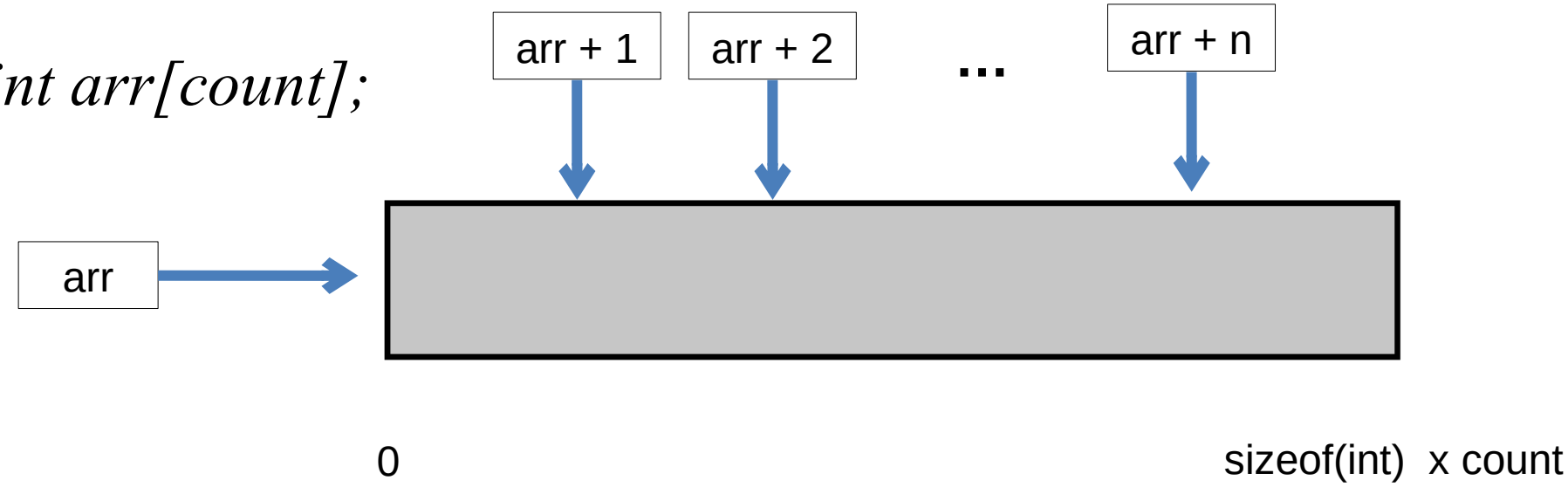
```
{
    int i =  5, intArray[ 6 ] = { 1, 2, 3, 4, 5, 6 }, k;
    float sum  = 0.0f,
                 floatArray[ 100 ] = { 1.0f, 5.0f, 20.0f };
    double  piFractions[ ] = {3.141592654,
                                    1.570796327, 0.785398163};

    int numbers[100] = {1, 2, 3, [10] = 10, 11, 12,
                                       [60] = 50, [42] = 420 };
};
```

Array

int arr[count];

Array

int arr[count];



Array: 2d Array memory layout

char mat[row][col]; // row = 3, col = 5

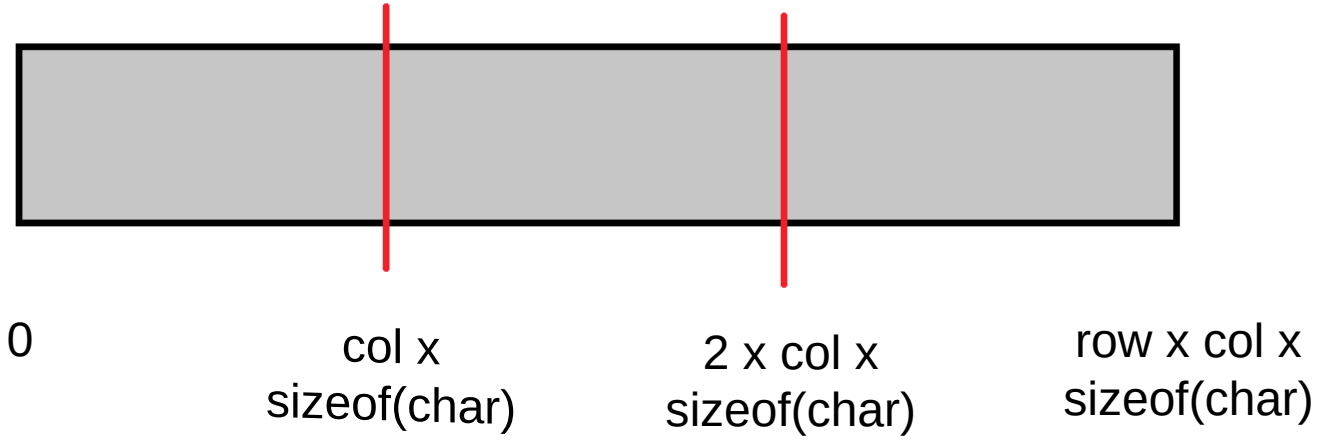
0

row x col x
sizeof(char)

* sizeof(char) = 1 Byte
* C supports row-major arrays

Array: 2d Array memory layout

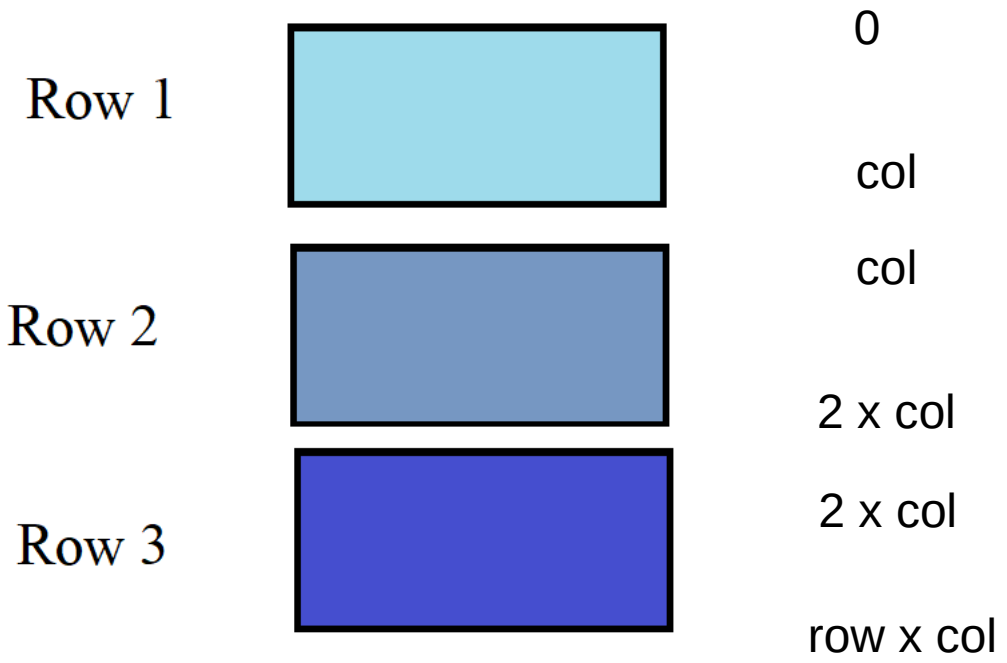
char mat[row][col]; // row = 3, col = 5



* sizeof(char) = 1 Byte
* C supports row-major arrays

Array: 2d Array memory layout

char mat[row][col]; // row = 3, col = 5



Array: 2d Array memory layout

char mat[row][col]; // row = 3, col = 5

```
row 0: 0x...020 0x...021 0x...022 0x...023 0x...024
row 1: 0x...025 0x...026 0x...027 0x...028 0x...029
row 2: 0x...02a 0x...02b 0x...02c 0x...02d 0x...02e
```

Flow Control

Flow Control: If Statements

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int temperature;
    scanf("%d\n", &temperature);
    if (temperature < 23) {
        turn_on_heater();
    } else if (temperature < 26) {
        turn_off_heater();
        turn_off_cooler();
    } else if (temperature < 40) {
        if (is_heater_enable()) {
            turn_off_heater();
        }
        turn_on_cooler();
    }
    return 0;
}
```

Flow Control: While Statements

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    unsigned char condition = 1;
    char cmd[32];
    while (condition) {
        fgets(cmd, 32, stdin);
        if (strcmp(cmd, "quit\n") == 0) {
            condition = 0;
        }
        /* execute the command and perform
         * related operations.
         * ...
         */
    }
    return 0;
}
```

Flow Control: Do-While Statements

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

int main(int argc, char *argv[])
{
    unsigned char condition = 1;
    do {
        /* do operations and related logic
         * ...
         */
    } while (condition);
    return 0;
}
```

Flow Control: For loop

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int count = 8;

    for (int i = 0; i < count; i++) {
        // ...
    }
    return 0;
}
```

Flow Control: For loop equivalent While loop

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
   /*for (int i = 0; i < count; i++) {

   }*/

    int count = 8;
    int i = 0;
    while (i < count) {
        // …
        // last instruction
        i++;
    }
    return 0;
}
```

Flow Control: For loop, Fibonacci Sequence

```c
/* Program to calculate the first 20 Fibonacci numbers. */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int i, fibonacci[ 20 ];
    fibonacci[ 0 ] = 0; fibonacci[ 1 ] = 1;
    for( i = 2; i < 20; i++ )
        fibonacci[ i ] = fibonacci[ i - 2 ] + fibonacci[ i - 1 ];
    for( i = 0; i < 20; i++ )
        printf( "Fibonacci[ %d ] = %f\n", i, fibonacci[ i ] );
}
```

Flow Control: Break and Continue

```
#include <stdio.h>
int main(int argc, char *argv[]) {
   unsigned char condition = 1;
   int array[] = {5, 6, 2, 8, 12, 19, 20, 13};
   int count = 8;
   int key = 19;
   int index = -1;
   int count_odd = 0;
   for (int i = 0; i < count; i++) {
        if (array[i] == key) {
            index = i;
            break;
        }
        if (array[i] % 2 == 0)
            continue;
        count_odd++;
   }
   return 0;
}
```

Flow Control: Switch-Case

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned char condition = 1;
    switch (value) {
        case 1:
            // ….
            break;
        case 2:
        case 3:
            // ….
            break;
        default:
            //….
    }
    return 0;
}
```

Functions

Functions

```
#include <stdio.h>
#include <stdbool.h>
bool is_even(int value){
    return value % 2 == 0;
}

int main(int argc, char *argv[])
{
   int val;
   scanf("%d\n", &val);
   if (is_even(val)) {
       printf("it is even\n");
   } else {
       printf("it is odd\n");
   }
   return 0;
}
```

Functions

```
#include <stdio.h>
void divide_by_2(int arr[], int size){
    // pass by reference
    for (int I = 0; I < size; I++) {
        arr[I] = arr[I] / 2;
    }
}

int main(int argc, char *argv[])
{
   int val;
   scanf("%d\n", &val);
   if (is_even(val)) {
       printf("it is even\n");
   } else {
       printf("it is odd\n");
   }
   return 0;
}
```

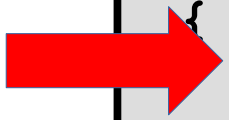
Functions: Trace a function call

```
#include <stdio.h>

void func(int a){
  int b = 10;
  return a / b;
}

int main(int argc, char *argv[])
{
    int val;
    int c;
    scanf("%d\n", &val);
    c = func(val);
    printf("%d\n" c);
    return 0;
}
```

Address	Value
0x00A1	….
0x00A2	….
0x00A3	….
0x00A4	….
0x00A5	….
0x00A6	….
0x00A7	….
0x00A8	…..
0x00A9	….

Functions: Trace a function call

```
#include <stdio.h>

void func(int a){
   int b = 10;
   return a / b;
}

int main(int argc, char *argv[])
{
    int val;
    int c;
    scanf("%d\n", &val);
    c = func(val);
    printf("%d\n" c);
    return 0;
}
```



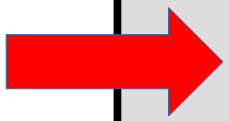
int val

int c

Address	Value
0x00A1	….
0x00A2	….
0x00A3	….
0x00A4	….
0x00A5	….
0x00A6	….
0x00A7	….
0x00A8	…..
0x00A9	….

Functions: Trace a function call

```
#include <stdio.h>

void func(int a){
   int b = 10;
   return a / b;
}

int main(int argc, char *argv[])
{
    int val;
    int c;
    scanf("%d\n", &val);
    c = func(val);
    printf("%d\n" c);
    return 0;
}
```



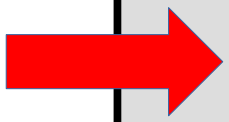
int val

int c

Address	Value
0x00A1	….
0x00A2	….
0x00A3	….
0x00A4	….
0x00A5	….
0x00A6	….
0x00A7	….
0x00A8	…..
0x00A9	….

Functions: Trace a function call

```
#include <stdio.h>

void func(int a){
   int b = 10;
   return a / b;
}

int main(int argc, char *argv[])
{
    int val;
    int c;
    scanf("%d\n", &val);
    c = func(val);
    printf("%d\n" c);
    return 0;
}
```



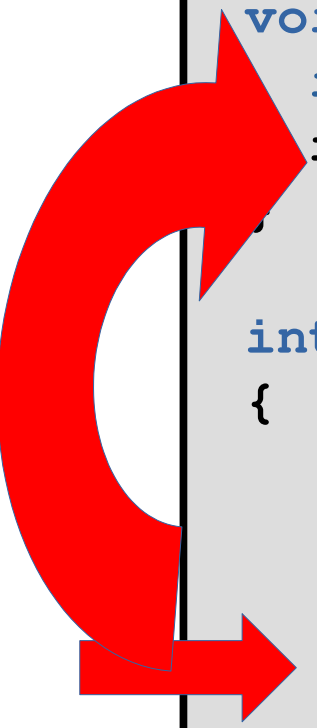
int value

int c

Address	Value
0x00A1	25
0x00A2	….
0x00A3	….
0x00A4	….
0x00A5	….
0x00A6	….
0x00A7	….
0x00A8	…..
0x00A9	….

Functions: Trace a function call

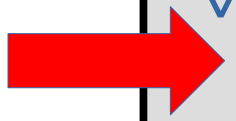
```
#include <stdio.h>

void func(int a){
   int b = 10;
   return a / b;
}


int main(int argc, char *argv[])
{
    int val;
    int c;
    scanf("%d\n", &val);
    c = func(val);
    printf("%d\n" c);
    return 0;
}
```



	Address	Value
int value	0x00A1	25
int c	0x00A2	….
	0x00A3	….
	0x00A4	….
int a	0x00A5	….
int b	0x00A6	….
	0x00A7	….
	0x00A8	…..
	0x00A9	….

Functions: Trace a function call

```
#include <stdio.h>

void func(int a){
  int b = 10;
  return a / b;
}

int main(int argc, char *argv[])
{
    int val;
    int c;
    scanf("%d\n", &val);
    c = func(val);
    printf("%d\n" c);
    return 0;
}
```

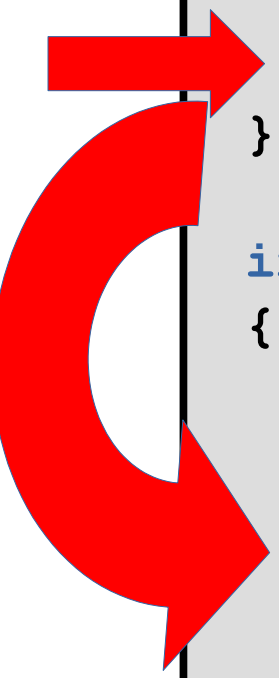


	Address	Value
int value	0x00A1	25
int c	0x00A2	….
	0x00A3	….
	0x00A4	….
int a	0x00A5	25
int b	0x00A6	10
	0x00A7	….
	0x00A8	…..
	0x00A9	….

Return register	2

Functions: Trace a function call

```
#include <stdio.h>

void func(int a){
   int b = 10;
   return a / b;
}

int main(int argc, char *argv[])
{
    int val;
    int c;
    scanf("%d\n", &val);
    c = func(val);
    printf("%d\n" c);
    return 0;
}
```

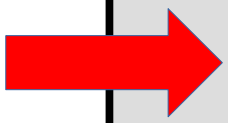


	Address	Value
int value	0x00A1	25
int c	0x00A2	….
	0x00A3	….
	0x00A4	….
int a	0x00A5	25
int b	0x00A6	10
	0x00A7	….
	0x00A8	…..
	0x00A9	….

Return register	2

Functions: Trace a function call

```
#include <stdio.h>

void func(int a){
   int b = 10;
   return a / b;
}

int main(int argc, char *argv[])
{
    int val;
    int c;
    scanf("%d\n", &val);
    c = func(val);
    printf("%d\n" c);
    return 0;
}
```



Address	Value
0x00A1	25
0x00A2	2
0x00A3	….
0x00A4	….
0x00A5	25
0x00A6	10
0x00A7	….
0x00A8	…..
0x00A9	….

int value

int c

Return register | 2

Struct and Typedef

Struct and Typedef

```
#include <stdio.h>

struct point {
    Int x;
    Int y;
};

typedef struct point point_t;

void print_point(struct point p) {
    printf("(%d, %d)\n", p.x, p.y);
}

int main(int argc, char *argv[])
{
    point_t p1 = {.x=5, .y=2};
   print_point(p1);
    return 0;
}
```

- You can define a structure to store values in a certain way.
- You can define a name for the struct.
- This is may be good for code readability and creating abstractions.

Struct and Typedef

```
struct obj_state {
    uint8_t id;
    uint8_t running;
    float prio;
    char *name[10];
};

int main()
{
    struct obj_state state1;
    return 0;
}
```

state1

Address	Value
0x00A1	10
0x00A2	120
0x00A3	….
0x00A4	….
0x00A5	25
0x00A6	10
0x00A7	0x00BC
0x00A8	…..
0x00A9	….

id

running

prio

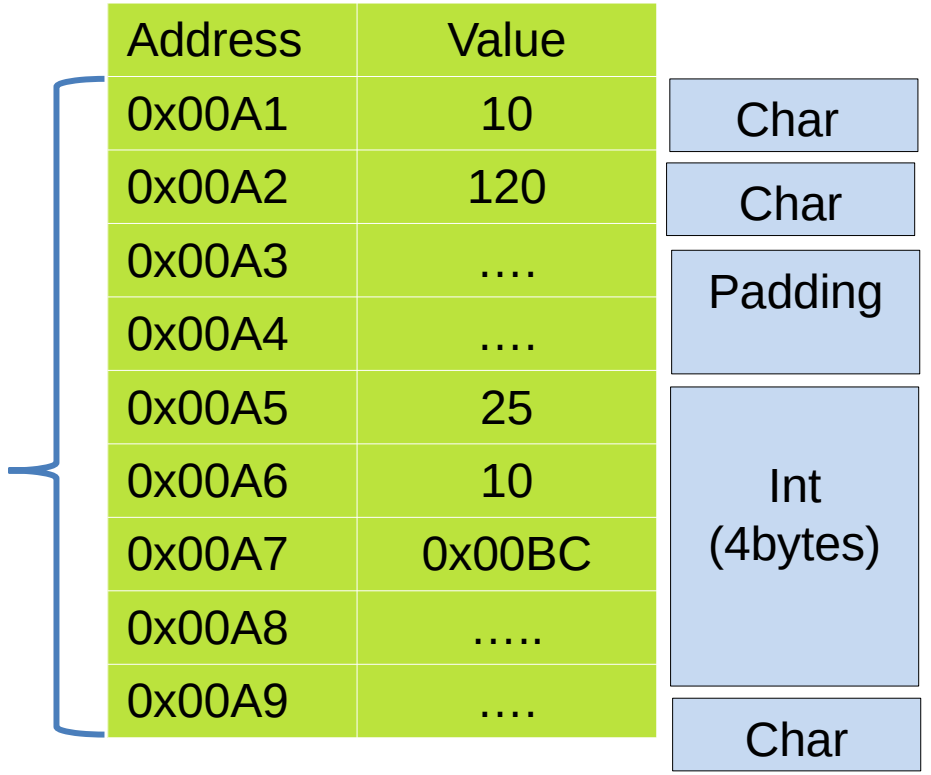
name

Struct and Typedef

- *Fields of a struct may not be contiguous because compiler may add padding for performance purposes.*

```
struct begin address: 0x...150
a: 0x...150 (expected: 0x...150)
b: 0x...151 (expected: 0x...151)
c: 0x...154 (expected: 0x...152)
d: 0x...158 (expected: 0x...156)
```

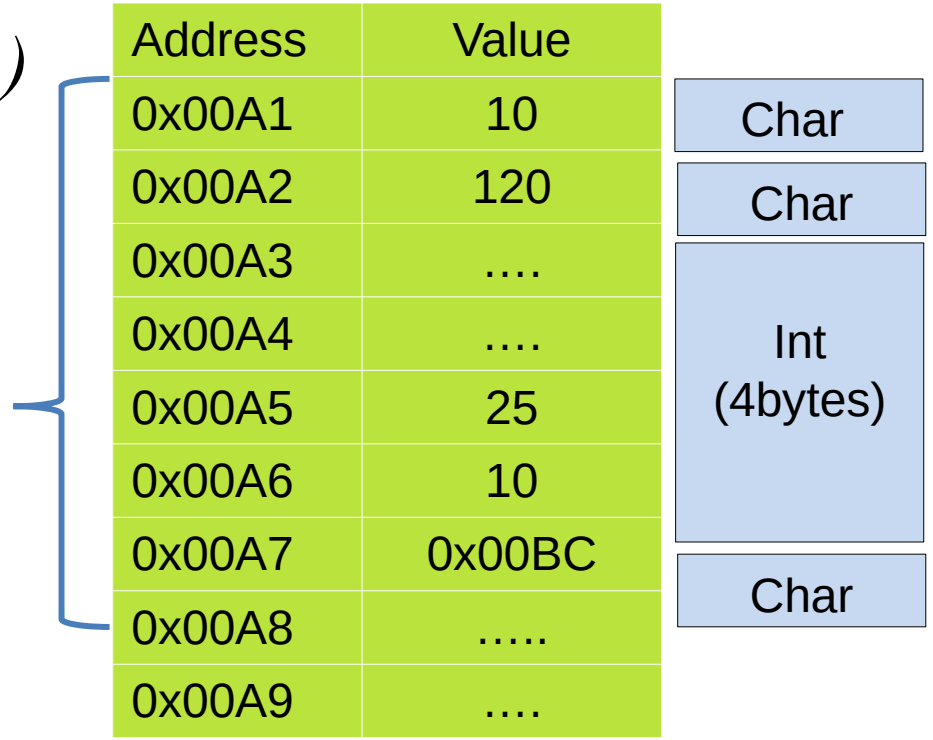
```
struct {
    char a,
    char b,
    int c,
    char d
};
```



Struct and Typedef

- *It is possible to give instructions to compiler not to add padding*
  - *For GCC __attribute__((__packed__))*

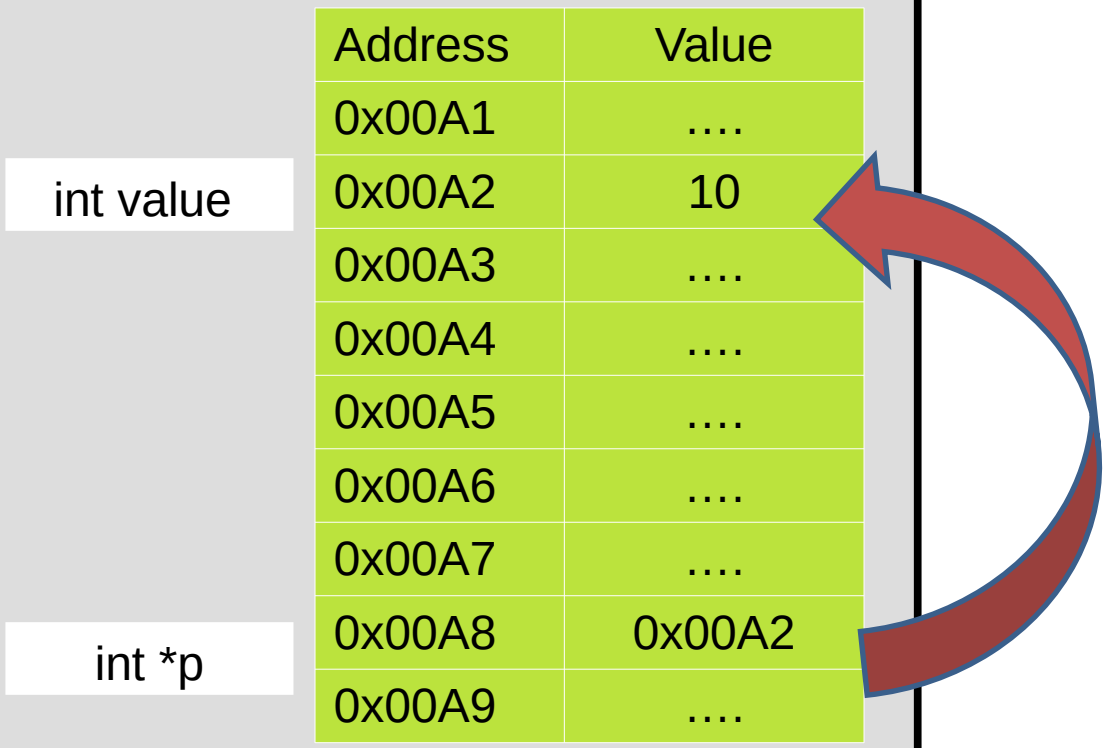
```
struct begin address: 0x...9c0
a: 0x...9c0 (expected: 0x...9c0)
b: 0x...9c1 (expected: 0x...9c1)
c: 0x...9c2 (expected: 0x...9c2)
d: 0x...9c6 (expected: 0x...9c6)
```



Pointers

Pointers

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int value = 10;
    int *p;
    p = &value;

    printf("value is: %d, "
           "(address: %x)\n",
            *p, p);
    return 0;
}
```



Pointers

```
#include <stdio.h>

struct rectangle { int width; int height; point_t top_left;};

void print_rect(struct rectangle *p) {
    printf("<w: %d, h: %d, x: %d, y: %d>\n",
           p->width, p->height, p->top_left.x,
           p->top_left.y);
}

int main(int argc, char *argv[]) {
   point_t p1 = {.x=2, .y=-3};
   struct rectangle r1 = {.width=10, .height=5, top_left=p1};

   print_rect(&r1);
   return 0;
}
```

- Pass the address of the structure to the function.
- Reduces memory copy.

Pointers

```
#include <stdio.h>

struct rectangle { int width; int height; point_t top_left;};

void print_rect(struct rectangle *p) {
    printf("<w: %d, h: %d, x: %d, y: %d>\n",
           p->width, p->height, p->top_left.x,
           p->top_left.y);
}

int main(int argc, char *argv[]) {
   point_t p1 = {.x=2, .y=-3};
   struct rectangle r1 = {.width=10, .height=5,

   print_rect(&r1);
   return 0;
}
```

- p->top_left.x
- (*p).top_left.x
- get the struct from address pointed to by `p` and select `top_left` member of the struct.

Pointers

Both instructions below are equivalent:

- p→width = 10;
- (*p).width = 10;

Pointers: Arithmetic

- When incrementing a pointer the address is changed with respect to the size of data type of the pointer.

```
{
    int64_t val = 10;
    int64_t *p64 = &val;
    printf("p64:%x,%x\n", p64, p64+1);

    int8_t *p8 = (int8_t *)(&val);
    printf("p8: %x, %x\n", p8, p8+1);
    return 0;
}
```

- P8 moved 1 byte
- P64 moved 8 bytes

```
p64, p64+1: 4a7d3c60, 4a7d3c68
p8,  p8+1:  4a7d3c60, 4a7d3c61
```

Pointers: sizeof()

- Size of an pointer is the address size:
  - On a 32 bit system sizeof(*p) == 4
  - On a 64 bit system sizeof(*p) == 8

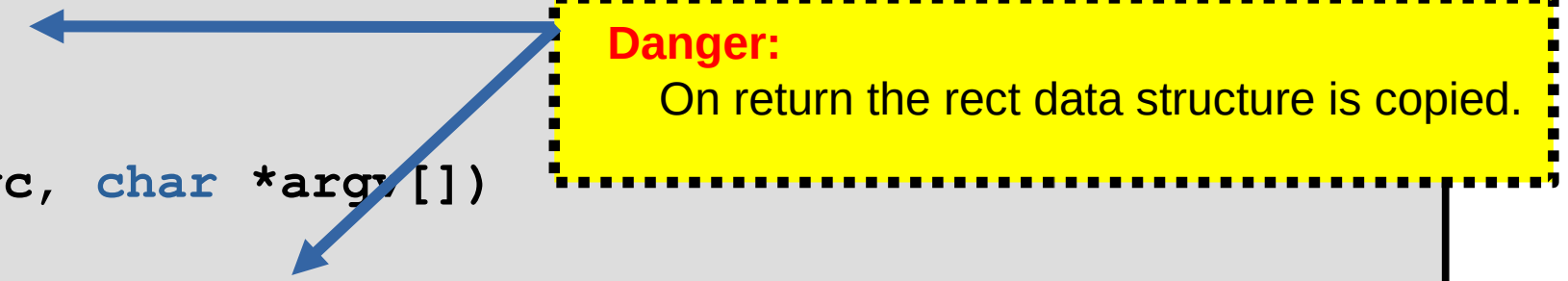
Memory Allocation

Memory Allocation

- Local variables are allocated from stack memory.
  - Local variables are freed when they are out of scope (for example function return)
- Allocating memory with `malloc` or `calloc` uses heap memory.
  - Memory should be explicitly freed using `free` function.

Memory Allocation

```
#include <stdio.h>
struct rectangle { int width; int height; point_t top_left;};

struct rectangle new_rect(int w, int h) {
    struct rectangle rect;
    rect.width = w;
    rect.hight = h;
    return rect;
}

int main(int argc, char *argv[])
{
   struct rectangle rect = new_rect(10, 5);
   // do some processing
   free(rect);
   return 0;
}
```



Memory Allocation

```
#include <stdio.h>

struct rectangle { int width; int height; point_t top_left;};

struct rectangle *new_rect(int w, int h) {
     struct rectangle rect;
     rect.width = w;
     rect.hight = h;
     return &rect;
}

int main(int argc, char *argv[])
{
    struct rectangle *rect = new_rect(10, 5);
    // do some processing
    free(rect);
    return 0;
}
```

Danger:
On return the context of the function is destroyed and, the returned pointer is invalid

Memory Allocation

```
#include <stdio.h>

struct rectangle { int width; int height; point_t top_left;};

struct rectangle *new_rect(int w, int h) {
    struct rectangle *rect = \
        malloc(sizeof(struct rectangle *));
    rect→width = w;
    rect→hight = h;
    return rect;
}

int main(int argc, char *argv[])
{
   struct rectangle *rect = new_rect(10, 5);
   // do some processing
   free(rect);
   return 0;
}
```

Allocate memory from heap

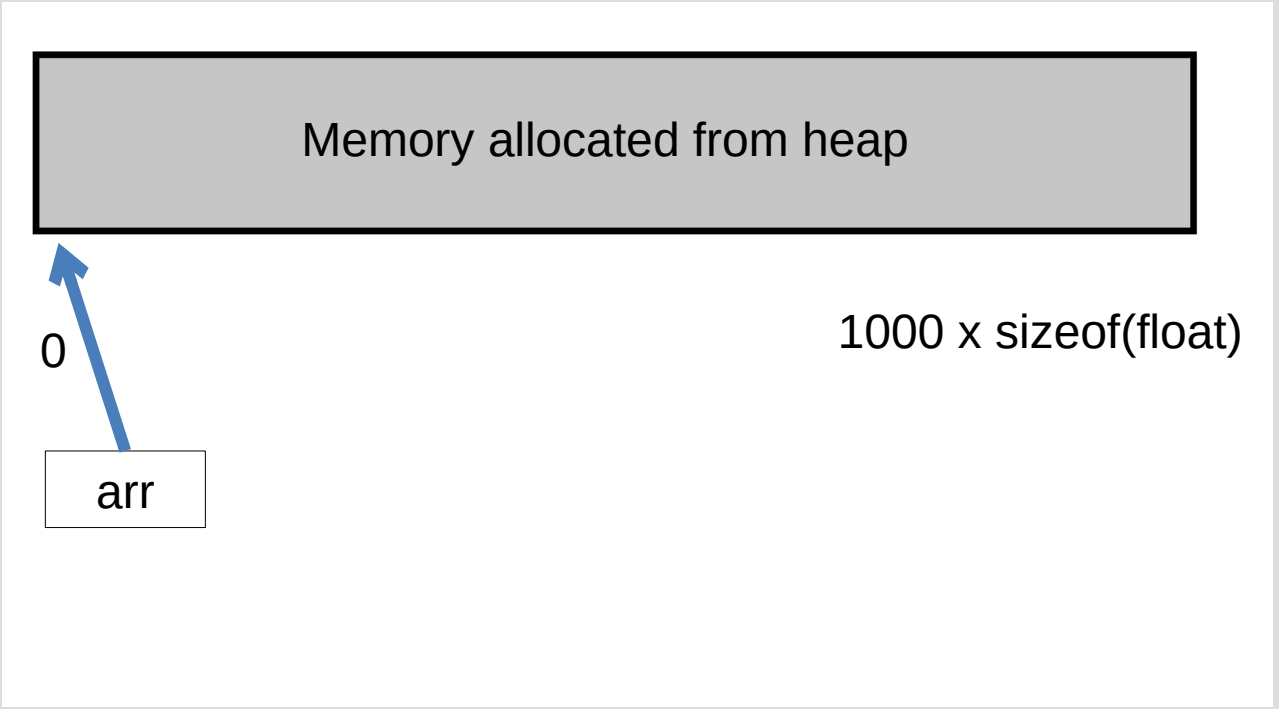
Pointers Revisited

Pointers Revisited: Pointer to Pointer

```
void new_rect(struct rectangle **p) {
    struct rectangle *r;
    r = malloc(sizeof(struct rectangle));

    *r = (struct rec..) {
            .width = 1,
            .height = 2,
            .top_left = (point_t) {.x=3, .y=4},
    };
    *p = r;
}

int main(int argc, char *argv[]) {
    struct rectangle *r1 = NULL;
    new_rect(&r1);

    print_rect(&r1);
    return 0;
}
```

Pointers Revisited: Allocate array from heap

```
int main(int argc, char *argv[]) {
    float *arr;
    arr = malloc( 1000 * sizeof(*arr));

    for (int i = 0; i < 1000; i++)
        arr[i] = 3.14;
    return 0;
}
```

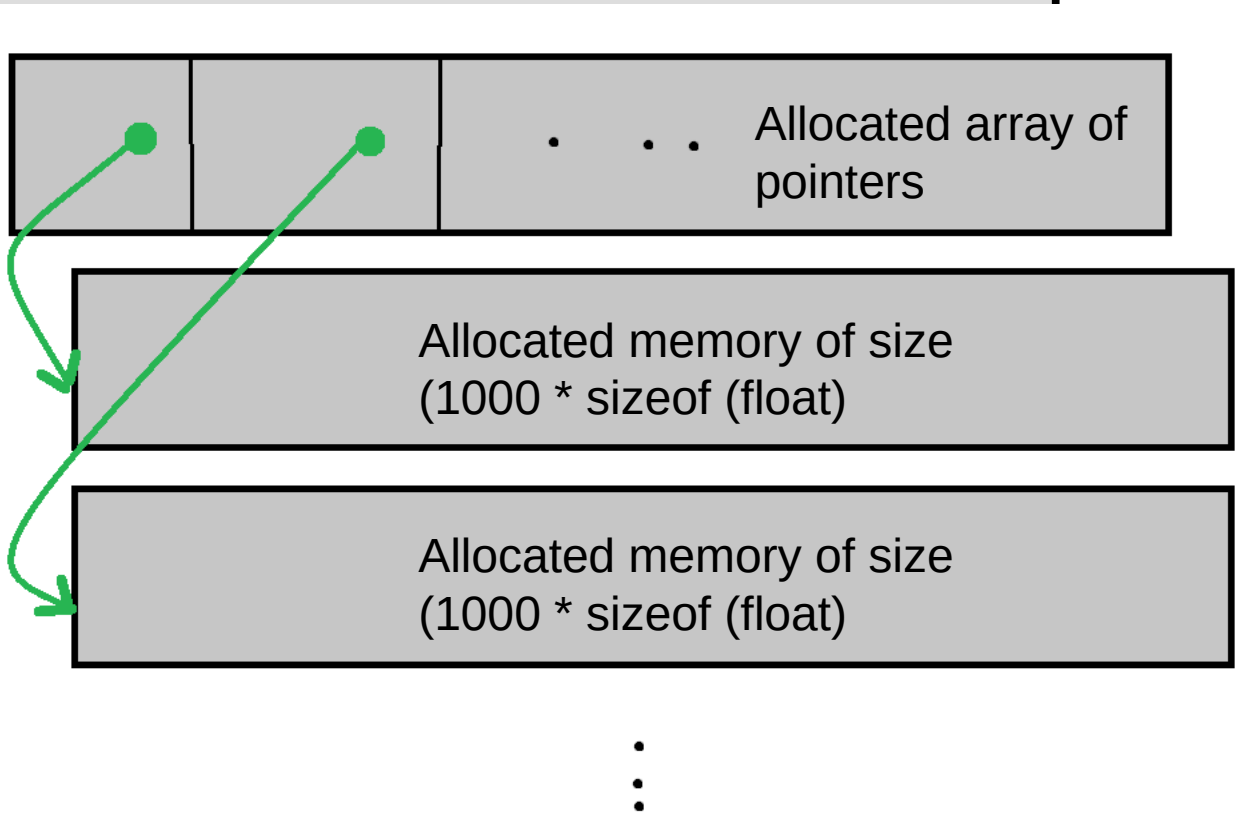


Pointers Revisited: Allocate 2d arrays

```
int main(int argc, char *argv[]) {
    // mat [50][1000]
    float **arr;
    arr = malloc ( 50 * sizeof(float *));

    for (int i = 0; i < 50; i++)
        arr[i] = malloc( 1000 * sizeof(float));

    for (int i = 0; i < 50; i++)
        for (int j = 0; j < 1000; j++)
            arr[i][j] = 3.14;

    return 0;
}
```

Pointers Revisited: Allocate 2d arrays

```
int main(int argc, char *argv[]) {
  // mat [50][1000]
  float **arr;
  arr = malloc ( 50 * sizeof(float *));

  for (int i = 0; i < 50; i++)
    arr[i] = malloc( 1000 *
                sizeof(float));
```



String Processing

String Processing

- *Strings are an array of characters*
  - *char str[100];*
  - *char *str = malloc(....);*
- *The end of string is usually determined by '\0'*
  - *It is called null-terminated string*

String Processing

- \n
- \r
- \t
- \0

String Processing

- *Header file <string.h>*
- *size_t strlen(const char *s);*
- *size_t strnlen(const char *s, size_t maxlen);*

String Processing

- *Header file <string.h>*
- *char *strcpy(char *dest, const char *src);*
- *char *strncpy(char *dest, const char *src, size_t n);*

String Processing

- *Header file <stdlib.h>*
- *int atoi(const char *nptr);*
- *long atol(const char *nptr);*
- *long long atoll(const char *nptr);*

String Processing

- *Header file <stdio.h>*
- *int scanf(const char *format, ...);*
- *int sscanf(const char *str, const char *format, ...);*

String Processing

- *Header file <stdio.h>*
- *int printf(const char *format, ...);*
- *int sprintf(char *str, const char *format, ...);*
- *int snprintf(char *str, size_t size, const char *format, ...);*

String Processing

- *%d: integer*
- *%ld: long*
- *%s: string*
- *%x: hex*
- *%p: pointer*

Pointer to Function

Pointer to Function

- To define a variable having type of pointer to a function:
  - <function return type> (*<variable name>)(<list of input parameters>)
  - int (*count_even)(int arr[], int count)
- typedef can be used to define a type and create abstraction

Pointer to Function

```
typedef int(*on_btn_clk_t)(struct event*);

int my_func(struct *event) {
  // …
  return 0;
}

int main(void)
{
   on_btn_clk_t _func = &my_func;
   // …
   if (condition) {
     _func(ev);
   }
   exit();
}
```

XV6 Shell

XV6 Shell

- XV6 is a UNIX like operating system implemented for educational purposes by MIT students.
- Last session we examined how an operating system boots. In this section we assume that operating system has been booted and the kernel is ready. We focus on the shell program letting users to interact with the system.

XV6 Shell

```
int
main(void)
{
    // ….
    exit();
}
```

- By convention starts from main function.

XV6 Shell

```
int
main(void)
{
  static char buf[100];
  int fd;
  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }
  // ….
  exit();
}
```

- Make sure at least three file descriptors are open
- 0: stdin
- 1: stdout
- 2: stderr

XV6 Shell

```
int
main(void)
{
  static char buf[100];
  int fd;
  // ….
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Chdir must be called by the parent, not the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        printf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));
    wait();
  }
  exit();
}
```

- Read a command and execute…

XV6 Shell

```
int
getcmd(char *buf, int nbuf)
{
  printf(2, "$ ");
  memset(buf, 0, nbuf);
  gets(buf, nbuf);
  if(buf[0] == 0) // EOF
    return -1;
  return 0;
}
```

Questions?

?