Introduction to C

Programming Workshop in C (67316)
Fall 2018
Lecture 6
6.11.2018

Exam questions

Malloc questions - reverse

read in integer for string length
read in string
reverse and print it

```
int main()
{
    int length = 0;
    scanf("%d", &length);
    char * str = (char*)malloc(length + 1);
    scanf("%s", str);

    for (int i = length - 1; i >= 0; --i)
    {
        putchar(str[i]);
    }

    free(str);
}
```

Lists

Linked List

```
typedef struct Node {

…
} Node;


Node *head = NULL; // global


void push(int new_data);

void printList();

void deleteList();
```

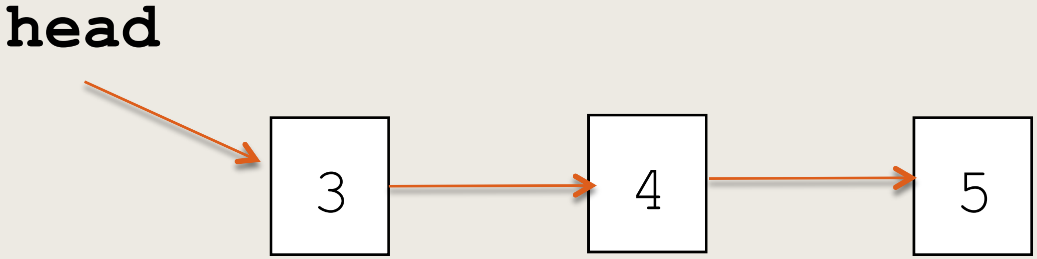
Linked List

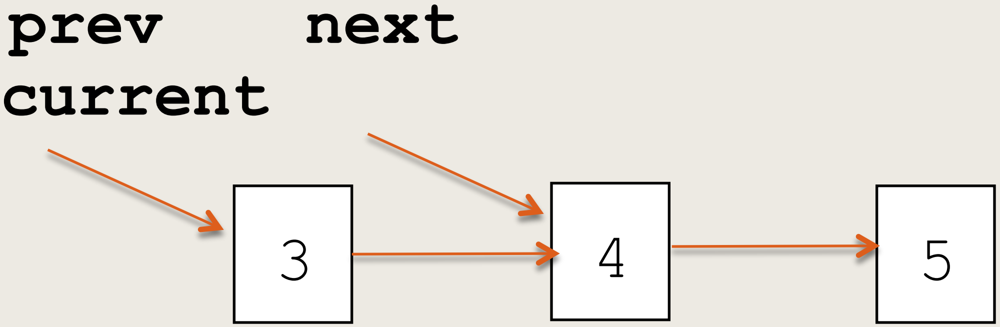
```
typedef struct Node {
  int data;
  struct Node* next;
} Node;


Node *head = NULL; // global


void push(int new_data) {
  Node* new_node =
   (Node*)malloc(sizeof(Node));
  new_node->data = new_data;
  new_node->next = head;
  head = new_node;
}
```

```
void printList() {
  Node *temp = head;
  while(temp != NULL) {
    printf("%d  ", temp->data);
    temp = temp->next;
  }
  printf("\n");
}

void deleteList() {
  Node *temp = head;
  while(temp != NULL) {
    Node *next = temp->next;
    free(temp);
    temp = next;
  }
  head = NULL;
}
```

Linked List - Reverse (exam question)

```
void reverse()
{
  Node* prev = NULL;
  Node* current = head;
  Node* next;
  while (current != NULL)
    {
      next  = current->next;
      current->next = prev; // flip the pointer
      prev = current;
      current = next;
    }
  head = prev;
}
```





Linked List - how to do non-global head?

```
typedef struct Node {
  int data;
  struct Node* next;
} Node;


Node *head = NULL; // global


void push(int new_data), Node *head )
{
  Node* new_node =
   (Node*)malloc(sizeof(Node));
  new_node->data = new_data;
  new_node->next = head;
  head = new_node;
}
```

Linked List - how to do non-global head?

```
void printList(Node *head) {
  Node *temp = head;
  while(temp != NULL) {
    printf("%d  ", temp->data);
    temp = temp->next;
  }
  printf("\n");
}

void deleteList(Node *head) {
  Node *temp = head;
  while(temp != NULL) {
    Node *next = temp->next;
    free(temp);
    temp = next;
  }
  head = NULL;
}
```

```
int main()
{
  Node *head = NULL;

  push(1, head);
  push(2, head);
  push(3, head);
  push(4, head);

  printList(head);
}
```

What will be printed?

Reminder – the swap function

Does nothing

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
int main()
{
    int x, y;
    x = 3; y = 7;
    swap(x, y);
    // now x==3, y==7
}
```

Works

```
void swap(int *pa, int *pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
int main()
{
    int x, y;
    x = 3; y = 7;
    swap(&x, &y);
    // x == 7, y == 3
}
```

the swap problem with pointers

Does nothing

```
void printList(Node *head) {
  Node *temp = head;
  while(temp != NULL) {
    printf("%d  ", temp->data);
    temp = temp->next;
  }
  printf("\n");
}

int main() {
  Node *head = NULL;
  push(1, head);
  push(2, head);
  push(3, head);
  push(4, head);

  printList(head);
}
```

Works

```
void printList(Node **head) {
  Node *temp = *head;
  while(temp != NULL) {
    printf("%d  ", temp->data);
    temp = temp->next;
  }
  printf("\n");
}

int main() {
  Node *head = NULL;
  push(1, &head);
  push(2, &head);
  push(3, &head);
  push(4, &head);

  printList(&head);
}
```

push is also modified

Does nothing

```c
void push(int new_data,
          Node *head )
{
  Node* new_node =
   (Node*)malloc(sizeof(Node));
  new_node->data = new_data;
  new_node->next = head;
  head = new_node;
}
```

Works

```c
void push(int new_data,
          Node **head )
{
  Node* new_node =
   (Node*)malloc(sizeof(Node));
  new_node->data = new_data;
  new_node->next = *head;
  *head = new_node;
}
```

Multi-dimensional arrays

Array of pointers, pointers to arrays

Multi-dimensional arrays

Static:

```
int arr[5][7]; // 5 rows, 7 columns
```

- Continuous memory: “array of arrays” (divided to 5 blocks of 7 ints)
- Size must be known at compile time
- Efficient: one memory access to reach an index

Semi-dynamic:

```
int *arr[5]; // array of 5 pointers to int
```

- Each row is in a different location
- Number of rows must be known at compile time
- Less efficient: two memory access to reach an index

Fully dynamic:

```
int **arr; // pointer to pointer to int
```

- Each row is in a different location
- Size may be unknown at compile-time
- Even less efficient: three memory access to reach an index

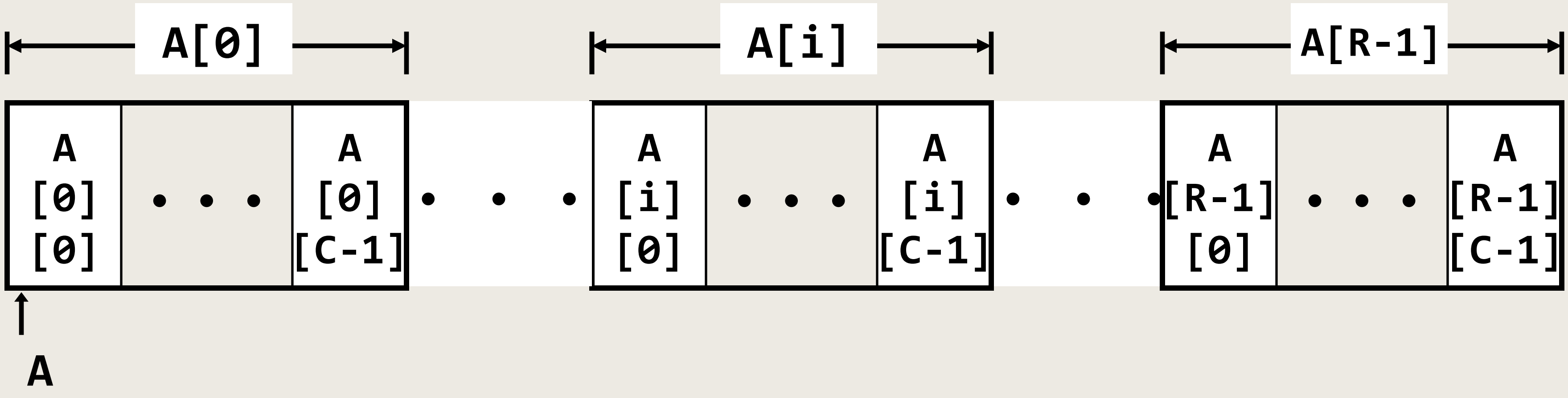
Static 2D array

```
int A[R][C];
```

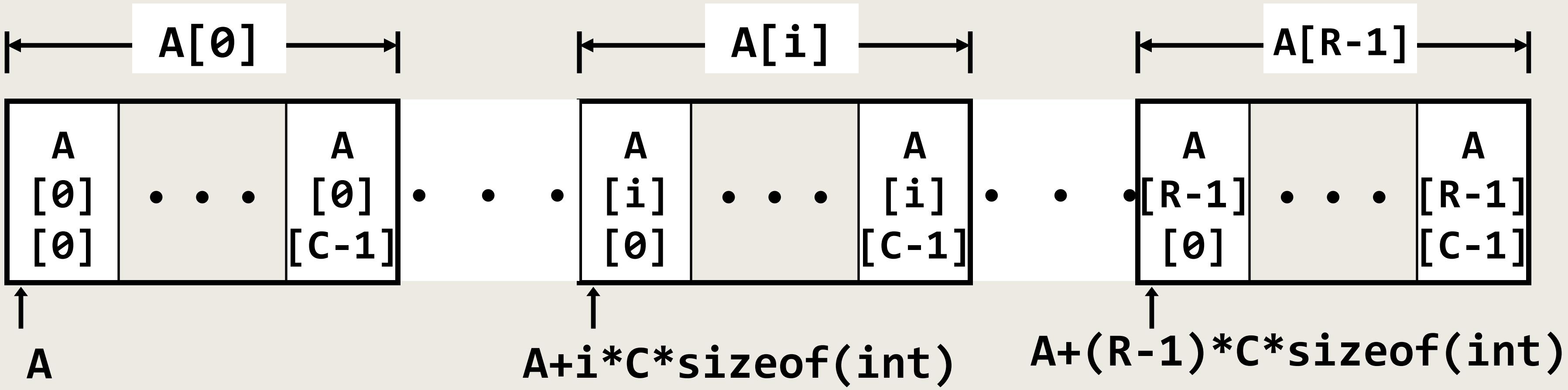
matrix representation

A[0][0]	A[0][1]	…	A[0][c-1]
…	…	…	…
A[i][0]	A[i][1]	…	A[i][c-1]
…	…	…	…
A[R-1][0]	A[R-1][1]	…	A[R-1][C-1]

Row-major ordering →



Static 2D array



```
int A[R][C];

A[i][j]=5; // → put 5 in:
           //     A + (i*C + j)*sizeof(int)

// C is the size of each row
```

Semi-dynamic arrays – array of pointers

```
int *pa[5]; // allocates memory for 5 pointers
for (i=0; i<5; i++)
{
    pa[i] = (int*) malloc( 7*sizeof(int) );
    // pa[i] now points to a memory of 7 ints
    // note that we can allocate
    // different size for each row
}

pa[i][j] = 5;
```

1. Go to `pa + i*sizeof(int*)` and take its value `val` – this is the i‘th row start address
2. Put 5 in `val + j*sizeof(int)`

Semi-dynamic arrays – array of pointers

```
pa[i][j] = 5;
```

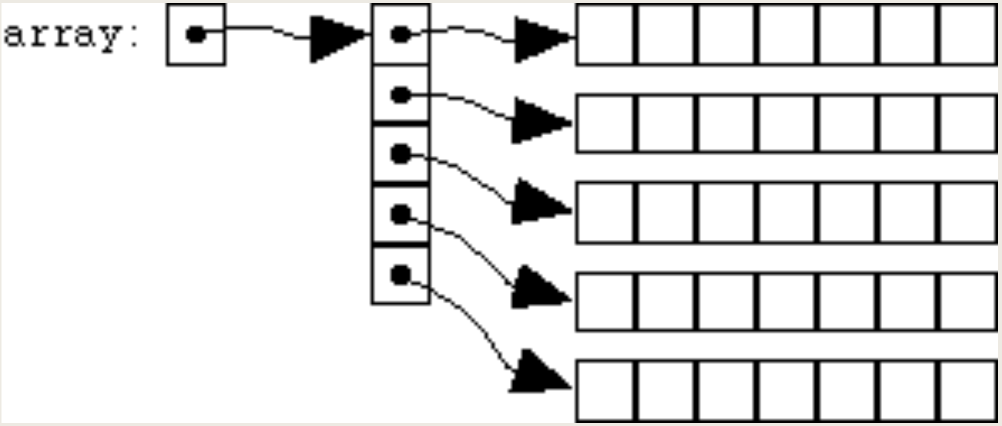
1. Go to `pa + i*sizeof(int*)` and take its value `val` – this is the i‘th row start address
2. Put 5 in `val + j*sizeof(int)`

With pointer arithmetic:

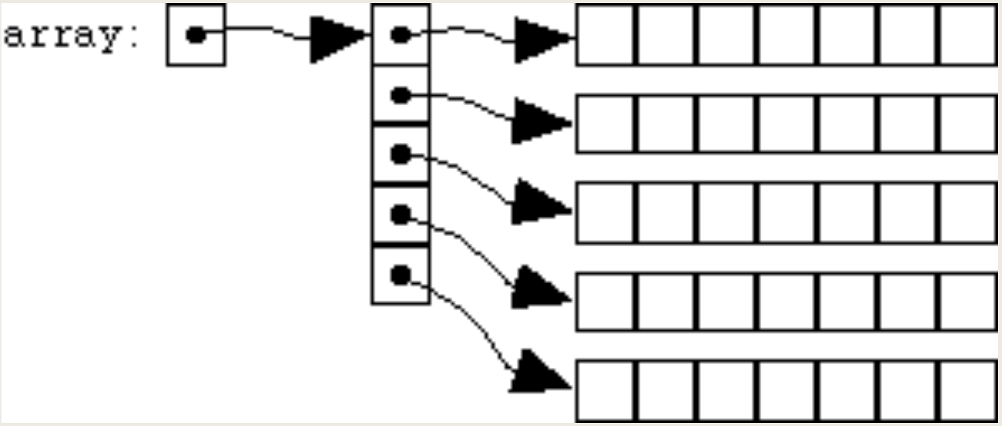
```
*(pa[i]+j)   = 5;
(*(pa+i))[j] = 5;
*(*(pa+i)+j) = 5;
```

Fully dynamically allocated arrays

```
int ** array;
array = (int**)malloc(5*sizeof(int*));
for (i=0; i<5; i++)
{
   array[i] = (int*)malloc(7*sizeof(int));
}
```



Fully dynamically allocated arrays



```
array[i][j] = 5;
```

1. Go to array and take its start address value `v1`
2. Go to the address `v1 + i*sizeof(int*)` and take its value `v2` `(i’th row start address)`
3. Put 5 in `v2 + j*sizeof(int)`

Fully dynamically allocated arrays

```
array[i][j] = 5;
```

1. Go to array and take its start address value `v1`
2. Go to the address `v1 + i*sizeof(int*)` and take its value `v2` `(i'th row start address)`
3. Put 5 in `v2 + j*sizeof(int)`

With pointers arithmetic (same as semi-dynamic):

```
*(array[i]+j)    = 5;
(*(array+i))[j] = 5;
*(*(array+i)+j) = 5;
```

Dynamically Multi-dimensional arrays

Semi/Full dynamically allocated multi-dimensional array:

- Memory not continuous
- Each row can have different size
- **Access:** `arr[i][j]`

Dynamically Multi-dimensional arrays

- Don’t forget to free all the memory – one `free` call for each one `malloc` call

  e.g., for full dynamically allocated 2D array:

```
for (i = 0; i < nrows; i++)
{
    free( array[i] );
    array[i] = NULL;
}
free( array );
array = NULL;
```

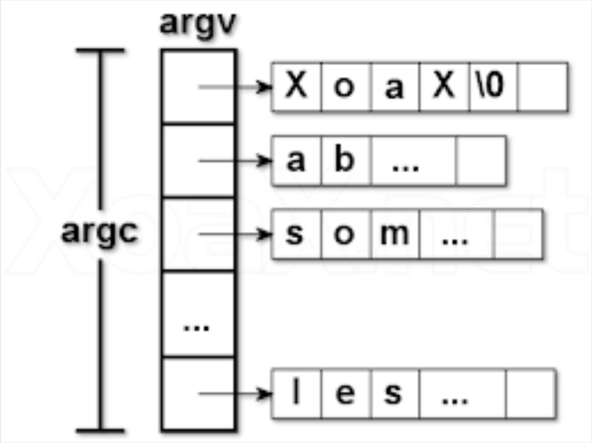
Passing arguments to a program with argc and argv

argc

- stands for “**arg**ument **c**ount”
- contains the number of arguments passed to the program

argv

- stands for “**arg**ument **v**ector”
- array of strings



```
> myprog 1 2 3
```

```
argc = 4 (program name is the first)
argv[0] => "myprog"
argv[1] => "1"
argv[2] => "2"
argv[3] => "3"
```

Passing arguments to a program with argc and argv

```c
int main(int argc, char *argv[])
{
   for(int i=0; i<argc; i++)
   {
     printf("%s ", argv[i]);
   }

   // prints the first character of program name
   printf(“%c”, argv[0][0]);

}
```

More efficient memory arrangement

Instead of allocating `int **arr`,
allocate `int *arr`

```
int *arr =(int*)malloc(5*7*sizeof(int))
```

- **Access:** `arr[i][j] -> arr[i*ncols + j]`
  - faster memory access
  - easier (and more efficient) implementation of iterators
- But:
  - less readable code (can partially hide with macro)

pointers to pointers to …

We also have pointers to pointers to pointers, etc.:

```
double ** mat1 = getMatrix();
double ** mat2 = getMatrix();
//allocate an array of matrices
double *** matrices =
(double***) malloc(n*sizeof(double**));
matrices[0] = mat1;
matrices[1] = mat2;
```

Multi-dimensional arrays

Sending array to functions:

- `void func( int x[5][7] ) //ok`
- `void func( int x[][7] )  //ok`

- `void func( int x[][] )   //error`

- `void func( int * x[] )   //something else`
- `void func( int ** x )    //same something else`

Pointers to arrays

```
int foo (char arr_a[][20]);
int bar (char arr_b[20]);
```

arr_a is a pointer to an array of 20 chars

arr_b is a pointer to a char

Therefore:

```
sizeof (arr_a) = sizeof (void*);
sizeof (*arr_a) = 20 * sizeof (char);
sizeof (arr_b) = sizeof (void*);
sizeof (*arr_b) = sizeof (char);
```