UNSW SYDNEY | Australia's Global University

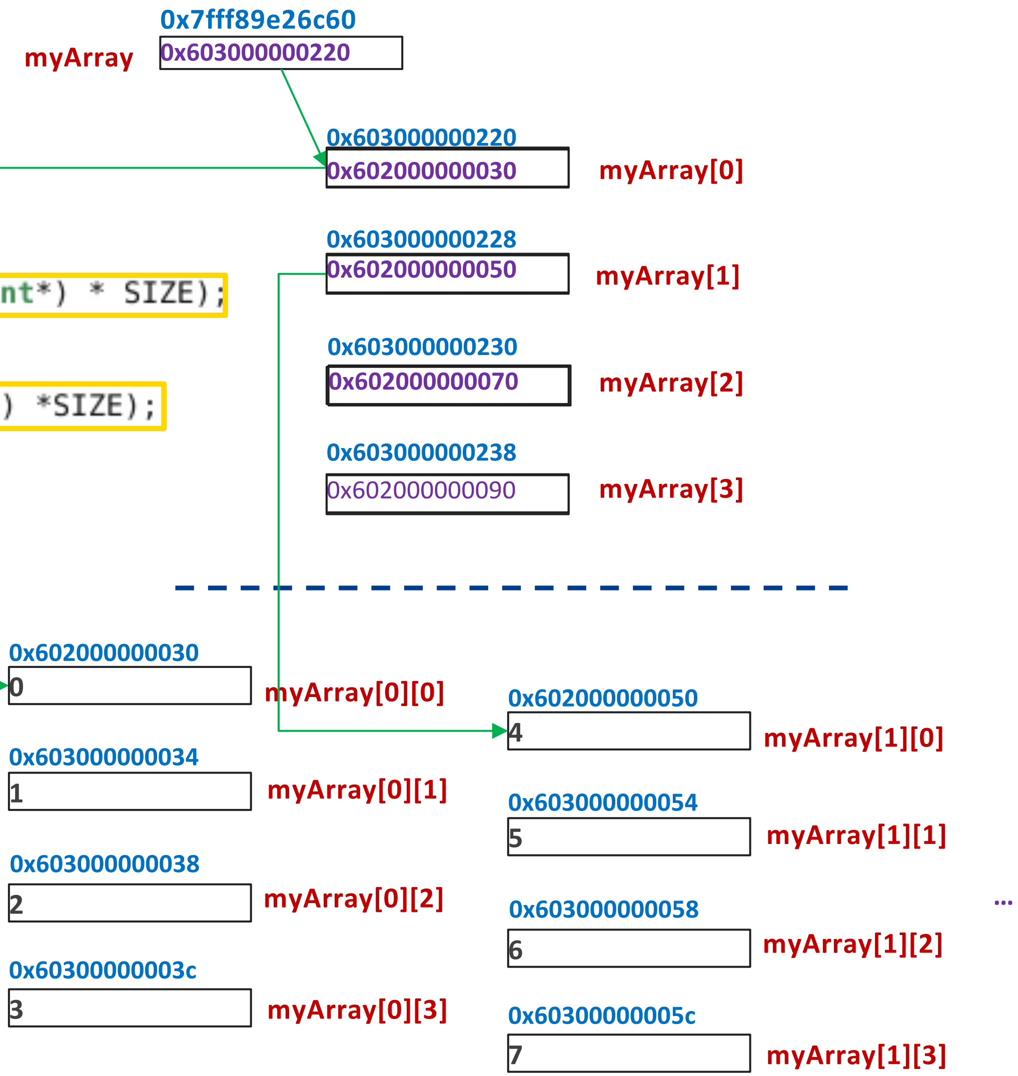
COMP1911 - Computing 1A

2D malloc

```c
#define SIZE 4

void printArray(int **arr);

int main(int argc, char *argv[]) {
    char *myString = argv[1];
    int **myArray = (int **) malloc(sizeof(int*) * SIZE);
    int i=0, j;
    while(i < SIZE){
        myArray[i] = (int*)malloc(sizeof(int) *SIZE);
        j=0;
        while(j < SIZE) {
            int c = myString[i*SIZE+j] - 48;
            myArray[i][j] = c;
            j++;
        }
        i++;
    }
    printArray(myArray);
    i = 0;
    while(i < SIZE) {
        free(myArray[i]);
        i++;
    }
    free(myArray);
    return 0;
}
```



“1D” malloc for 2D array

```
int main() {
    int r = 3;
    int c = 4;
    int *arr = (int *)malloc(r*c*sizeof(int));
    int i = 0;
    int j;
    int count = 0;
    printf("%p\n", arr);
    while(i<r){
        j = 0;
        while(j<c){
            *(arr+i*c+j) = count;
            count = count + 1;
            j = j + 1;
        }
        i = i + 1;
    }
    i = 0;
    while(i<r){
        j = 0;
        while(j<c){
            printf("%p\t", arr+i*c+j);
            printf("%d\n", *(arr+i*c+j));
            j = j + 1;
        }
        i = i + 1;
    }
    printf("\n");
    free(arr);
    return 0;
}
```

14. Stacks and Queues

In this lecture we will cover:

- **Stacks**
- **Queues**
- **Multi-file C Programs**

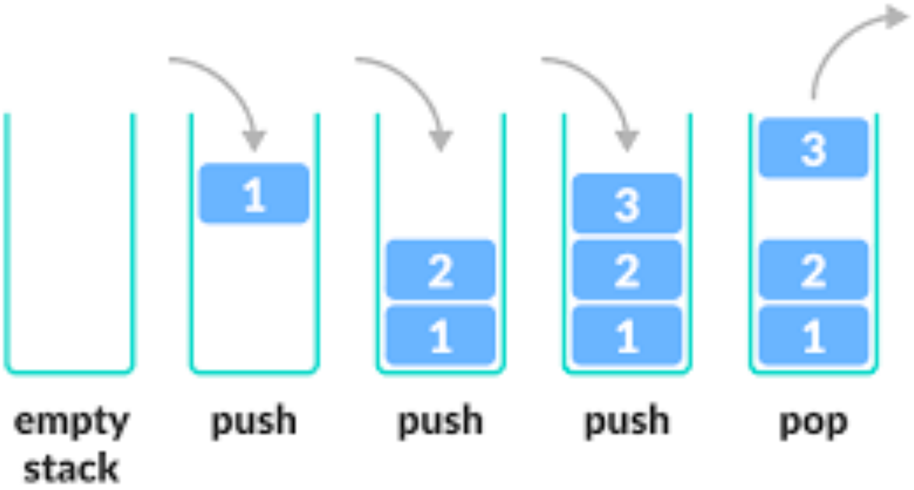
Stacks and Queues

- Stacks and Queues are used in many computing applications,
- they forming auxiliary data structures for common algorithms,
- they appear as components of larger structures.
- They are also good examples to practice programming with arrays.

Stacks

- A stack is a collection of items such that the last item to enter is the first one to exit, i.e.

  "last in, first out" (LIFO)
- based on the idea of a stack of books, or plates



Stack Functions

- Essential Stack functions:
  - push() // add new item to the top of the stack
  - pop() // remove top item from stack
- Additional Stack functions:
  - top() // fetch top item (but don't remove it)
  - size() // get the number of items in the stack

Stack Applications

- page-visited history in a Web browser
- undo sequence in a text editor
- checking for balanced brackets
- HTML tag matching
- postfix calculator
- chain of function calls in a program
- Assignment 2 Part 4!!!

If the current character is a starting **bracket** ('(' or '{' or '[') then push it to stack. If the current character is a closing **bracket** (')' or '}' or ']') then pop from stack. If the popped character is the matching starting **bracket** then fine, else **brackets** are not **balanced**.

Queues

- a queue is a collection of items such that the first item to enter is the first one to exit, i.e. “first in, first out” (FIFO)
- based on the idea of queueing at a bank, shop, etc.

Queue Functions

- Essential Queue functions:
  - enqueue() // add new item to queue
  - dequeue() // remove front item from queue
- Additional Queue functions:
  - front() // fetch front item (but don't remove it)
  - size() // number of items

Queue Applications

- waiting lists, bureaucracy
- access to shared resources (printers, etc.)
- phone call centres
- multiple processes in a computer

Multi-file C Programs

- Large C programs spread across many C files e.g. Linux operating system has 50,000+ .c files.
- By convention .h files used to share information between files.
- .h files contain:
  - function prototypes
  - type definitions
- .h files should not contain code (function definitions)
- #include used to incorporate .h file

  put #include at top of .c file

Example: Include File

answer.h

```c
int answer(double x);
```

answer.c

```c
#include "answer.h"
int answer(double x) {
    return x * 21;
}
```

main.c

```c
#include <stdio.h>
#include "answer.h"
int main(void) {
    printf("answer(2) = %d\n", answer(1));
    return 0;
}
```

Multi-file Compilation

dcc main.c answer.c -o answer

./answer

42

Can also compile file separately creating bf .o files which contain machine code for one file.

dcc -c main.c

dcc -c answer.c

dcc main.o answer.o -o answer

./answer

42

Useful with huge programs because faster to re-compile only part changed since last compilation.

-c generate object files

-o writes the build output to an output file.

Implementing Stacks and Queues

- To implement a stack or a queue, we need an array and some kind of variable to keep track of the current size and maybe even the maximum size.
- Instead of having to keep many variables to represent our stack or queue, why not package it up using a struct? For example:

```
struct stack{
    int items[MAX_SIZE];
    int size;
};
```

Stack Function Prototypes

Suppose we are storing ints in our stack

```
typedef struct stack * Stack;
Stack stackCreate(void);
void stackPush(Stack stack, int item);
int stackPop(Stack stack);
int stackTop(Stack stack);
int stackSize(Stack stack);
void stackDestroy(Stack stack);
```

Note: Why is it important that we pass our stacks around by reference (using a pointer)?

Stack Application: Postfix Notation

Some early calculators and programming languages used a convention known as Reverse Polish notation (RPN) or Postfix Notation where the operator comes after the two operands rather than between them:

1 2 +

result = 3

3 2 *

result = 6

4 3 + 6 *

result = $\boxed{}$

1 2 3 4 + * +

result = $\boxed{}$

Postfix Notation: What is the result?

4 3 + 6 *

Total Results: 0

Postfix Notation: What is the result?

1 2 3 4 + * +

Total Results: 0

Postfix Calculator

A calculator using RPN is called a Postfix Calculator; it can be implemented using a stack:

- when a number is entered: push it onto the stack
- when an operator is entered: pop the top two items from the stack, apply the operator to them, and push the result back onto the stack.

Postfix Calculator Examples

Example 1: 4 2 3 5 1 - + * +

Example 2: 4 5 + 7 2 - *

Queue Function Prototypes

Suppose we are storing ints in our queue

```
typedef struct queue * Queue;
Queue queueCreate(void);
void enqueue(Queue queue, int item);
int dequeue(Queue queue);
int queueFront(Queue queue);
int queueSize(Queue queue);
void queueDestroy(Queue queue);
```

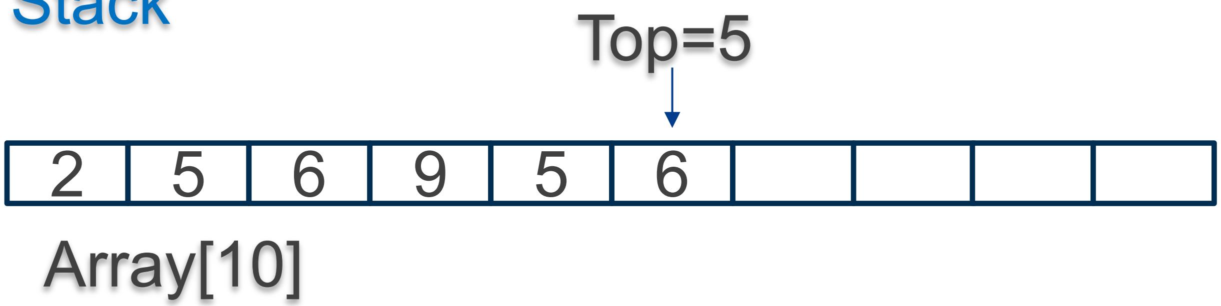
Note: Why is it important that we pass our queues around by reference (using a pointer)?

Implementing Queues

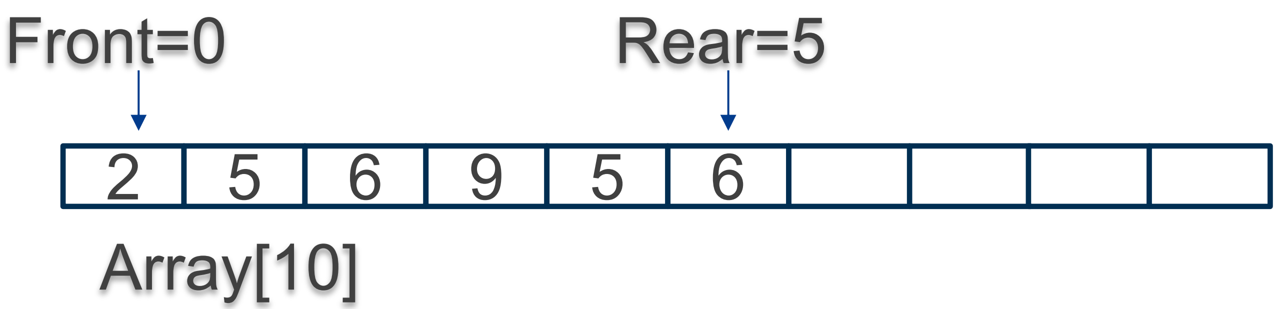
- a stack can be implemented using an array, by adding and removing at the end which is easy to do.
- for a queue, we need to either add or remove from the front
  - What issue do we face when we add or remove an item from the start of an array?

Stacks & Queues

Stack

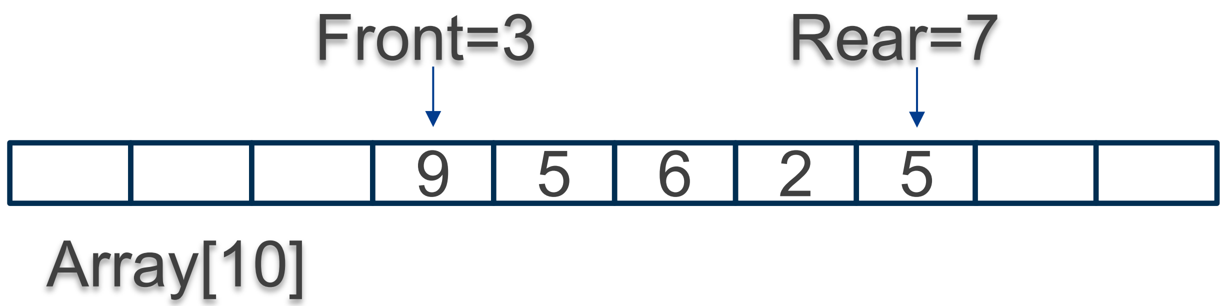


Queue



Queues

Queue



Priority Queue

Priority Queue is an extension of FIFO queue

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

insert(item, priority): Inserts an item with given priority.
getHighestPriority(): Returns the highest priority item.
deleteHighestPriority(): Removes the highest priority item.

```
struct item {
    int item;
    int priority;
}
item q[MAX];
int number;
```

Double Ended Queue

Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

insertFront(): Adds an item at the front of Queue.
insertLast(): Adds an item at the rear of Queue.
deleteFront(): Deletes an item from front of Queue.
deleteLast(): Deletes an item from rear of Queue.