CH-230-A

Programming in C and C++

C/C++

Tutorial 8

Dr. Kinga Lipskoch

Fall 2019

Queues

- A queue is a FIFO (First-In First-Out) data structure, often implemented as a simply linked list
- However:
  - New items can only be added to end of list
  - Items can be removed from the list only from the beginning
  - Just think of line waiting in front of the movies

Operations on the Queue

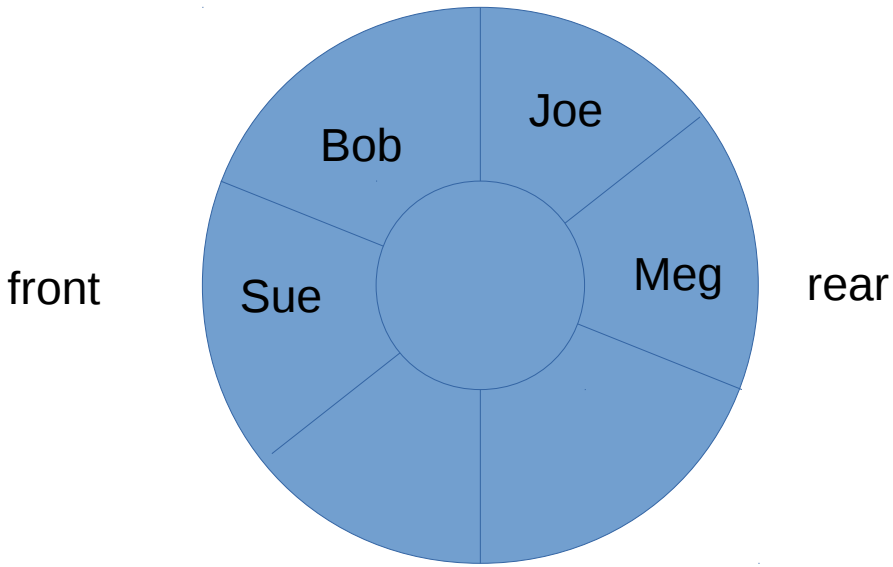
- Initialize queue
- Determine whether queue is empty
- Determine whether queue is full
- Determine number of items in queue
- Add item to queue (always at end)
- Remove item from queue (always from front)
- Empty queue

Data Representation

- Array might be used for queue
  - Simple implementation, but all elements need to be moved each time item is removed from queue
- Wrap-around array
  - Instead of moving elements, use array where indexes wrap around
  - Front and rear pointers point to begin and end of queue

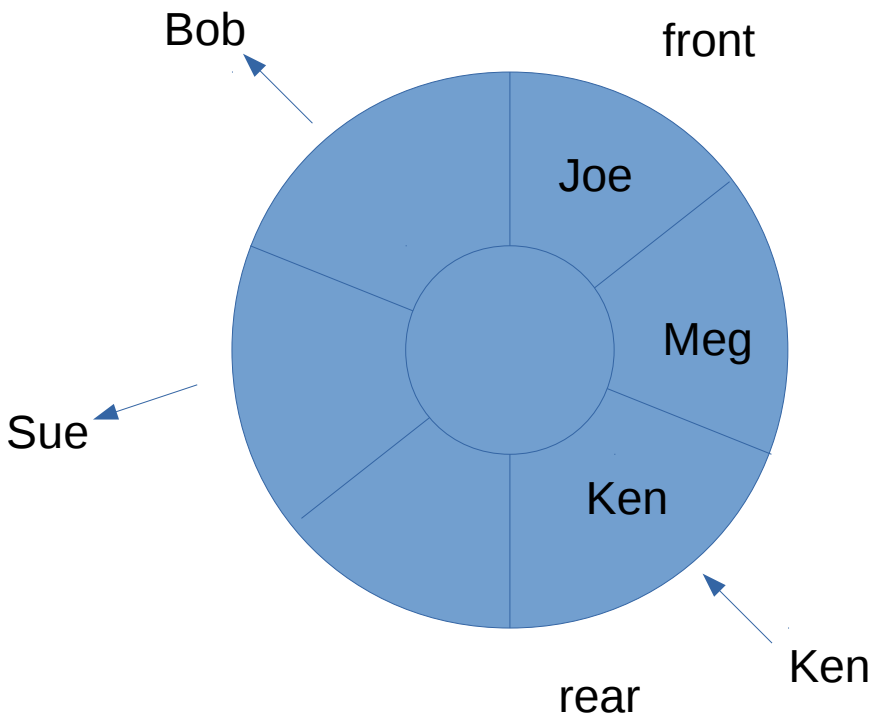
Queue (1)

4 people in the queue



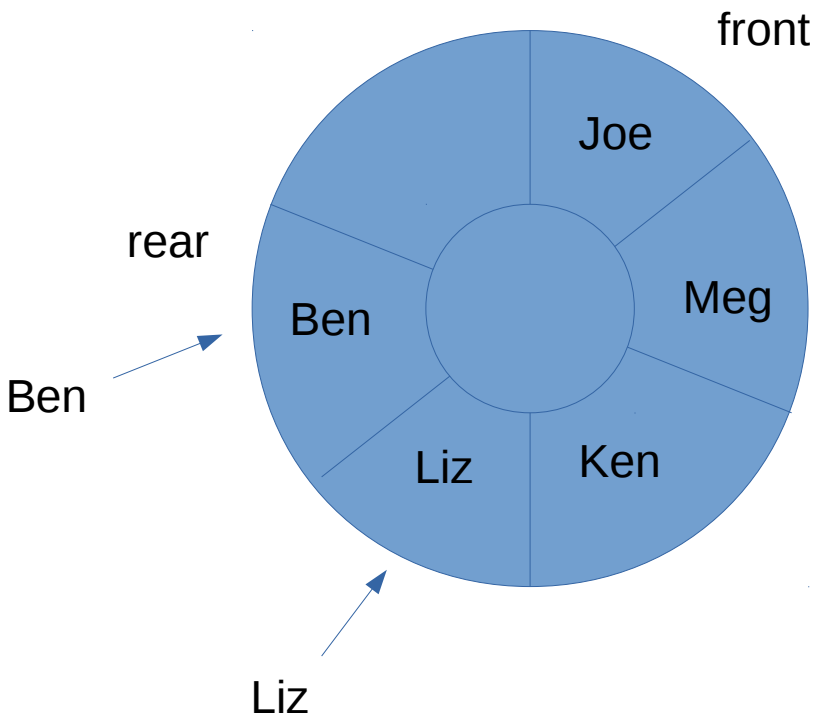
Queue (2)

Sue and Bob leave, while Ken joins queue



Queue (3)

Circular queue wraps around



Queue Implementation (1)

- Use linked list or circular linked list
- Should work with anything, but let's start with integers

```
typedef int Item;
```

- Linked list is built from nodes

```
1 struct node {
2     Item item;
3     struct node *next;
4 };
5 typedef struct node Node;
```

Queue Implementation (2)

- Queue needs to keep track of front and rear items
- Just use pointers for this
- Counter to keep track of items in queue

```
1 struct queue {
2     Node *front;
3     Node *rear;
4     int items;
5 };
6 typedef struct queue Queue;
```

Header Files and Conditional Inclusion

- Have seen that conditional statements can control preprocessing itself
- To make sure that contents of file `myheader.h` is included only once

```
1 #ifndef _MYHEADER_H
2 #define _MYHEADER_H
3
4     // contents of myheader.h goes here
5
6 #endif
```

Interface and Complete Implementation

- Header file contains data types and prototypes
  - `queue.h`
  - Needs to be included by implementation (and users of queue)
- Implementation of queue
  - `queue.c`
- User of queue
  - `testqueue.c`
- Makefile with targets like `all`, `testqueue`, `doc`, `clean`, `clobber`
  - `Makefile`
- Configuration file for doxygen
  - `Doxyfile`
- Testcase input and output
  - `test1.in` `test1.out`

Adding an Item to a Queue

1. If queue is full do not do anything
2. Create a new node
3. Copy item to the node
4. Set next pointer to `NULL`
5. Set front node if queue was empty
6. Set current rear node's next pointer to new node if queue already exists
7. Set rear pointer to new node
8. Add 1 to item count

Removing an Item from a Queue

1. If queue is empty do not do anything
2. Copy item to waiting variable
3. Reset front pointer to the next item in queue
4. Free memory
5. Reset front and rear pointers to `NULL`, if last item is removed
6. Decrement item count

File Handling in C

- Input and output can come from/go into files
- C treats files as streams of data
- A stream is a sequence of bytes (either incoming or outgoing)
- The language does not provide basic constructs for file handling, but rather the standard library does

Communicating with Files

- Communication with files from the outside
- Output redirection
  - `file > outputfile`
- Input redirection
  - `file < inputfile`

Working with Files

- The paradigm is the following:
  - Open the file
  - Read/write
  - Close the file
- In C the information concerning a file are stored in a `FILE` structure (defined in `stdio.h`)
- The C stdio library implements buffered I/O: Data is first written to an internal buffer, which is eventually written to a file

Standard Streams

- `stdin`
  - Standard input is stream data (often text) going into a program
  - Unless redirected, standard input is expected from the keyboard which started the program
- `stdout`
  - Standard output is the stream where a program writes its output data
  - Unless redirected, standard output is the text terminal which initiated the program
- `stderr`
  - Standard error is another output stream typically used by programs to output error messages or diagnostics
  - It is a stream independent of standard output and can be redirected separately

File Modes

Streams can be handled in two modes: (only important for MS Windows)

- Text streams: sequence of characters logically organized in lines. Lines are terminated by a newline ('`\n`')
  - Sometimes pre/post processed
  - Example: text files
- Binary streams: sequence of raw bytes
  - Examples: images, mp3, user defined file formats, etc.

Opening a File

- To open a file the `fopen` function is used

```
FILE *fopen(const char * name, const char * mode)
```

- `name`: name of the file (OS level)
- `mode`: indicates the type of the file and the operations that will be performed

```
FILE *fptr;
fptr = fopen("myfile.txt", "r");
```

Mode Strings

String	Meaning
`"r"`	Open for reading, positions at the beginning
`"r+"`	Open for reading and writing, positions at the beginning
`"w"`	Open for writing, truncate if exists, positions at the beginning
`"w+"`	Open for reading and writing, truncate if exists, positions at the beginning
`"a"`	Open for appending, does not truncate if exists, positions at the end
`"a+"`	Open for appending and reading, does not truncate if exists, positions at the end

A `b` or a `t` can be added to indicate it is a binary/text file

Closing a File

- `int fclose(FILE *fp);`
- Forgetting to close a file might result in a loss of data
- After a file is closed it is not possible anymore to read/write

```
1       FILE *fptr;
2       fptr = fopen("myfile.txt", "r");
3       if (fptr == NULL) {
4         printf("Some error occurred!\n");
5         exit(1);
6       }
7       ...
8       /* do some operations */
9       fclose(fptr);
10      ...
```

Reading/Writing

Prototype	Use
`int getc(FILE *fp)`	Returns next `char` from `fp`
`int putc(int c, FILE *fp)`	Writes a `char` to `fp`
`int fscanf(FILE* fp, char * format, ...)`	Gets data from `fp` according to the format string
`int fprintf(FILE* fp, char * format, ...)`	Outputs data to `fp` according to the format string

Line Input and Line Output

```
char *fgets(char *line, int max, FILE *fp);
```

- Already seen with `stdin`
- Used for files as well

```
int fputs(char *line, FILE *fp);
```

- Outputs/writes a string to a file

Files: Example 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4    char ch;
5    FILE *fp;
6    fp = fopen("file.txt", "r");
7    if (fp == NULL) {
8      printf("Cannot open file!\n");
9      exit(1);
10   }
11   ch = getc(fp);
12   while (ch != EOF) {
13     putchar(ch);
14     ch = getc(fp);
15   }
16   fclose(fp);
17   return 0;
18 }
```

Files: Example 2

```
1  # include <stdio.h>
2  # include <stdlib.h>
3  int main () {
4    char ch;
5    FILE * fp;
6    fp = fopen("file.txt", "r") ;
7    if (fp == NULL) {
8      printf("Cannot open file!\n");
9      exit(1);
10   }
11   while((ch=getc(fp))!=EOF) {
12     putchar(ch);
13   }
14   fclose(fp);
15   return 0;
16 }
```

Files: Example 3

```
1  # include <stdio.h>
2  # include <stdlib.h>
3  int main () {
4    char ch;
5    FILE * fp;
6    fp = fopen("file.txt", "r") ;
7    if (fp == NULL) {
8      printf("Cannot open file!\n");
9      exit(1);
10   }
11   while(!feof(fp)) {
12     ch=getc(fp);
13     if (ch!=EOF)
14       putchar(ch);
15   }
16   fclose(fp);
17   return 0;
18 }
```