JTSK-320112

Programming in C II

C-Lab II

Lecture 5 & 6

Dr. Kinga Lipskoch

Spring 2017

Planned Syllabus

- The C Preprocessor
- Bit Operations
- Pointers and Arrays (Dynamically Allocated Multi-Dimensional Arrays)
- Pointers and Structures (Linked Lists)
- Compiling, Linking and the `make` Utility
- Pointers and Functions (Function Pointers)
- **Stacks and Queues**
- **Modifiers and Other Keywords**
- **Binary I/O (File Handling)**

Stacks (1)

- A stack is a container where items are retrieved according to the order of insertion
- For a stack, the element deleted from the set is the one most recently inserted
- It is called Last-In First-Out policy: LIFO

Stacks (2)

Abstract operations on a stack:

- `push(x, s)` insert item `x` at top of stack `s`
- `pop(s)` remove (and return) the top item of stack `s`
- `init(s)` create an empty stack
- `isFull(s)` determine whether stack is full
- `isEmpty(s)` determine whether stack is empty

Stacks (3)

- Easiest implementation uses an array with an index variable that represents top of stack

```
1 struct stack {
2   unsigned int count;
3   int array[10];    // Container
4 };
```

- Linked list implementation is also possible
  - Advantage: no overflow

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
```

Queues

- A queue is a FIFO (First-In First-Out) data structure, often implemented as a single 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

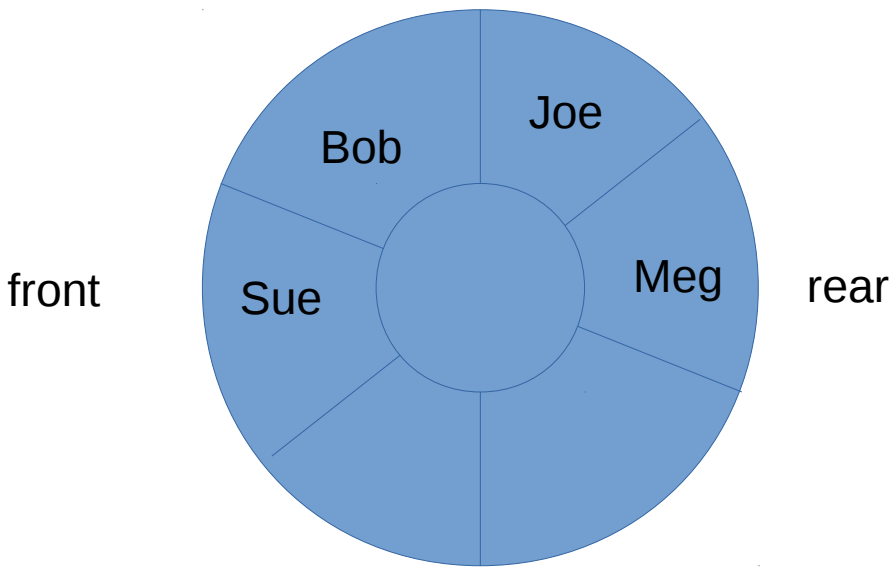
- Initialize queue
- Determine whether queue is empty
- Determine whether queue is full
- Determine number of items in queue
- Add item to queue (always at end)
- Remove item from queue (always from front)
- Empty queue

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

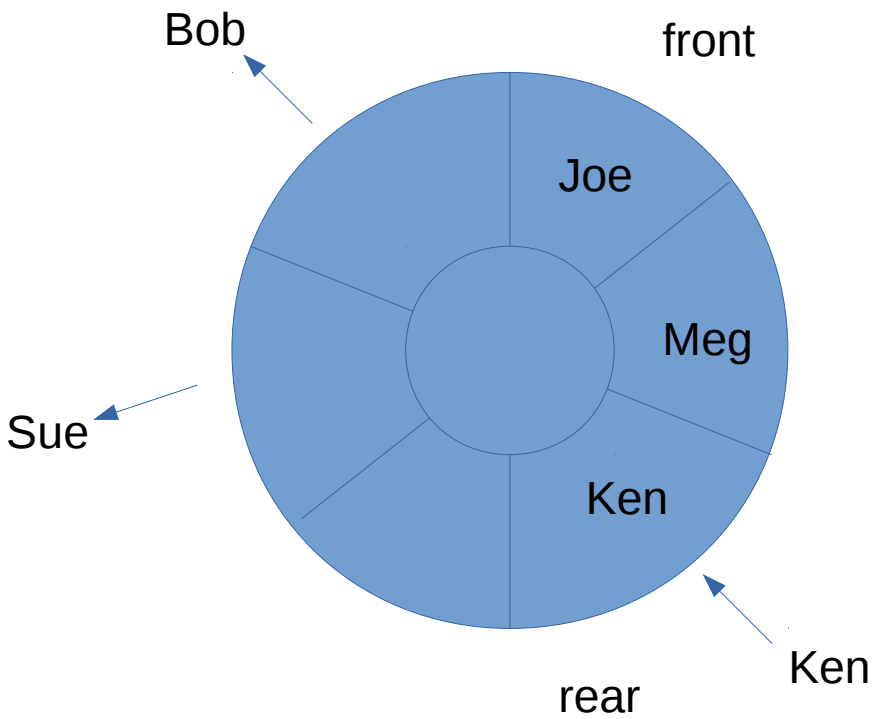
Queue (1)

4 people in the queue



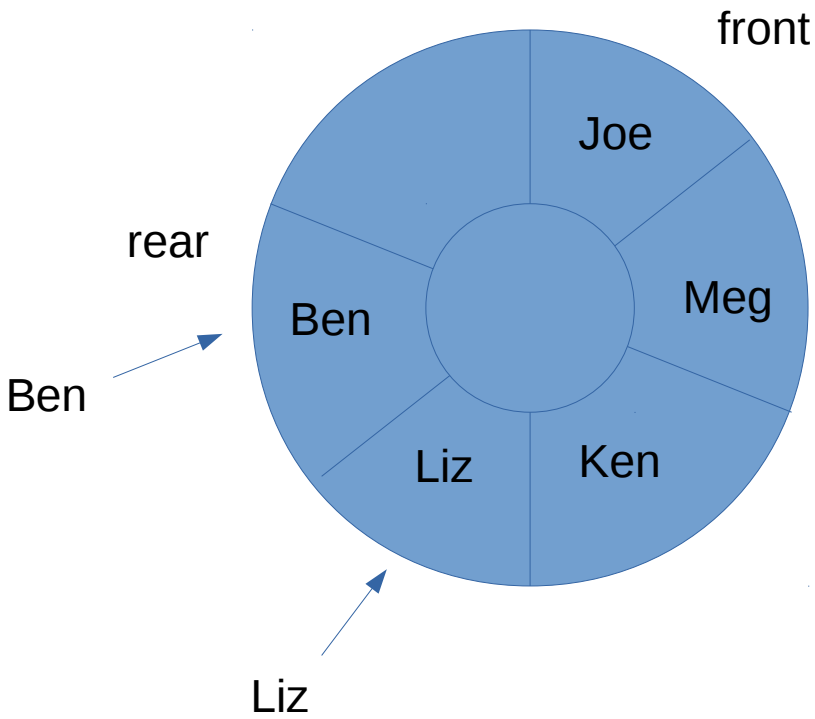
Queue (2)

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;
```

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

Shared Variables / Functions Among Different Files

- Must be defined (once) at global scope in one source file
- Can be declared in any other file which needs that variable
- When declaring variables defined in other files, the `extern` keyword must be used
- `extern int ext_var; /* declaration */`

The extern Modifier

The extern modifier indicates that variable or function is defined outside current file

```
1  #include <stdio.h>
2  extern int counter;
3  extern void inc_counter(void);
4  int main() {
5    int i;
6    for (i = 0; i < 10; i++)
7      inc_counter();
8    printf("Counter is %d\n", counter);
9    return 0;
10 } // main.c
11
12 int counter = 0;
13 void inc_counter(void) {
14   ++counter;
15 } // count.c
```

```
gcc -Wall -o prog main.c count.c
```

Modifiers

- `main()` uses variable `counter`
- `extern` declaration indicates that counter is declared outside this source file
- `counter` is defined in `count.c`

Modifier	*Meaning*	*C++ "equivalent"*
`extern`	variable/function is defined in other file	
`"none"`	variable/function is defined here and can be used by other files as well	`public`
`static`	variable/function is local to this file	`private`

The `static` Modifier

- `static` for globally defined data
  - limits scope to file in which it is declared
  - private to this file
- `static` for variable inside function
  - variable retains value across function calls
  - allocation from static memory and not from stack

Do not Declare Same Variable in Two Files

```c
#include <stdio.h>
int flag = 0;
int main() {
  printf("Flag is %d\n", flag);
  return 0;
} // submain.c

int flag = 1; // sub.c
```

```
gcc -o prog submain.c sub.c
/tmp/cc02iB1n.o:(.bss+0x0): multiple definition of `flag'
/tmp/ccSseHVA.o:(.data+0x0): first defined here
collect2: ld returned 1 exit status
```

, Operator (1)

The , operator evaluates its first operand and discards the result, and then evaluates the second operand and returns this value (and type)

```
1 if (total < 0) {
2   printf("This is a message\n");
3   total = 0;
4 }
```

could be rewritten as:

```
1 if (total < 0)
2   printf("This is a message\n"), total = 0;
```

Syntactically not easy to read

, Operator (2)

Only place where useful, is in the `for` statement:

```
1 for (two = 0, three = 0; two < 10;
2       two+=2, three+= 3)
3   printf("%d %d\n", two, three);
```

Type Qualifiers: `register`

- Request to store variable in CPU's register
- You cannot apply address operator to register variable
- Optimizing compilers are often smarter in determining good register usage (for the target CPU type) than programmers since code optimizations can also change the function of variables
- No guarantee that compiler uses the hint
- Use only after algorithmic and data structure optimizations were done

Type Qualifiers: `volatile`

- Tells the compiler that the value of a variable might not only be changed by the program but from somewhere else
- Compiler should not optimize away accesses
- Compiler needs to reread variable each time it is used

Type Qualifiers: `restrict`

- Is an optimization hint for compiler
- Compiler can choose to ignore it

```
1 int *restrict x;
2 int *restrict y;
```

- Compiler can assume that `x` and `y` are not pointing to the same location

Function memset()

```
#include <string.h>
void * memset(void *s, int c, size_t n);
```

- The `memset()` function fills the first `n` bytes of the memory area pointed to by `s` with the constant byte `c`
- Examples of syntactically correct with "logically" correct and incorrect usages

  `memset_ex.c`

Communicating with Files

- Simple reading and writing so far in "Programming in C I"
- 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
  - *Example:* images, mp3, user defined file formats, etc.

Opening a File

- To open a file the `fopen` function has be 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

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

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 writing. Does not truncate if exists. Positions at the end.

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 no more possible to read/write

```
1 FILE *fptr;
2 fptr = fopen("myfile.txt", "r");
3 if (fptr == NULL) {
4   fprintf(stderr, "Cannot open file!\n");
5   exit(1);
6 }
7 /* do something */
8 fclose(fptr);
```

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, *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

getc() and putc()

- getc() and putc() work like getchar() / putchar()
- `ch = getchar(); // read from standard input`
- `ch = getc(fp);  // provide file pointer to read from`
- `putc(ch, fp);   // char first, then fp`

`EOF` (End Of File)

- Program needs to stop when it reaches end of file
- `getc()` returns special value `EOF`, when trying to read character but reached end of file

Version 0 (contains two issues)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4     char ch;
5     FILE *fp;
6     fp = fopen("file.txt", "r");
7
8     while (ch != EOF) {
9        ch = getc(fp);
10       putchar(ch);
11    }
12    fclose(fp);
13    return 0;
14 }
```

Version 1

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  char ch;
  FILE *fp;
  fp = fopen("file.txt", "r");
  if (!fp) {
    fprintf(stderr, "Cannot open file!\n");
    exit(1);
  }

  ch = getc(fp);
  while (ch != EOF) {
    putchar(ch);
    ch = getc(fp);
  }
  fclose(fp);
  return 0;
}
```

Version 2

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  char ch;
  FILE *fp;
  fp = fopen("file.txt", "r");
  if (!fp) {
    fprintf(stderr, "Cannot open file!\n");
    exit(1);
  }

  while ((ch = getc(fp)) != EOF) {
    putchar(ch);
  }
  fclose(fp);
  return 0;
}
```

fflush(), feof(), ferror()

- `int fflush(FILE *stream)` flushes the output buffer of a stream
  - fflush_ex.c
- `int feof(FILE *stream)` tests the end-of-file indicator for the given stream
  - feof_ex.c
  - myfile.txt
- `int ferror(FILE *stream)` tests the error indicator for the given stream
  - ferror_ex.c

`fseek()` and `ftell()`

- Enables to use a file just like an array and move directly to a specific byte in a file that has been opened via `fopen()`
- `ftell()` returns current position of file pointer as a `long` value

`fseek(fp, offset, mode)`

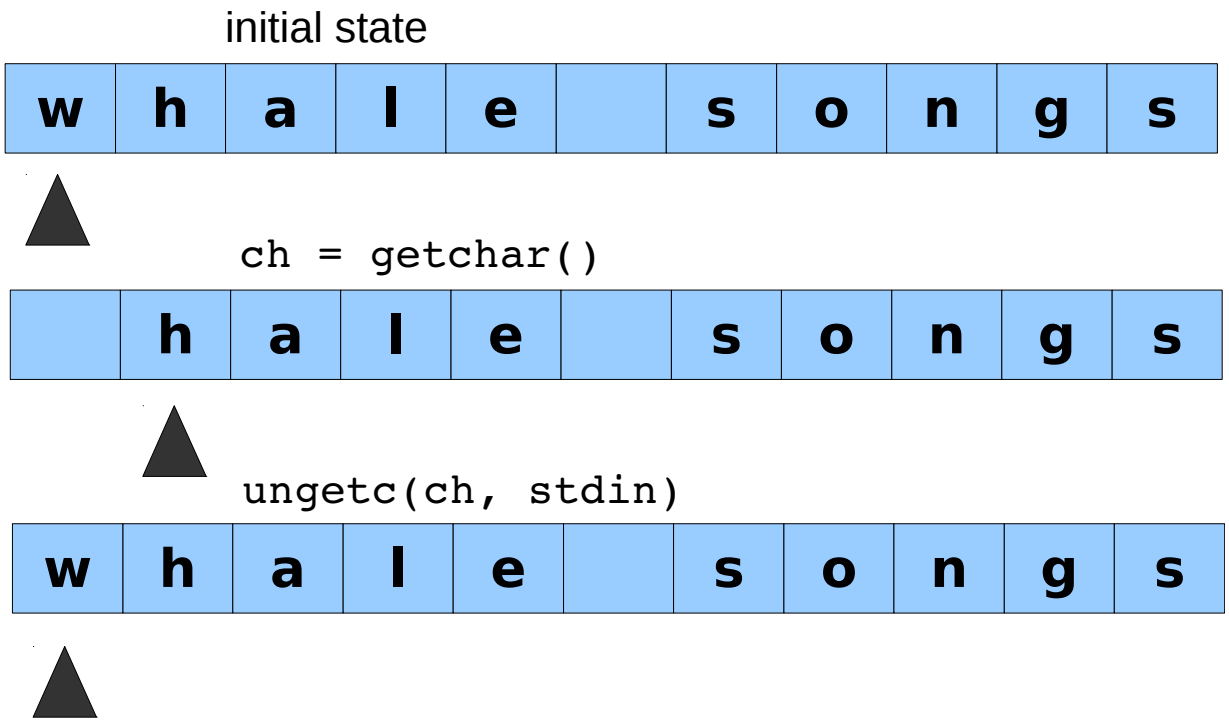
- `fp` is a file pointer, points to file via `fopen()`
- `offset` is how far to move (in bytes) from the reference point
- `mode` specifies the reference point

Mode	measure offset from
`SEEK_SET`	beginning of file
`SEEK_CUR`	current position
`SEEK_END`	end of file

Examples

- `fseek(fp, 0L, SEEK_END);`
  - set position to offset of 0 bytes from file end therefore set position to end of file
- `long last = ftell(fp);`
  - assigns to `last` the number of bytes from the beginning to end of file

ungetc()


initial state
w h a l e s o n g s
ch = getchar()
h a l e s o n g s
ungetc(ch, stdin)
w h a l e s o n g s

Binary I/O

- `fread()` and `fwrite()`
- Standard I/O is text-oriented
  - Characters and strings
- How to save a double
  - Possible as string but also other

```
double num = 1/3.0;
fprintf(fp, "%lf", num);
```

- Most accurate way would be to store the bit pattern that program internally uses
- Called binary when data is stored in representation the program uses

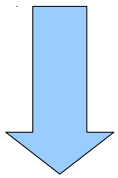
I/O as Text

- All data is stored in binary form
- But for text, data is interpreted as characters

```
short int num = 12345      // a 16-bit number
```

stores 12345 as binary number in num

00110000	00111001



```
fprintf(fp, "%d", num);
```

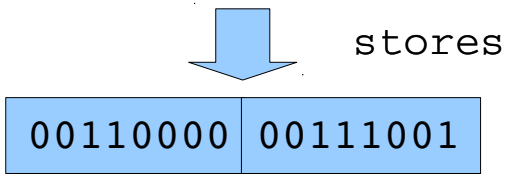
writes binary code for
characters '1', '2', '3', '4', '5' to file

00110001	00110010	00110011	00110100	00110101

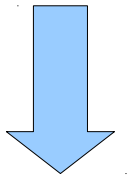
I/O as Binary

If data is interpreted as numeric data in binary form, data is stored as binary

```
short int num = 12345     // a 16-bit number
```

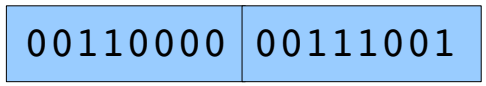


stores 12345 as binary number in num



```
fwrite(&num, sizeof(short int), 1, fp);
```

writes binary code the value 12345 to file



fwrite() (1)

- `size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *fp)`
- Writes binary data to a file
- `size_t` - type is type that `sizeof()` returns, typically `unsigned int`
- `ptr` - address of chunk of data to be written
- `size` - size in bytes of one chunk
- `nmemb` - number of chunks to be written
- `fp` - file pointer to write to

fwrite() (2)

```
1 char buffer[256];
2 fwrite(buffer, 256, 1, fp);
```

- Writes 256 of bytes to the file

```
1 double price[10];
2 fwrite(price, sizeof(double), 10, fp);
```

- Writes data from the price array to the file in 10 chunks each of size double
- Return number of items successfully written, may be less if write error

fread()

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp)`
- Takes same set of arguments that `fwrite()` does
- `ptr` pointer to which data is read to

```
1 double price[10];
2 fread(price, sizeof(double), 10, fp);
```

- Reads `10` size double values into the `price` array
- Returns number of items read, maybe less if read error or end of file reached

Final Exam: Details

- On Saturday, the 11th of March, 2017, 10:00 - 12:00
- Location: IRC Conference Hall and IRC East Wing
- Exam consists of programming exercises to be solved on paper
  - Two hours to solve exercises
  - Similar to the programming assignments
  - You may not use books or other documentation while taking the exam
  - You may not use mobile phones, calculators or any other electronic devices
- `Practice sheet`
- Final tutorial will be given by the TAs before the exam