Week 4 Lecture Notes

algorithms and data structures using c

Contents

[Queues as static arrays 1](#_Toc45105557)

[Circular queues 2](#_Toc45105558)

[Binary File I/O 4](#_Toc45105559)

[fwrite 5](#_Toc45105560)

[fread 5](#_Toc45105561)

[Enumerated Types 7](#_Toc45105562)

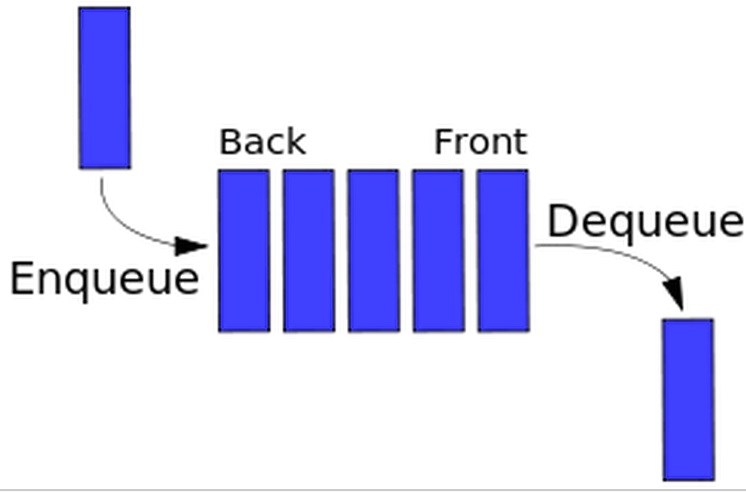
[Unions 7](#_Toc45105563)

# Queues as static arrays

A queue is a first-in-first-out (FIFO) data structure. It is like a line at the grocery. The first one in the line is the first one that gets served. Well, until some other line opens up and then the person at the back of your line gets to go to the front of the new line! That’s not FIFO! No one likes a LIFO (Last In First Out like a stack) data structure at the grocery!

As the image below shows, a queue has a front/head and a rear/tail. To enqueue or shift means to put an item in the queue at the rear. To dequeue or unshift is to take an item off the front of the queue.

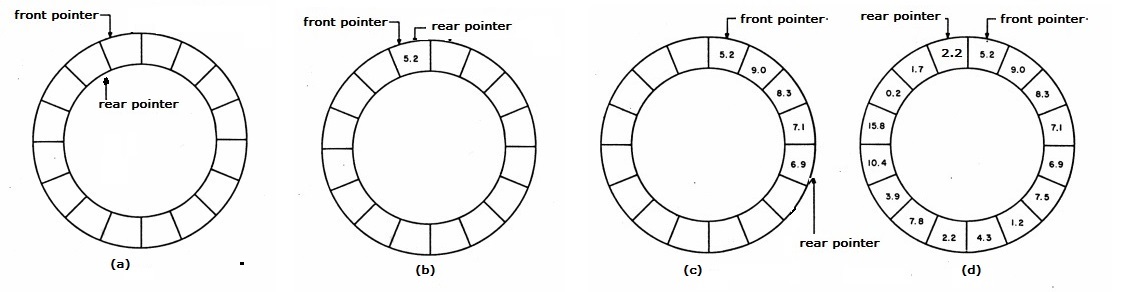
(Image from www.wikipedia.org)

Operating systems use queues all the time. Programs wait for their turn at the CPU in a queue. Print jobs wait to be printed in a queue. Customers waiting for a table at restaurant that doesn’t take reservations are waiting in a queue.

Imagine implementing a queue with a static array. You enqueue an item, so it’s at array[0]. Then, you enqueue another. You could move element 0 to element 1 and put the new item at element 0. So, every time you enqueue, you’d have to shift all the items in the queue down one. That sounds inefficient, doesn’t it? Another option is to enqueue the first item at element 0. Then enqueue the second at element 1, and so on. But, as soon as you needed to dequeue an item, all that shifting would occur. You’d dequeue element 0 and have to shift all the other elements down one so the value that was in element 1 is now at the front of the queue in element 0, and so on. Still, not too efficient. Both of these options are linear implementations.

## Circular queues

Fortunately, there’s another option, a circular queue of which a visual is shown below. A circular queue is still an array, but the index of the front and rear are cycled around as described below.



(Image from http://www.cirsovius.de/)

To implement a circular queue with a non-circular array, we need to remember the index of the front and the rear, a count of the number of items in the queue then use a little modular arithmetic to make the front and rear indices cycle around the queue. As shown in the image above, when the queue is empty, the front index = 0, the rear index = *n*-1 (where *n* = the number of elements in the array), and count is 0. The front index is the index of the item that gets to be dequeued next. The rear index is the index of the item at the back of the queue. Items added to a queue go at the end, so when we add the very first item to the queue, we add it to the rear. The front index doesn’t change, but the rear index must be “incremented” so that is goes from *n-1* to 0, and the new item placed at the rear, index 0. In a queue with one item, the front and the rear are the same. As more items are enqueued in the image, the rear index is “incremented” first then the new item is placed at that index. The front index is not altered when enqueuing. It is altered when an item is dequeued. We can continue to enqueue items onto the queue until the queue is full where the front index is still 0, the rear index is *n-1,* and the count of the number of items is n.

How do we “increment” the rear index so that it goes from n-1 back to 0 when that very first item was enqueued? Suppose in our code the size of the array is QUEUE\_SIZE. Use modular arithmetic to make the rear index “go around” instead of just adding 1 to rear:

rear = (rear + 1 ) % QUEUE\_SIZE;

What happens when rear has the index of the last element in the array, QUEUE\_SIZE-1? Well, we add 1 to it which makes rear’s value QUEUE\_SIZE. What is  
QUEUE\_SIZE % QUEUE\_SIZE? What is 6 % 6? 3 % 3? 100 % 100? Zero! We just made our index go “around”. As we dequeue items, we also have to make the front index “go around” so it is incremented in the same way.

Some implementation details:

* The queue structure must have an array, a count of the number of items in the queue, a front of the queue index and a rear of the queue index. The indices are integers.
* Initializing the queue: to initialize the queue, set count to 0, front index to 0 and rear index to QUEUE\_SIZE-1 (the last index in the array)
* To enqueue an item:

1. if count < the size of the array (so the queue isn’t already full)
   * increment count
   * rear index = (rear index + 1 ) % size of the array (“increment” the rear index)
   * array[rear index] = new item

* To dequeue an item:

1. if count > 0 (if the queue isn’t empty)
   * decrement the count
   * item to return = array[front index]
   * front index = (front index + 1 ) % size of array (“increment” the front index)

* To determine if the queue is empty:

1. if count is 0
   * return true
2. else
   * return false

As far as algorithm efficiency goes, what’s the order of enqueuing or dequeuing? They are both O(1) algorithms because it doesn’t matter how big the queue is, only a constant number of operations needs to happen.

**Pair Programming 4a:** Create the following files then execute and test your program.

* queueItem.h (similar to stackItem.h) with a typedef for a QueueItem
* queue.h (similar to stack.h with a QUEUE\_SIZE of 10) with constants, a typedef for the Queue structure described above and prototypes for functions:
  + void initQueue( Queue \*): initializes queue
  + int enqueue( Queue\*, QueueItem ): enqueues the QueueItem and returns true if it was able, false if not (because queue is full)
  + int dequeue( Queue\*, QueueItem\* ): dequeues an item and places it in the QueueItem parameter (the QueueItem it points to). Returns true if dequeue successful, false if not (because queue is empty)
  + int isEmpty( Queue q ): returns true if queue is empty, false if not
  + void printQueue( Queue ): prints the queue with the code shown below.

|  |
| --- |
| void printQueue( Queue q ) {     int i;     printf("QUEUE\n");     if ( q.count > 0 ) {        for( i = q.front; i != q.rear; i = (i+1) % QUEUE\_SIZE ) {           printf ("| %4d ", q.array[i] );        }     }     if ( q.count > 0 )        printf ("| %4d |\n", q.array[q.rear] );     else        printf( "| |\n" );  } |

* queue.c: has the function definitions of the function prototypes in queue.h
* mainQueue.c: copy mainQueue.c from ~caarnold/cisp1020/week4 and use it to test your queue.
* Makefile

# Binary File I/O

No matter what kind of file you are reading from/writing to, you will always have to follow these steps listed below, and the same general rules apply even if the “file” is a port or a database. All functions, types, constants below are in stdio.h.

1. Declare some kind if “file” variable
2. Open the file
3. Access the file
4. Close the file – as soon as you are done accessing it! Not at the end of the program!

All files are stored in binary, right? I mean, the computer does \*everything\* using binary data. Yes, that’s right. But, try opening your last executable a.out file with an editor like nano or vim. Or, on your PC/laptop, try opening a word processing document like MS Word with a text editor like Notepad or Notepad++. Looks wild, huh? That’s because executable files are not stored with the binary sequence of ACII characters. Characters are still stored in binary. For example: “Mom” is 77 111 109 (in decimal) which is 010011010110111101101101 (binary). But, a Word document has a lot more than just characters in it. A Word document has information about the page set up, paragraphs, fonts, font styles (like bold and italics). All kinds of things that are stored with binary numbers but not binary ACSII characters. Since all of that extra binary data are not ASCII characters, Notepad can’t display them. The executable a.out file has the binary sequences of machine instructions, also not ASCII characters. So, again, an editor can’t display these non-ASCII binary sequences.

Let’ say we want to store the integer 123456789. If we use printf or a C++ ifstream object to write this value to a text file, printf converts it to nine, one-byte ASCII digits: the ASCII value for the digit 1 followed by the ASCII value for the digit 1, etc. If we write this value in its binary, integer format, we would write just 4 bytes with a value of 111010110111100110100010101 (leading zeros left out). In general, writing numbers in their binary sequence instead of their ASCII sequence is more efficient because, in general, the binary sequence takes up less space.

## fwrite

<http://www.tutorialspoint.com/c_standard_library/c_function_fwrite.htm>

When reading/writing data in a structure from/to a text file, it is cumbersome to read/write every single data member separately, isn’t it? So many lines of code. And, if the data has strings of differing lengths, some of them with spaces, it’s even worse. So, it’s often nice to read/write the structure data in binary because we can read/write the entire structure with one statement and not even worry about white space! What if you could:

fwrite( &aStructure, sizeof( aStructure ), 1, out );

and that would write the entire contents of a structure. Well, you can!

Given an Employee structure type and an array, e, of 5 Employee structures, the following code writes all 5 structures to a file called binary.out in one statement. See ~caarnold/cisp1020/week4/binaryWrite.c for a complete program.

|  |
| --- |
| FILE\* out;  int i;  out = fopen( "binary.out", "wb" );  fwrite( e, sizeof( Employee ), 5, out );  fclose( out ); |

Note how you can also write one structure at a time by putting the following statement in a for loop with index i.

fwrite( &e[i], sizeof( Employee ), 1, out );

fwrite can also be used to write one data member of a structure at a time:

fwrite( e[i].firstName, MAX\_NAME\_LENGTH, 1, out );

fwrite( e[i].lastName, MAX\_NAME\_LENGTH, 1, out );

fwrite( &e[i].age, sizeof( int ), 1, out );

fwrite( &e[i].salary, sizeof( double ), 1, out );

**PITFALL:** If you write one structure at a time, you \*must\* read one structure at a time. If you write the data members of the structure in separate statements, then you have to read them in separate statements. The size of a structure in not always equal to the sum of the size of its data members, so writing then one way then reading them another won’t always work.

## fread

<http://www.tutorialspoint.com/c_standard_library/c_function_fread.htm>

fread is used to read data from a binary file. For example:

fread( &e[i], sizeof( Employee), 1, in );

Recall, when reading in data from a file into an array, a while or do while loop is more appropriate than a for loop since at the time of reading, the number of items is unknown. Some other useful I/O functions (that can be used with text or binary files):

* feof <http://www.tutorialspoint.com/c_standard_library/c_function_feof.htm>. Returns 0 (false) if it’s not the end of the file.

FILE\* in;

in = fopen( “data.dat”, “rb” );

if ( in != NULL ) {

fread( &e[i], sizeof( Employee ) , 1, in );

while( !feof( in ) ) {

// do stuff then read the next employee

fread( &e[i], sizeof( Employee ) , 1, in );

}

* ferror <http://www.tutorialspoint.com/c_standard_library/c_function_ferror.htm>. Returns non –zero if an error occurred such as trying to read from a file opened in write mode.
* fseek <http://www.tutorialspoint.com/c_standard_library/c_function_fseek.htm>. Move file pointer to another byte in the file. Mostly used in binary files when fixed-sized structures are used as we’ve discussed above. For example, if I created a binary file writing entire Employee structures, and I knew I wanted the 11th structure, then I could seek to the 11th structure and read:

// seek 10\*sizeof(Employee) bytes from beginning of file.

fseek( in, 10\*sizeof( Employee ), SEEK\_SET );

fread( &e[i], sizeof( Employee ) , 1, in );

* ftell <http://www.tutorialspoint.com/c_standard_library/c_function_ftell.htm>. Returns the current byte position of the file pointer. I can get the length of the file in bytes with this function by seeking to the end of the file and using ftell to “tell” me at what byte position the end is.

fseek( in, 0, SEEK\_END ); // seek 0 bytes from the end of the file

long fileSizeInBytes = ftell( in );

**Pair Programming 4b:** Copy ~caarnold/cisp1020/week4/binaryReadStart.c to binaryRead.c and complete it so it reads an unknown number of Employee structures (at most 100, though) into an array of Employee structures using the input file binary.out (also in ~caarnold/cisp1020/week4). The structures read have to be \*exactly\* like the structures written by binaryWrite.c discussed above. So, you \*must\* read entire structures at a time (not individual data members) and use the exact same structure declaration as the Employee structure in binaryWrite.c. The first name declared first with space for 30 characters. The last name declared second, age (an int) next then salary (a double) declared last. The binary data in this file was written in this format. Even if a name didn’t actually take up 30 characters, 30 characters were written. After reading in the data from the binary file, write it out to stdout so you can see if your reading was successful.

# Enumerated Types

Enumerated types are like constants and, like constants, they are used to make the program clearer for the programmer. For example, I could create an enumerated type called boolean\_t, then declare a variable of that value. I should only assign the values of “false” or “true” to my boolean\_t variable. Notice I didn’t say “could”. I could assign a boolean\_t variable the value of 101, for example, but I shouldn’t. Enumerated types are just integers, but we use them to make our code more readable and understandable.

typedef enum {

   false, true;

} boolean\_t;

The values of true and false in the enumerated type are constant integers numbered starting at 0, so false is 0 and true is 1. It would have been a bad idea to list “true” first since its value would be 0 (which is false)! An example of declaring a boolean\_t variable

boolean\_t flag;

An example of assigning a boolean\_t variable using the enumerated type values:

flag = true;

An example of using a boolean\_t variable in a Boolean expression:

if ( flag ) {

You can start your enumerated type at an integer other than zero. For example:

typedef enum {

JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC;

} month\_t;

An example of declaring a month\_t variable:

month\_t month;

An example of using the enumerated type constants which are in order:

switch( month ) {

case JAN:

etc.

See how nice and readable that is!

Beware: The compiler doesn’t verify valid values for enumerated types. I can still:

month = 100;

even though I should never do that.

Why use enumerated types instead of #define? Values are generated for you, in order, which in sometimes nice. Some debuggers print the value of the enumerated type in symbolic form (i.e., the debugger will print “true” instead of 1).

**Before class:**See ~caarnold/cisp1020/week4/enum.c

# Unions

Unions are data structures that allow the same memory to be referenced in different ways; to store data of different types in the same memory location. Only one type is used at a time. Where a structure can store, for example, an integer *and* a character, a union can store an integer *or* a character. They are most often used in low-level code that interfaces with hardware.

For example, create a type definition:

typedef union{

char c;

int i;

double x;

} data\_t;

Then, declare the variable:

data\_t myUnion;

Access the data members using the dot operator just as you would a structure:

myUnion.c, myUnion.i, myUnion.x

The key here is that only one of these data members c, i and x, have valid data at any given time. There’s not space for data members c *and* i *and* x. There’s enough space in the variable myUnion for the biggest data member, the double x. If I print out the size of myUnion, I will see 8 because a double has 8 bytes. The integer only needs 4 of those bytes and the character only 1 of those eight bytes.  So, I can treat the data in myUnion as a character, an integer or a double. If I put a character in myUnion.c:

myUnion.c = ‘A’;

Then I can print myUnion.c:

printf( “%c\n”. myUnion.c );

**Before class:** See ~caarnold/cisp1020/week4/union.c

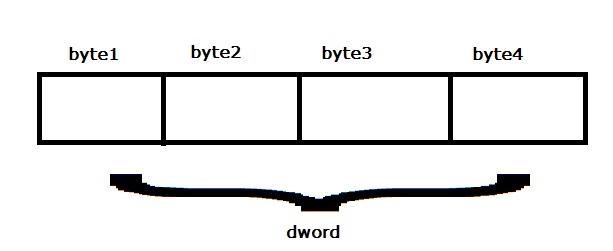
What if I assign:

myUnion.c = ‘A’;

Then print myUnion.x?

printf(“%lf\n”, myUnion.x );

Well, it doesn’t really make any sense to do that. First, what does it mean to interpret the 8 bytes in myUnion as a double when I just put a character (1 byte) in it? And, second, since the character only takes up one of those 8 bytes, what’s in the other 7 bytes? Well, technically, the behavior is undefined: implementation dependent. It is up to the programmer to keep track of which data member is currently in use. If we just assigned myUnion.c, the we have to remember that it is the character we are currently working with.

You can use a union in a structure and/or a structure in a union. Here’s an example if a union used to access a 4-byte hardware register either one byte at a time or the entire 4-byte register at once (image from stackoverflow.com).

typedef union {

struct {

unsigned char byte1;

unsigned char byte2;

unsigned char byte3;

unsigned char byte4;

} bytes;

unsigned int dword;

} HW\_Register;

To declare a variable (don’t declare a variable called “register” – that’s a keyword)

HW\_Register reg;

The union HW\_Register variable reg is 4 bytes. The entire 4-bytes can be accessed at once such as:

reg.dword

Or, each byte can be accessed separately such as

reg.bytes.byte1

Note: the “endianness” of the machine could be important here. Big endian machines store the most significant byte of a word in the smallest address. Little endian machines store the most significant byte of a 4-byte word in the largest addressed byte. A “word” is typically the size of a register and the amount of data read from/written to memory at a time. For example, 64 bits/8 bytes. You learn more about endianness in a Computer Organization class. PS11 uses little endian. How do I know? I wrote a little program to find out that prints values and memory addresses (then I looked it up to verify my findings).

Things to remember:

* A union’s size is that of it biggest data member
* Unions are usually used in conjunction with structures. A union might have several structures in it (so, its size would be that of its largest structure) or a structure could have a union in it.
* Used a lot in embedded/low-level programming to access device register contents, for example. They can be used to access individual bytes of a larger type, to store variable data and to share storage in order to save space.