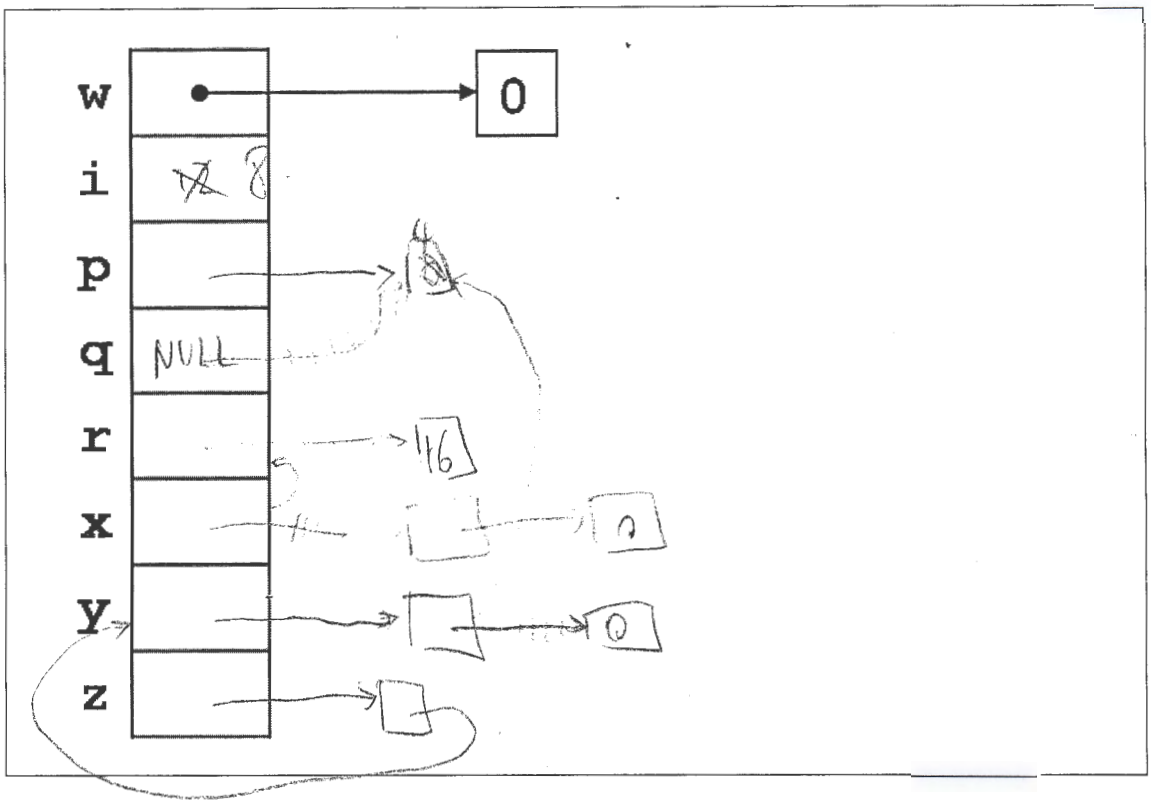
2pts 1. **Pointer Illustration**

Clearly and carefully illustrate the contents of memory after the following code runs. We've drawn the contents of `w`, a pointer that points into allocated memory where the number 0 is stored.

```
int* w = alloc(int);
int i = 12;
int* p = alloc(int);
int* q = p;
int* r = alloc(int);
int** x = alloc(int*);
int** y = alloc(int*);
int** z = y;
*r = i + 4;
*y = q;
**z = 4;
*x = r;
q = NULL;
i = *p + **y;
```



15-122: Principles of Imperative Computation, Fall 2017

Written Homework 5

Due: Monday 2nd October, 2017 by 9pm

Name: Shaojie Zhang.

Andrew ID: shaojiez

Section: D

This written homework covers big-O notation, some reasoning about searching and sorting algorithms, pointers, interfaces, and stacks and queues. You will use some of the functions from the arrayutil.c0 library, as well as the data structure interfaces introduced in lecture this week.

This is the first homework to emphasize interfaces. It's important for you to think carefully and be sure that your solutions respect the interface involved in the problem.

Instructions

You can prepare your submission in one of two ways:

Just edit (preferred) Use any PDF editor (e.g., Preview on Mac, iAnnotate on mobile, Acrobat Pro installed on all non-CS cluster machines and most platforms) to typeset your answers in the given spaces — you can even draw pictures. *That's it.*

Print and Scan Alternatively, print this file, write your answers *neatly* by hand, and then scan it into a PDF file. *This is pretty labor-intensive.*

Once you have prepared your submission, submit it on Gradescope. You have unlimited submissions.

Question:	1	2	3	Total
Points:	2	4.5	2.5	9
Score:				

Evaluation Summary Once this homework is graded, you will be able to find a summary of your performance on Gradescope.

2. Implementing an Image Type Using a Struct

In a previous programming assignment, we worked with one-dimensional arrays that represented two-dimensional images. Suppose we want to create a data type for an image along with an interface that specifies functions to allow us to get a pixel of the image or set a pixel of the image.

(You are allowed to assume that `p1 == p2` is an acceptable way of comparing pixels for equality.)

1.5pts **2.1** Complete the interface for the `image` type. Add appropriate preconditions and postconditions for each image operation (you may not need all the lines we provided). The first two functions should have at least one meaningful postcondition, but you don't have to give every conceivable postcondition.

```
// typedef ______* image_t;

int image_getwidth(image_t IMG)
/*@ requires A != NULL; @*/
/*@ ensures \result >= 0; @*/

int image_getheight(image_t IMG)
/*@ requires A != NULL; @*/
/*@ ensures \result >= 0; @*/

pixel_t image_getpixel(image_t IMG, int row, int col)
/*@ requires IMG != NULL; @*/
/* @ ensures IMG->data[row] != NULL; @*/

void image_setpixel(image_t IMG, int row, int col, pixel_t P)
/*@ requires IMG != NULL; @*/
/* @ ensures P == image_getpixel(IMG, row, col) @*/

image_t image_new(int width, int height)
/*@ requires 0 <= width && 0 <= height; @*/
/* @ ensures \result != NULL; @*/
```

3pts **2.2** In the implementation of the `image_t` type, we have the following type definitions:

```
1  struct image_header {
2      int width;
3      int height;
4      pixel_t[] data;
5  };
6  typedef struct image_header image;
7  typedef image* image_t;
```

And the following data structure invariant:

```
8  bool is_image(image* IMG) {
9    return IMG != NULL
10       && IMG->width > 0
11       && IMG->height > 0
12       && IMG->width <= int_max() / IMG->height
13       && is_arr_expected_length(IMG->data, IMG->width * IMG->height);
14 }
```

The client does not need to know about this function, since it is the job of the implementation to preserve the validity of the image data structure. But the implementation must use this specification function to assure that the image is valid before and after any image operation.

Write an implementation for `image_setpixel`, assuming pixels are stored the same way they were stored in the programming assignment. Include any necessary preconditions and postconditions for the implementation.

```
int image_setpixel (image_t IMG, int row, int col, pixel_t p)
//@ requires IMG != NULL;
//@ requires is_image(IMG);
//@ ensures \result != NULL;
int i = row * image_getwidth(IMG) + col;
IMG->data[i] = p;
}
```

Write an implementation for `image_getwidth`. Include any necessary preconditions and postconditions for the implementation.

```
int image_getwidth( image_t *, IMG )
//@ requires IMG != NULL;
//@ ensures \result >= 0;
{
  return IMG -> width;
}
```

//@ requires is_image(IMG)

3. **Stacks, queues, and interfaces**

1pt **3.1** Consider the following interface for `stack_t` that stores elements of the type **`string`**:

```
/* Stack Interface */
// typedef ______* stack_t;

bool stack_empty(stack_t S)     /* O(1), check if stack empty */
/*@requires S != NULL; @*/;

stack_t stack_new()             /* O(1), create new empty stack */
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/;

void push(stack_t S, string x) /* O(1), add item on top of stack */
/*@requires S != NULL; @*/;

string pop(stack_t S)           /* O(1), remove item from top */
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/
```

Write a client function `stack_bottom(stack_t S)` that returns the bottom element of the given stack, assuming the stack is not empty. The final stack should be identical to the initial stack. For this question, use only the interface since, as a client, you do not know how this data structure is implemented. Do not use any stack functions that are not in the interface (including specification functions like `is_stack` since these belong to the implementation).

```
string stack_bottom(stack_t S)
//@requires S != NULL;
//@requires !stack_empty(S);
{
  stack_t N = stack_new();
  while stack_empty(S)==false{
    push(N, pop(S));
  }
  string result = pop(N);
  push(S, result);
  while stack_empty(N)==false;
    push(S, pop(N));
  return result.

}
```

1.5pts **3.2** Below is the queue interface from lecture.

```
// typedef ______* queue_t;

bool queue_empty(queue_t Q)           /* O(1) */
/*@requires Q != NULL; @*/;

queue_t queue_new()                   /* O(1) */
/*@ensures \result != NULL; @*/
/*@ensures queue_empty(\result); @*/;

void enq(queue_t Q, string e)         /* O(1) */
/*@requires Q != NULL; @*/;

string deq(queue_t Q)                 /* O(1) */
/*@requires Q != NULL; @*/
/*@requires !queue_empty(Q); @*/ ;
```

The following is a client function `queue_size` that is intended to compute the size of the queue while leaving the queue unchanged.

```
int queue_size(queue_t Q)
//@requires Q != NULL;
//@ensures \result >= 0;
{
  int size = 0;
  queue_t C = Q;
  while (!queue_empty(Q)) {
    enq(C, deq(Q));
    size++;
  }
  while (!queue_empty(C)) enq(Q, deq(C));
  return size;
}
```

Explain why the function `queue_size` does not work and give a corrected version below:

(Explanation)

The queue is changed in the function. (Aliasing)

(Correct code)

```
int queue_size(queue_t Q)
//@requires Q != NULL;
//@ensures \result >= 0;
{
  int size = 0;
  queue_t C = queue_new();
  while queue_empty(Q) == false {
      enq(C, deq(Q));
      size++;
  }
  Q = C;
  return size;
}
```