15-122: Principles of Imperative Computation, Spring 2020

Written Homework 5

Due on Gradescope: Monday 17th February, 2020 by 9pm

Name: ______________________________

Andrew ID: ______________________________

Section: ______________________________

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.

Preparing your Submission You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- ***pdfescape*** or ***dochub***, two web-based PDF editors that work from anywhere.
- ***Preview***, the Mac's PDF viewer.
- ***Acrobat Pro***, installed on all non-CS cluster machines, works on many platforms.
- ***iAnnotate*** works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though.*

Submitting your Work Once you are done, submit this assignment on Gradescope. ***Always check it was correctly uploaded.*** You have unlimited submissions.

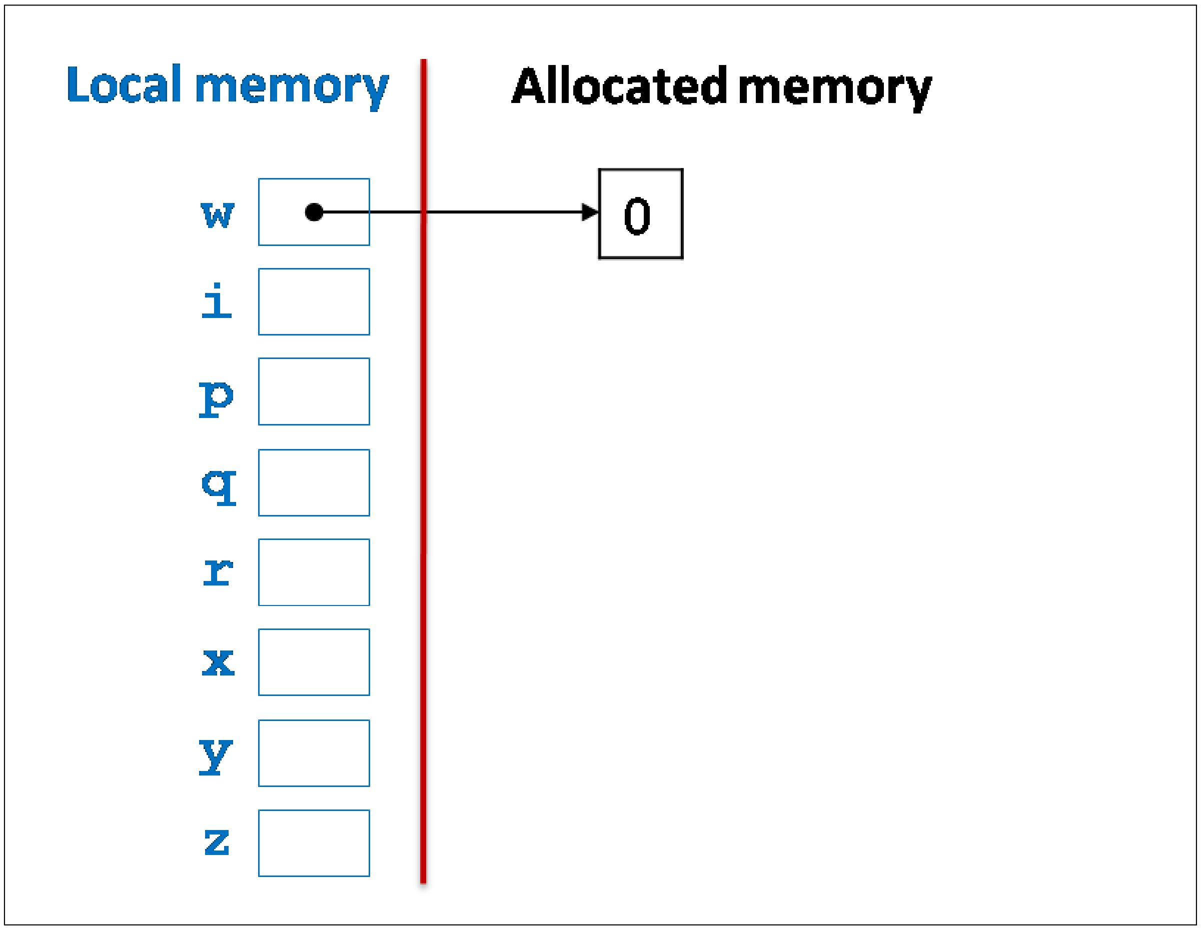
Question:	1	2	3	Total
Points:	2	4.5	2.5	9
Score:				

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 = 42;
int* p = alloc(int);
int* q = p;
int* r = alloc(int);
int** x = alloc(int*);
int** y = x;
int** z = alloc(int*);
*r = i + 7;
*x = q;
**y = *r - i;
p = NULL;
*z = r;
i = *q + **x + *w;
```



2. **Implementing an Image Type Using a Struct**

   In a previous programming assignment, we worked with one-dimensional arrays that represented two-dimensional images. Neither the width nor the height of an image could be 0. 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. This type should still be implementable as a one-dimensional array (but allow other choices).

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

1.5pts

2.1 Complete the interface for the image data 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;

__________ image_getwidth(________________________________________)

    ________________________________________________________________

    ________________________________________________________________

    ________________________________________________________________

__________ image_getheight(_______________________________________)

    ________________________________________________________________

    ________________________________________________________________

    ________________________________________________________________
```

```
__________ image_getpixel(image_t IMG, int r, int c)

  ____________________________________________________________

  ____________________________________________________________

  ____________________________________________________________

  ____________________________________________________________

  ____________________________________________________________

_______ image_setpixel(image_t IMG, int r, int c, pixel_t p)

  ____________________________________________________________

  ____________________________________________________________

  ____________________________________________________________

  ____________________________________________________________

  ____________________________________________________________

__________ image_new(__________ w, __________ h)

  ____________________________________________________________

  ____________________________________________________________

  ____________________________________________________________

  ____________________________________________________________

  ____________________________________________________________
```

In the implementation of the image data type, we have the following type definitions:

```
struct image_header {
    int width;
    int height;
    pixel_t[] data;
};
typedef struct image_header image;
typedef image* image_t;
```

And the following data structure invariant:

```
bool is_image(image* IMG) {
  return IMG != NULL
      && IMG->width > 0
      && IMG->height > 0
      && IMG->width <= int_max() / IMG->height
      && is_arr_expected_length(IMG->data, IMG->width * IMG->height);
}
```

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.

1.5pts **2.2** 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.

1.5pts

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

3. **Queues, Stacks, and Interfaces**

1pt

3.1 Consider the following interface for queues that stores elements of type `bool`:

```
/* Queue Interface */
// typedef ______* queue_t;

bool queue_empty(queue_t Q)     // O(1), check if queue empty
/*@requires Q != NULL; @*/;

queue_t queue_new()             // O(1), create new empty queue
/*@ensures \result != NULL; @*/
/*@ensures queue_empty(\result); @*/;

void enq(queue_t Q, bool e)     // O(1), add item at back of queue
/*@requires Q != NULL; @*/;

bool deq(queue_t Q)             // O(1), remove item from front
/*@requires Q != NULL; @*/
/*@requires !queue_empty(Q); @*/ ;
```

Using this interface, write a client function `queue_back(queue_t Q)` that returns the element at the *back* of the given queue, assuming the queue is not empty. The back of a queue is the most recently inserted element — do not confuse it with the element returned by `deq`, the front of the queue, i.e., the element that has been in the queue the longest. Upon returning, the queue `Q` should be identical to the queue passed to the function. For this task, use only the interface since, as a client, you do not know how this data structure is implemented. Do not use any queue functions that are not in the interface (including specification functions like `is_queue` since these belong to the implementation).

```
bool queue_back(queue_t Q)
//@requires Q != NULL;
//@requires !queue_empty(Q);
{

}
```

Below is the stack interface from lecture (with elements of type `bool`).

```
// 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, bool x)         // O(1), add item on top of stack
/*@requires S != NULL; @*/;

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

The following is a client function `stack_reverse` that is intended to return a copy of its input stack S with its elements in reverse order while leaving S unchanged.

```
stack_t stack_reverse(stack_t S)
//@requires S != NULL;
//@ensures \result != NULL;
{
  stack_t RES = stack_new();
  stack_t TMP = S;
  while (!stack_empty(S)) {
    bool x = pop(S);
    push(TMP, x);
    push(RES, x);
  }
  while (!stack_empty(TMP)) push(S, pop(TMP));
  return RES;
}
```

0.5pts

3.2 Explain why `stack_reverse` does not work.

1pt **3.3** Give a corrected version of `stack_reverse`.

```
stack_t stack_reverse(stack_t S)
//@requires S != NULL;
//@ensures \result != NULL;
{

}
```