15-122: Principles of Imperative Computation, Spring 2023

Written Homework 5

Due on Gradescope: Sunday 12th February, 2023 by 9pm

Name: ______________________________

Andrew ID: ______________________________

Section: ______________________________

This written homework covers pointers, interfaces, and stacks and queues.

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.
- ***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*.

Caution Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

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

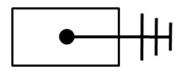
Question:	1	2	3	Total
Points:	4	6	2.5	12.5
Score:				

4pts 1. **Pointer Illustration**

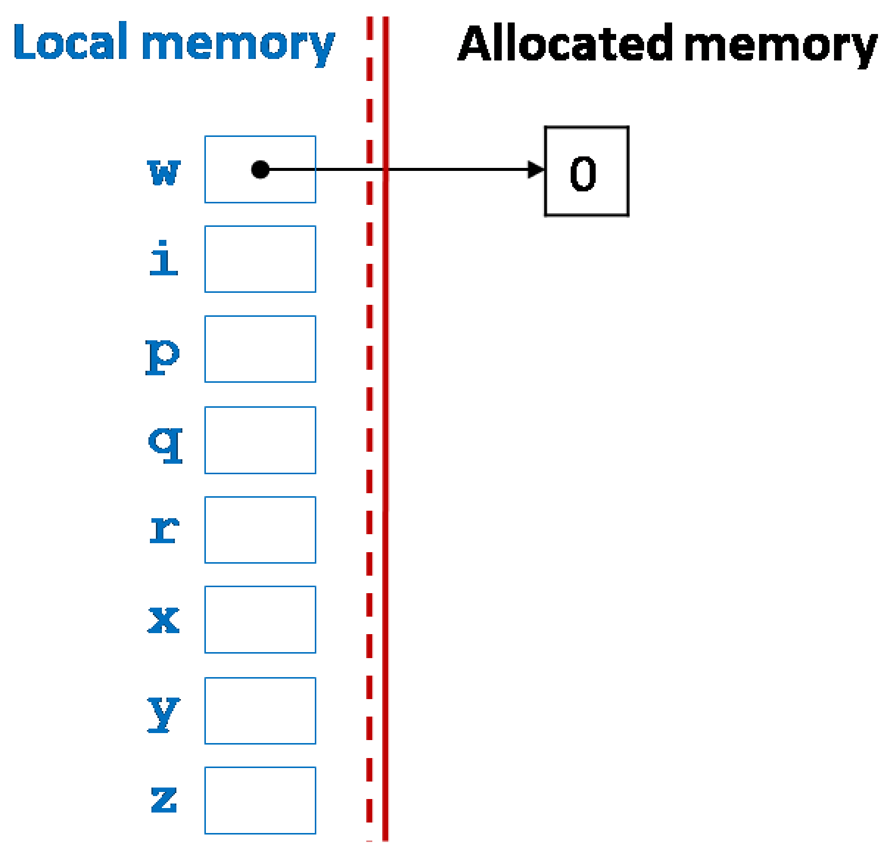
Clearly and carefully illustrate the contents of memory after the following code runs. We've drawn the contents of w, a pointer that points into allocated memory where the number 0 is stored.

```
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;
```

You may draw a variable or memory location containing NULL either as



or as



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

In a previous programming assignment, you implemented image manipulations where

- we used a one-dimensional array to represent a two-dimensional image,
- neither the width nor the height of the image could be 0,
- their product should be at most `int_max()` so that every position in the array can be indexed with an **`int`**.

We want to write a library that packages an abstract type of images and some basic operations to work with them. Your first job is to complete the interface of this library. Your interface should support *any representation* of images, not just the one you saw in the programming assignment.

You may assume the client has access to the pixel interface and some basic image utility functions reproduced in appendices A and B at the end of this homework. Feel free to use the added function `pixel_equal` if you need it.

1pt

2.1 Complete the interface for the image data type. Add appropriate preconditions and postconditions for each image operation to ensure the client can't use the interface unsafely. The postconditions should give the client enough information to prove the safety of subsequent calls. *(You may not need all the lines we provided.)*

```
// typedef ______* image_t;

__________ image_getwidth(________________________________)
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________

__________ image_getheight(_______________________________)
    ____________________________________________________
    ____________________________________________________
    ____________________________________________________
```

1pt **2.2** (Continued)

```
__________ image_getpixel(image_t A, int row, int col)

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

void image_setpixel(image_t A, int row, int col, pixel_t px)

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________
```

1pt **2.3** (Continued)

```
__________ image_new(__________ width, __________ height)

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________
```

In the implementation of the image data type, we have the following concrete 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* A) {
  return A != NULL
      && A->width > 0
      && A->height > 0
      && A->width <= int_max() / A->height
      && is_arr_expected_length(A->data, A->width * A->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.

Note that the type `pixel_t` is abstract and can only be manipulated using the functions listed in Appendix A.

1.5pts **2.4** Write an implementation for `image_setpixel`, assuming pixels are stored in the pixel array in the exact same way they were stored in the programming assignment. Include any necessary preconditions and postconditions for the implementation.

1.5pts

2.5 Write an implementation for `image_getheight`. 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 In one or two sentences, explain why `stack_reverse` does not work.

1pt

3.3 Give a corrected version of `stack_reverse`. Your code may only use stacks.

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



}
```

A Appendix: the pixel interface

See your *pixels* programming assignment for the contracts these functions obey and what they do. However, *don't assume the implementation in the assignment*.

```
// typedef _______ pixel_t;

int get_red(pixel_t p);
int get_green(pixel_t p);
int get_blue(pixel_t p);
int get_alpha(pixel_t p);

pixel_t make_pixel(int alpha, int red, int green, int blue);

// Returns true if p and q are the same pixel, false otherwise
bool pixel_equal(pixel_t p, pixel_t q);  // ADDED
```

B Appendix: image utility functions — `imageutil.c0`

See your *images* programming assignment for the contracts these functions obey and what they do. However, *don't assume the implementation in the assignment*.

```
bool is_valid_imagesize(int width, int height);

int get_row(int index, int width, int height);
int get_column(int index, int width, int height);

bool is_valid_pixel(int row, int col, int width, int height);

int get_index(int row, int col, int width, int height);
```