iii Design Document

COMP 40 : Machine Structure and Assembly

Programming Language

by George Pesmazoglou

and Alexandru Ungureanu

Prof. Noah Mendelsohn

Part A – Uarray2.h

1. **Purpose :** The abstract data type we are representing in this project is a 2-dimensional unboxed array. An unboxed data structure stores the actual data, and not the pointers to the data.

2. **Functions and Contracts:**

   - **`T UArray2_new(int width, int height, int size)`** creates a new 2D unboxed array with the given width and height that holds elements of the given size and returns the new array (of type T)
   - **`void UArray2_free(T *uarray2)`** frees the memory allocated for the array
   - **`int UArray2_width(T uarray2)`** returns the width of the 2D array given as parameter
   - **`int Uarra2_size(T uarray2)`** returns the size of a uarray2 element
   - **`int UArray2_height(T uarray2)`** returns the height of the 2D array given as parameter
   - **`void *Uarray_at(T uarray2, int w, int h)`** returns a pointer to the element at position [w, h] in the array given as parameter
   - **`void Uarray2_map_col_major(T uarray2, void apply(int w, int h, T uarray2, void *element, void *cl), void *cl)`** applies the

function “apply” to every element in the 2D array given as parameter by going through columns first; cl is a pointer that is passed in each call of the “apply function”

- **`void Uarray2_map_row_major(T uarray2, void apply(int w, int h, T uarray2, void *element, void *cl), void *cl)`** applies the function “apply” to every element in the 2D array given as parameter by going through rows first; cl is a pointer that is passed in each call of the “apply function”

Note: T is defined as a pointer to a Uarray2_T “object”.

3. Example:

```
Uarray2_T array = Uarray2_new (5, 3, sizeof(long));
int width = Uarray_width(array);   // the width of the array
int height = Uarray_height(array);   // the height of the array
int *p1 = (int *) Uarray2_at(array, 0, 0);   // top-left corner
*p1 = 1;   // top-left corner element gets value 1
int *p2 = (int *) Uarra2_at(array, width – 1, height – 1);   //bottom-right
*p2 = 2;   // bottom-right element gets value 2
UArray2_free(&array);   // free the array
```

4. **Representation and Invariants:** We will use one Hanson Uarray (unboxed 1D array) for each column and store those Hanson Uarray's in another array to create the 2D unboxed array.

   Invariants:

   - at each point, an element in the 2D array is the actual data we want to use, and not a pointer to the data (unboxed representation)
   - at each point, the element at position [i, j] is the j^{th} element in the i^{th} 1D array (since each column is represented as a 1D unboxed array)

5. **Correspondence to the world of ideas:**

   - the unboxed 2D array corresponds to a matrix
   - each 1D array used to represent the columns corresponds to a vector of elements
   - the array used to store the column-uarray2's corresponds to an ordered sequence of vectors

6. **Test cases:** In order to test the implementation of the functions in the Uarra2.h interface, we are going to run the following tests:

   - creating a 2D array of the given width and height
   - modifying element(s) of the array
   - check if space for each element is allocated correctly
   - output the elements of the 2D array using the map function

- modifying all the elements in the array using the map function
- valgrind test to see if space is deallocated correctly
- access an element outside the width and height (error)
- access an element when initializing an array of zero size (error)
- access an element after having freed the array (error)

7. **Programming Idioms:**
    - handle void * values of known types
    - use unboxed structures

Part B – Bit2.h

1. **Purpose :** The abstract data type we are representing in this project is a 2-dimensional unboxed bit array. An unboxed data structure stores the actual data, and not the pointers to the data.

2. **Functions and Contracts:**
    - **`T Bit2_new(int width, int height)`** creates a new 2D unboxed bits array with the given width and height that holds bits (0 or 1 value)
    - **`void Bit2_free(T *bit2)`** frees the memory allocated for the bits array
    - **`int Bit2_width(T bit2)`** returns the width of the 2D bits array given as parameter

- **`int Bit2_height(T bit2)`** returns the height of the 2D bits array given as parameter
- **`void Bit2_put(T bit2, int w, int h, int value)`** puts the given value (can be 0 or 1) at position [w, h] in the given bits 2D array
- **`void Bit2_get(T bit2, int w, int h)`** returns the bit value at position [w, h] in the given bits 2D array.
- **`void Bit2_map_col_major(T bit2, void apply(int w, int h, T bit2, int value, void *cl), void *cl)`** applies the function "apply" to every element in the 2D bits array given as parameter by going through columns first; cl is a pointer that is passed in each call of the "apply function"
- **`void Bit2_map_row_major(T bit2, void apply(int w, int h, T bit2, int value, void *cl), void *cl)`** applies the function "apply" to every element in the 2D bits array given as parameter by going through rows first; cl is a pointer that is passed in each call of the "apply function"

Note: T is defined as a pointer to a Bit2_T "object".

3. Example:

```
Bit2_T bArray = Bit2_new (7, 5);

int width = Bit2_width(bArray);   // the width of the bits array

int height = Bit2_height(bArray);   // the height of the array

Bit2_put (bArray, 0, 0, 1);   // value 1 in top-left corner

int bit = Bit2_get(bArray, 0, 0);   //get the value

printf("%d", bit);
```

```
Bit2_free(&bArray);   // free the array
```

4. **Representation and Invariants:** We will use one Hanson Bit Vector (1D bits array) for each column and store those Hanson Bit Vectors in an array to represent a 2D array of bits. Invariants:
    - at each point, a bit in the 2D bits array can be either 1 or 0
    - at each point, the bit at position [i, j] is the j^{th} element in the i^{th} 1D Bit Vector (since each column is represented as a 1D bits array)
    - at each point, an element of the bit map is the actual bit we want to use (since we can't store pointers to bits)

5. **Correspondence to the world of ideas:**
    - the bit 2D array corresponds to a matrix of 1s and 0s (a bit map)
    - each 1D bit array used to represent the columns corresponds to a vector of bits (a vector of 1s and 0s)
    - the bit 1D array used to store the column-arrays corresponds to an ordered sequence of vectors

6. **Test cases:** In order to test the implementation of the functions in the Bit2.h interface, we are going to run the following tests:
    - creating a 2D bit array of the given width and height
    - modifying bits in the array

- getting bit values from the array
- output the elements of the 2D bit array using the map function
- modifying all the elements in the bit array using the map function
- valgrind test to see if space is deallocated correctly
- access a bit outside the width and height (error)
- try to access a bit after having freed the array (error)

7. Programming Idioms:

- use unboxed structures
- represent a bit as an integer of value 1 or 0