Computer Programming using C

Lecture 11: Buffers and linked lists: solutions

Dr Julian Miller
Room P/M/101
Ext 2383; E-mail: jfm@ohm
http://www.elec.york.ac.uk/intsys/users/jfm7

Contents

• Solution to buffers exercises
• Linked lists (solutions)
• In the lab this week: Doubly linked lists
Buffers

• In lab 14, you looked at two kinds of buffers, LIFO (stack) and FIFO (queue)
• In exercise 14.1 you were asked to complete a `buffer_remove` function
• `buffer_remove` retrieves and removes an item of data from the stack
Exercise 14.1: solution

```
/*
 * Remove the next value from the buffer and return it
 * solution to exercise 14.1
 */
int buffer_remove(int *buffer, int *buffer_top)
{
    int item;
    /* note *buffer_top -1 indicates location of top of stack */
    /* Update the next free slot indicator */
    (*buffer_top)--;
    /* remove from the buffer */
    item = buffer[*buffer_top];
    return item;
}
```

Exercise 14.2

- The programs provided for lab 14 don’t check whether the buffer is full or empty. This means that the program could crash under certain circumstances.
- You were asked to warn the user in these cases (and disallow possible situations that might cause an error)
Exercise 14.2: solution

- Idea: construct a validation function that checks for overflow and underflow situations and warns the user.

```c
int buffer_validate(int buffer_top, int buffer_state)
{
    int i;
    int valid = 1;
    if (buffer_state)
    {
        if (buffer_top == BUFFER_SIZE)
        {
            gotoxy(4, 15);
            printf("buffer full, can't add to it");
            valid = 0;
        }
        else
        {
            if (buffer_top == 0)
            {
                gotoxy(4, 15);
                printf("buffer empty, can't remove from it");
                valid = 0;
            }
        }
    }
    /* wait */
    for (i=0;i<100000000;i=i);
    return valid;
}
```

Exercise 14.2: calling `buffer_validate`

- In the main function modify the switch statement as follows

```c
/* Handle the key press */
switch(key)
{
    /* User wants to add a data item */
    case 'a':
        case 'A':
        valid = buffer_validate(buffer_top,1);
        if (valid)
           add_item(buffer, &buffer_top);
        break;
    /* User wants to remove a data item */
    case 'r':
    case 'R':
    valid = buffer_validate(buffer_top,0);
    if (valid)
        remove_item(buffer, &buffer_top);
    break;
    /* User wants to quit the program */
    case 'q':
    case 'Q':
       quit = 1;
}
```
Exercise 14.3: Implementing a FIFO buffer

- FIFO is a queue. This differs from the stack in where we add items. In a stack items are added and remove from the same place, the top of the stack.
- In a queue, items are removed from the top (as with stack) but added at another place. This is best handled with two markers to tell us where the queue begins and ends.
- buffer_head, buffer_foot
- So in exercise 14.3 you were asked to implement a queue.

FIFO buffer (queue): demo
Exercise 14.3: Initialise

- In buffer.c
- All we need to do is initialise two variables one for the head of the queue and one for the foot

```c
/* Initialize the buffer */
void buffer_init(int *buffer, int *buffer_head, int *buffer_foot) {
    /* All we need to do is initialise the buffer head and foot */
    *buffer_head = 0;
    *buffer_foot = 0;
}
```

---

Exercise 14.3: buffer_add

- Now we need to modify the routine that adds things to the queue buffer

```c
/* Add the value specified to the buffer.
   Adding to queues mean increasing the foot
   and adding the item there. Notice buffer_foot
   never gets smaller! Uh, oh!
*/
void buffer_add(int *buffer, int *buffer_foot, int item) {
    /* Add the data to the buffer */
    buffer[*buffer_foot] = item;
    /* Update the next free slot indicator */
    (*buffer_foot)++;
}
```
Exercise 14.3: buffer_remove

- Also we need to modify the routine that removes things from the queue buffer

```c
/* Remove the next value from the buffer and return it
   solution to exercise 14.1
   Things get removed from queues at the head.
   Notice: buffer_head never gets smaller! Uh, oh!
 */
int buffer_remove(int *buffer, int *buffer_head)
{
    int item;
    /* remove from the buffer */
    item = buffer[*buffer_head];
    /* Update the next free slot indicator */
    (*buffer_head)++;
    return item;
}
```

Exercise 14.3: buffer_validate

- Also we need to change the buffer validation to take into account the changes and to prevent buffer overflow.

```c
int buffer_validate(int buffer_head, int buffer_foot, int buffer_state)
{
    int valid = 1;
    if (buffer_state) /* dealing with adding */
    { if (buffer_foot >= BUFFER_SIZE)
        { getoxy(4, 15);
          printf("buffer overflow, can't add to it");
          valid = 0;
        }
    } else /* dealing with removing */
    { if (buffer_head >= BUFFER_SIZE)
        { getoxy(4, 15);
          printf("buffer overflow, can't remove from it");
          valid = 0;
        }
    }
    /* wait */
    for (i=1;i<100000000;i++)
    return valid;
```
Exercise 14.3: Calling
buffer_validate

```c
switch (key)
{
  /* User wants to add a data item */
  case 'A':
    valid = buffer_validate(buffer_head, buffer_foot, 1);
    if (valid)
      add_item(buffer, &buffer_foot);
    break;

  /* User wants to remove a data item */
  case 'R':
    valid = buffer_validate(buffer_head, buffer_foot, 0);
    if (valid)
      remove_item(buffer, &buffer_head);
    break;

  /* User wants to quit the program */
  case 'Q':
    case 'q':
    quit = 1;
}
```

Exercise 14.4: Implementing a
circular FIFO buffer

- The implementation of a queue in Exercise 14.3 is not ideal we can only add items at the buffer_foot irrespective of where that is in the actual array. To solve this we need a circular queue where the item after the top element is the bottom element.
- To achieve this we need alter buffer_add and buffer_remove so that
  - buffer_head, buffer_foot
  - Wrap around.
FIFO buffer (circular queue): demo

Exercise 14.4: `buffer_add`

```c
/* Add the value specified to the buffer.
   Adding to queues means increasing the foot
   and adding the item there.
   Notice: now buffer_foot can never just wraps around
*/
void buffer_add(int *buffer, int *buffer_foot, int item, int *num_items)
{
    /* Add the data to the buffer */
    buffer[*buffer_foot] = item;
    /* Update the next free slot indicator */
    *buffer_foot = (*buffer_foot)++;
    if (*buffer_foot == BUFFER_SIZE)
        *buffer_foot = 0;
    *num_items = (*num_items)++;
}
```
Exercise 14.4: buffer_remove

```c
/*
 * Remove the next value from the buffer and return it
 * solution to exercise 14.1.
 * Things get removed from queues at the head.
 * Notice: now buffer_head just wraps round
 */
int buffer_remove(int *buffer, int *buffer_head, int *num_items)
{
    int item;
    /* remove from the buffer */
    item = buffer[*buffer_head];
    /* write something in to show it has been removed */
    buffer[*buffer_head] = -1;
    /* update the next free slot indicator */
    *buffer_head = (*buffer_head) + 1;
    if (*buffer_head == BUFFER_SIZE)
        *buffer_head = 0;
    /*num_items = (*num_items) --;
    return item;
*/
}```

Exercise 14.4: buffer_validate

```c
int buffer_validate(int buffer_head, int buffer_foot,
                    int buffer_state, int num_items)
{
    int i;
    int valid = 1;
    if (buffer_state) /* dealing with adding */
    {
        if (num_items == BUFFER_SIZE)
        {
            gotoxy(4, 15);
            printf("Queue full. Can't add");
            valid = 0;
        }
        else /* dealing with removing */
        {
            if (num_items == 0)
            {
                gotoxy(4, 15);
                printf("Queue empty. Can't remove");
                valid = 0;
            }
        }
    } /* wait */
    for (i = 1; i < 100000000; i++)
    {
        return valid;
    }
```
Linked lists

• In last weeks lab15 you encountered linked lists and were edit and add to a linked list database program. The data structure looked like:

The user menu

As it stood:
1. you could not go to the last student
2. You could not remove a student details
Exercise 15.4

1. You were asked to add a ‘last’ pointer to the list data structure
2. Modify list_init so that last is initialised to be NULL
3. Modify the function list_last so that the current pointer pointed to the last record
4. Modify the function list_add so that the last pointer was given the address of the last record

Exercise 15.4: solution

```c
typedef struct long_list_type
{
    student_type *first;
    student_type *current;
    student_type *last;
} list_type;

void list_init(list_type *list)
{
    /* Just set all the pointers to NULL */
    list->first = NULL;
    list->current = NULL;
    list->last = NULL;
}

void list_last(list_type *list)
{
    list->current = list->last;
}
```
Exercise 15.4: list_add

```c
void list_add(list_type *list, student_type* new_student)
{
    /* At the moment, this always adds the student */
    /* to the beginning of the list */
    new_student->next = list->first;
    list->first = new_student;
    /* Make the new student the first one on the list */
    list->first = new_student;
    /* Make the student we have just added the current one */
    list->current = new_student;
    /* if this student is the last */
    /* then get its address and assign to last */
    if (list->last == NULL)
        list->last = list->first;
}
```

Exercise 15.5: creating list_remove

Three conditions we need to think about

1. Database is empty
   - This is dealt with in function query_remove

2. Only one record in the database
   - We need to free the memory and make first, current and last pointers NULL

3. Two or more records in database
   - Bypass the first record and free the memory
Exercise 15.5: list_remove, step 2

2. Only one record in the database
   - We need to free the memory and make first, current and last pointers NULL

```c
void list_remove(list_type *list)
{
    /* declare a variable to hold the address of the first student */
    student_type *temp;
    /* copy the location of the first student data into
    a temporary pointer variable */
    temp = list->first;

    /* if no students in database then this isn’t called
    assert_gives you deals with this */
    if (list->first->next == NULL)
    {
        /* case 1: there is only one student in the list.
        In this case we have return the list to its initial state
        */
        list_init(list);
    }
    else
    {
        /* case 2: there are two or more students in the list.
        */
    }
}
```

Exercise 15.5: list_remove, step 3

3. Two or more records in database
   - Bypass the first record and free the memory
Exercise 15.5: list_remove, step 3. The code

```c
else {
    /* case 2: there are two or more students in the list.
    so we have to
    1. make first student point to the second student
    2. free up the memory allocated to the first student */

    /* Make the new first student pointer point to the next student */
    list->first = list->first->next;

    /* Make the current pointer point to the new first student */
    /* we don't have to change last as it still should point to
    the last student */
    list->current = list->first;

    /* now we can delete the block of memory pointed to by temp */
    free(temp);
}
```

In the lab this week: Doubly linked lists

In the program it has a `previous` option. This is much easier to implement with doubly linked lists. These are very useful also for FIFO (queues)