CS110: Computer Programming Lab
Department of CSE
IIT, Guwahati Jan-May 2018

1 BASIC POINTER-BASED DATA-STRUCTURES, GLIMPSES OF OBJECT-ORIENTATION, TESTS-FIRST CODING, ABSTRACTION – COHESION AND COUPLING

1.1 INTRODUCTION: MODULE 04 STAGE 02

Students must refer to Module 02 Stage 01 for protocol details related to CS110 Lab sessions and assessment practices. The information is not repeated here to save space in this document.

In summary, diligently work through the drill training documents before requesting for and working on the assessment exercise(s) assigned to you. No more than one assessment would be permitted in a single lab session to any student. (Option of revising a Good grade to a better grade for the immediately previous stage is available in addition to this single assessment.)

1.2 LEARNING AIMS OF MODULE 04 STAGE 02

- User defined elementary data structures
- Introduction to Object-Orientation
- Interfaces for Object Classes
- Access disciplines for queue and stack
- Examples of other access disciplines
- Simple examples of lists
- Ordered lists – key-based access to data
- Abstraction – High intra-module cohesion, Low inter-module coupling

1.3 ASSESSMENT COMPLETION CHECKLIST

The following items will be checked by the assessing tutor before recording successful completion of this stage by the student on CS110 course records.

1. Is the program appropriately commented? And, do the comments help in understanding the program code?
2. Is the amount of comments in the program appropriate? That is, the amount of comments are neither too little nor too much.

3. Is the name of the programmer and date of program creation included in the demonstrated program?
4. Are all constants used in the program code from the exercise statement? The program output should only be computed from the program inputs and constants listed in the problem description.
5. Are the identifiers used as variables helpful and describe the variable use correctly? (In all code segments in this document variable names ending in P represent a pointer!)
6. Are the variable type declarations appropriate?
7. Is the program correctly indented? Is it easy to read and understand?
8. Has the student constructed test cases? Can the student explain coverage and usefulness of the test cases?
9. Student's code should have at least three new and useful `assert()` declarations.
10. Does the program run correctly?

2 LEARNING GUIDE AND DRILL INSTRUCTIONS

Think of a student getting ready for an open notes examination. The student is permitted to take normal-size handwritten notes on some limited number of pages. These pages are called *cheat-sheets*.

What goes in the head of this student a day before the examination? The questions that challenge the student are:

1. What information is needed for the examination?
2. How can the information be added efficiently onto the cheat-sheets?
3. What cheat-sheet organisation will support efficient access of the information during the examination?
4. What keywords should be highlighted to improve access?
5. What were the questions in the previous examinations? And,
6. How can the student use these past exam questions to improve the quality of the cheat-sheet contents and their organisation?

There are other issues of interest when multi-page cheat-sheets are permitted. Students may wish to organise them based on the topics. Each cheat-sheet must be self-contained with all information that a single question may ask. That is, the cheat-sheets must have low coupling – the need to refer to multiple sheets for a single answer should be small.

There is a related property; cohesion. Unconnected information on a single cheat-sheet would make their use difficult in the examination.

Programmers face similar concerns! They seek efficient and effective organisations of information. They want to organise their programs so that the efforts needed to explain, understand and track-errors in the program are localised to relatively small domain of functions and/or files.

2.1 OBJECTS AND CLASSES

Programs are used to model the real world computational needs. The real world is made of objects with attributes (C `struct` is a set of object attributes) and behaviours (C functions/procedures can be viewed as describing the behaviours). To give an example: IIT has students, courses, teachers, classrooms, hostels and so on. Each of these are object classes. Objects in these classes have established behavioural patterns and expectations. A student enrols in a course, a teacher teaches a course, a course is taught in a classroom.

C is a procedural language. C++ and Java are programming languages based on the Object-Orientation (O-O) approach. Yet, it may be worth our efforts here in this drill to view `struct` data-structures as objects and organise functions around them to build object classes based on their behaviours.

The properties that are difficult to implement in C would be ignored in this drill training. We, however, will discuss interfaces of the classes. An interface is the behavioural front that all objects in a class present to the other objects in the program. Before we discuss this in a little more detail in the next subsection, let us spend a few paragraphs looking at the O-O issues that we shall ignore later as C may not have convenient constructs to implement these features.

Inheritance is a powerful feature in O-O methodology. Students and teachers are both objects from a common class human. They share many common properties and behaviours as both are specialised class of common class human. A student is a human. Also, a teacher is a human. Both object classes inherit these (human-based) properties and behaviours from their more general class `human`.

Yet, students and teachers have properties and behaviours that specialise them into different classes. Teachers teach courses. Students learn courses. Teachers assign grades to the students in the courses. Students receive grades for the courses.

Full featured O-O programming languages provide constructs to support sharing of the common properties (data values and function codes) among the specialising classes. This is called inheritance.

Sometimes the specialising class may alter the inherited behaviours. For example, penguins are birds that do not fly. Humans are animals that do not have four legs.

As said previously, we shall only focus on class interface in this drill as language C does not fully support O-O paradigm.

There are students Ram and Abdul. They interact with one-another through interactions that involve exchanges of information. Each of them maintains personal information that the other student cannot access. Object-Oriented programming groups the objects with the same behaviour into classes. There may be several objects from a single class; each object has its distinct private data but common behaviour (we have already noted the example of students). In the extreme approach that we will describe here, the classes define methods (functions) through which other objects can interact with the objects in the class. The data internal and specific to the object is inaccessible to the other objects and can only be enquired and changed by requesting the object through defined methods, if such a method is available.

2.2 METHODS, CONTRACTS AND SPECIFICATIONS

The programmer developing code for a method in a class must provide specification (contract) of the method.

Contract is specification of a method. The method must ensure that if a program segment using the object meets its obligations (pre-conditions) as set in the method contract before calling the method, the object must provide results for the method that matches the contract outcome obligations (post-conditions). Thus, a correct code for class method must match the specifications (contract) of the method. We will return to this issue a little later with the tests-first approach.

You can immediately see the benefits. The approach delivers low coupling; as all interactions are through a small number of well-specified methods defined for the class interface. Each class code delivers cohesion as only this code is able to access and use the private information of the object – mistakes when noticed can be corrected by looking carefully at the class code. Code within a class is not concerned with the implementation details of the other classes and/or even the codes in the application programs that use the objects of the class.

In this document we only provide a glimpse of this approach and explain only some concepts using `queue` as our first example of a class. In fact, we will look at queues in detail and then look at other classes only at the interface specification level.

Queue is a class of objects – many queues can be formed but each queue has similar behaviour. At the same time different queues may be for different kinds of entities. Queues for humans and queues for cars are both queues but have stored objects with different attributes.

Behavioural discipline of all queues is, however, similar. Entities join the queues and leave the queues. The entities join at the rear of a queue and leave from the front. This gives a first-in-first-out discipline to the queues.

2.3 OBJECT CLASS INTERFACE

A puritan view of a class interface is as a set of functions (called methods) providing the only way to access the object. No attribute of any object can be accessed from outside the code through a variable or as a member in any `struct`. We will follow this extremist view in this drill for the data-structures we discuss in this document!

Objects can be referred through pointers. Pointers can be pure references without revealing any type detail of the referred objects. This is done by expressing the reference-pointer type as `void *`.

Queues and Stacks are data-structures that organise the objects based on their arrival sequence. Others data-structures, like sorted lists use object-keys to identify and locate the object in the data-structures.

The data-structures, we use in this drill, will have no more than one link/reference in the data-structure `struct` to organise the objects. This is in line with the data-structures discussed in CS101 classes. However, there will be another reference member built into these elements to refer to the objects "saved" in the data-structure.

In later courses you may learn about more sophisticated data-structures that use multiple links to build the storage structures.

The following sections describe a very basic but useful discipline for inserting and retrieving objects in a popular and commonly understood discipline called `queue`.

3 CLASS QUEUE AND ITS INTERFACE

The following interface (File `queue.h`) provides a useful view of class queue:

```
#ifndef QUEUE_H_INCLUDED
#define QUEUE_H_INCLUDED

/* Returns a reference (pointer) to a new queue */
void * mkQueue(void);

/* Removes a queue specified by a valid reference */
int rmQueue (void *);

/* Returns number of entries in a validly referenced queue */
int sizeQ (void *);

/* Place objP  in queueP and returns new count of entries */
int joinQ (void * queueP, void * objP);

/* Return earliest arrived objP in queue */
void * leaveQ (void *);

#endif // QUEUE_H_INCLUDED
```

Methods `mkQueue()` and `rmQueue()` are two methods included here on the same purpose as make directory (`mkdir`) and remove directory (`rmdir`) commands in Linux.

Type `void *` is a pure reference to the objects – it carries no further information. This paucity of information in the reference ensures that the application code using queue has no access to the internal details of any queue. The code using the queue can only use the reference that it has through the methods defined in the interface and implemented in the `.c` code file(s) private to the queue implementation.

The remaining three methods are quite obvious. `sizeQ()` tells the count of entities in the queue. Each time a new entity is added by method `joinQ()` the number will go up by one. It will reduce by 1 when method `leaveQ()` is invoked.

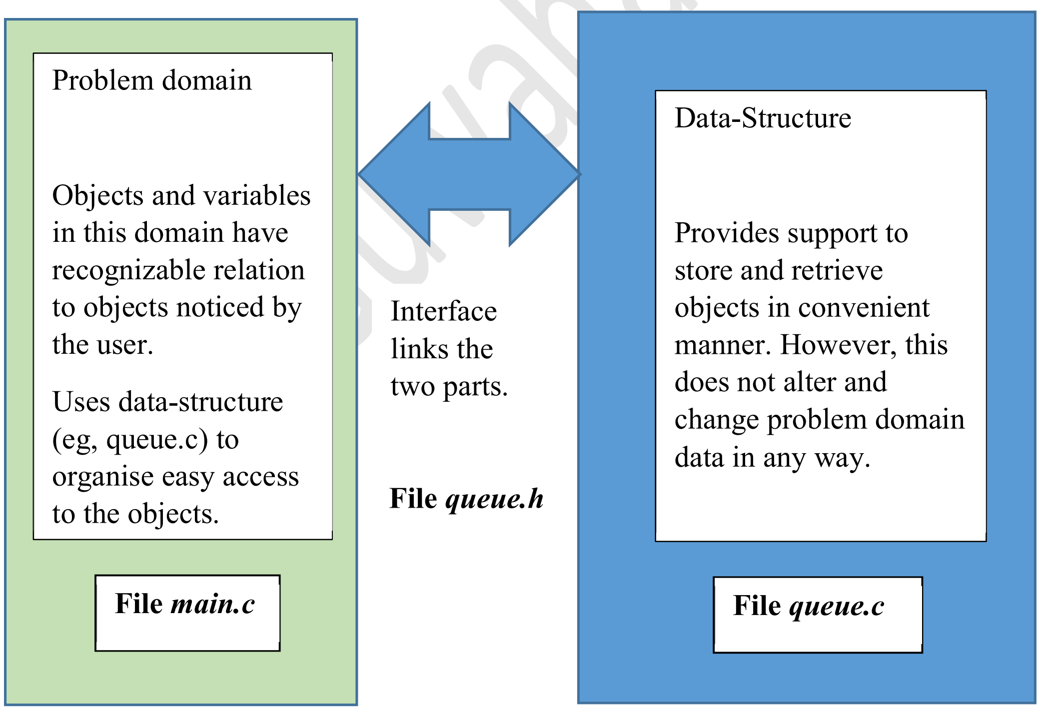
Note that like the application using the queue, queue also receives and returns references to the application object. Objects that are stored in the data-structure queue can not to be changed or used by the methods and functions in file `queue.c`. The code implementing queue cannot access information from the object as the type is pure reference `void *` -- without any information to see the insides of the referenced object. To see the contents in the referenced `struct` one needs to know the type of the `struct` being referenced. (An example is added as an appendix to this document to explain the point further.)

The contract specifications of various methods are expressed in the next few sentences.

Method `leaveQ()` will return reference to the earliest application object who joined the queue but has not left the queue yet. The entity will be removed from the queue. If the queue is empty, it will return a `NULL` pointer.

Method `joinQ()` adds an entity into the specified queue and correctly remembers its arrival order. We have already described the obligations in contract for method `sizeQ()`.

The three C compiler directives in file `queue.h` (`#ifndef`, `#define`, `#endif`) may be ignored and are primarily used to avoid multiple includes of the interface code into a file being compiled. Our simple examples are not likely to cause any problem if these directive lines are removed. (Again, students seeking an example must see the appendix to this document.).



The diagram is instructive. It shows that an application program is made of three separate files. What is important here is the fact that file `main.c` does not know or even care about the code in file `queue.c`. Similarly, code in `queue.c` is unaware of any detail of file `main.c`. The two parts are connected by the use of `#include "queue.h"` directives in two `.c` files.

File `queue.h` described above is a common interface file. This file will be included (`#include "queue.h"`) in files `main.c` and `queue.c`. File `main.c` is application program using class queue. File `queue.c` provides method (behaviour) codes and memory storage needed for queue(s) and its entities.

For compilation of a program spread over multiple files located in a single directory, you may use a `gcc` command that lists all `.c` files to be complied. Also note the use of `#include` directive in the program codes. Be sure to locate your .h file in the same directory where `.c` files are stored.

Command needed on a Linux computer is: `gcc queue.c main.c` to get executable file `a.out`

3.1 Tests First Coding

The methods included in the `queue.h` interface are contracts. As they are obligations on the implementer of code file `queue.c`; we can write test code even before the code is written. Later when methods and functions in file `queue.c` are available, the tests will provide an easy check that the agreed contracts are being fulfilled. We list one test suite here, but one may think of multiple test suites to ensure that all obligations are being met by the developed class code!

Test suites may also be called acceptance tests – the implementers of class queue have completed their obligations when a code is written that passes all agreed test suites.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "queue.h"

#include "queue.h"

int main()
{
    /* ****************************
     * Create a test suite to check
     * obligations of queue.c code
     *******************************/

    void * testQP;

    struct testObj
    {
        int id;
    } one, two, * returnedItemP;
```

```c

    /* Create a queue and test it */
    testQP = mkQueue();
    if (testQP == NULL)
    {
        printf("Queue not created.\n");
        return 1; /* Terminate and report  failure */
    }

    /* Test size of newly created queue */
    if (sizeQ(testQP) != 0)
    {
        printf ("Newly created queue has non-zero size.\n");
        return 1; /* Terminate and report  failure */
    }

    /* Test contents of a new queue */
    if (leaveQ(testQP) != NULL)
    {
        printf("Newly created queue returns a non-NULL \
               entity for leaveQ().\n");
        return 1; /* Terminate and report  failure */
    }

    one.id = 1; /* Assigne identity to application objet */
    two.id = 2;

    /* Test joinQ() for retur value */
    if (joinQ(testQP, &one) != 1)
    {
        printf("Size does not match expectations \
               after joinQ()\n");
        return 1; /* Terminate and report  failure */
    }

    /* Remove item and see its identity */
    returnedItemP = leaveQ(testQP);
    if (returnedItemP->id != 1)
    {
        printf("Returned entity not as expected\n");
        return 1; /* Terminate and report  failure */
    }

    /* Work with two entities in queue */
    joinQ(testQP, &two);
    joinQ(testQP, &one);
    returnedItemP = leaveQ(testQP);
    if (returnedItemP->id != 2)
    {
        printf("Returned entity not as expected\n");
        return 1; /* Terminate and report  failure */
```

```
    }

    /* Try deleting a non-empty queue */
    if (rmQueue(testQP))
    {
        printf("Deleted non-empty queue!\n");
        return 1; /* Terminate and report  failure */
    }

    /* Try removing an empty queue */
    leaveQ(testQP);
    if (!rmQueue(testQP))
    {
        printf("Empty queue not removed\n");
        return 1; /* Terminate and report  failure */
    }

    /* ******************************
     * My small test suite did not fail.
     * Implementation accepted.
     ******************************/
    printf("Satifsied all tests\n");
    return 0; /* Happy terminate */
}
```

While tests are no guarantee for a perfect and error-free program, well-designed sets of tests is the most common way of enhancing trust in the correctness of program codes. We have already described them as acceptance agreement between the code developers and their clients.

3.2 AN IMPLEMENTATION OF QUEUE.C

An implementation of the interface is presented here. The code is located in file `queue.c`. The code using this queue will be located in separate file `main.c`.

The students must also take note of the use of asserts in the code to reduce errors. This is important here as the code must run successfully with every possible code that relies on the implementation of the interface methods. Asserts provide us with extra protection against mistakes.

```
#include "queue.h"
#include <stdlib.h>
#include <assert.h>

/* Each element of queue has ...*/
struct queue_elem
{
    /* ... a referernce to application object, and ... */
    void * objP;
    /* ... another element in the queue */
    struct queue_elem * nextP;
```

```
};

/* A queue will have a reception struct; with ... */
struct queue
{
    /* ...a reference to the most recent entry to queue...*/
    struct queue_elem * recentElemP;
    /* ...and the one who joined the earliest */
    struct queue_elem * ancientElemP;
};

void * mkQueue(void)
{
    struct queue * theQP;

    /* Allocate new space and initialise pointers */
    theQP = (struct queue *)malloc(sizeof (struct queue));
    theQP->ancientElemP = NULL;
    theQP->recentElemP = NULL;

    return theQP;
};

int rmQueue (void * queueP)
{
    /* Get details of the pointed structure */
    struct queue * theQP = (struct queue *) queueP;
    assert (theQP != NULL); /* Not a SPAM queue */

    /* Can not remove non empty queue */
    if (theQP->recentElemP != NULL)
        return 0; /* Deleted */
    assert (theQP->recentElemP == NULL &&
            theQP->ancientElemP == NULL);
    free (theQP);
    return 1; /* Failed to delete */
}

int sizeQ (void * queueP)
{
    struct queue * theQP = (struct queue *) queueP;

    int size = 0;
    struct queue_elem * qeP;

    assert (theQP != NULL); /* Not a SPAM queue */

    if (theQP->recentElemP == NULL &&
            theQP->ancientElemP == NULL)
        return 0;
```

```
    /* Queue has existing entries */
    assert (theQP->recentElemP != NULL &&
            theQP->ancientElemP != NULL);

    qeP = theQP->ancientElemP;
    do
    {
        size++;
        qeP = qeP->nextP;
    }
    while (qeP != NULL);

    return size;
}

int joinQ (void * queueP, void * objP)
{
    struct queue * theQP = (struct queue *) queueP;
    assert (theQP != NULL); /* Not a SPAM queue */
    struct queue_elem * qElemP;

    qElemP = (struct queue_elem *)
             malloc (sizeof(struct queue_elem));
    assert(qElemP != NULL);

    qElemP->objP = objP;
    qElemP->nextP = NULL;

    /* If empty queue */
    if (theQP->recentElemP == NULL &&
            theQP->ancientElemP == NULL)
    {
        theQP->ancientElemP = theQP->recentElemP = qElemP;
        return 1;
    }

    /* Queue has existign entries */
    assert (theQP->recentElemP != NULL &&
            theQP->ancientElemP != NULL);
    theQP->recentElemP->nextP = qElemP;
    theQP->recentElemP = qElemP;
    return sizeQ(theQP);
}

void * leaveQ (void * queueP)
{
    struct queue * theQP = (struct queue *) queueP;
    assert (theQP != NULL); /* Not a SPAM queue */
    void * returnP;
    struct queue_elem * removedElemP;

```

```
    /* Queue has no element */
    if (theQP->ancientElemP == NULL)
        return NULL;

    assert (theQP->recentElemP != NULL &&
            theQP->ancientElemP != NULL); /* Not expected */

    removedElemP = theQP->ancientElemP;
    theQP->ancientElemP = removedElemP->nextP;

    /* If the element being removed is most recent in queue */
    if (removedElemP == theQP->recentElemP)
        theQP->recentElemP =NULL;

    returnP = removedElemP->objP;
    free (removedElemP);
    return returnP;
}
```

The code was tested against the test suite defined previously. After that a small application was developed and it is described in the next subsection.

3.3 An Application Program

The program below is an application program developed to use (object) class `queue`. Admittedly, the application is not a hugely exciting program. More interesting applications are included as assessment exercises for this drill.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "queue.h"

int main()
{
    /* Object struct used by application */
    struct student
    {
        char name[20];
        long rollNo;
    };

    /* **************************************
     * Pointer to the main queue object that
     * will hold student objects
     **************************************/

    struct queue * studentsP;

    studentsP = mkQueue();

    struct student * stdP;
```

```
    int numbers;

    /* Create some students and let them join a queue */
    for (numbers=1; numbers<20; numbers++)
    {
        stdP = (struct student *)
                        malloc(sizeof(struct student));
        strcpy(stdP->name, "Name Is  ");
        /* For test use only */
        stdP->name[8] = 'A' + numbers - 1;
        stdP->rollNo = numbers;
        joinQ (studentsP, stdP);
    }

    /* List all students in the queue */
    while (sizeQ(studentsP) != 0)
    {
        stdP = (struct student *) leaveQ(studentsP);
        printf ("%3ld: %s\n", stdP->rollNo, stdP->name);
    }

    rmQueue (studentsP);

    return 0;
}
```

4 ALTERING QUEUE CODE WITHOUT AFFECTING CLIENTS

One big advantage of abstraction and low coupling across the files is that one can alter the code in a file without requiring any significant modification in the other files of the program.

The approach also helps in finding errors and mistakes by localising them into a single file.

In this section, we will show some examples of improving code. Later, we will also show an example of adding new functionality to the class. This new feature will be added without causing any change in the existing applications.

4.1 AN ALTERNATE IMPLEMENTATION FOR METHOD SIZEQ

If size of a queue is required often in an application, the method provided is not efficient. Instead, one may keep size value and update it as the objects join and leave the queue.

The following version of file `queue.c` implements this change. The new implementation has changed `struct queue`. We have also changed some `assert()` checks to take advantage of readily available `size` value. Yet we require no change in the client applications; the original test suite and application program are unchanged.

```
#include "queue.h"
#include <stdlib.h>
#include <assert.h>

/* Each element of queue has ...*/
struct queue_elem
{
    /* ... a reference to object, and ... */
    void * objP;
    /* ... element who joined after this */
    struct queue_elem * nextP;
};

/* A queue will be referecne to a struct; with ... */
struct queue
{
    /* a reference to the most recent entry to queue...*/
    struct queue_elem * recentElemP;
    /* and the one who joined the earliest */
    struct queue_elem * ancientElemP;
    /* Size of queue */
    int size;
};

void * mkQueue(void)
{
    struct queue * theQP;

    theQP = (struct queue *)malloc(sizeof (struct queue));
    theQP->ancientElemP = NULL;
    theQP->recentElemP = NULL;
    theQP->size = 0;

    return theQP;
};

int rmQueue (void * queueP)
{
    struct queue * theQP = (struct queue *) queueP;

    /* Can not remove non empty queue */
    if (theQP->recentElemP != NULL)
        return 0;
    assert (theQP->recentElemP == NULL &&
            theQP->ancientElemP == NULL);
    free (theQP);
    return 1;
}
```

```
int sizeQ (void * queueP)
{
    struct queue * theQP = (struct queue *) queueP;
    return theQP->size;
}

int joinQ (void * queueP, void * objP)
{
    struct queue * theQP = (struct queue *) queueP;
    struct queue_elem * qElemP;
    if (theQP == NULL)
        return 0; /* No queue! */
    qElemP = (struct queue_elem *)
             malloc (sizeof(struct queue_elem));
    qElemP->objP = objP;
    qElemP->nextP = NULL;
    /* If empty queue */
    if (theQP->size == 0)
    {
        theQP->ancientElemP = theQP->recentElemP = qElemP;
        theQP->size = 1;
        return 1;
    }

    /* Queue has existign entries */
    assert (theQP->size > 0);
    theQP->recentElemP->nextP = qElemP;
    theQP->recentElemP = qElemP;
    theQP->size++;
    return sizeQ(theQP);
}

void * leaveQ (void * queueP)
{
    struct queue * theQP = (struct queue *) queueP;
    void * returnP;
    struct queue_elem * removedElemP;

    /* Queue has no element */
    if (theQP->size == 0)
        return NULL;

    assert (theQP->recentElemP != NULL &&
            theQP->ancientElemP != NULL &&
            theQP->size > 0);
    removedElemP = theQP->ancientElemP;
    theQP->ancientElemP = removedElemP->nextP;
    /* If the element being removed is most recent in queue */
    theQP->size--;
    if (theQP->size == 0)
```

```
        theQP->recentElemP =NULL;

    returnP = removedElemP->objP;
    free (removedElemP);

    return returnP;
}
```

4.2 ADDING NEW METHODS TO CLASS QUEUE

In addition to changes in the implementation that may be caused by changes in the algorithms or removal of program errors there may be changes made to improve the functionality in the interface.

New functions may be added for more interesting applications. We describe a few new methods in the interface file `queue.h`.

```
#ifndef QUEUE_H_INCLUDED
#define QUEUE_H_INCLUDED

/* Returns a reference to a new queue */
void * mkQueue(void);

/* Removes a queue specified by a valid reference */
int rmQueue (void *);

/* Returns number of entries in a validly referenced queue */
int sizeQ (void *);

/* Place a new entry and returns new count of entries */
int joinQ (void * queueP, void * objP);

/* Returns reference to the most ancient arrival in queue */
void * leaveQ (void * queueRef);

/* Force delete queue and its malloc() memory */
int forceRmQueue (void *);

/* Create a new queue by copying the given queue */
void * duplicateQ (void *);

/* Reverse the order of queue entries */
void reverseQ (void *);

#endif // QUEUE_H_INCLUDED
```

Method `forcedRmQueue()` is suggested on the same lines as command `rmdir -F` in Linux. It must free three sets of allocated memory: the memory allocated by the application code for the object references stored in the queue, the memory allocated to the `queue_elem` and also the memory allocated to the `struct queue`. In writing its implementation, in the assessment exercises, students would be asked to assume that application objects saved in queue were allocated memory by call to function `malloc()` .

To support testing, we require the method to return the number of application objects freed (by calls to `stdlib` function `free()`) by this method.

Method `duplicateQ()` is added to duplicate an existing queue. It must create a new queue with a newly allocated memory for `struct queue` and all its `queue_elem`. However, the references to the application objects in `struct queue_elem` will be copied from the queue being duplicated.

Students are cautioned that they should avoid invoking method `forcedRMQueue()` on a queue that has been duplicated by method `duplicateQ()`. It is not safe to call `forcedRMQueue()` on either the original queue or its duplicated queue.

We skip the specification of method `reverseQ()` and leave it to the students for an obvious interpretation of its meaning as its specification. An implementation of the method is provided below.

These new functions should be tested in a new augmented tests-first suite. We leave the task of creating required test suites as subtasks in the relevant assessment exercises.

```
/* Reverse the queue order */
void reverseQ (void * queueP)
{
    struct queue * theQP = (struct queue *) queueP;
    struct queue_elem *nxtP, *thisP, *tempP;

    if (sizeQ(queueP) <= 1)
        return; /* Nothing to be done */

    /* Reverse one element at a time */
    thisP = theQP->ancientElemP;
    nxtP = thisP->nextP; /* thisP != NULL */
    thisP->nextP = NULL;
    while (nxtP != NULL)
    {
        tempP = nxtP->nextP;
        nxtP->nextP = thisP;
        thisP = nxtP;
        nxtP = tempP;
    }

    /* Swap pointers in the queue struct */
    thisP = theQP->ancientElemP;
    theQP->ancientElemP = theQP->recentElemP;
    theQP->recentElemP =thisP;
}
```

Perhaps, adding a call to `reverseseQ()` in the students queue in the application program previously listed is a quick way to reveal the effects of this method.

5 OTHER DISCIPLINES

In the assessment exercises we shall be requiring the students to implement additional methods within class queue.

Other exercises would require the students to develop a similar code for other access disciplines including stacks and sorted lists.

5.1 DISCIPLINE STACK

A stack is a class very similar to class queue described above in this drill document. Its interface is listed below. The items join and leave a stack in last-in-first-out basis.

```
#ifndef STACK_H_INCLUDED
#define STACK_H_INCLUDED

/* Returns a reference to a new stack */
void * mkStack(void);

/* Removes empty stack specified by a valid reference */
int rmStack (void *);

/* Place a new entry into stack */
void push (void * stackP, void * objP);

/* Returns reference to the most recent arrival in stack*/
void * pop (void * queueRef);

#endif // STACK_H_INCLUDED
```

5.2 A SORTED LIST

A list is a collection of objects that are organised in a fashion similar to the one discussed for queues earlier. However, the objects may join and leave a list following a different discipline.

Each object has a key. The application code specifies a key when entering an object into the list. Later the application uses the key to retrieve an object reference from the list. The lists may also be accessed for the object by other methods for example, get first object in the list.

```
#ifndef LIST_H_INCLUDED
#define LIST_H_INCLUDED

/* Returns a reference to a new list */
void * mkList(void);

/* Place a new entry */*/
```

```
void insert (void * listP, void * objP, long key);

/* Returns/removes reference to object with key */
void * remove (void * listP, long key);

/* Returns reference/removes first object in list */
void * first (void *);

#endif // LIST_H_INCLUDED
```

In the assessment exercises you will be asked to write codes for several variants of the lists. Perhaps in all these variants, it will help if you use the following struct as `list_elem`:

```
/* Each element of list has ...*/
struct list_elem
{
    /* ... a reference to object, and ... */
    void * objP;
    /* Each element has a key */
    long key;
    /* ... elements are joined together by ... */
    struct list_elem * nextP;
};
```

Let us describe three variants and assign them descriptive names.

5.2.1 SortedList

In a sorted list we would expect the list elements to be organised in ascending order of their key values. Thus, method `first()` will return reference to the object in `sortedList` with smallest key value.

5.2.2 QueuedList

The list elements are stored in order matching the queue. Thus, method `first()` will be similar to `leaveQ()`. However, it would also be possible to retrieve objects using key through method `remove()`.

5.2.3 StackedList

This list uses stack-like ordering of list elements. Method `first()` is like method `pop()` of stack in its behaviour.

6 ASSESSMENT EXERCISES

The assessment exercise attached to this drill are of four kind.

6.1 ENHANCEMENTS OF CLASS QUEUE INTERFACE

The enhancements have already been mentioned earlier in this document.

6.2 Uses of Class Queue in Applications

The exercise descriptions would suggest ways to use the queue objects. Very often a single problem will have different ways of implementation.

6.3 Implementation of Class Stack

These exercises will come with an additional task of building tests-first suite so that you can validate your implementations of class `stack` and demonstrate them to your tutor.

6.4 Implementation of Interfaces for Lists

These lists will require new entries to be added to and/or removed from the lists at locations within a list. Typically there will be a search for the location where the change is to occur. The search would usually provide a pointer to the `list_elem` whose `nextP` value is going to change to add or remove the target `list_elem`.

Some further guidance will be provided in the exercise statement. Some exercises may have same specification of the problem but different possible implementations. These exercises are written as separate assessment exercises.

7 Error Reporting and Suggestions for Improvements

My sincere apologies if the document has errors or mistakes. Please report errors in this document to vmm@iitg.ernet.in. Also, I welcome suggestions and advice to improve the quality of the document for the students of CS110.

8 Appendix

8.1 Pointer with Type and Pointer without Type

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    struct test
    {
        int id;
    };

    /* *********************************
     * canSeeIdP will let us assign id.
     *
     * cannotSeeIdP is pure reference and
     * has no access to id.
```

```
     *
     * *********************************/
    struct test * canSeeIdP;
    void * cannotSeeIdP;

    /* *********************************
     * malloc() return void * reference
     * to allocate memory
     **********************************/
    cannotSeeIdP = malloc(sizeof(struct test));
    /* Coerce to teh desired type */
    canSeeIdP = (struct test *) cannotSeeIdP;

    /* The following assignment (currently commented
     * out) failed compilation */
    // cannotSeeIdP->id = 10;

    /* *******************************
     * This is ok as type of pointer
     * is informative
     ********************************/
     canSeeIdP->id = 10;
     printf("Id is %d\n", canSeeIdP->id);
}
```

8.2 COMPILER DIRECTIVES IN .H FILE (THIS PART IS OPTIONAL)

It is OK to write this in a .c file!

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "queue.h"
#include "queue.h"
```

Rest of the file has been deleted

When first `#include "queue.h"` is noticed, compiler notes that it does not know `QUEUE_H_INCLUDED`. So `#ifndef` directive continues to read file as `ifndef` means *If Not Defined*.

The compiler however, marks `QUEUE_H_INCLUDED` as defined when it sees `#define` directive in the second line in file `queue.h`. All lines in file `queue.h` upto `#ifend` are included and processed by the compiler.

However, second `#include "queue.h"` adds no new line and skips all lines in file `queue.h`. This time compiler finds that `QUEUE_H_INCLUDED` is defined!

8.3 AUDITING THE ASSETS

This is optional section for those who may wish to explore the properties of the data-structures more deeply.

All data-structures/classes we discussed in this drill have a similar arrangement. Each class had a header `struct` acting as the reception for the data-structure object. In all examples in this drill this `struct` was named by the best known name for the data-structure. Thus, we used the names like queue, stack, sortedList.

The references created and given to the application code for these data-structures by their make methods were always to these `struct` objects. These header `struct` contain one or more pointers to the element struct of the class. As more entries are added to a storage object, more element struct are created and linked into the main object. For example, a queue as a queue grows because more objects invoke method `joinQ()`, it increases count of linked `struct queue_elem`.

In a first year course, we have restricted to simple data-structures. Each `struct queue_elem` has exactly one member `nextP` pointing at a `struct queue_elem`! In our main implementation of class queue, the queue header has two pointers to `struct queue_elem`.

Here are some easy audit facts to count the different kinds of the `struct` and pointers in a `queue`.

Let N be the number of `struct queue_elem` in class queue implemented in our drill lessons.

Question 1: How many pointers are there to `struct queue_elem` in a correctly implemented queue?

Answer 1: N + 1. Why? There are two such pointers in the header `struct queue`. Each element `struct queue_elem` has one pointer. But, the most recently inserted element has `nextP` that is `NULL`.

Question 2: How many distinct application objects a queue can hold?

Answer 2: No more than N. Why? Each `struct queue_elem` can have one reference to application object. However, some of them may be references to the same object. Also some references can be `NULL`.

Question 3: How many pointers can point to a `struct queue_elem`?

Answer 3: There are N such `struct` and there are N + 1 pointers to them. This means all but one of these `struct` have exactly one pointer pointing at it. There is exactly one `struct` with two pointers pointing at it.

Note: If this property is violated queue is broken and implementation is incorrect!

9 LAST UPDATE: MONDAY 12 MARCH 2018