#### CMPS 12M

**Introduction to Data Structures Lab**

**Lab Assignment 5**

The goal of this assignment is to learn how to implement data structures in C. We will discuss the *typedef* and *struct* commands, header files, information hiding, constructors and destructors, and memory management. You will write a program in C that creates the Queue data structure (also referred as the Queue ADT).

**Creating New Data Types in C**

The struct keyword is used to create a new aggregate data type called a *structure* or just *struct*, which is the closest thing C has to Java's class construct. Structs contain data fields, but no methods, unlike Java classes. A struct can also be thought of as a generalization of an array. An array is a contiguous set of memory areas all storing the same type of data, whereas a struct may be composed of different types of data. The general form of a struct declaration is

struct structure\_tag{

// data field declarations

};

Note the semicolon after the closing brace. For example

struct person{

int age;

int height;

char first[20];

char last[20];

};

The above code does not allocate any memory however, and in fact does even not create a complete type called person. The term person is only a tag which can be used with struct to declare variables of the new type.

struct person fred;

After this declaration fred is a local variable of type struct person, i.e. a symbolic name for an area of stack memory storing a person structure. By comparison, a Java reference variable is a pointer to heap memory. As we shall see, it is possible and desirable to declare C structures from heap memory as well. The variable fred contains four components, which can be accessed via the component selection (dot ".") operator:

fred.age = 27;

fred.height = 70;

strcpy(fred.first, "Fredrick");

strcpy(fred.last, "Flintstone");

See the man pages or google to learn more about strcpy() in the library string.h.

The struct command is most often used in conjunction with typedef, which establishes an alias for an existing data type. The general form of a typedef statement is:

typedef existing\_type new\_type;

For instance

typedef int feet;

defines feet to be an alias for int. We can then declare variables of type feet by doing

feet x = 32;

Using typedef together with struct allows us to declare variables of the structure type without having to include struct in the declaration. The general form of this combined typedef statement is:

typedef struct structure\_tag{

/\* data field declarations \*/

} new\_type;

The structure\_tag is only necessary when one of the data fields is itself of the new type, and can otherwise be omitted. Often the tag is included simply as a matter of convention. Also by convention structure\_tag and new\_type are the same identifier, since there is no reason for them to differ. Going back to the person example above we have

typedef struct person{

int age;

int height;

char first[20];

char last[20];

} person;

We can now declare

person fred;

and assign values to the data fields of fred as before. It is important to remember that the typedef statement itself allocates no memory, only the declaration does. To allocate a person structure from heap memory, we do

person\* pFred = malloc(sizeof(person));

The variable pFred points to a person structure on the heap. Note that pFred itself is a symbolic name for an area of stack memory storing the *address of* a block of heap memory storing a person structure. This is essentially the situation one has in java when declaring a reference variable of some class type. To access the components of the person structure pointed to by pFred, we must first dereference (i.e. follow) the pointer using the indirection (value-at) operator \*. Unfortunately the expression \*pFred.first is not valid since the component selection (dot ".") operator has higher precedence than value-at \*. We could insert parentheses to get (\*pFred).first, but this leads to some unwieldy expressions. Fortunately C provides a single operator combining the value-at and dot operators called the indirect component selection (arrow ->) operator. Note this operator is represented by two characters with no separating space. To assign values to the components of the person pointed to by pFred, do

pFred->age = 27;

pFred->height = 70;

strcpy(pFred->first, "Fredrick");

strcpy(pFred->last, "Flintstone");

Thus the C operator that is equivalent to the familiar dot operator in Java is not component selection (dot "."), but indirect component selection (arrow "->"). The following example defines a new data type called NodeObj that has a pointer to NodeObj as one of its members.

typedef struct NodeObj{

int item;

struct NodeObj\* next;

} NodeObj;

In this case the NodeObj tag is necessary since the definition itself refers to NodeObj. Observe however that within the body of the structure definition, NodeObj is referred to as struct NodeObj since the typedef statement is not yet complete. Outside the structure definition we can simply use NodeObj as a new type name. Another typedef statement defines Node as being a pointer to NodeObj.

typedef NodeObj\* Node;

To declare and initialize a reference (pointer) to a NodeObj we do

Node N = malloc(sizeof(NodeObj));

N->item = 5;

N->next = NULL;

Two rules to remember when using structures to define ADTs in C: (1) always use typedef and struct together as in the last example do define new structure types and pointers to those types, and (2) always declare your structure variables as pointers to heap memory and access their components via the arrow operator. Do not declare structure variables from stack memory. Our goal here is to emulate as closely as possible the class construct as it appears in the Java language.

Please take the time to try the samples above to increase your understanding of how structs are used in C.

**Information Hiding**

The C language does not include access modifiers such as Java's public and private keywords. To enforce the principle of information hiding we split the definition of an ADT into two files called the *header file* (with suffix .h), and the *implementation file* (with suffix .c). The header file constitutes the ADT interface, and is roughly equivalent to a Java interface file. It contains the prototypes of all public ADT operations together with typedef statements defining exported types. One of the exported types in the header file is a pointer (also called a *handle* or *reference*) to a structure that encapsulates the data fields of the ADT. The definition of that structure is placed in the implementation (.c) file, along with definitions of any private types, together with function definitions, both public and private. The implementation (.c) file will #include its own header (.h) file. ADT operations are defined to take in and return references to the structure type, exactly as is the case in Java.

A client module will then #include the header (.h) file giving it the ability to declare variables of the reference type, as well as functions that either take or return reference type parameters. The client cannot dereference this pointer however, since the structure it points to is not defined in the header file. The ADT operations take reference arguments, so the client does not need to (and is in fact unable to) directly access the structure these references point to. The client can therefore interact with the ADT only through the public ADT operations and is prevented from accessing the interior of the so called 'black box'. This is how information hiding is accomplished in C. One establishes a function as public by including its prototype in the header file, and private by leaving it out of the header file. Likewise for the reference types belonging to an ADT.

We illustrate with the following C implementation of an IntegerLinkedList. This example has been abridged to save space.

//-----------------------------------------------------------------------------

// IntegerLinkedList.h

// Header file for the IntegerLinkedList ADT

//-----------------------------------------------------------------------------

#ifndef \_INTEGER\_LINKEDLIST\_H\_INCLUDE\_

#define \_INTEGER\_LINKEDLIST\_H\_INCLUDE\_

#include<stdio.h>

// LinkedList

// Exported reference type

typedef struct LinkedListObj\* LinkedList;

// Node

// Exported reference type

typedef struct NodeObj\* Node;

//constructor for node

Node newNode(int x);

// newLinkedList()

// constructor for the LinkedList type

LinkedList newLinkedList(void);

// freeLinkedList()

// destructor for the LinkedList type

void freeLinkedList(LinkedList\* pS);

//-----------------------------------------------------------------------------

// prototypes of ADT operations deleted to save space

//-----------------------------------------------------------------------------

// printLinkedList()

// prints a text representation of S to the file pointed to by out

// pre: none

void printLinkedList(FILE\* out, LinkedList S);

// insert()

// insert number into linked list

// pre: none

void insert(int number, LinkedList S);

// find()

// find pointer to node containing number (read next code snippet for details), return // null if none exists

// pre: none

Node find(int number, LinkedList S);

// delete()

// delete number from linked list

// pre: none

void delete(int n, LinkedList S);

#endif

This file contains some preprocessor commands we haven't yet seen for conditional compilation, namely #ifndef and #endif. If the C compiler encounters multiple definitions of the same type, or multiple prototypes of any function, it is considered a syntax error. Therefore when a program contains several files, each of which may #include the same header file, it is necessary to place the content of the header file within a conditionally compiled block (sometimes called an “include guard”), so that the prototypes etc. are seen by the compiler only once. The general form of such a block is

#ifndef \_MACRO\_NAME\_

#define \_MACRO\_NAME\_

statements

#endif

If \_MACRO\_NAME\_ is undefined then the statements between #ifndef and #endif are compiled. Otherwise these statements are skipped. The first operation in the block is to #define \_MACRO\_NAME\_. Notice that the macro is not defined to *be* anything, it just needs to be defined. It is customary to choose \_MACRO\_NAME\_ in such a way that it is unlikely to conflict with any "legitimate" macros. Therefore the name usually begins and ends with an underscore \_ character.

The next item in IntegerLinkedList.h is the typedef command defining LinkedList to be a pointer to struct LinkedListObj. The definition of struct LinkedListObj, which contains the data fields for the IntegerLinkedList ADT, will be placed in the implementation file IntegerLinkedList.c.

**Memory Management**

Structure types usually have their own constructor that allocates heap memory and initializes data fields, as well as a destructor that balances calls to malloc() and calloc() in the constructor with corresponding calls to free(). Observe that the arguments to freeNode() and freeLinkedList() are not the reference types Node and LinkedList, but are instead pointers to these types. The reason for this is that the destructor must alter, not just the object a reference points to, but also the reference itself by setting it to NULL. Why must the destructor do this? Recall that maintaining a pointer to a free block on the heap is a major memory error in C. The responsibility for setting such pointers safely to NULL should lie with the ADT module, not the with client module. To accomplish this reference types are themselves passed by reference to their destructor.

As in Java, all ADT operations should check their own preconditions and exit with a useful error massage when one is violated. This message should state the module and function in which the error occurred, and exactly which precondition was violated. The purpose of this message is to provide diagnostic assistance to the designer of the client module. In C however there is one more item to check. Every ADT operation should verify that the reference that is its main argument is not NULL. This check should come before the checks of preconditions since any attempt to dereference a NULL handle will result in a segmentation fault. The reason this was not necessary in java was because calling an instance method on a null reference variable causes a NullPointerException to be thrown, which provides some error tracking to the client programmer.

**Naming Conventions**

Suppose you design an ADT in C called Blah. Then the header file will be called Blah.h and should define a reference type Blah that points to a structure type BlahObj.

typedef struct BlahObj\* Blah;

The header file should also contain prototypes for ADT operations. The implementation file will be called Blah.c and should contain the statement

typedef struct BlahObj{

// data fields for the Blah ADT

} BlahObj;

together with constructors and destructors for the BlahObj structure. File Blah.c will also contain definitions of all public functions (i.e. those with prototypes in Blah.h) as well as definitions of private types and functions. The general form for the constructor and destructor (respectively) are

Blah newBlah(arg\_list){

Blah B = malloc(sizeof(BlahObj));

assert( B!= NULL );

// initialize the fields of the Blah structure

return B;

}

and

void freeBlah(Blah\* pB){

if( pB!=NULL && \*pB!=NULL){

// free all heap memory associated with \*pB

free(\*pB);

\*pB = NULL;

}

}

Again note that the destructor passes its Blah argument by reference, so it can set this pointer to NULL. Given a Blah variable B a call to freeBlah() would look like

freeBlah(&B);

The general form for an ADT operation is

return\_type some\_op(Blah B, other\_parameters){

if( B==NULL ){

fprintf(stderr, "Blah Error: some\_op() called on NULL Blah reference\n");

exit(EXIT\_FAILURE);

}

// check preconditions

// do whatever some\_op() is supposed to do

}

Most ADTs should also contain a printBlah() function that prints a text representation of a Blah object to a file stream. This function is roughly equivalent to the toString() function in Java.

void printBlah(FILE\* out, Blah B){

if( B==NULL ){

fprintf(stderr,

"Blah Error: printBlah() called on NULL Blah reference\n");

exit(EXIT\_FAILURE);

}

// calls to fprintf(out, text\_representation\_of\_B)

}

**What to Turn In**

Implement a Queue of ints in C. This should support the obvious operations of enqueue and dequeue. We require an additional function, print, that prints the current queue in order. You should implement the queue as a linked list.

The interface for the Queue ADT should be in the file queue.h. This contains the descriptions of the above functions. The actual code for them will be in queue.c.

Finally, write a “client” called queueClient.c, which is compiled into the executable queueClient. (Write a Makefile to achieve this.) This client contains the main function that will read the input, and perform all the operations.

This executable takes command line inputs to an input and output file in the same directory. The input file has a series of instructions for the client. For each instruction, the output file contains a line.

e <int>: enqueue the int. The output file contains line “enqueued <int>”

d: dequeue. The output file prints the int dequeued. If queue is empty, print “empty”

p: print. Print the queue in order, where the beginning appears first. If queue is empty, print empty line

For example, the input file could be:

e 45

e 42

e 21

p

d

p

d

d

p

d

e 32

The output file should contain:

enqueued 45

enqueued 42

enqueued 21

45 42 21

45

42 21

42

21

empty

enqueued 32

Do make clean to delete old binaries and make check to check your for memory leaks. If you’re using Booleans, remember to include stdbool.h.

The Lab5 directory should contain:

README

Makefile

queue.c

queue.h

queueClient.c

As always start early and ask for help if anything is not completely clear.

**Grading:**

* (10 points) Everything done correctly
* (8 points) Code correct, but Makefile is incorrect
* (8 points) Everything correct, but valgrind reports a memory leak. queueClient does not crash.
* (6 points) queueClient makes occasional mistakes.
* (3 points) queueClient.c compiles and doesn’t crash.