Homework Two

Q1. Modify the stack implementation in the lecture notes (Stack.h and Stack.c) to implement a stack of integers.

Solution:

IntStack.h

```
// Integer Stack ADO header file
void StackInit();     // set up empty stack
int  StackIsEmpty();  // check whether stack is empty
void StackPush(int);  // insert int on top of stack
int  StackPop();      // remove int from top of stack
```

IntStack.c

```
// Integer Stack ADO implementation
#include "IntStack.h"
#include <assert.h>

#define MAXITEMS 10

static struct {
   int item[MAXITEMS];
   int top;
} stackObject;  // defines the Data Object

void StackInit() {        // set up empty stack
   stackObject.top = -1;
}

int StackIsEmpty() {     // check whether stack is empty
   return (stackObject.top < 0);
}

void StackPush(int n) {  // insert int on top of stack
   assert(stackObject.top < MAXITEMS-1);
   stackObject.top++;
   int i = stackObject.top;
   stackObject.item[i] = n;
}
```

```
int StackPop() {          // remove int from top of stack
   assert(stackObject.top > -1);
   int i = stackObject.top;
   int n = stackObject.item[i];
   stackObject.top--;
   return n;
}
```

Q2. Write a test program for your stack code in **Q**1 that does the following:

- initialise the stack
- prompt the user to input a number *n*
- check that *n* is a positive number
- prompt the user to input *n* numbers and push each number onto the stack
- use the stack to output the *n* numbers in reverse order

An example of the program executing could be

```
Enter a positive number: 3
Enter a number: 2017
Enter a number: 12
Enter a number: 24
24
12
2017
```

Solution:

```
#include <stdio.h>
#include "IntStack.h"

int main(void) {
   int i, n, number;

   StackInit();

   printf("Enter a positive number: ");
   if (scanf("%d", &n) == 1 && (n > 0)) {   // test if scanf
successful and returns positive number
```

```
        for (i = 0; i < n; i++) {
        printf("Enter a number: ");
        scanf("%d", &number);
        StackPush(number);
        }
        while (!StackIsEmpty()) {
        printf("%d\n", StackPop());
        }
    }
    return 0;
}
```

Q3. Modify your program in **Q**2 so that it takes the *n* numbers from the command line. An example of the program execution could be

```
prompt$./tester 2017 12 24
24
12
2017
```

Solution:

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

int main(int argc, char *argv[]) {
    int i;

    StackInit();
    for (i = 1; i < argc; i++) {
        StackPush(atoi(argv[i]));
    }
    while (!StackIsEmpty()) {
        printf("%d\n", StackPop());
    }
    return 0;
}
```

Q4. A stack can be used to convert a positive decimal number *n* to a different numeral system with base *k* according to the following algorithm:

while n>0 **do**
  push *n*%*k* onto the stack
  n = *n* / *k*
end while

The result can be displayed by printing the numbers as they are popped off the stack. Example (*k*=2):

```
n = 13        --> push 1 (= 13%2)
n = 6  (= 13/2) --> push 0 (= 6%2)
n = 3  (=  6/2) --> push 1 (= 3%2)
n = 1  (=  3/2) --> push 1 (= 1%2)
n = 0  (=  1/2)
Result: 1101
```

Using your stack code in Q1, write a C program to implement this algorithm to convert to base *k*=2 a number given on the command line. Design a Makefile to compile this program along with the integer stack implementation.

An example of program compilation and execution could be

```
prompt$ make
gcc -Wall -Werror -c binary.c
gcc -Wall -Werror -c IntStack.c
gcc -o binary binary.o IntStack.o
./binary 13
1101
./binary 128
10000000
./binary 127
1111111
```

Solution:

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

```
int main(int argc, char *argv[]) {
   int i;

   StackInit();
   for (i = 1; i < argc; i++) {
      StackPush(atoi(argv[i]));
   }
   while (!StackIsEmpty()) {
      printf("%d\n", StackPop());
   }
   return 0;
}
```

Makefile

```
binary : binary.o IntStack.o
     gcc -o binary binary.o IntStack.o

binary.o : binary.c IntStack.h
     gcc -Wall -Werror -c binary.c

IntStack.o : IntStack.c IntStack.h
     gcc -Wall -Werror -c IntStack.c
```

Q5. Implement a queue of integers in C using an array to store all the integers. All the function prototypes of the integer queue are defined in IntQueue.h as follows:

```
// Integer queue header file
void queueInit();       // set up an empty queue
int  isEmpty();    // check whether the queue is empty
void enqueue(int);  // insert int at the end of queue
int  dequeue();    // remove int from the front of queue
```

Solution:

IntQueue.c

```
// Integer Queue ADO implementation
#include "IntQueue.h"
#include <assert.h>
```

```
#define MAXITEMS 10
static struct {
   int item[MAXITEMS];
   int top;
} queueObject;  // defines the Data Object

void QueueInit() {          // set up empty queue
   queueObject.top = -1;
}

int QueueIsEmpty() {        // check whether queue is empty
   return (queueObject.top < 0);
}

void QueueEnqueue(int n) {  // insert int at end of queue
   assert(queueObject.top < MAXITEMS-1);
   queueObject.top++;
   int i;
   for (i = queueObject.top; i > 0; i--) {
      queueObject.item[i] = queueObject.item[i-1]; // move all
elements up
   }
   queueObject.item[0] = n; // add element at end of queue
}

int QueueDequeue() {        // remove int from front of queue
   assert(queueObject.top > -1);
   int i = queueObject.top;
   int n = queueObject.item[i];
   queueObject.top--;
   return n;
}
```

Q6. Given the following definition:

int data[12] = {5, 3, 6, 2, 7, 4, 9, 1, 8};

and assuming that *&data[0] == 0x10000*, what are the values of the following expressions?

data + 4

*data + 4
*(data + 4)
data[4]
*(data + *(data + 3))
data[data[2]]

Solution:

data + 4	== 0x10000 + 4 * 4 bytes == 0x10010
*data + 4	== data[0] + 4 == 5 + 4 == 9
*(data + 4)	== data[4] == 7
data[4]	== 7
*(data + *(data + 3))	== *(data + data[3]) == *(data + 2) == data[2] == 6
data[data[2]]	== data[6] == 9

Q7. Consider the following piece of C code:

```
typedef struct {
  int   studentID;
  int   age;
  char  gender;
  float WAM;
} PersonT;

PersonT per1;
PersonT per2;
PersonT *ptr;

ptr = &per1;
per1.studentID = 3141592;
ptr->gender = 'M';
ptr = &per2;
ptr->studentID = 2718281;
```

```
ptr->gender = 'F';
per1.age = 25;
per2.age = 24;
ptr = &per1;
per2.WAM = 86.0;
ptr->WAM = 72.625;
```

What are the values of the fields in the *per1* and *per2* record after execution of the above statements?

Note that ptr->t means the same as (*ptr).t

Solution:

per1.studentID	== 3141592
per1.age	== 25
per1.gender	== 'M'
per1.WAM	== 72.625
per2.studentID	== 2718281
per2.age	== 24
per2.gender	== 'F'
per2.WAM	== 86.0

Q8. Write a C program that takes 1 command line argument and prints all its *prefixes* in decreasing order of length. You are *not* permitted to use any library functions other than printf(). You are also *not* permitted to use any array other than argv[].

An example of the program execution could be

```
prompt$ ./prefixes Programming
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
```

```
Prog
Pro
Pr
P
```

Solution:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   char *start, *end;

   if (argc == 2) {
      start = argv[1];
      end = argv[1];
      while (*end != '\0') {    // find address of terminating
'\0'
         end++;
      }
      while (start != end) {
         printf("%s\n", start); // print string from start to '\0'
         end--;                 // move end pointer up
         *end = '\0';           // overwrite last char by '\0'
      }
   }
   return 0;
}
```