CSE 15
Introduction to Data Structures
Lab Assignment 3

The goal of this assignment is to learn how to implement ADTs in C. We will discuss the *typedef* and *struct* commands, header files, information hiding, constructors and destructors, and memory management. You will then create a stack ADT in C.

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 the class construct in C++, Java or Python. 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 the keyword `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. the name of a local variable (so stack memory) storing a `person` structure. By comparison, a reference variable in an OOP language like C++, Java or Python 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 one of the C reference pages linked on the course webpage 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 struct` 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 just 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 then 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 would do

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

The variable `pFred` points to a `person` structure on the heap. Note that `pFred` itself is the name of a local variable (stack memory) storing the *address of* a block of heap memory, which itself stores 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 `"->"`).

Abstract Data Types (ADTs)
Recall that an ADT consists of two things:

(1) A set $\mathcal{S}$ of discrete mathematical structures called the *states* of the ADT.
(2) An associated set of *operations* on these states.

In the Integer List ADT discussed in class, $\mathcal{S}$ is the set of finite sequences of integers (up to a fixed maximum length MAX_LENGTH.) The six ADT operations are:

1. isEmpty(L): Return true if list L is in the empty state, false otherwise.
2. size(L): Return the length of the integer sequence L.
3. get(L, index): Return the integer at position index in L. Pre: $1 \leq \text{index} \leq \text{size}(L) \leq \text{MAX\_LENGTH}$
4. add(L, index, x): Inserts x into L at position index. Pre: $1 \leq \text{index} \leq \text{size}(L) + 1 \leq \text{MAX\_LENGTH}$
5. delete(L, index): Deletes integer at position index in L. Pre: $1 \leq \text{index} \leq \text{size}(L) \leq \text{MAX\_LENGTH}$
6. deleteAll(L): Resets L to the empty state.

These operations fall into two general categories called *Access Functions* (1-3), which return information about a state but do not alter it, and *Manipulation Procedures* (4-6) which do alter the state. Most ADT implementations also include functions that don't fall neatly into these categories.

To implement this ADT in C, we use typedef and struct to define a new type for the an Integer List object.

```
typedef struct IntegerListObj{
   int* item;     // array of items
   int numItems;  // number of items
} IntegerListObj;
```

We then define a pointer type to this new type called `IntegerList`, which is analogous to a reference type in Java.

```
typedef struct IntegerListObj* IntegerList;
```

We can now declare and initialize a reference (i.e. pointer) to a `IntegerListObj` by doing

```
IntegerList L = malloc(sizeof(IntegerListObj));
L->item = calloc(MAX_LENGTH, sizeof(int));
L->numItems = 0;
```

The reference L is now a program representation of an instance of the Integer List ADT in the empty state. There are two rules to remember when using structs to define ADTs in C. First, always use `typedef` and `struct` together as above to define new structure types and pointers to those types. Second, 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, as was done in our first few examples. (Our goal is to emulate as closely as possible the notion of a class in an OOP language like Python or Java.)

Information Hiding
The Information Hiding principle is the idea that the user (client) of an ADT need not know (and cannot know) its internal details. In OOP languages this is usually accomplished through the use of access modifiers like

`public` and `private`, which do not exist in C. 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. The header (`.h`) file defines a pointer (`IntegerList` above) to a struct (`IntegerListObj`) that encapsulates the data fields of the ADT. The struct itself is placed in the implementation (`.c`) file, along with definitions of any private types, and function definitions, both public and private. The implementation (`.c`) file will `#include` its own header (`.h`) file. ADT operations are defined to take a reference argument that is a pointer to the struct type.

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.

Integer List ADT

The full implementation of the Integer List ADT can be found in Examples/Lecture/IntegerListADT/Array/ on the class webpage. An abridged version of this example is presented here to save space.

```
// IntegerList.h
// Header file for the Integer List ADT in C
#include<stdlib.h>
#include<stdio.h>

#ifndef _List_H_INCLUDE_
#define _List_H_INCLUDE_

#define MAX_LENGTH 1000

// Exported type ------------------------------------

// IntegerList
// Exported reference type
typedef struct IntegerListObj* IntegerList;

// Constructors-Destructors -------------------------

// newIntegerList()
// Constructor for the IntegerList ADT
IntegerList newIntegerList();

// freeIntegerList()
// Destructor for the List ADT
void freeIntegerList(IntegerList* pL);

// Prototypes for ADT operations deleted to save space, see webpage
```

```
// Other Operations ---------------------------------

// printIntegerList()
// Prints a space separated list of items in L to file
// stream pointed to by out.
void printIntegerList(FILE* out, IntegerList L);

// equals()
// Returns true (1) if L and R are matching sequences
// of integers, false (0) otherwise.
int equals(IntegerList L, IntegerList R);

#endif
```

This file contains preprocessor commands for conditional compilation that we have not yet seen, 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 `IntegerList.h` is the `typedef` command defining `IntegerList` to be a pointer to `struct IntegerListObj`. The definition of `struct IntegerListObj`, which contains the data fields for the IntegerList ADT, will be placed in the implementation file below. Next are prototypes for the constructor `newIntegerList()` and destructor `freeIntegerList()`, followed by prototypes of ADT operations (skipped in this document, but given on the class webpage.) Finally a prototype is included for functions `printIntegerList()` and `equals()`.

```
// IntegerList.c
// Array based implementation of Integer List ADT in C
#include<stdlib.h>
#include<stdio.h>
#include"IntegerList.h"

// Private Types and Functions ----------------------------

// IntegerListObj
typedef struct IntegerListObj{
   int* item;     // array of IntegerList items
   int numItems;  // number of items in this IntegerList
} IntegerListObj;
```

```
// arrayIndex()
// transforms an IntegerList index to an Array index
int arrayIndex(int listIndex){
   return listIndex-1;
}

// Constructors-Destructors ---------------------------------

// newIntegerList()
// Constructor for the IntegerList ADT
IntegerList newIntegerList(){
   IntegerList L = malloc(sizeof(IntegerListObj));
   L->item = calloc(MAX_LENGTH, sizeof(int));
   L->numItems = 0;
   return L;
}

// freeIntegerList()
// Destructor for the List ADT
void freeIntegerList(IntegerList* pL){
   if( pL!=NULL && *pL!=NULL ){
      free((*pL)->item);
      free(*pL);
      *pL = NULL;
   }
}

// ADT operations and other functions deleted to save space, see webpage
```

This file defines the private type `IntegerListObj` , which encapsulates the data fields for the Integer List ADT. Type `IntegerList`, defined in the header (`.h`) file, is a pointer to this struct, and is the reference through which the client interacts with ADT operations. The implementation file also contains one private helper function `arrayIndex()` used for translating between array indices and list indices. It also contains a constructor `newIntegerList()` that allocates heap memory and initializes data fields, and destructor `freeIntegerList()` that balances calls to `malloc()` and `calloc()` in the constructor with corresponding calls to `free()`.

Observe that the argument to `freeIntegerList()` is not the reference type `IntegerList`, but instead a pointer to this type. The reason for the added level of indirection 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 serious memory error in C. It is our policy that the responsibility for setting such pointers safely to `NULL` will lie with the ADT module, not the with client module. To accomplish this, reference types are themselves passed by reference to their destructor.

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 the 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 must verify that its reference argument is not `NULL`. This check should come before the checks of other preconditions since any attempt to dereference a `NULL` reference will result in a segmentation fault.

Study the full example carefully. Also study another version of the IntegerList ADT, which can be found on the webpage under /Examples/Lecture/IntegerListADT/ArrayDoubling/. This version, also based on an array data structure, creates a new array from heap memory whenever the current array is full, and replaces the old with the new. This process is called array doubling, and it removes the need for the constant macro

MAX_LENGTH, allowing the precondition for function add() to change from $1 \leq \text{index} \leq \text{size}(L) + 1 \leq$ MAX_LENGTH to the simpler $1 \leq \text{index} \leq \text{size}(L) + 1$.

A Word on Naming Conventions

Suppose you are designing 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 struct called `BlahObj`.

```
typedef struct BlahObj* Blah;
```

The header file should also contain prototypes for other 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` type. 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 ); // optional, not required
   // 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 any 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 other preconditions
   // do whatever some_op() is supposed to do
}
```

Most ADTs should also contain a print function that prints a string representation of the corresponding object to an arbitrary file stream.

```
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, arg_list) giving a string
   // representation of B
}
```

Every ADT should be tested thoroughly in isolation before it is used in a larger program. The goal of such tests is to exercise every logical pathway of execution in each exported function, so it will not fail under any circumstances. This includes tests of the errors (and error messages) that arise when a client violates a precondition. A test client called `BlahTest.c` or `BlahClient.c` should be written for this purpose, along with a `Makefile` that builds the whole thing.

A Google search on the phrase "implementing an ADT in C" will bring up many documents describing procedures that differ in some respects from the one outlined in this document. They are all essentially the same however, and their differences are mostly in the names given to files, types and functions. It is the policy of this class that students adhere to the naming conventions in this document for the sake of uniformity, clarity and ease of grading.

The IntegerStack ADT

The Integer Stack ADT is similar to the Integer List in that its set $\mathcal{S}$ of states is the same, namely the set of finite sequences of integers (although there will be no limit on the length of a sequence.) A stack has somewhat less functionality though. The defining feature of a stack is that all insertions and deletions take place at one end of the list, referred to as the *top* of the stack. Thus the last integer added is always the first one to be deleted. This is often called a LIFO structure (for Last In First Out). We usually picture a stack as vertical list rather than horizontal, as in the List ADT. The five Integer Stack operations are:

1. isEmpty(S): Return true if stack S is in the empty state, false otherwise.
2. push($S,\ x$): Inserts x on top of stack S.
3. pop(S): Delete and return the integer at top of stack S. Pre: !isEmpty(S)
4. peek(S): Return integer at top of stack S. Pre: !isEmpty(S)
5. popAll(S): Resets S to the empty state.

A header (`.h`) file for the Integer Stack ADT can be found at /Examples/Labs/lab3/IntegerStack.h on the course website. The header contains prototypes for the above ADT operations, a constructor, destructor, print function and equals function. The examples directory also contains a simple test client, and a Makefile.

What to Turn In

Complete the Integer Stack ADT by creating an implementation (`.c`) file called `IntegerStack.c` that implements all of the functions in the header, along with any necessary helper functions and types. Use the array doubling technique described above (and illustrated in the second version of the Integer List ADT) to enable the stack to have unlimited size. Think carefully about where (i.e. at what index) in the underlying array is the top of the stack. There are basically two choices: index 0 or index $n-1$ (where n is the number

of integers in the stack.) One of these choices leads to a very efficient implementation, the other to an inefficient implementation. You are required to choose the efficient one.

You may not alter the header file `IntegerStack.h`, or the test client `IntegerStackTest.c` in any way, even to place an identifying comment block at the beginning. You may alter the `Makefile` provided on the webpage as you see fit, but at minimum, it must build the executable `IntegerStackTest`, contain a `clean` utility, and contain a `memcheck` target that runs `IntegerStackTest` under `Valgrind`. Also create a file called `README` that serves as a table of contents for the project. It should contain a list of submitted files, together with their roles in the program and any special notes to the grader. Submit the following five files to the assignment lab3.

`README`	written by you
`Makefile`	provided, but you may alter
`IntegerStack.c`	written by you
`IntegerStackTest.c`	provided, do not alter
`IntegerStack.h`	provided, do not alter

I believe that this is a very easy project, since much of it is done for you, and you can emulate existing examples to figure out the rest. As always, start early and ask for help if anything is not completely clear.