CMPS 12M
Introduction to Data Structures Lab
Lab Assignment 4 (for students coming from CMPS 11 or 12A)

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 write a program in C that implements a rudimentary Blockchain 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 to 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.

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 IntegerStack based on a linked list. This example has been abridged to save space. The unabridged version is posted on the webpage.

```
//-----------------------------------------------------------------------------
// IntegerStack.h
// Header file for the IntegerStack ADT
//-----------------------------------------------------------------------------

#ifndef _INTEGER_STACK_H_INCLUDE_
#define _INTEGER_STACK_H_INCLUDE_

// Stack
// Exported reference type
typedef struct StackObj* Stack;

// newStack()
// constructor for the Stack type
Stack newStack(void);

// freeStack()
// destructor for the Stack type
void freeStack(Stack* pS);

//-----------------------------------------------------------------------------
// prototypes of ADT operations deleted to save space, see webpage
//-----------------------------------------------------------------------------

// printStack()
// prints a text representation of S to the file pointed to by out
// pre: none
void printStack(FILE* out, Stack 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 `IntegerStack.h` is the `typedef` command defining `Stack` to be a pointer to `struct StackObj`. The definition of `struct StackObj`, which contains the data fields for the IntegerStack ADT, will be placed in the implementation file. Next are prototypes for the constructor `newStack()` and destructor `freeStack()`, followed by prototypes of ADT operations (skipped in this document, but given on the class webpage.) Finally a prototype is included for a function `printStack()` corresponding roughly to the `toString()` method in java. An abridged version of the implementation file follows.

```
//-----------------------------------------------------------------------------
//   IntegerStack.c
//   Implementation file for IntegerStack ADT
//-----------------------------------------------------------------------------
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<assert.h>
#include"IntegerStack.h"

// private types --------------------------------------------------------------

// NodeObj
typedef struct NodeObj{
   int item;
   struct NodeObj* next;
} NodeObj;

// Node
typedef NodeObj* Node;

// newNode()
// constructor of the Node type
Node newNode(int x) {
   Node N = malloc(sizeof(NodeObj));
   assert(N!=NULL);
   N->item = x;
   N->next = NULL;
   return(N);
}

// freeNode()
// destructor for the Node type
void freeNode(Node* pN){
   if( pN!=NULL && *pN!=NULL ){
      free(*pN);
      *pN = NULL;
   }
}

// StackObj
typedef struct StackObj{
   Node top;
   int numItems;
} StackObj;

// public functions -----------------------------------------------------------

// newStack()
// constructor for the Stack type
Stack newStack(void){
   Stack S = malloc(sizeof(StackObj));
   assert(S!=NULL);
   S->top = NULL;
   S->numItems = 0;
   return S;
}
```

```
// freeStack()
// destructor for the Stack type
void freeStack(Stack* pS){
   if( pS!=NULL && *pS!=NULL ){
      if( !isEmpty(*pS) ) popAll(*pS);
      free(*pS);
      *pS = NULL;
   }
}

//-----------------------------------------------------------------------------
// definitions of ADT operations deleted to save space, see webpage
//-----------------------------------------------------------------------------

// printStack()
// prints a text representation of S to the file pointed to by out
// pre: none
void printStack(FILE* out, Stack S){
   Node N;
   if( S==NULL ){
      fprintf(stderr,
              "Stack Error: calling printStack() on NULL Stack reference\n");
      exit(EXIT_FAILURE);
   }
   for(N=S->top; N!=NULL; N=N->next) fprintf(out, "%d ", N->item);
   fprintf(out, "\n");
}
```

This implementation file defines several private types, namely `NodeObj`, `Node` and `StackObj`. Type `Node` is a pointer to `NodeObj`, which is the basic building block for a linked list. Type `StackObj` encapsulates the data fields for a stack. Recall that type `Stack` was defined in the header file to be a pointer to the structure `struct StackObj`. Type `Stack` is the reference through which the client interacts with ADT operations.

Memory Management

Each of the structure types defined in the above example 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 `freeStack()` are not the reference types `Node` and `Stack`, 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 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 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

A blockchain is one of the technologies underlying crypto-currencies such as BitCoin. One feature of a blockchain is that each block in the chain stores a hash value for the previous block. Consequently, if any block in the chain is modified, the chain will no longer be valid. For this assignment you will create a very rudimentary blockchain. Each block in your chain will contain three fields: data (a string that is the information stored in the block), an integer id indicating the blocks position in the chain with block 0 being the first, and a long integer value that is the hash value for the previous block in the chain. The hash value of the "previous" block in the chain will be zero for the first block in the chain.

The blockchain itself will contain the list of Block values in the chain. For this assignment it will suffice to use a fixed size array of Block and an integer to keep track of the current number of blocks in the chain. Of course in practice the list of Block values could be stored in many forms, including any of the linked list variations covered in class. Because of data hiding, any program that uses your blockchain will be independent of your choice of how to store the list of Block values in the chain.

Your Blockchain ADT will have methods append(), valid(), size(), get(), and printBlockchain() as specified in Blockchain.h.

Your Block ADT must have methods data(), hash(), previousHash(), and printBlock() that prints the block in the format id:data.

The interface for the Blockchain ADT is embodied in the file `Blockchain.h` posted on the webpage. A test client called `BlockchainClient.c` is also included. Submit both of these files unaltered with your project. The webpage also contains a `Makefile` for the Stack ADT. Alter this Makefile to make the executable `BlockchainClient` from the source `BlockchainClient.c`. Compare the output of `BlockchainClient` with the file `BlockOut`. Note that this test client and its output are provided as a convenience to you and should not be deemed to certify your `Blockchain.c` and `Block.c` as error free. For that you will need to construct your own tests in a file called `BlockChainTest.c` which you will also submit. Note that the Makefile provided includes `clean` and `check` utilities which you should leave in place. Do `make clean` to delete old binaries and `make check` to check your `Blockchain.c` for memory leaks.

We will be discussing hash functions later in the quarter. For this exercise the hash value of a block should be computed as a simple sum of the char values in the data string, the block id, and the hash value from the previous block.

Submit lab4.zip that contains the lab4 directory. The lab4 directory should contain:

```
README
Makefile
Block.c
Blockchain.c
BlockchainClient.c
Block.h
Blockchain.h
```

and a pair programming log from logTemplates.txt named log.txt

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