CMPS 12M
Introduction to Data Structures Lab
Winter 2011

Lab Assignment 8

Due Wednesday March 9

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 which recreates the Dictionary ADT in java which was assigned as programming assignment 2 from the Winter 2009 offering of CMPS 12B. Begin by reading the specifications for that assignment at http://www.soe.ucsc.edu/classes/cmps012b/Winter09/pa2.pdf.

Creating New Data Types in C

The `struct` keyword is used to create a new aggregate data type called a *structure*, which is the closest thing C has to java's class construct. Structures contain data fields, but no methods, unlike java classes. A structure 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 structure may be composed of different types of data. The general form of a structure declaration is

```
struct structure_tag{
   /* data field declarations */
};
```

Observe the semicolon after the closing brace. For example

```
struct person{
   int age;
   int height;
   char first[20];
   char last[20];
};
```

This does not however create a complete type name 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;
```

In this example `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 variable of some class type is necessarily a reference (i.e. 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 if you are not familiar with the function `strcpy` in the library `string.h`. The `struct` keyword 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 the keyword `struct` in the declaration. The general form of this combined `typedef` statement is:

```
typedef struct tag{
   /* data field declarations */
} new_type;
```

The `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 as a matter of convention. Also by convention `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;
```

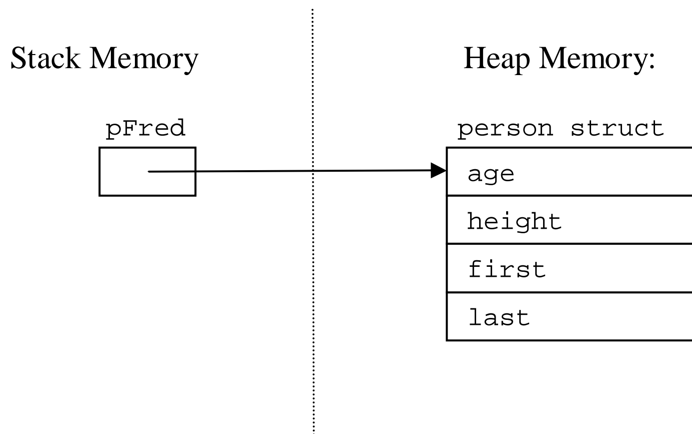
We can now allocate a `person` structure from stack memory by declaring

```
person fred;
```

and assign values to the data fields of `fred` as before. It is important to remember that the `typedef` statement itself does not allocate any 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 which stores the *address of* a block of heap memory which 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 as in `(*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.) 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 which 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 `Node` which has a pointer to `Node` as one of its members.

```
typedef struct Node{
   int item;
   struct Node* next;
} Node;
```

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

```
typedef Node* NodeRef;
```

To declare and initialize a reference to a `Node` we do

```
NodeRef N = malloc(sizeof(Node));
N->item = 5;
N->next = NULL;
```

Two rules to remember when using structures to define ADTs: (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.

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 in C then, 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 structure definition however, is placed in the implementation file, along with the definitions of any

private types, and function definitions, both public and private. The implementation file will `#include` it's own header file, and the ADT operations are defined so as to take and return handles to the ADT structure type.

A client module can then `#include` the header file giving it the ability to declare variables of the handle type, as well as functions which either take or return handle type parameters. However, the client cannot dereference this pointer since the structure it points to is not defined in the header file. The ADT operations take handle arguments, so the client does not need to (and is in fact unable to) directly access the structure which these handles point to. Therefore the client can 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, and likewise for the defined data types belonging to an ADT.

Example

We illustrate the construction of an IntegerStack ADT in C. The header file `IntegerStack.h` and implementation file `IntegerStack.c` can be found on the CMPS 12B webpage from Winter 2009 located at http://www.soe.ucsc.edu/classes/cmps012b/Winter09/Labs/lab5/.

First observe that the header file `IntegerStack.h` contains some preprocessor commands for conditional compilation, namely `#if`, `#endif`, and the preprocessor operator `defined`. If the C compiler encounters multiple definitions of the same type or function, or multiple prototypes for the same function, it is considered a syntax error. Therefore when a program consists of 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, so that the prototypes etc. are seen only once. The general form of such a block is

```
#if !defined (_macro_name_)
#define _macro_name_
statement sequence
#endif
```

If `_macro_name_` is undefined, the lines between `#if` and `#endif` are compiled, otherwise they are skipped. The first operation within 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 the underscore _ character.

The next item in `IntegerStack.h` is the `typedef` statement which defines `StackRef` to be an alias for a pointer to the type `struct Stack`. The definition of `struct Stack` , 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 for ADT operations, then finally a prototype for a function called `printStack`, which corresponds roughly to the `toString()` method in java.

The implementation file `IntegerStack.c` defines several private types, namely `Node`, `NodeRef`, and `Stack`. Type `NodeRef` is a pointer to `Node`, which is the basic building block for the linked list underlying the stack ADT. Type `Stack` encapsulates the data fields for a stack. Recall the type `StackRef` was defined in the header file `IntegerStack.h` to be a pointer to the type `struct Stack`. Type `StackRef` is the so-called handle through which the client interacts with the stack ADT.

Memory Management

Each of the structure types defined in the above example have their own constructor, which allocates heap memory and initializes data fields, as well as their own destructor which balances the calls to `malloc` and `calloc` in the constructor with corresponding calls to `free`. Observe that the arguments to the destructors `freeNode` and `freeStack` are not the handles `NodeRef` and `StackRef` respectively, but pointers to these handle types. The reason for this is that the destructor must alter not just the structure the handle points to, but the handle itself by setting it to `NULL`.

As in java, all ADT operations should check their own preconditions and exit with a useful error massage when one of them 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 program. In C however there is one more item to check. Each ADT operation should check that the handle which 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 programmer.

Naming Conventions

Suppose you are designing an ADT in C called `Blah`. In some other programming classes you may use names like `BlahPtr` or `BlahHndl` instead of `BlahRef`. Of course no name is inherently better, but for the sake of consistency, you are required to adhere to the naming conventions outlined here. In particular the header file should be called `Blah.h` and should contain a `typedef` defining the reference (or handle) type for the ADT

```
typedef struct Blah* BlahRef;
```

along with prototypes for the `Blah` ADT operations. The implementation file should be called `Blah.c` and should contain the statement

```
typedef struct Blah{
   /* data fields for the Blah ADT */
} Blah;
```

together with constructors and destructors for the `Blah` structure, and functions which implement the ADT operations. The general forms for the constructor and destructor are

```
BlahRef newBlah(arg_list){
   BlahRef B = malloc(sizeof(Blah));
   assert( B!= NULL );
   /* initialize the fields of the Blah structure */
   return B;
}
```

and

```
void freeBlah(BlahRef* pB){
   if( pB!=NULL && *pB!=NULL){
      /* free all heap memory associated with *pB */
      free(*pB);
      *pB = NULL;
   }
}
```

Note that the destructor passes its `BlahRef` argument by reference, so it can set the handle to NULL. Given a `BlahRef` variable `B`, a call to `freeBlah` would look like

```
freeBlah(&B);
```

The general form for an ADT operation is

```
return_type some_op(BlahRef B, other_parameters){
   if( B==NULL ){
      fprintf(stderr, "Blah Error: calling some_op on NULL BlahRef\n");
      exit(EXIT_FAILURE);
   }
   /* check preconditions */
   /* do whatever some_op is supposed to do */
}
```

Most ADTs should also contain a `printBlah` function which prints a text representation of a `Blah` to a file stream.

```
void printBlah(BlahRef B, FILE* out){
   if( B==NULL ){
      fprintf(stderr, "Blah Error: calling printBlah on NULL BlahRef\n");
      exit(EXIT_FAILURE);
   }
   /* calls to fprintf(out, text_which_represents_B ) */
}
```

What to Turn In

Use the Stack ADT example above as the starting point for your Dictionary ADT in C. The Dictionary ADT in this assignment is largely the same as the one in pa2 from Winter 2009, with a few minor differences (other than the fact that this one is in C and pa2 was in java.) Again the Dictionary ADT will be based on an underlying linked list data structure. The elements of the Dictionary will be (*key*, *value*) pairs as before, but now *key* and *value* will be ints, not Strings. As in pa2 there are basically two design options to deal with these pairs. Either design your inner `Node` structure to have two `int` fields called `key` and `value` (along with its `next` field), or design another private inner structure called `Pair`, which encapsulates `key` and `value`, then let `Node` have an `item` field of type `PairRef`. The ADT operations are identical to those in Winter 2009 pa2 except of course that certain return types and formal parameters are now `int` rather than `String`. One other difference is that you must write a destructor, where none was necessary in java. You will also write a function called `printDictionary` which replaces the java `toString` method. Its output should be identical to `System.out.println(myDictionary)` in the java version.

The interface for the Dictionary ADT is embodied in the file `Dictionary.h` which is posted on the webpage from Winter 2009, also located at http://www.soe.ucsc.edu/classes/cmps012b/Winter09/Labs/lab5/. Also included is a test client called `DictionaryClient.c`. Turn in both these files with your project but do not alter them in any way. The webpage also contains a `makefile` for the Stack ADT. Alter this makefile so as to make an executable called `DictionaryClient` from the source file `DictionaryClient.c`. Compare the output of `DictionaryClient` with the file `model_out` to check that your Dictionary is working properly. Include utilities called `clean` and `check` in your makefile. The `check` utility should simply call

```
valgrind DictionaryClient
```

to check for memory leaks. In order to receive full credit, your Dictionary operations must produce no memory leaks. In addition to the implementation file `Dictionary.c`, you will write the usual ADT test platform `DictionaryTest.c`. This file should be used by you to test the Dictionary ADT in isolation before

you link it with `DictionaryClient.c`. Its contents are not specified, other than to say that you should include enough calls to ADT operations to convince the grader that you actually used it to test your work.

Submit the following files to lab8:

`README`: Table of contents for this project.
`makefile`: Described above.
`Dictionary.c`: Implementation file constituting your main task in this assignment.
`DictionaryTest.c`: Described above.
`Dictionary.h`: Given at http://www.soe.ucsc.edu/classes/cmps012b/Winter09/Labs/lab5/
`DictionaryClient.c`: Given at http://www.soe.ucsc.edu/classes/cmps012b/Winter09/Labs/lab5/