The Oregon ADT Library

J. Sventek

December 12, 2017

When constructing sophisticated applications, one often needs to use well-known abstract data types (ADTs). If one is using the Java programming language, the Java collection classes provide a rich set of such data type implementations. C++ has the Standard Template Library, along with many other contributed template libraries. When programming in C, one often reinvents just enough of a required ADT to get the job done. In addition to the requirement for general availability of complete implementations of these data types, 21st century programming often demands thread-safe versions, as well.

Through the judicious use of the `void *` type, C provides the ability to create generic abstract data types, similar to the way Java collection classes were done before the incorporation of generics. The Oregon ADT library[1] is a small set of abstract data types that have proved generally useful. Each such ADT is patterned after a corresponding Java collection class and is a complete implementation. Two versions are provided for each ADT, one thread-safe and the other not. Each ADT provides an iterator factory method; each such iterator can then be used to linearly progress through the ADT elements in a standard way.

An ADT instance is referred to using a *const* pointer to a dispatch table portrayed in the following diagram:

`void *self`	private instance data
`ret-type`$_1$ `(*method`$_1$`)(const ADT *adt, m`$_1$`-arguments)`	1st method
`ret-type`$_2$ `(*method`$_2$`)(const ADT *adt, m`$_2$`-arguments)`	2nd method
`ret-type`$_3$ `(*method`$_3$`)(const ADT *adt, m`$_3$`-arguments)`	3rd method
`* * *`	
`ret-type`$_N$ `(*method`$_N$`)(const ADT *adt, m`$_N$`-arguments)`	N^{th} method
`const Iterator* (*itCreate)(const ADT *adt)`	iterator factory method

Each ADT also provides a constructor, or class method, for creating an instance of the ADT. Such a constructor will have a signature of the form:

```
const ADT *ADT_create(constructor-arguments);
```

You might ask "Why the use of *const* in the signatures of the constructor and the iterator factory method? And as the first argument to each of the methods?"

[1] The set of ADT implementations may be obtained from `https://github.com/jsventek/ADTsv2`

The latter answer is simple - since the constructor returns a *const* pointer to the instance, the argument passed to the instance's methods should also be *const* to prevent the user from having to perform endless casts.

The answer to the former question is to provide partial support for information hiding. To be sure, the `void *self` entry cannot be dereferenced by a user, but if the pointer to the ADT instance did not have the *const* attribute, the user could easily change the fields in the structure; the *const* attribute causes C compilers to flag such changes as an error, although determined users can certainly use `memcpy()` and casts to get around this if they wish.

How would a user create and use such an ADT? The following pseudocode will give you an idea.

```
const ADT *adt = ADT_create(constructor-arguments);
const Iterator *it;

adt->method1(adt, m1-arguments);
/* any number of manipulations of the adt */
it = adt->itCreate(adt);
while (it->hasNext(it)) {
    void *v;
    it->next(it, &v);
    /* suitably cast v, doing something with it */
}
it->destroy(it);
adt->destroy(adt, destroy-arguments);
```

The following describes the general structure for a non-thread-safe ADT, a thread-safe ADT, and how these link to and interact with the corresponding iterator classes. This is done through the example of a `Stack` ADT. We need to be able to create `Stack` instances, perform `destroy` and `clear` operations on an instance, as well as `push`, `pop`, and `peek` elements to/from an instance. Since this is a generic `Stack`, the items pushed and popped will be `void *` elements. We also want the stack to resize itself when it is full and to be able to specify an initial size for the stack when it is created. We want to be able to create an iterator over the current contents of the stack, and to provide some additional, generally-useful methods, as well.

1 The non-thread-safe interface

The following definitions appear in the header file, `stack.h`:

```
#include "iterator.h"         /* needed for factory method */

typedef struct stack Stack;   /* forward reference */
const Stack *Stack_create(long capacity);
struct stack {
    void *self;
```

```
    void (*destroy)(const Stack *st, void (*freeFxn)(void *element));
    void (*clear)(const Stack *st, void (*freeFxn)(void *element));
    int (*push)(const Stack *st, void *element);
    int (*pop)(const Stack *st, void **element);
    int (*peek)(const Stack *st, void **element);
    long (*size)(const Stack *st);
    int (*isEmpty)(const Stack *st);
    void **(*toArray)(const Stack *st, long *len);
    const Iterator (*itCreate)(const Stack *st);
};
```

A client of the ADT possesses variables of type `const Stack *`, an approximate equivalent to the object reference used in Java.

1.1 Creation

The method for creating an instance of a stack has the following function signature:

```
const Stack *Stack_create(long capacity);
```

If `capacity` has a value of `0L`, then a stack with a default capacity is created; this default capacity is documented in a comment in the header file.

`Stack_create()` allocates an instance of `Stack` on the heap, initializes its fields, and returns a pointer to that `Stack` instance as the value of the function. If there is a heap allocation failure, the value `NULL` is returned.

1.2 Destroy

C requires that a programmer manage all memory allocated on the heap programmatically. The ADTs in the Oregon library assume that the calling program has retained responsibility for the management of any `void *` elements stored in an ADT unless explicitly transferred to the ADT.

Destruction of a stack is one of those instances when the management responsibility *can* be transferred to the ADT.

Given a `Stack` instance, `st`, the stack can be destroyed with any of the following invocations:

```
st->destroy(st, NULL);
st->destroy(st, free);
st->destroy(st, freeFxn);
```

If the 2nd argument to `destroy()` is not `NULL`, `destroy()` visits each element on the stack and invokes that function on that element; the second invocation above shows the use of the

standard `free()` function, while the third invocation shows the use of a custom function; the presumption is that `freeFxn()` knows how to return any heap storage associated with an element. `destroy()` then returns any heap storage associated with the `Stack` implementation, and finally returns `st` to the heap. After `destroy()` returns, it is illegal to attempt to invoke any of the `Stack` methods on `st`.

1.3 Clear

Rather than destroy a stack, a program's logic may require that it be able to purge all remaining elements from a stack.

Given a `Stack` instance, `st`, the stack can be purged with any of the following invocations:

```
st->clear(st, NULL);
st->clear(st, free);
st->clear(st, freeFxn);
```

The 2nd argument to `clear()` has the same meaning as for `destroy()`. Upon return from `clear()`, the stack represented by `st` is empty i.e. a `pop()` operation will generate an error.

1.4 Push, pop, and peek operations

A common pattern used in C programs is to reserve the function return value to indicate success or failure of the function call. In most cases, a return value of 1 (which is true in C) indicates success, while a return value of 0 (which is false in C) indicates failure.

If a function is to return anything other than status, that information must be returned through a pointer argument. This will be seen in the signatures below.

```
int (*push)(const Stack *st, void *item);
int (*pop)(const Stack *st, void **item);
int (*peek)(const Stack *st, void **item);
```

In each case, if the return value of the function is 1, then the operation has been successful. For a successful `pop()` or `peek()`, the item popped or peeked is returned in `*item`. An unsuccessful `push()` indicates that the stack is full and cannot be extended; an unsuccessful `pop()` or `peek()` indicates that the stack was empty.

Given a `Stack` instance, `st`, here are example invocations of these methods:

```
char *p;      /* assume a stack of strings */

st->push(st, "a string");
```

```
st->peek(st, (void **)&p);  /* p will point to "a string" */
st->pop(st, (void **)&p);   /* ditto, but the stack is now empty */
```

1.5 Other reasonable operations

Although not listed in our initial enumeration of required operations, three other operations are of general utility to programmers, and are often included in complete implementations of collection classes.

```
int (*isEmpty)(const Stack *st);
```

`isEmpty()` returns 1 if the stack is empty, or 0 if it has at least one element.

```
long (*size)(const Stack *st);
```

`size()` returns the number of elements currently on the stack. Obviously, `isEmpty() == 1` if `size() == 0L`.

```
void **(*toArray)(const Stack *st, long *len);
```

An array of `void *` pointers to the elements on the stack is returned, with the number of elements in the array returned in `*len`. The 0^{th} element of the returned array points to the element that would be returned by a `peek()` call to the `Stack`. The array of `void *` pointers is allocated on the heap, so *must* be returned by a call to `free()` when the caller has finished using it.

1.6 Iterating over Collections

A common programming pattern used with collection classes is to use an iterator over the elements of the class. This enables the programmer to visit the elements of the class without knowledge of the structure of the class.[2] This normally requires that the collection class provide a factory method[3] in its interface that can be called to yield an iterator over the collection class. Each of the Oregon ADT classes provide a factory method that yields a generic iterator to enable the use of iterators over instances of that collection class.

The generic `Iterator` interface is as follows:

```
#ifndef _ITERATOR_H_
#define _ITERATOR_H_
```

[2] See description of the Iterator pattern in E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995, ISBN 0-201-63361-2, pp 257-271. See also https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html or any Python programming text.

[3] Gamma, Helm, Johnson, Vlissides, ibid., pp 107-116.

```
/* BSD header removed to conserve space */

/* interface definition for generic iterator
 *
 * patterned roughly after Java 6 Iterator class
 */

typedef struct iterator Iterator; /* forward reference */

/* creates an iterator from the supplied arguments; it is for use by the
 * iterator factory methods in ADTs
 *
 * NB - iterator assumes responsibility for elements[] if create is successful
 * i.e. it_destroy will free the array of pointers
 *
 * returns pointer to iterator if successful, NULL otherwise
 */
const Iterator *Iterator_create(long size, void **elements);

/* now define struct iterator */
struct iterator {
    /* the private data of the iterator */
    void *self;

    /* returns 1/0 if the iterator has a next element */
    int (*hasNext)(const Iterator *self);

    /* returns the next element from the iterator in '*element'
     *
     * returns 1 if successful, 0 if unsuccessful (no next element)
     */
    int (*next)(const Iterator *self, void **element);

    /* destroys the iterator */
    void (*destroy)(const Iterator *self);
};

#endif /* _ITERATOR_H_ */
```

The following pseudocode shows how a program in possession of a collection class creates and uses such an `Iterator` to visit the elements in the collection class:

```
const Class *cl;      /* the instance of the collection class */
const Iterator *it;   /* the iterator instance */

it = cl->itCreate(Class *cl);    /* create the iterator */
while (it->hasNext(it)) {        /* while more elements */
    void *el;

    (void)it->next(it, &el);     /* obtain the next element */
    /*
```

```
     * cast el to the appropriate type and use it
     */
}
it->destroy(it);                /* returns Iterator heap storage */
```

Note that this pseudocode casts the return value of `next()` to `(void)`; this is because the while loop condition has already determined that there is another value.

1.7 Iterating over the Stack

As described above, the Stack ADT must provide a factory method to create an `Iterator` over the `Stack`; this `Iterator` can then be manipulated and destroyed using the methods for a generic iterator.

```
const Iterator *(*itCreate)(const Stack *st);
```

If the `itCreate()` method is successful, a pointer to the iterator is returned; if not, `NULL` is returned. The first `next()` call to the `Iterator` will return the same element as a `peek()` call on the `Stack`; subsequent `next()` calls will traverse down the stack.

1.8 The complete `stack.h` header file

```
#ifndef _STACK_H_
#define _STACK_H_

/* BSD header removed to conserve space */

/*
 * interface definition for generic stack
 *
 * patterned roughly after Java 6 Stack interface
 */

#include "iterator.h" /* needed for factory method */

#define DEFAULT_CAPACITY 50L            /* initial capacity if unspecified */

typedef struct stack Stack; /* forward reference */

/*
 * create a stack with the specified capacity; if capacity == 0L,
 * DEFAULT_CAPACITY is used
 *
 * returns a pointer to the stack, or NULL if there are malloc() errors
 */
const Stack *Stack_create(long capacity);

/*
```

```
 * now define struct stack
 */
struct stack {
    /*
     * the private data of the stack
     */
    void *self;

    /*
     * destroys the stack; for each occupied position, if freeFxn != NULL,
     * it is invoked on the element at that position; the storage associated
     * with the stack is then returned to the heap
     */
    void (*destroy)(const Stack *st, void (*userFunction)(void *element));

    /*
     * clears all elements from the stack; for each occupied position,
     * if freeFxn != NULL, it is invoked on the element at that position;
     * the stack is then re-initialized
     *
     * upon return, the stack is empty
     */
    void (*clear)(const Stack *st, void (*userFunction)(void *element));

    /*
     * pushes 'element' onto the stack; if no more room in the stack, it is
     * dynamically resized
     *
     * returns 1 if successful, 0 if unsuccessful (malloc errors)
     */
    int (*push)(const Stack *st, void *element);

    /*
     * pops the element at the top of the stack into '*element'
     *
     * returns 1 if successful, 0 if stack was empty
     */
    int (*pop)(const Stack *st, void **element);

    /*
     * peeks at the top element of the stack without removing it;
     * returned in '*element'
     *
     * return 1 if successful, 0 if stack was empty
     */
    int (*peek)(const Stack *st, void **element);

    /*
     * returns the number of elements in the stack
     */
    long (*size)(const Stack *st);
```

```
    /*
     * returns true if the stack is empty, false if not
     */
    int (*isEmpty)(const Stack *st);

    /*
     * returns an array containing all of the elements of the stack in
     * proper sequence (from top to bottom element); returns the length of the
     * array in '*len'
     *
     * returns pointer to void * array of elements, or NULL if malloc failure
     *
     * The array of void * pointers is allocated on the heap, so must be
     * returned by a call to free() when the caller has finished using it.
     */
    void **(*toArray)(const Stack *st, long *len);

    /*
     * creates generic iterator to this stack;
     * successive next calls return elements in the sequence top to bottom
     *
     * returns pointer to the Iterator or NULL if malloc failure
     */
    const Iterator *(*itCreate)(const Stack *st);
};

#endif /* _STACK_H_ */
```

2 The non-thread-safe implementation

2.1 The preliminaries

```
/* BSD header removed to conserve space */

/*
 * implementation for generic stack
 */

#include "stack.h"
#include <stdlib.h>

#define MAX_INIT_CAPACITY 1000L

typedef struct st_data { /* private data for stack instance */
    long capacity;
    long delta;
    long next;
    void **theArray;
} StData;
```

As always with an ADT, the implementation must include the corresponding interface definition, thus assuring that we have the correct function signatures in the implementation. We then flesh out the opaque structure type that represents a stack. From this we can see that we maintain the current `capacity` of the stack, the `delta` value by which to increment the `capacity` if we run out of room, and the `next` index into the stack to be used for a `stack_push`. Note that the top of stack is at index `next-1`; if `next` is 0, the stack is empty. Finally, we have an array of `void *` pointers for the elements on the stack.

2.2 `destroy()` and `clear()`

Both of these require that we traverse the current elements on the stack, invoking the user-specified function to free any memory associated with each element. Thus, we define a local function in the implementation to do this common processing, leaving any different processing to the two ADT methods.

Note that in order to return the heap storage for the dispatch table in `destroy()`, we need to first cast it to `void *`; this overrides the `const` nature of the function argument, and prevents warning messages from the compiler.

```
/*
 * local function - traverses stack, applying user-supplied function
 * to each element; if freeFxn is NULL, nothing is done
 */

static void purge(StData *std, void (*freeFxn)(void*)) {
    if (freeFxn != NULL) {
        long i;

        for (i = 0L; i < std->next; i++)
            (*freeFxn)(std->theArray[i]);    /* user frees elem storage */
    }
}

static void st_destroy(const Stack *st, void (*freeFxn)(void *element)) {
    StData *std = (StData *)st->self;

    purge(std, freeFxn);
    free(std->theArray);                /* free array of pointers */
    free(std);                          /* free structure with instance data */
    free((void *)st);                   /* free dispatch table */
}

static void st_clear(const Stack *st, void (*freeFxn)(void *element)) {
    StData *std = (StData *)st->self;

    purge(std, freeFxn);
    std->next = 0L;
}
```

2.3 push()

```
static int st_push(const Stack *st, void *element) {
    StData *std = (StData *)st->self;
    int status = 1;

    if (std->capacity <= std->next) {    /* need to reallocate */
        size_t nbytes = (std->capacity + std->delta) * sizeof(void *);
        void **tmp = (void **)realloc(std->theArray, nbytes);

        if (tmp == NULL)
            status = 0;                  /* allocation failure */
        else {
            std->theArray = tmp;
            std->capacity += std->delta;
        }
    }
    if (status)
        std->theArray[std->next++] = element;
    return status;
}
```

The "complicated" code here is simply to detect when the stack needs to be resized. If the capacity has been exhausted, then `realloc()` is invoked; if it is unsuccessful, then we will return a failure status, since the stack is full. If it is successful, or if the stack had room for the new `element`, appropriate fields are modified, and we then place `element` into the next location in the array and increment the `next` field.

2.4 pop() and peek()

The logic here is nearly identical, except for the side effect of decrementing the `next` field in `pop()`. The code is so simple that there is insufficient scope for placing the common logic in a local function, so the common code is duplicated in the two methods.

```
static int st_pop(const Stack *st, void **element) {
    StData *std = (StData *)st->self;
    int status = 0;

    if (std->next > 0L) {
        *element = std->theArray[--std->next];
        status = 1;
    }
    return status;
}

static int st_peek(const Stack *st, void **element) {
    StData *std = (StData *)st->self;
    int status = 0;
```

```
    if (std->next > 0L) {
        *element = std->theArray[std->next - 1];
        status = 1;
    }
    return status;
}
```

2.5 `size()` and `isEmpty()`

No further discussion is need, as the code is self-evident.

```
static long st_size(const Stack *st) {
    StData *std = (StData *)st->self;

    return std->next;
}

static int st_isEmpty(const Stack *st) {
    StData *std = (StData *)st->self;

    return (std->next == 0L);
}
```

2.6 `toArray()` and `itCreate()`

As can be seen in the Iterator interface presented in section 1.6, the method that creates a generic iterator in the `Iterator` ADT requires an array of `void *` pointers and a length. Thus, `toArray()` and `itCreate()` both require an array of `void *` pointers to the elements. A local function, `arrayDupl()`, is defined, and then used by the two public methods in the ADT.

```
/*
 * local function - duplicates array of void * pointers on the heap
 *
 * returns pointer to duplicate array or NULL if malloc failure
 */
static void **arrayDupl(StData *std) {
    void **tmp = NULL;

    if (std->next > 0L) {
        size_t nbytes = std->next * sizeof(void *);
        tmp = (void **)malloc(nbytes);
        if (tmp != NULL) {
            long i;

            for (i = 0L; i < std->next; i++)
                tmp[i] = std->theArray[i];
        }
```

```
    }
    return tmp;
}

static void **st_toArray(const Stack *st, long *len) {
    StData *std = (StData *)st->self;
    void **tmp = arrayDupl(std);

    if (tmp != NULL)
        *len = std->next;
    return tmp;
}

static const Iterator *st_itCreate(const Stack *st) {
    StData *std = (StData *)st->self;
    const Iterator *it = NULL;
    void **tmp = arrayDupl(std);

    if (tmp != NULL) {
        it = Iterator_create(std->next, tmp);
        if (it == NULL)
            free(tmp);
    }
    return it;
}
```

Note that both `toArray()` and `itCreate()` return NULL if the stack is empty.

2.7 Template dispatch table

Every instance of `Stack` will contain function pointers to `destroy()`, `clear()`, `push()`, `pop()`, `peek()`, `size()`, `isEmpty()`, `toArray()`, and `itCreate()`. In order to minimize the number of statements needed to load up a new `Stack` instance, we create a template dispatch table with those function pointers already filled in. We will use it in `Stack_create()` below.

```
static Stack template = {
    NULL, st_destroy, st_clear, st_push, st_pop, st_peek, st_size,
    st_isEmpty, st_toArray, st_itCreate
};
```

2.8 `Stack_create()`

```
const Stack *Stack_create(long capacity) {
    Stack *st = (Stack *)malloc(sizeof(Stack));

    if (st != NULL) {
        StData *std = (StData *)malloc(sizeof(StData));
```

```
        if (std != NULL) {
            long cap;
            void **array = NULL;

            cap = (capacity <= 0L) ? DEFAULT_CAPACITY : capacity;
            cap = (cap > MAX_INIT_CAPACITY) ? MAX_INIT_CAPACITY : cap;
            array = (void **)malloc(cap * sizeof(void *));
            if (array != NULL) {
                std->capacity = cap;
                std->delta = cap;
                std->next = 0L;
                std->theArray = array;
                *st = template;
                st->self = std;
            } else {
                free(std);
                free(st);
                st = NULL;
            }
        } else {
            free(st);
            st = NULL;
        }
    }
    return st;
}
```

First, we `malloc()` a struct to represent the new stack. If that was successful, we `malloc()` a struct to store the private data members of the stack (to be stored in the `self` member of the `Stack` instance. If that was successful, compute the initial capacity, and allocate an array of `void *` pointers of that `capacity`. If that was successful, fill in the various fields of the `self` struct. Then we copy the `template` struct into the `Stack` instance, and store the `self` struct pointer into the `self` member of the `Stack` instance. If there are `malloc()` errors anywhere in this sequence, free up the previously allocated heap storage, and set the `Stack` pointer to NULL. Finally return the `Stack` pointer.

3 A test program for the non-thread-safe ADT

The following program reads lines of text from a file specified in `argv[1]`, and then exercises most of the methods in the interface.

```
/* BSD header removed to conserve space */

#include "stack.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

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

```
char buf[1024];
char *p;
const Stack *st = NULL;
long i, n;
FILE *fd;
char **array;
const Iterator *it = NULL;
int status = 1;             /* assume error */

if (argc != 2) {
   fprintf(stderr, "usage: ./sttest file\n");
   goto cleanup;
}
if ((st = Stack_create(0L)) == NULL) {
   fprintf(stderr, "Error creating stack of strings\n");
   goto cleanup;
}
if ((fd = fopen(argv[1], "r")) == NULL) {
   fprintf(stderr, "Unable to open %s to read\n", argv[1]);
   goto cleanup;
}
/*
 * test of push()
 */
printf("===== test of push\n");
while (fgets(buf, 1024, fd) != NULL) {
   if ((p = strdup(buf)) == NULL) {
      fprintf(stderr, "Error duplicating string\n");
      goto cleanup;
   }
   if (!st->push(st, p)) {
      fprintf(stderr, "Error pushing string to stack\n");
      goto cleanup;
   }
}
fclose(fd);
n = st->size(st);
/*
 * test of pop()
 */
printf("===== test of pop\n");
for (i = 0; i < n; i++) {
   if (!st->pop(st, (void **)&p)) {
      fprintf(stderr, "Error retrieving %ld'th element\n", i);
      goto cleanup;
   }
   printf("%s", p);
   free(p);
}
printf("===== test of destroy(NULL)\n");
/*
 * test of destroy with NULL freeFxn
```

```
 */
st->destroy(st, NULL);
if ((st = Stack_create(0L)) == NULL) {
   fprintf(stderr, "Error creating stack of strings\n");
   goto cleanup;
}
fd = fopen(argv[1], "r");            /* we know we can open it */
while (fgets(buf, 1024, fd) != NULL) {
   if ((p = strdup(buf)) == NULL) {
      fprintf(stderr, "Error duplicating string\n");
      goto cleanup;
   }
   if (!st->push(st, p)) {
      fprintf(stderr, "Error pushing string to stack\n");
      goto cleanup;
   }
}
fclose(fd);
printf("===== test of toArray\n");
/*
 * test of toArray
 */
if ((array = (char **)st->toArray(st, &n)) == NULL) {
   fprintf(stderr, "Error in invoking st->toArray()\n");
   goto cleanup;
}
for (i = 0; i < n; i++) {
   printf("%s", array[i]);
}
free(array);
printf("===== test of iterator\n");
/*
 * test of iterator
 */
if ((it = st->itCreate(st)) == NULL) {
   fprintf(stderr, "Error in creating iterator\n");
   goto cleanup;
}
while (it->hasNext(it)) {
   char *p;
   (void) it->next(it, (void **)&p);
   printf("%s", p);
}
it->destroy(it);
it = NULL;
printf("===== test of destroy(free)\n");
/*
 * test of destroy with free() as freeFxn
 */
st->destroy(st, free);
st = NULL;
status = 0;
```

```
cleanup:
   if (it != NULL)
      it->destroy(it);
   if (st != NULL)
      st->destroy(st, free);
   return status;
}
```

4 The thread-safe interface

The non-thread-safe ADT gives us all of the functionality that we require. All we have to do now is create appropriate critical sections around calls to the non-thread-safe methods to guarantee the prevention of race conditions.

Rather than create another complete implementation, but this time with appropriate Pthread[4] logic to create the critical sections, it is far easier and less error prone to simply encapsulate an instance of a non-thread-safe ADT inside of a thread-safe instance. Each method will then act like a Java synchronized method[5].

Note that one often wishes to create a transaction across two or more separate calls e.g. insert a new entry into a collection ONLY if it is *not* already there. This has been tackled by enabling a client thread to obtain the lock associated with the thread-safe ADT; once in possession of this lock, the client may invoke as many of the other methods as it wishes before releasing the lock. Thus, each of our "synchronized" methods must also be invokable in the scope of one of these larger transactions.

There is a general problem with using iterators if the structure can change while you are traversing it. When you are given a thread-safe iterator by the factory method in one of these ADTs, you also possess the lock on the class instance, and will retain that lock until you invoke `destroy()` on that iterator.

Here is the interface specification for the thread-safe Stack ADT, `TSStack`. Note that it is a different class from the non-thread-safe Stack ADT, `Stack`.

```
#ifndef _TSSTACK_H_
#define _TSSTACK_H_

/* BSD header removed to conserve space */

/*
 * interface definition for thread-safe generic stack
 *
 * patterned roughly after Java 6 Stack interface
```

[4]See https://en.wikipedia.org/wiki/POSIX_Threads
[5]See https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

```
 */

#include "tsiterator.h" /* needed for factory method */

typedef struct tsstack TSStack; /* forward reference */

/*
 * create a stack with the specified capacity; if capacity == 0L, a
 * default initial capacity (50 elements) is used
 *
 * returns a pointer to the stack, or NULL if there are malloc() errors
 */
const TSStack *TSStack_create(long capacity);

/*
 * now define struct tsstack
 */
struct tsstack {
/*
 * the private data of the stack
 */
    void *self;

/*
 * destroys the stack; for each occupied position, if freeFxn != NULL,
 * it is invoked on the element at that position; the storage associated with
 * the stack is then returned to the heap
 */
    void (*destroy)(const TSStack *st, void (*freeFxn)(void *element));

/*
 * clears all elements from the stack; for each occupied position,
 * if freeFxn != NULL, it is invoked on the element at that position;
 * the stack is then re-initialized
 *
 * upon return, the stack is empty
 */
    void (*clear)(const TSStack *st, void (*freeFxn)(void *element));

/*
 * obtains the lock for exclusive access
 */
    void (*lock)(const TSStack *st);

/*
 * releases the lock
```

```
 */
    void (*unlock)(const TSStack *st);

/*
 * pushes 'element' onto the stack; if no more room in the stack, it is
 * dynamically resized
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
    int (*push)(const TSStack *st, void *element);

/*
 * pops the element at the top of the stack into '*element'
 *
 * returns 1 if successful, 0 if stack was empty
 */
    int (*pop)(const TSStack *st, void **element);

/*
 * peeks at the top element of the stack without removing it;
 * returned in '*element'
 *
 * return 1 if successful, 0 if stack was empty
 */
    int (*peek)(const TSStack *st, void **element);

/*
 * returns the number of elements in the stack
 */
    long (*size)(const TSStack *st);

/*
 * returns true if the stack is empty, false if not
 */
    int (*isEmpty)(const TSStack *st);

/*
 * returns an array containing all of the elements of the stack in
 * proper sequence (from top to bottom element); returns the length of the
 * array in '*len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 *
 * The array of void * pointers is allocated on the heap, so must be returned
 * by a call to free() when the caller has finished using it.
 */
```

```
    void **(*toArray)(const TSStack *st, long *len);

/*
 * creates generic iterator to this stack;
 * successive next calls return elements in the proper sequence (top to bottom)
 *
 * returns pointer to the Iterator or NULL if malloc failure
 */
    const TSIterator *(*itCreate)(const TSStack *st);
};

#endif /* _TSSTACK_H_ */
```

Note that it defines the new class, `TSStack`, and that the same functionality is available on the `TSStack` class as was available on the `Stack` class, albeit an instance of the thread-safe class is a `TSStack *` and the iterator factory method returns a `TSIterator *`. Two new methods are defined:

```
void (*lock)(const TSStack *st);
void (*unlock)(const TSStack *st);
```

These methods enable a thread to acquire the lock associated with the `TSStack` instance in order to make two or more invocations on the instance in the scope of a transaction. Note that one does *not* need to do this in order to make a thread-safe, single invocation on the instance.

5 The thread-safe implementation

5.1 The preliminaries

```
/* BSD header removed to conserve space */

/*
 * implementation for thread-safe generic stack
 */

#include "tsstack.h"
#include "stack.h"
#include <stdlib.h>
#include <pthread.h>

#define LOCK(st) &((st)->lock)

typedef struct tsst_data {
```

```
    const Stack *st;
    pthread_mutex_t lock; /* this is a recursive lock */
} TSStData;
```

No magic here. The data structure behind a `TSStack` instance consists of a `Stack` instance and a PThread mutex. We define the `LOCK` macro to make it easy to specify the PThread mutex associated with the class instance.

5.2 `destroy()` and `clear()`

Each of these invoke the equivalent methods on the encapsulated `Stack` instance while holding the associated mutex lock. `destroy()` then destroys the mutex lock before freeing the `TSStack` structure.

```
static void st_destroy(const TSStack *st, void (*freeFxn)(void *element)) {
    TSStData *std = (TSStData *)st->self;

    pthread_mutex_lock(LOCK(std));
    std->st->destroy(std->st, freeFxn);
    pthread_mutex_unlock(LOCK(std));
    pthread_mutex_destroy(LOCK(std));
    free(std);
    free((void *)st);
}

static void st_clear(const TSStack *st, void (*freeFxn)(void *element)) {
    TSStData *std = (TSStData *)st->self;

    pthread_mutex_lock(LOCK(std));
    std->st->clear(std->st, freeFxn);
    pthread_mutex_unlock(LOCK(std));
}
```

5.3 `lock()` and `unlock`

```
static void st_lock(const TSStack *st) {
    TSStData *std = (TSStData *)st->self;

    pthread_mutex_lock(LOCK(std));
}

static void st_unlock(const TSStack *st) {
    TSStData *std = (TSStData *)st->self;
```

```
    pthread_mutex_unlock(LOCK(std));
}
```

These two methods simply invoke the lock or unlock functions on the associated mutex. Again, these routines are *only* called when the client thread wishes to create a transaction across two or more of the other methods in the interface.

5.4 `push()`, `pop()` and `peek()`

```
static int st_push(const TSStack *st, void *element) {
    TSStData *std = (TSStData *)st->self;
    int result;

    pthread_mutex_lock(LOCK(std));
    result = std->st->push(std->st, element);
    pthread_mutex_unlock(LOCK(std));
    return result;
}

static int st_pop(const TSStack *st, void **element) {
    TSStData *std = (TSStData *)st->self;
    int result;

    pthread_mutex_lock(LOCK(std));
    result = std->st->pop(std->st, element);
    pthread_mutex_unlock(LOCK(std));
    return result;
}

static int st_peek(const TSStack *st, void **element) {
    TSStData *std = (TSStData *)st->self;
    int result;

    pthread_mutex_lock(LOCK(std));
    result = std->st->peek(std->st, element);
    pthread_mutex_unlock(LOCK(std));
    return result;
}
```

Each of these invoke the equivalent methods on the encapsulated `Stack` instance while holding the associated mutex lock.

5.5 `isEmpty()`, `size()` and `toArray()`

Each of these invoke the equivalent methods on the encapsulated `Stack` instance while holding the associated mutex lock.

```
static long st_size(const TSStack *st) {
    TSStData *std = (TSStData *)st->self;
    long result;

    pthread_mutex_lock(LOCK(std));
    result = std->st->size(std->st);
    pthread_mutex_unlock(LOCK(std));
    return result;
}

static int st_isEmpty(const TSStack *st) {
    TSStData *std = (TSStData *)st->self;
    int result;

    pthread_mutex_lock(LOCK(std));
    result = std->st->isEmpty(std->st);
    pthread_mutex_unlock(LOCK(std));
    return result;
}

static void **st_toArray(const TSStack *st, long *len) {
    TSStData *std = (TSStData *)st->self;
    void **result;

    pthread_mutex_lock(LOCK(std));
    result = std->st->toArray(std->st, len);
    pthread_mutex_unlock(LOCK(std));
    return result;
}
```

5.6 `itCreate()`

This is slightly more complicated, since we want the caller to possess the lock on the `TSStack` instance until the caller invokes `destroy()` on the iterator.

```
static const TSIterator *st_itCreate(const TSStack *st) {
    TSStData *std = (TSStData *)st->self;
    const TSIterator *it = NULL;
    void **tmp;
    long len;
```

```
    pthread_mutex_lock(LOCK(std));
    tmp = std->st->toArray(std->st, &len);
    if (tmp != NULL) {
        it = TSIterator_create(LOCK(std), len, tmp);
        if (it == NULL)
            free(tmp);
    }
    if (it == NULL)
        pthread_mutex_unlock(LOCK(std));
    return it;
}
```

The code first locks the mutex; it then obtains a `toArray` from the encapsulated `Stack` instance. If that is successful, it invokes `TSIterator_create()` to create the iterator; note that the signature for `TSIterator_create()` requires the lock as an additional argument. If a `TSIterator` was successfully returned, it is returned to the caller; if not, then allocated heap memory is returned *and* the mutex is unlocked.

5.7 Template dispatch table

As for the `Stack` implementation, we create a template dispatch table with all function pointers already filled in. We will use it in `TSStack_create()` below.

```
static TSStack template = {
    NULL, st_destroy, st_clear, st_lock, st_unlock, st_push, st_pop,
    st_peek, st_size, st_isEmpty, st_toArray, st_itCreate
};
```

5.8 `TSStack_create()`

In order to enable the transaction capability over two or more method calls, we have used *RECURSIVE* Pthread mutexes. This is why the code in `TSStack_create` is dominated by Pthread calls. Other than that, the implementation is quite straight-forward.

```
const TSStack *TSStack_create(long capacity) {
    TSStack *st = (TSStack *)malloc(sizeof(TSStack));

    if (st != NULL) {
        TSStData *std = (TSStData *)malloc(sizeof(TSStData));

        if (std != NULL) {
            std->st = Stack_create(capacity);
```

```
            if (std->st != NULL) {
                pthread_mutexattr_t ma;
                pthread_mutexattr_init(&ma);
                pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_RECURSIVE);
                pthread_mutex_init(LOCK(std), &ma);
                pthread_mutexattr_destroy(&ma);
                *st = template;
                st->self = std;
            } else {
                free(std);
                free(st);
                st = NULL;
            }
        } else {
            free(st);
            st = NULL;
        }
    }
    return st;
}
```

First, we create a `TSStack` instance. If that is successful, we create a `TSStData` structure to hold the instance data for the `TSStack`. If that is successful, we then initialize a `pthread_mutexattr_t` structure, set the mutex type to `RECURSIVE`, initialize the mutex lock, and destroy the attribute structure. We return the `TSStack` pointer if all goes well.

5.9 `tssttest`

The test program for the thread-safe stack is identical to that for the non-thread-safe stack, with three changes:

- `#include "tsstack.h"` instead of `"stack.h"`;
- change the three occurrences of `Stack` to `TSStack`; and
- change the one occurrence of `Iterator` to `TSIterator`.

When building `tssttest`, we need to link the main program to `tsstack.o` and `tsiterator.o`; additionally, we need to link to `stack.o` and `iterator.o`, since we are encapsulating the non-thread-safe stack.