Recitation 10: From C1 to Shining C

Thursday March 21st

File inclusions

The first thing we do in a C file is to include the header files of any library whose functions we will use.[1] For example, the implementation of BST dictionaries we will be developing may include the following header files:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "myheaders/bst.h"
```

Here, `<stdio.h>` is the header file of the system library that supplies `printf` among other functions, `<stdlib.h>` gives us access to `malloc` and `free`, while `<stdbool.h>` lets us use the type `bool` and the values `true` and `false` just like in C0. The last one is the header file of the BST dictionary itself. Differently from the others, it is user-defined as the local file `bst.h` store in the directory `myheaders/`. System header files are enclosed in angled brackets while user-defined header files are enclosed in double quotes.

Contracts

Contracts are a core part of the C0 and C1 languages. Unfortunately, we don't have the power to build contracts into C. Don't lose hope, though! We provide you with a supplemental contracts library so you can continue to program with contracts. To use it, you'll need to put `#include "path/to/contracts.h"` at the top of your C file. Also, you'll need to pass the `-DDEBUG` flag instead of `-d` when you compile with gcc if you want contracts to be checked.

The file `contracts.h` provides you with `REQUIRES`, `ENSURES`, and `ASSERT`. You can treat these as C functions[2] that replace our contracts in the following way:

(a) `//@requires` can be replaced by `REQUIRES` at the very beginning of the function.

(b) `//@ensures` can be replaced by `ENSURES` before every return statement and/or at the end of the function.

(c) `//@loop_invariant` can be replaced by `ASSERT` before the loop runs and at the end of each loop iteration.

(d) `//@assert` can be replaced by `ASSERT`.

[1] See the appendix for what's in a header file.

[2] To be precise, these are actually C macros that preprocessing reduces to native C `assert` statements during compilation. Shhh, don't tell.

Checkpoint 0

Rewrite the following C0 function into C. Only the contracts will need to be changed.

```
1  int length(list* start, list* end)
2  //@requires is_segment(start, end);
3  //@ensures \result >= 0;
4  {
5    size_t length = 0;
6    while (start != end)
7    //@loop_invariant is_segment(start, end);
8    {
9      length++;
10     start = start->next;
11   }
12   return length;
13 }
```

Memory allocation

In C0 and C1, we had the functions `alloc`, which allocated enough memory for a singleton of some type, and `alloc_array`, which allocated enough memory for an array of some type. In C, there is only[3] `malloc`, which takes one argument — the amount of memory you want.

For example, an equivalent of `alloc(struct list_node)` in C is `malloc(sizeof(struct list_node))`, and the equivalent of `alloc_array(int, 3)` in C is `malloc(3*sizeof(int))`.

However, `malloc` can return `NULL` when out of memory. Usually you would have to check the return value to see if it is `NULL`, but we provide you with a replacement for `malloc` that does this check for you: `xmalloc`. To use it, put `#include "path/to/xalloc.h"` at the beginning of your C file.

Freeing memory

After you are done using any memory referenced by a pointer returned by `malloc` or `calloc`, you must free it or your program will have "memory leaks" (in C0 and C1, something called a garbage collector does this automatically). You can free such memory by passing a pointer to it to `free`. Once you free it, it is undefined to access that memory. Also, don't free memory that was previously freed. That also results in undefined behavior.

When we design libraries for data structures like stacks and BST's, it's important to specify whether the client or library is responsible for freeing each piece of memory that is `malloc`ed. Usually, whoever allocates the memory "owns" it and is responsible for freeing it, but in data structures like BST's, we may want to transfer ownership of the memory to the data structure. Thus, we ask the client to supply a "freeing" function in `bst_new` for BST elements, so that when the client calls `bst_free`, we can call their "freeing" function on every element in the BST. If the function pointer the client gives us is `NULL`, then we don't free the BST elements.

Checkpoint 1

Rewrite the functions `bst_new` (which should now take in an additional pointer to a "freeing function") into C. You will also need to update the definition of `struct bst_header`. Then rewrite `tree_insert` and `bst_insert`.

[3] Note that `malloc` does not initialize the memory it allocates to 0. If you require this, there is a variant of `malloc` called `calloc` (and a corresponding `xcalloc`) that does this.

```
typedef int compare_fn(void* e1, void* e2);

typedef struct tree_node tree;
struct tree_node {
  void* data;
  tree* left;
  tree* right;
};

typedef struct bst_header bst;
struct bst_header {
  tree* root;
  compare_fn* compare;
};

bst* bst_new(compare_fn* compare)
//@requires compare != NULL;
//@ensures is_bst(\result);
{
  bst* B = alloc(struct bst_header);
  B->root = NULL;
  B->compare = compare;
  return B;
}

tree* tree_insert(tree* T, void* e, compare_fn* compare)
//@requires e != NULL && compare != NULL && is_tree(T, compare);
//@ensures is_tree(\result, compare);
{
  if (T == NULL) {     /* create new node and return it */
    T = alloc(struct tree_node);
    T->data = e;
    T->left = NULL; T->right = NULL;
    return T;
  }
  int r = (*compare)(e, T->data);
  if (r == 0) {
    T->data = e;       /* modify in place */
  } else if (r < 0) {
    T->left = tree_insert(T->left, e, compare);
  } else {
    //@assert r > 0;
    T->right = tree_insert(T->right, e, compare);
  }
  return T;
}

void bst_insert(bst* B, void* e)
//@requires is_bst(B);
```

```
//@requires e != NULL;
//@ensures is_bst(B);
{
  B->root = tree_insert(B->root, e, B->compare);
  return;
}
```

Checkpoint 2

Consider the client code below.

```
int int_compare(void *x, void *y) {
  REQUIRES(x != NULL && y != NULL);
  if (*(int*)x < *(int*)y) return -1;
  if (*(int*)x == *(int*)y) return 0;
  return 1;
}

int main() {
  bst *B = bst_new(&int_compare, &free);
  int *x = xmalloc(sizeof(int));
  *x = 5;
  int *y = xmalloc(sizeof(int));
  *y = 7;
  int *z = xmalloc(sizeof(int));
  *z = 9;
  bst_insert(B, (void*)x);
  bst_insert(B, (void*)y);
  bst_insert(B, (void*)z);
  bst_free(B);
  return 0;
}
```

Draw a memory diagram that represents the state immediately before the `main` function returns.

What would happen if `bst_free(B)` just called `free(B)`?

Checkpoint 3

How can we actually free all of our allocated memory? Fill in `bst_free` below.

```
80 void tree_free(tree *T, free_fn *elemfree) {
81   if (T != NULL) {
82     if (elemfree != NULL) {
83       ____________________________________________________________________________;
84     }
85     ____________________________________________________________________________;
86     ____________________________________________________________________________;
87     free(T);
88   }
89   return;
90 }
91
92 void bst_free(bst *B) {
93   REQUIRES(____________________________________________________________________);
94   ____________________________________________________________________________;
95   free(B);
96   return;
97 }
```

Checkpoint 4

Great! When we run Valgrind for the test case in the earlier checkpoint, it seems like we have no memory leaks now that we've written a `bst_free`. However, we need to be mindful that this is just one test case, and writing more can expose other issues. For example, consider:

```
int main() {
  bst *B = bst_new(&int_compare, &free);
  int *x = xmalloc(sizeof(int));
  *x = 5;
  int *y = xmalloc(sizeof(int));
  *y = 7;
  int *z = xmalloc(sizeof(int));
  *z = 7;                                          // MODIFIED
  bst_insert(B, (void*)x);
  bst_insert(B, (void*)y);
  bst_insert(B, (void*)z);
  bst_free(B);
  return 0;
}
```

Valgrind will report a memory leak on this code. What causes it? Update the C code you wrote earlier to fix this issue.

Throughout this recitation, your TAs may show you some common Valgrind errors and how to fix them. Feel free to use the space below to take notes!

Appendix: Header files

In C0 and C1, we usually wrote the interface and implementation of our data structures in the same file. Unfortunately, this means that if we want to show clients our data structure's interface, we end up showing them our implementation too! C solves this by separating the interface and implementation respectively into a header file and a source file. A header file for BST's is shown below:

```
#ifndef BST_H
#define BST_H

typedef struct bst_header bst;
typedef int compare_fn(void *e1, void *e2);
typedef void free_fn(void *e);

bst *bst_new(compare_fn *elem_compare, free_fn *elem_free);
void bst_insert(bst *B, void *e);              /* e cannot be NULL! */
void bst_free(bst *B);

#endif
```

Header files usually only contain type and function declarations for the client and no actual code.

Aside: the **#ifndef**, **#define**, and **#endif** lines are known as header guards, and they prevent the header file from being included too many times in a file.