15-122: Principles of Imperative Computation

Recitation 18 Josh Zimmerman, Arjun Hans

Macros

C has a concept of *macros*. Macros are simple find and replace expressions that get run at compile time. These can be useful for constants, things that edit the syntax of your code, or statements that you may or may not want to run depending on compilation flags.

For example, if you have a default array size that you use in your program (say, 256), you could write the line

```
#define ARRAY_SIZE 256
```

If you did that, everywhere `ARRAY_SIZE` appeared in your code, the compiler would replace it with `256`.

We can also define macros that take arguments. These simply do a textual find-and-replace when they are "called".

This can lead to many problems.

For instance, consider this definition of a macro:

```
#define MAX(a,b) a > b ? a : b
```

What happens if we call `5 * MAX(4 + 3, 10)`?

Well, we just do simple replacement, so this is expanded to `5 * 4 + 3 > 10 ? 4 + 3 : 10`. Due to order of operations, this is equivalent to `(5 * 4 + 3) > 10 ? (4 + 3) : 10`, or in other words, if `(5 * 4 + 3) > 10` then 7, else 10. This does not compute the max correctly.

We need to add parentheses to make this do what we want.

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

This gets us: `5 * ((4 + 3) > (10) ? (4 + 3) : 10)`, which works the way we want and evaluates to `5 * 10`.

It's also very important to never have a statement with side effects in the arguments to a macro. For example, `MAX(x++, y++);` will expand to:

`((x++) > (y++) ? (x++) : (y++))`. This is bad, because one of `x++` and `y++` will be evaluated twice.

There's something called an `inline` function that provides a way to avoid these problems and to get many of the performance benefits that macros provide. If you're interested in more details, you can ask a TA, CA, or instructor about it or look up more information online, but we won't cover it yet.

ASSERT, REQUIRES, and ENSURES

In C, we have defined ASSERT, REQUIRES, and ENSURES macros that approximate the functionality of the similar statements in C0 .

If you compile with the -DDEBUG flag, these will be compiled into assert statements (and so be checked at runtime, just like normal contracts). Otherwise, they have no effect.

```
% gcc -Wall -Wextra -Werror -std=c99 -pedantic -g -DDEBUG foo.c
```

One drawback to C is that we have no way of implementing \length . We also don't have \result . We can work around the lack of \result by just having an ENSURES statement immediately before any return, though.

This is the lookup function for binary search trees. Cut-and-pasted into C, the only problem is that the contracts won't be checked anymore (they're just comments now). Add the C contract macros so that the C0 contracts are checked with the -DDEBUG flag:

```
elem tree_lookup(tree* T, key k)
//@requires is_ordtree(T);
//@ensures \result == NULL || key_compare(elem_key(\result), k) == 0;
{
  if (T == NULL) return NULL;
  int r = key_compare(k, elem_key(T->data));
  if (r == 0) {
    return T->data;
  } else if (r < 0) {
    return tree_lookup(T->left, k);
  } else {
    //@assert r > 0;
    return tree_lookup(T->right, k);
  }
}
```

We also don't have //@loop_invariant either. This is not quite as easy to replace, but still doable.

```
while (condition)
//@loop_invariant foo;
{
   // do something
}
```

can be rewritten in C by doing the following:

```
ASSERT(foo);
while (condition) {
   // do something
   ASSERT(foo);
}
```

If you `malloc`, you MUST `free`, and you MUST `free` ONLY what you `malloc`'d

In C0 , memory is automatically managed—once you're done with it it automatically goes away.

In C, this is NOT the case. C gives you much more power than C0 does and lets you do much more than C0 does. However, with this power comes great responsibility. We have several functions we can use to help us manage memory.

The function `malloc` takes in a number of bytes and returns either `NULL` or a pointer to some area of memory with enough space for the amount of memory you asked for. `malloc` does not guarantee that the memory it gives you a pointer to is initialized to anything, so you CANNOT use the values of memory that you without first initializing them. This point is very important, so it's worth reemphasizing. **THE DATA IN THE AREA THAT MALLOC RETURNS A POINTER TO IS GARBAGE AND CANNOT BE USED WITHOUT BEING INITIALIZED!**

The function `calloc` is like `malloc`, except for two details. First, it takes two arguments. The first argument is the number of elements you want to allocate and the second is the size (in bytes) of each element. `calloc` initializes all of the memory it returns to 0, but is slower than `malloc` because it needs to take the time to do this.

The function `free` takes in a pointer and reclaims the memory that it points to. The argument it takes must be either `NULL` (in which case it does nothing) or a pointer to some memory returned by `malloc` or `calloc` that has not been already `freed`. (Freeing things multiple times is a really bad bug, as is never freeing something that you use.)

The contract that you must follow when using memory in C is that if you allocate some memory with `malloc` or `calloc`, you must call `free` to tell the system that you are done using the memory. If you do not do this, your program has a memory leak, and if it runs for long enough, it will eventually use up all of the memory on your system. You must only call `free` on memory you got from `malloc` or `calloc`, and must only free something once.

The tool `valgrind` is very useful here: if you run it on your code it can tell you if you have memory leaks. (Note that if `valgrind` reports "no leaks are possible", this only means that there were no leaks in the specific execution that just happened. It could be that under different conditions, your program is leaky.)

It's an error to free the same memory multiple times. If you do so, you get undefined behavior, which, to reiterate, means that anything could happen.

Make sure you understand how you'd free the memory allocated to a binary search tree. This implementation should *NOT* free any of the elements of the binary search tree. In other words, we don't treat the binary search tree of owning its constituent elements. Why is it a bad idea to call `free(T->data)` on every node? What other choices could we have made here?

```
void tree_free(tree *T) {
  REQUIRES(is_ordtree(T));
  if(T != NULL) {

  }
  return;
}

void bst_free(bst B) {
  REQUIRES(is_bst(B));

}
```

`xmalloc` and `xcalloc`

One really unfortuante aspect of `malloc` and `calloc` is that they are allowed to return `NULL` instead of a valid pointer. Rather than littering your code with `NULL` checks, all the C assignments in this class will come with a library `xalloc.h` that has versions of `malloc` and `calloc` that always safely abort rather than calling `NULL`.

```c
#include <stdio.h>

#ifndef _XALLOC_H_
#define _XALLOC_H_

/* xcalloc(nobj, size) returns a non-NULL pointer to array of nobj
 * objects, each of size size and * exits if the allocation fails.
 * Like calloc, the * array is initialized with zeroes */
void *xcalloc(size_t nobj, size_t size);

/* xmalloc(size) returns a non-NULL pointer to an object of size size
 * and exits if the allocation fails. Like malloc, no initialization
 * is guaranteed. */
void *xmalloc(size_t size);

#endif
```

The type `size_t` is the size of memory sizes and array accesses, and it is an *unsigned* type (it won't ever be negative). We'll talk more about `void*` later.

String manipulation and `strcat`

In C, strings are represented as *null-terminated* arrays. (They're also sort of equivalent to pointers, though. We often write the type of string as `char *s`, and sometimes but more rarely write it as `char s[]`.)

What this means is that to represent the string "hi", we'd have an array of 3 `char`s: `[‘h’, ‘i’, ‘\0’]`. Note that `char`s are internally stored (for the version of gcc we're using) in 8-bit two's complement, so this is the same as `[104, 105, 0]`. It's *very* important to remember that strings have a null character at the end. Forgetting this is a very, *very* common source of bugs, since behavior is undefined in C if you go outside the bounds of an array.

In the best case, your program will crash when this happens. In the worst, it will appear to keep working, but will overwrite variables and corrupt internal program state, leading to problems that are very difficult to debug.

In fact, many modern security vulnerabilities come from failing to properly check length of arrays or from using arrays that are too small. You'll see this in more detail in 15-213, where you'll write a *buffer overflow* attack that exploits such a vulnerability in a poorly written program.

Here's a function similar to C0 's `string_join`. It takes two strings, appends the second to the end of the first, and returns the result:

```c
char *strcat(char *dest, char *src) {
  size_t offset = strlen(dest);
  size_t i; // Declare here so we can use it outside of the loop
  for (i = 0; src[i] != 0; i++) {
    dest[i + offset] = src[i];
  }
  dest[i + offset] = 0; // null-terminate
  return dest;
}
```

I'm going to show a few examples of this now: one which works and has defined behavior, one which keeps the program running and appears to work, and one which crashes immediately.