C track: assignment 5

Goals

In this assignment you will write a program to simulate a simple one-dimensional cellular automaton, which are related to (but are not the same as) two-dimensional cellular automata like Conway's game of Life. We will use this example to introduce pointers and pointer arithmetic, as well as dynamic memory allocation. Pointers are the most confusing topic in C, so we will spend a lot of time discussing them before we get to the program you have to write.

Language concepts covered this week

- pointers
- dynamic memory allocation
- the `sizeof` operator
- type casts

Other concepts covered this week

Memory leaks.

Pointers

To quote from K&R:

> A pointer is a variable that contains the address of a variable.

What's this about addresses, though? In C, as in many computer languages, variables refer to particular locations in memory. This location is known as the *address* of the variable. Numerically, the address is the number of the memory location in bytes. It is normally represented using a hexadecimal number (*e.g.* `0x00124AF4`; recall that hexadecimal numbers in C are entered with a leading "0x"). In fact, you can directly get at the address in C by using the `&` operator *e.g.* `&foo` is the address of the variable `foo`. (**Note**: the `&` operator has another meaning ("bitwise AND") that the compiler determines from the context.) This address can be stored in a pointer. However, pointers have specific types (*e.g.* "pointer to integer"), so, in general, you can only store the address of an integer in a pointer to an integer. Pointers are declared as follows:

```
int *p;  /* declares 'p' to be a pointer to an integer */
```

The declaration uses the `*` operator. Confusingly, this operator also stands for multiplication. However, it's usually easy to see which meaning is which depending on the context.

What can you do with a pointer? The most basic thing is that you can store a memory address there:

```
int *p;
int q = 79;

p = &q;  /* now 'p' stores the address of the integer variable 'q'. */
```

Once you have the address of a variable stored in a pointer, you can retrieve the contents of the variable by *dereferencing* the pointer:

```
int *p;
int q = 79;
int r = 21;
int sum;

p = &q;
sum = *p + r;  /* = 100 */
```

The expression `*p` means to fetch the value at the address stored in the pointer `p`. This unary use of the `*` operator is called the pointer dereferencing operator. If there is any chance that it might be confused with the multiplication operator, you should surround it with parentheses *e.g.*

```
sum = (*p) + r;
```

These two operations, storing addresses into a pointer and dereferencing the pointer, are the two most basic pointer operations. You can also do *pointer arithmetic*, which we describe below.

Dynamic memory allocation

Up to now, when we wanted an object we just declared it inside a function e.g.

```
int foo(void)
{
    int bar[10];  /* create an array with ten integers */
    double d;     /* create a double */

    /* etc. */

}
```

Objects created this way are called "automatic" variables. What this means is that the compiler automatically allocates the space for these objects when the function begins

execution (specifically, on the call stack, which we will discuss in the lecture) and frees the memory when the function completes its execution. However, it is also possible, and often necessary, to "manually" create new objects on the fly (which is called "dynamic memory allocation"). This happens, for instance, when you don't know in advance how many data objects you will need. For dynamic memory allocation we use the system function `malloc`, which is short for "memory allocator". `malloc`, which is defined in `<stdlib.h>`, has the following declaration:

```
void *malloc(size_t size);
```

This means that you pass it a positive integer representing a size (`size_t` is an alias (typedef) for an integer type) and it returns a pointer which points to the newly-allocated block of memory of the appropriate size (in bytes). But what's this "`void *`" stuff? A pointer to nothing? That sounds very Zen-like... Actually, `void *` is C's way of saying "a pointer to some block of memory that can hold any type of data whatsoever". So `malloc` will work regardless of what type of data you need. Only one question remains: how do you know how many bytes of data you need? You could figure it out by determining the exact size of all the primitive data types in bytes (which is not required to be the same for different machines), or you could do the right thing, which is to use the `sizeof` operator. `sizeof(int)` returns the size of an `int` in bytes, `sizeof(double)` returns the size of a `double` in bytes, and so on. So `malloc` is normally used like this:

```
int *numbers;

/* Allocate memory for ten ints. */
numbers = (int *)malloc(10 * sizeof(int));
```

The `sizeof` operator looks like a function call, but it's actually a language-defined operator (it can't be a real function, since its argument is a type name, not a C expression). `sizeof` may not return the same size for a given data type on all machines (*e.g.* ints are 4 bytes on some machines and 8 bytes on others), so using `sizeof` helps to keep your code portable. `sizeof` can also work on pointers or structs (as we'll see in the next lab).

Another language feature we see here is a *type cast*, which until now we've only used to convert between primitive types (e.g. `int` to `float`). The `(int *)` in the line before the `malloc` is a type cast. Recall that `malloc` returns an object of type "`void *`", which is a pointer to any type. The type cast says to the compiler, "treat this data object as if it were a pointer to an int". It is important to realize that a pointer to an int can be treated as an array as long as the memory for the array has been allocated. In fact, the array indexing operator `[]` is actually just syntactic sugar for pointer arithmetic (see below).

Now you can use `numbers` as if it were an array of integers, because it *is* an array of integers. You can set and access elements of `numbers` just as with arrays:

```
numbers[0] = 1;
numbers[1] = 1;
numbers[2] = numbers[0] + numbers[1];
/* etc. */
```

However, the memory that is allocated by `malloc` is not taken from the same location as the memory that is used for automatic variables. Instead, it's taken from a region called the "heap", which is just a region of memory set aside for this purpose. We refer to objects allocated by `malloc` as "heap-allocated" as opposed to automatic variables, which are "stack-allocated", because they live on a data structure called the "stack".

The most significant difference between heap-allocated and stack-allocated objects is this: once you reach the end of a function, stack-allocated objects have their memory automatically reclaimed, but heap-allocated objects do not. Instead, you have to manually reclaim the memory for heap-allocated objects by calling the `free` function, which is also defined in `<stdlib.h>`:

```
free(numbers);
```

If you don't do this, you have a **memory leak**, which is the bane of C programmers. Memory leaks are one of the worst kind of bugs, because they very often do not cause any problems at all. However, if you are allocating a lot of memory which you never free, sooner or later you will run out of memory and your program (and maybe your computer) will crash for no apparent reason at what appears to be a random location. Tracking down memory leaks is sometimes so difficult that there are a number of commercial tools (such as Purify) to help programmers do this, as well as some freely-available tools such as Valgrind.

For our purposes, the thing to remember is this: if you `malloc` a block of memory, sooner or later you are going to have to `free` it. However, you don't necessarily have to `free` it in the same function that you `malloc`'ed it. If that was the case, you might as well not use `malloc` at all, since it would be no better than a stack-allocated automatic variable. Instead, what typically happens is that one function will `malloc` the memory and another will `free` it. We will see this in this lab.

Notes on dynamic memory allocation

Note 1: There is a function related to `malloc` called `calloc` which is in many ways superior. Its function prototype is:

```
void *calloc(size_t nmemb, size_t size);
```

It is like `malloc` except for two features:

1. `calloc` takes two arguments: the number of things you are allocating (`nmemb`) and the size of each thing in bytes (`size`). In contrast, with `malloc` you have to multiply these two numbers together to give the single argument.

2. After allocating memory, `calloc` sets all the allocated memory to zero values. This is often useful and can save you from writing a loop to do the same thing manually.

We recommend that you use `calloc` instead of `malloc`.

Note 2: If `malloc` or `calloc` is unable to allocate memory, it will return `NULL` to indicate that it failed. It's important to always check the return values of `malloc` and `calloc` to make sure that the memory allocation completed correctly, or to print a reasonable error message and exit if it doesn't. Many C programmers don't do this, unfortunately. **You must *always* check the return value of `malloc` and `calloc` in your programs every time you use them!** Failure to do so in your labs will result in a redo.

For instance, a good call to `calloc` might look something like this:

```
int *numbers;
numbers = (int *) calloc(10, sizeof(int));

/* Check that the calloc call succeeded. */
if (numbers == NULL)
{
    fprintf(stderr, "Error! Memory allocation failed!\n");
    exit(1);  /* abort the program */
}
/* Do what you want with the numbers array... */
```

Note 3: To make it easier for you to avoid memory leaks in your program, we're supplying you with a simple memory leak checker (see below for the files) to help you find these leaks. It is a "drop-in" replacement for `malloc`, `calloc`, and `free`. To use it, you have to put this at the top of your file:

```
#include <stdlib.h>
#include "memcheck.h"
```

Make sure that those two lines are in exactly that order. When you compile your program, you will also have to compile the file "`memcheck.c`" and link it in to the final program. After that, to check for memory leaks, just put the following line in the `main` function right before you exit the program:

```
print_memory_leaks();
```

This will print an error message for every place you allocated memory but didn't free it. It will also tell you where you allocated the unfreed memory. This will be very annoying to use at first, but it will also be very eye-opening. You should fix your code until no errors are reported.

Note 4: The use of dynamic memory allocation with explicit `free`ing of variables is such a fertile source of hard-to-find and/or hard-to-fix bugs in C programs that there exist things called *garbage collectors* (GCs) whose job it is to do this automatically for the programmer. Garbage collectors involve some overhead in terms of execution speed and/or total memory use, but they make the programmer's job much easier. For this reason, most programming languages (such as Java, Python, and Scheme) provide garbage collection as a standard service to the user. It turns out that having a garbage collector in a language like C (which has pointers) is harder than in a language like Java (which doesn't have pointers), and this, along with the performance cost, is the reason

most programmers don't use a GC in their C programs. However, it's possible, and using garbage collectors in C programs is covered in the advanced C track of CS 11. We won't use a GC for our programs in this track.

Pointer arithmetic

It is possible, and often necessary, to add and subtract integers from pointers. This is known as *pointer arithmetic*. It is a bit of a misnomer, since the permitted operations are quite limited. Here we will only discuss adding an integer to a pointer to get another pointer, which is by far the most common example of pointer arithmetic.

When you add 1 to a pointer, you generate a new pointer which is `sizeof(N)` larger than the original pointer, where `N` is the data type that the pointer points to. So if you have:

```
int *numbers = (int *) malloc(10 * sizeof(int));
```

then `numbers + 1` really equals `numbers + sizeof(int)` in terms of the addresses (*i.e.* address `0x00001000` will become `0x00001004` if ints are 4 bytes long), `numbers + 2` represents `numbers + 2 * sizeof(int)`, etc. This is how arrays are actually implemented in the C language. The syntax `numbers[4]` is actually shorthand for `*(numbers + 4)`. This means that we generate the address of the fourth integer and then dereference it to get the value of the fourth component of the `numbers` array.

Pointer arithmetic is often used explicitly to iterate through an array. To do this, you use the ++ operator, which in this context is the same as adding 1 to the pointer:

```
int i;

for (i = 0; i < 10; i++)
{
    printf("next number: %d\n", *numbers);
    numbers++;
}
```

This will print out each element of the `numbers` array in turn (we're using the term array loosely here). You might well ask why we don't just use array indexing. Well, you can, but using pointer arithmetic in this way is sometimes more efficient. Let's compare the cost of the code above with the array-based equivalent:

```
int i;

for (i = 0; i < 10; i++)
{
    printf("next number: %d\n", numbers[i]);
}
```

In the first case, each time through the loop you have to dereference the `numbers` pointer and increment it. In the second case, you have to compute `numbers[i]` which is equivalent to `*(numbers + i)`. Thus, you have to do a pointer addition and a dereference. Incrementing a pointer by one is usually faster than adding an arbitrary

integer to it. With multidimensional arrays or arrays of structs, the difference can often be quite significant.

As a general rule, **don't** use pointers this way unless you've determined that there will be a sizeable speed increase (except in this assignment!). We've seen cases where replacing array indexing with pointer manipulations in a critical section of code has sped up a program by a factor of five. We've also seen many more cases where it made little or no difference.

Dangers of pointers

Letting programmers directly access pointers is dangerous. When a pointer is set to a value created by `malloc`, the new storage contains arbitrary values (we say it contains "garbage") at each location. If a C programmer dereferences any location in this storage before data has been put in it, then there is no predicting what the result might be. This is why we recommend you use `calloc` (see above) instead of `malloc` in your code -- you then know that all the memory values are zero.

In addition, if you define a pointer value and try to dereference it before any valid address has been put into the pointer, the computer will often try to access memory that it isn't allowed to, leading in most cases to a *core dump*, which means that the program aborts and dumps a huge file called `core` to your directory (the core file can be useful for debugging). Worse, the memory accessed may actually be memory that is in use by the program, which will not lead to a core dump but to memory corruption, a source of some particularly nasty bugs that will only manifest themselves much later in the program.

Programming with pointers is programming without a net. You have to know exactly what you're doing, or your program will crash.

Program to write

Now, finally, we get to the program you have to write ;-) However, **if you haven't read the above sections, go back and read them now, because you're responsible for them.** Also, there are actually two versions of the program that you'll have to write (see below).

One-dimensional cellular automata

A cellular automaton (CA) consists of a collection of cells which can hold data values, generally organized into some regular geometric arrangement (usually a square grid or a line). A one-dimensional CA (1dCA) has all the cells arranged in a line. The values in the cells change over time. Each cell's value at the next time interval depends on its current value as well as the value of its two immediate neighbors. We set the first and last cells to be always empty. In this case you will be implementing a 1dCA with two

states (full and empty, which you should implement as 1 and 0).

The 1dCA you will implement has the following update rule:

1. If the current cell is empty, and one of the two adjacent cells is full (but not both), set the current cell to be full.
2. Otherwise set the current cell to be empty.

You have to iterate through all the cells to generate the new pattern (the next "generation"). You need to have two arrays to store the data, because you are not allowed to modify one generation until the next generation has been completely computed. The 1dCA should be randomly seeded with 1's and 0's to start with (try having about 50% of the cells 1s initially). If successive generations are printed on a screen, one generation per line, with (say) a "." for 0s and a "*" for 1s, you will get a chaotic-looking pattern that has a fractal-like quality to it and is quite pretty.

Be sure you get the correct update rule in your program; every year, some students get some part of it wrong, and have to redo the assignment. If the pattern generated by your program doesn't look cool, you've almost certainly done something wrong.

Command-line arguments

Your program should take two command-line arguments: the numbers of cells in the 1dCA and the number of generations to compute. The program should print out this many generations to the screen. Use dynamic memory allocation to create the arrays you need to hold the 1dCA data. Make sure you check the return value of every `malloc` or `calloc` call and exit with an error message if the call returns `NULL`. Also make sure you free your newly-allocated data before your program exits.

NOTE: if your program doesn't get the correct number of arguments, it should exit with a usage message as usual. Specifically, **make sure you test what the program does when no arguments are given!** Every year, a large percentage of the programs core dump when no arguments are given; this is just bad programming.

Note that you can do this entire program without using pointer arithmetic. To give you practice in using pointer arithmetic, write the function which updates the board in two ways:

1. only using array operations;
2. only using pointer operations.

Define two separate programs for these two cases (with different names). The programs will share all of their code except for the board updating function. For the second program, you **don't** have to use explicit pointer arithmetic in any functions other than the update function; use array operations everywhere else. Also note that for both programs you have to use dynamic memory allocation when allocating the arrays (because you don't know the size of the arrays in advance).

Don't just translate the array operations into equivalent pointer operations! Instead, use the iteration idiom shown above, suitably modified to work with your program. Specifically, you should use *three* pointers to iterate through the array: for any given value of an index variable `i`, one points to `cell[i-1]`, one to `cell[i]` and one to `cell[i+1]` (assuming that the array is called `cell`). Each of the three pointers will get incremented once for each new cell value that is being computed. In addition, you'll need another pointer into another array which is the array you're writing into. In this way you won't have to use array operations at all, and the only pointer arithmetic you'll need is to increment by one (++). This requirement applies throughout the entire function; even if you have to copy an array at the end, use pointer operations to do this. Once again, don't just translate the array code into the equivalent pointer code; every year, several students in this track do that, and they end up having to redo the assignment.

As always, try to decompose your program into small, easy-to-understand functions instead of having a few huge functions. Make each function do one thing only (don't mix printing with updating, for instance).

Don't forget to include the memory leak checking code in your file (see above). The assignment will not be graded until your program has no memory leaks.

Other stuff to do

Write your own `Makefile` for this lab, based on the examples you've seen before. Each of the two programs you have to write should have a different name. You should have separate targets to compile each program, and for each program you should have separate targets for the executable program and for the object code which you compile from the source code file. In other words, if your array program source code file is called "`1dca_array.c`" you should have targets for "`1dca_array`" and for "`1dca_array.o`", and the "`.o`" file will be a dependency for the executable file and will be used to compile it. There should also be a target for "`memcheck.o`", the memory leak checker, which will be built from "`memcheck.c`"; "`memcheck.o`" will also be a dependency for the two executables. Make sure you get all the dependencies right (it isn't hard; just think about what every target depends on *i.e.* what files would, if changed, require the target to be rebuilt)

In addition, you should also have an "`all`" target that compiles both programs (*i.e.* you type "`make all`" and both programs are built). It's convenient to make the "`all`" target the first target in the program, so just typing "`make`" will build both programs. **Note that you can (and should) write the "`all`" target in a single line (hint: what are its dependencies?).** You should also have a "`clean`" target that removes all compiled files (object files and executables), and a "`check`" target that calls the style checker on both programs (note that the style checker can take multiple filenames as arguments).

Name your programs anything you want, except don't call them "life" or some variant of the word "life". "Life" refers to a specific kind of two-dimensional cellular automaton, not the one-dimensional cellular automaton we're writing here.

Supporting files

1. memcheck.h (header file for the memory leak checker).
2. memcheck.c (the memory leak checking code itself).
3. The ever-annoying style checker.

To hand in

The source code for the two versions of your program, along with your `Makefile`.

References

- Darnell and Margolis, chapter 7.
- K&R, chapter 5.
- Some background information on one-dimensional cellular automata.
