Pointer Arithmetic

coursera.org/learn/pointers-arrays-recursion/supplement/3P2jM/pointer-arithmetic

Like all types in C, pointers are variables with numerical values. The precocious programming student may wonder if the value of variables can be manipulated the way one can manipulate the value of integers and floating point numbers. The answer is a resounding Yes!

Consider the following code:

```
1
2
3
4
5
6
7
int x = 4;
float y = 4.0;
int *ptr = &x;


x = x + 1;
y = y + 1;
ptr = ptr + 1;
```

Lines 1–3 initialize the values of 3 variables of various types (integer, floating point number, and pointer-to-an-integer). Lines 5–7 add 1 to each of these variables. For each type, adding 1 has a different meaning. For *x*, adding 1 performs integer addition, resulting in the value 5. For *y*, adding 1 requires converting the 1 into a floating point number (1.0) and then performing floating point addition, resulting in 5.0. For both integers and floating point numbers, adding 1 has the basic semantics of "one larger". For the integer pointer *ptr* (which initially points to *x*), adding 1 has the semantics of "one integer later in memory". Incrementing the pointer should have it point

to the “next” integer in memory. In order to do this, the compiler emits instructions which actually add the number of bytes that an integer occupies in memory (Lines 1–3 initialize the values of 3 variables of various types (integer, floating point number, and pointer-to-an-integer). Lines 5–7 add 1 to each of these variables. For each type, adding 1 has a different meaning. For x, adding 1 performs integer addition, resulting in the value 5. For y, adding 1 requires converting the 1 into a floating point number (1.0) and then performing floating point addition, resulting in 5.0. For both integers and floating point numbers, adding 1 has the basic semantics of “one larger”. For the integer pointer ptr (which initially points to x), adding 1 has the semantics of “one integer later in memory”. Incrementing the pointer should have it point to the “next” integer in memory. In order to do this, the compiler emits instructions which actually add the number of bytes that an integer occupies in memory (*e.g.*, +1 means to change the numerical value of the pointer by +4). Likewise, when adding N to a pointer to any type T, the compiler generates instructions which add (N * the number of bytes for values of type T) to the numeric value of the pointer—causing it to point N Ts further in memory. This is why pointer arithmetic does not work with pointers to void; since the compiler has no idea how big a “thing” is, it does not know how to do the math to move by N “things”.

The code we have written here is legal as far as the compiler is concerned, however, our use of pointer arithmetic is rather nonsensical in this context. In particular, we have no idea what box *ptr* actually points at when this snippet of code finishes. If we had another line of code that did *ptr = 3;, the code would still compile, but would have *undefined behavior* — we could not execute it by hand and say with certainty what happens. Specifically, when *ptr = &x*, it is pointing at one box (for an integer) which is all by itself—it is not part of some sequence of multiple integer boxes (which we will see shortly). Incrementing the pointer will point it at *some* location in memory, we just do not know what. It *could* be the box for y, the return address of the function, or even the box for ptr itself.

The fact that we do not know what happens here is not simply a matter of the fact that we have not learned what happens—it is a matter of the fact that the rules of C give the compiler a lot of freedom in how it lays out the variables in memory. One compiler may place one piece of data after *x*, while another may place some other data after *x*. In fact, the same compiler may change how it arranges variables in memory when given different command line options changing its optimization levels.

We will consider all code with undefined behavior, such as this, to be erroneous. Accordingly, if you were to execute this code by hand, when you perform *ptr = ptr + 1;*, you should change the value of *ptr* to just be a note that it is *&x+1*, and not any valid arrow. If you then dereference a pointer which does not validly point at a box, you should stop, declare your code broken, and go fix it. We will note that simply performing arithmetic on pointers such that they do not point to a valid box is not, by itself, an error—only dereferencing the pointer while we do not know what it points at is the problem. We *could* perform *ptr = ptr - 1*; right after this code, and know with certainty that ptr points

at *x* again. We might also just go past a valid sequence of boxes at the end of a loop, but not use the invalid pointer value. We will generally not do these things, and you should know the difference between what is and is not acceptable coding practice.

e.g.

Completed