CM 10227: Lecture 4

Dr Rachid Hourizi and Dr. Michael Wright

October 27, 2016

This Week

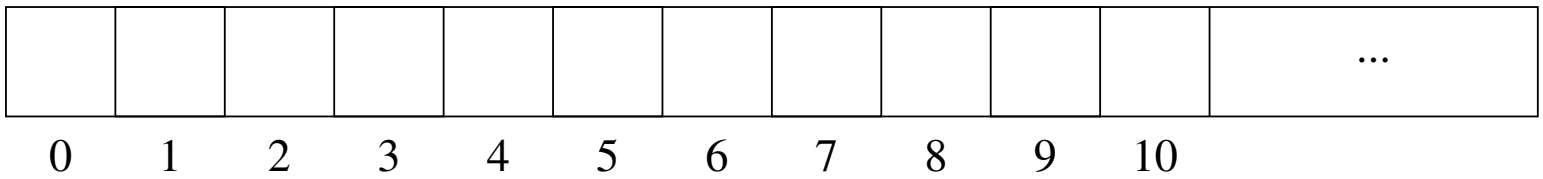
- Complex Collections
- Abstract Data Types

- Memory Allocation and Pointers

Pointers

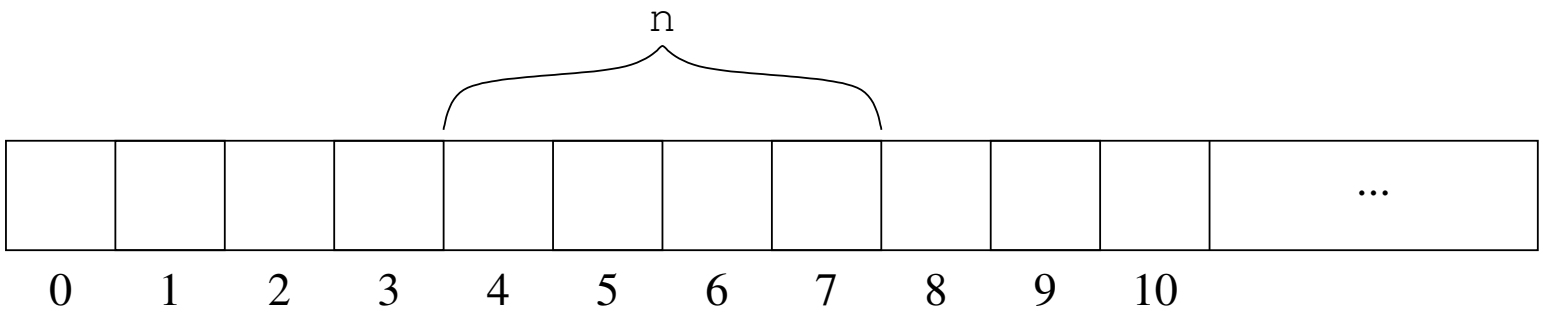
We now turn to one of the features of C that (a) some people find difficult, and (b) makes C so useful: pointers

- We start by reviewing the way memory is laid out in hardware
- Recall that (thanks to the universal adoption of von Neumann's model) memory can be regarded as a big array of bytes; conventionally numbered from 0 upwards



When a program is compiled, variables are mapped in some useful way to memory location by the system (compiler and OS program loader)

- So if we have a (4 byte) integer `n` in our code, the system might choose to place it at memory address 4 (a very unlikely place in real systems)



Then every access of `n` in our code becomes a read or write of bytes 4–7 of memory

- We say byte 4 is the *address* of the variable `n`
- It's where the variable lives in memory

C gives us access to these addresses in our program: that is, we can find out where a variable has been placed

- Other languages might not reveal this kind of information, preferring to hide these details from the programmer
- But for low-level programs that manipulate bits and bytes of memory this is just what they need
- To get the address of a variable use the & operator

```c
#include <stdio.h>
int main(void)
{
    int n = 1234;
    printf("n has value %d and address %p\n", n, &n);
    return 0;
}
```

- Produces
  n has value 1234 and address 0x7fff251f6d5c

Note the difference between the *value* of `n` and the *address* of `n`

- The value of `n` will always be 1234; the address (this example: 140732877607788 in decimal) will likely be different on different OSs, different on different compilers, possibly different on different runs on the same machine
- It all depends on where in memory `n` happens to be placed when the program is loaded to be run

But addresses are just integers

- C does treat them slightly differently from normal integers to make certain nice things happen, but, at base, they are just integers
- The %p in `printf` prints addresses in hexadecimal, as that is often useful to the programmer

Addresses are first-class values in C: this means you can use and manipulate them just like any other values (like integers, doubles, etc.)

- They are just integers, after all
- Variables that hold addresses are called *pointer variables*
- (Though it's not the variables that are pointers, but their values. . .)

So a pointer variable contains a simple integer (the address), but to make things work nicely, C distinguishes between pointers and integers, and also between pointers to different types

- So a pointer to an integer is treated as different to a pointer to a double
- And both are treated as different from a ordinary integer
- This is a bit subtle: they are all simple integers underneath; it's just how the compiler *manipulates* those integers that will be different for different types

So the *interpretation* of that pointer integer is what is important

- This is to make manipulations of them much more convenient
- Now, memory doesn't "know" what kind of data is being stored at a particular address; memory is just a bunch of bytes

If I gave you 10000000100100100001111111011011 and asked "what does that mean?" you could legitimately say "anything you like"

- It is purely the job of the program (and programmer) to say what a particular bunch of bits is supposed to mean
- The type of a variable or the type of a pointer encodes the information as to what bits they refer to "mean"
- Thus `int n = 99;` says "allocate four bytes of memory somewhere and (while we access these bytes through this `n`) interpret the bits in those bytes as an integer"

At one point the program might store an integer at a given address; later it might store a double there

- It is up to the program to interpret the bits at a given address in whatever way it wants
- Don't make the mistake of assuming the computer magically "knows" what a bunch of bits means. That's the job of the program
- Note: while C makes this quite plain, the same is true for all computer languages

We can declare pointer variables, e.g., `pn`

```
int n;
int *pn;
pn = &n;
```

The `*` is read as "pointer to"; the variable `pn` has type "pointer to `int`"

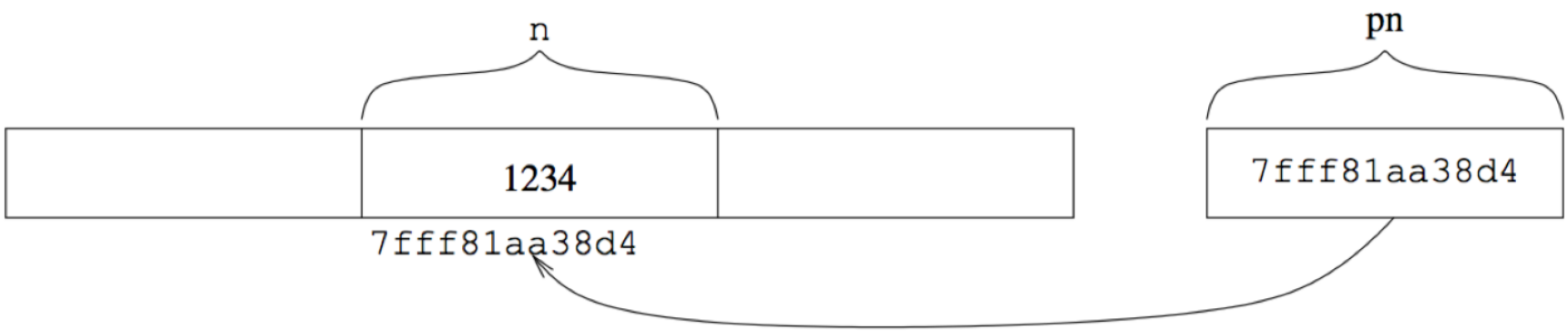
- We also say "`pn` is an `int` pointer"; sometimes "`pn` is an integer reference" or even "`pn` is a reference to an `int`"
- "pointer to" and "reference to" are the same as "address of"

In the opposite direction to &, given a pointer value we can get at the value stored at that address using the * operator

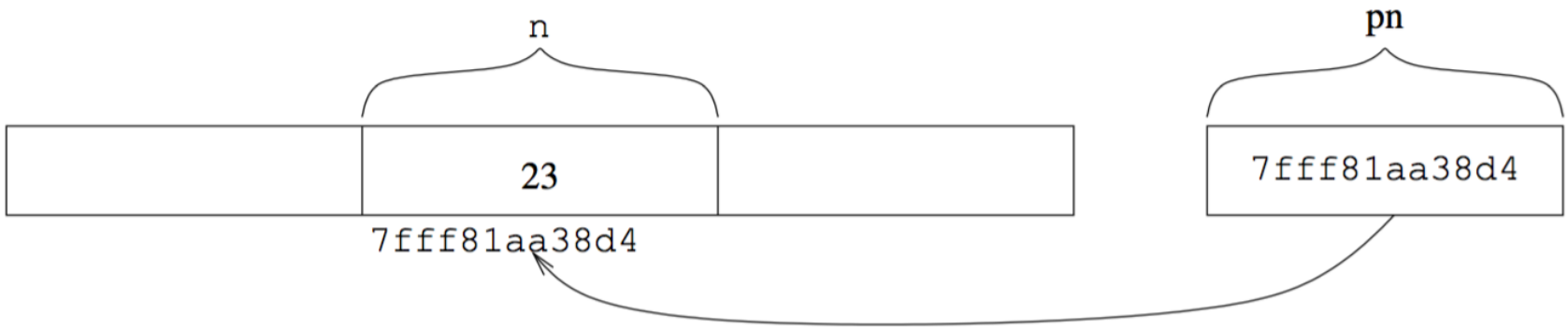
```
int n = 1234, *pn = &n;

printf("n␣has␣value␣%d,␣pn␣has␣value␣%p\n", n, pn);
printf("the␣value␣pn␣points␣to␣is␣%d\n", *pn);
*pn = 23;
printf("n␣has␣value␣%d,␣pn␣has␣value␣%p\n", n, pn);
```

```
n has value 1234, pn has value 0x7fff81aa38d4
the value pn points to is 1234
n has value 23, pn has value 0x7fff81aa38d4
```

Initial values



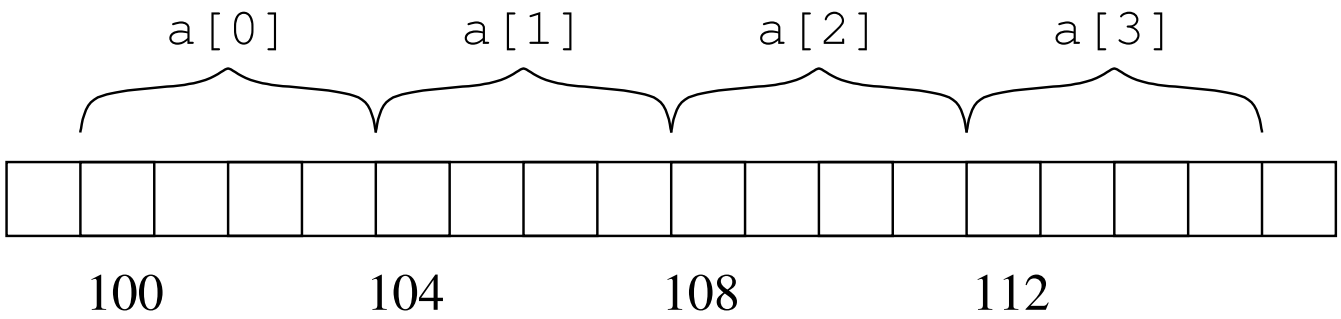
After `*pn = 23`



Arrays and Pointers

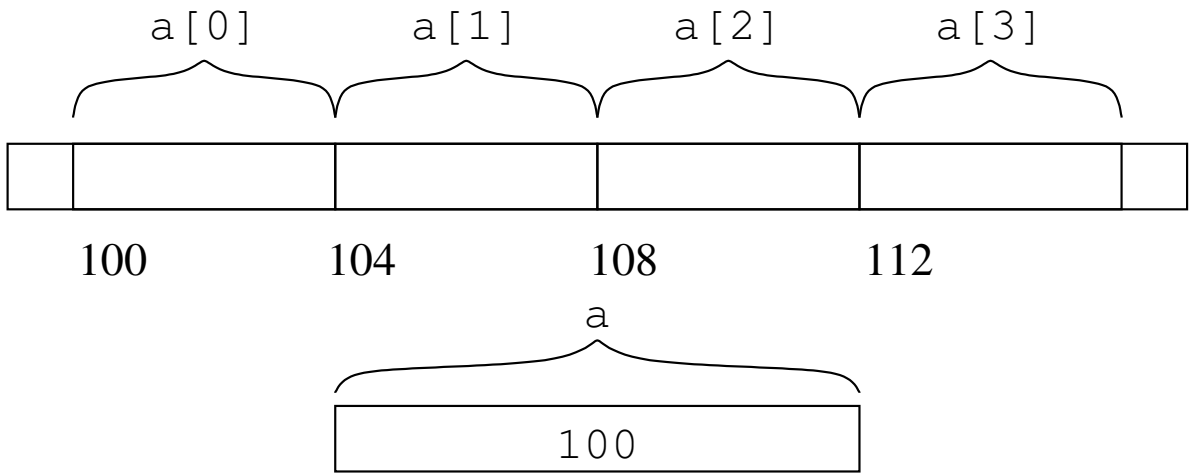
Pointers are intimately associated with arrays in C

- Consider an array `int a[4];`
- In memory, C arrays are laid out simply



To be definite, we fix on using 4 byte (32 bit) integers

In fact the variable a contains the address of the start of the array



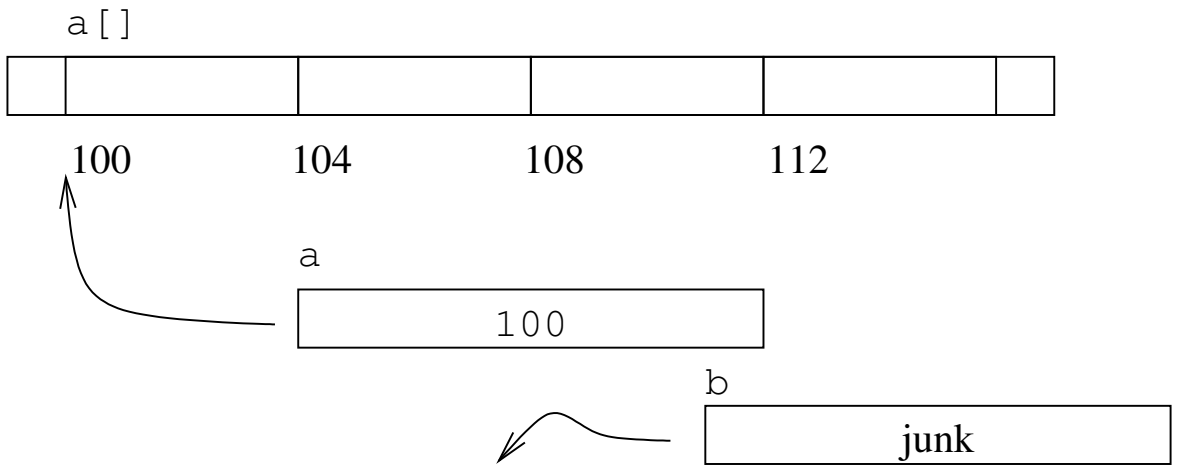
- So a actually has type `int*` (with a caveat)

In the declarations of an array and a pointer

```
int a[4];
int *b;
```

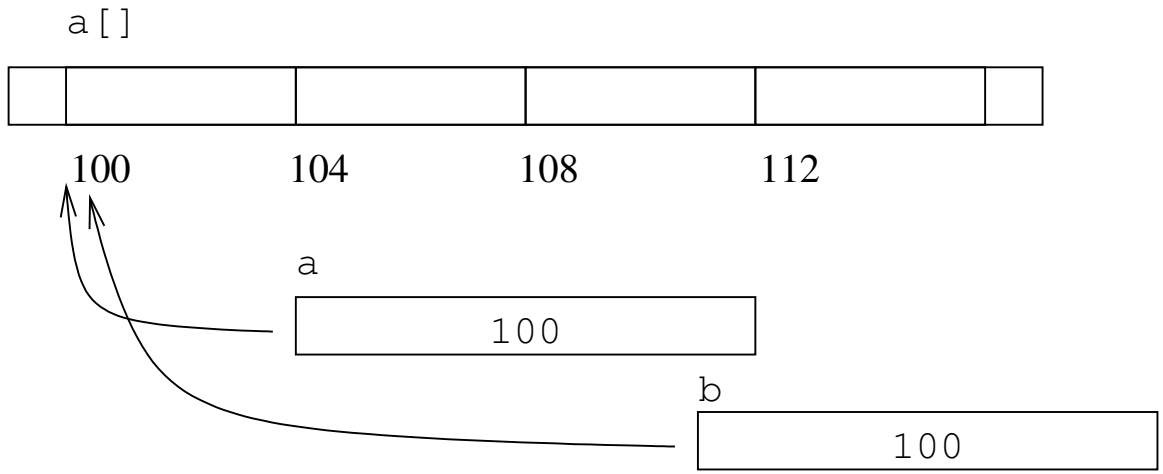
we need to be clear about what is happening

- a is a variable of type pointer to integer **and** a chunk of memory (e.g., 16 bytes) is reserved somewhere for the array; the value of the variable a will be the address of that chunk of memory
- b is a variable of type pointer to integer, with no particular value, and no chunk of memory is reserved



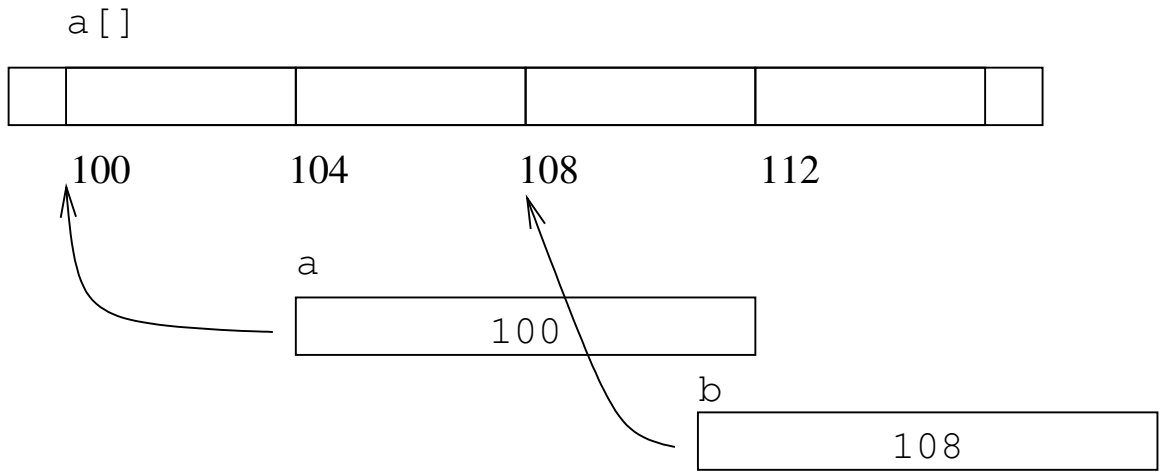
`int a[4]` sets up both `a` and the space for the array;
`int *b` just sets up `b`

b is a pointer variable, so we can set its value: `b = a;`



- And now `b[1]` makes sense; it is the same as `a[1]`
  `b[1]` is at address $100 + 1 \times 4 = 104$

We could equally do `b = a + 2`



- And now `b[1]` makes sense; it is the same as `a[3]`
  `b[1]` is at address $108 + 1 \times 4 = 112$
- And now `b[-1]` makes sense; it is the same as `a[1]`
  `b[-1]` is at address $108 + (-1) \times 4 = 104$

Another difference in this declaration is that a is a *constant* variable (!)

- We can't change the value of a
- This is what we usually want from arrays: if we are thinking of a as indicating the start of an array we don't want its value wandering about in memory
- And b is explicitly a variable pointer: if we need something variable, use a pointer

```
void foo(void)
{
     int a[4];
     a++;
}
```

gives an error message in the compiler

```
void foo(void)
{
     int a[4], *b = a;
     b++;
}
```

is OK as b is allowed to vary

Structures and Pointers

Arrays are fixed-size structures in C

- Once declared, their length cannot be altered
- Some languages allow variably sized arrays: there is a hidden cost to this, though, in speed of access to the elements of the array
- Modern programs need dynamic structures, like lists and trees, that can grow and shrink
- Lists and other dynamic datastructures are made easy in C by the use of structures and pointers

```
#include <stdio.h>

struct Stack
{
    int head;
    int max_size;
    int filo[1000];
};
```

We can define

```
struct IntList {
  int val;
  struct IntList *next;
};
```

This structure contains an integer value and a pointer to the next item in the list

We can define a few values

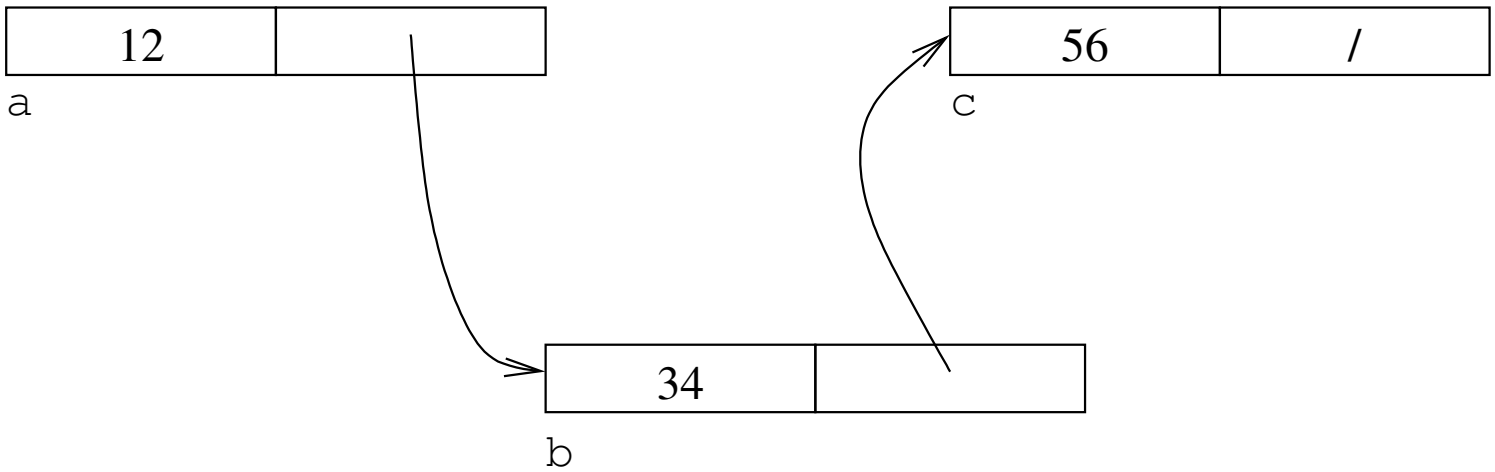
```
struct IntList a, b, c;
a.val = 12; a.next = &b;
b.val = 34; b.next = &c;
c.val = 56; c.next = 0;
```

N.B. this is not the right way to do this kind of thing

- So a is the head of the list; b is next; then c
- We conventionally terminate the list with a 0 pointer as this turns out to be useful later (think about Boolean values)

Now the address values are distracting, not realistic and will vary depending on the compiler, runtime, and other factors.

- So the convention is to use box and pointer pictures. There are no particular values for the addresses, instead arrows indicate the relationships between the boxes
- The actual locations of the structures in memory are not relevant here: but the relationships between the structures are



Suppose we are given the head of the list, `a`

- Getting the value in `a` is easy: just `a.val`
- How to get the next value in the list?
- `a.next` is a pointer to `b`...
- ... so we need `*(a.next)` to follow the pointer to get at the `struct b` ...
- ... then `(*(a.next)).val` for the value in `b`

- This is ugly, but is such a common usage C provides the arrow `->` operator, to prettify code
- So `expr->name` is the same as `(*expr).name`

- Thus `a.next->val` same as `(*(a.next)).val` ,but easier to read
- Further, `a.next->next->val` is the value in c
- The first accessor is a dot, as `a` is a struct; the others are arrows as they follow pointers to structs

```
struct IntList {
  int val;
  struct IntList *next;
};

int main(){
    struct IntList a, b, c;
    a.val = 12; a.next = &b;
    b.val = 34; b.next = &c;
    c.val = 56; c.next = 0;
}
```

```c
void printlist(struct intlist *l)
{
   struct IntList *ptr;
   for (ptr = l; ptr != NULL; ptr = ptr->next) {
      printf("%d\n", ptr->val);
   }
}
```

- We pass a pointer to the structure into `printlist`
- The pointer variable `ptr` will iterate down the items in the list
- The test for termination of loop is `ptr != NULL` as `ptr` is `NULL` at the end of the list
- The `ptr` is updated at each iteration to point to the next item in the list

```
int main(){
    struct IntList a, b, c;
    a.val = 12; a.next = &b;
    b.val = 34; b.next = &c;
    c.val = 56; c.next = 0;

    printlist(&a);
}
```

We can also do this with recursion...

```
void printlistrec(struct IntList *l)
{
  if (l) {
    printf("%d\n", l->val);
    printlistrec(l->next);
  }
}
```

Questions