Introduction (0:00-2:00)

- In the news: iPhone software broken again!
- This is a good example of the constant struggle between those who make software and those who try to crack software
- Notice that the adversary has advantage: the software developer has to think of and protect against every possible bug; the hacker only has to find one
- This is partially a result of the nature of the language, C. We will see why this is true as we get into pointers later today.

This Week's Problem Set: The Game of Fifteen (2:00-9:30)

- `make`
  - In the past, you would compile, for instance, hello.c by typing `gcc hello.c` at the command line
  - You might add in some extra stuff like `gcc –o hello hello.c –lcs50`
  - In this week's problem set, all of that stuff you normally type is encoded in the command `make`
  - You can type `make find` at the command line, and the computer will interpret that as `gcc –o find helpers.c find.c –lcs50`
  - You can also just type "`make`" at the command line and that will compile all the necessary files
  - This is because the make command has been defined in the Makefile, which has been written for you
  - You can take a look at the Makefile in your ps3 directory to see how the computer interprets the make command
- Random number generation
  - A computer cannot generate truly random numbers because it is a deterministic machine.
  - That is, given a particular input, it will always produce the same output
  - It can't come up with truly random data "off the top of its head"
  - But we can write "pseudorandom" number generators, which produce sequences of numbers that *appear* random
  - That is, the sequence has no apparent pattern that would allow an observer to think of the next number
  - Furthermore, the generation of each new number in the sequence depends on a function that operates on the previous number in the sequence. The first number depends on the "seed" – an input that you must feed to it when you use it.
  - This means that, as long as you use a new, unique seed, you will get a difference sequence than you got the previous time

    - A common seed used is the current time (or time elapsed since some fixed point in the past)
    - In this week's problem set we include a file generate.c which is a pseudorandom number generator
- Your task in this weeks' problem set is to implement the game of 15 that we handed out a few weeks ago
    - The game sits in the main function in what's essentially an infinite loop waiting for the user to provide a move
    - When the user make a move, we pass in the tile number to a function move which returns a Boolean indicating whether or not it is a legal move
    - If the Boolean is false, the program informs the user. Either way, it returns to the start of the loop
    - On each pass, the program checks to see if the gameboard is in a winning position
- You'll notice that we've only provided you with a framework, but you'll have to fill in the holes (i.e., write all the functions that we have left empty!)

How to Approach This Problem Set (9:30-12:30)

- With this problem set, you're going to want to take baby steps
- With short programs like caesar.c, you could probably write the whole thing, compile, and then debug. But with larger more complex programs, this is a very poor strategy because you will end up with a broken program and no idea how to fix it.
- A good programming strategy is to get a basic thing working, test it, get it working, and then repeat with another piece. Gradually put these small pieces together to build up a fully functional program.
- For instance, with caesar.c, you might go through this sequence of versions
    - Version 1: program that gets a single commandline argument, checks number of arguments and quites
    - Version 2: check that argument to make sure its numerical
    - Version 3: ask the user for a string
    - Version 4: iterate over the string and rotate them, not worrying about the modulo arithmetic
    - Version 5: work out the mathematical details
- This way, at each step you have something correct, even if very basic
- It is also easy to isolate problems
- Otherwise, if you write the whole program and compile, you (a) will have errors all over the place and (b) will have no idea where the errors are

Counting in Hexadecimal (12:30-

- Numbers in hexadecimal are in base 16

- We use 16 digits to represent numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
- Using a single digit, then, we can count up to the decimal number 15
- If you ever see in a program 0x followed by a number, this tells the compiler that what follows is a hexadecimal number
- `int x = 0xa` means we want to put the decimal number 10 into the variable x
- There is a nice correspondence between hex numbers and binary numbers
- Recall that we need 4 bits to represent 16 possible digits
- So the 16 binary digits correspond to the 16 possible 4-bit numbers:
  - 0 → 0000
  - 1 → 0001
  - e → 15 → 1110
  - f → 16 → 1111
- This means that something like 0xff simply refers to 1111 1111 in binary, or 255 in decimal
- So we can use hex to represent large numbers more compactly in a program
- For instance, we could represent $2^{32} - 1$, we know that we just want 32 bits worth of 1's, which can be nicely written as 0xffffffff
- Hex will also be used to represent color codes in HTML!

Pointers (18:00-30:00)

- Remember buggy3.c?

```
int
main(int argc, char * argv[])
{
    int x = 1;
    int y = 2;
    swap(x, y);
}

void
swap(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}
```

- It was supposed to swap the values in the variables x and y, but it didn't work
- Why? Because of variable scope.
- When we pass x and y to the swap function we only pass their values. The swap function then makes copies of these variable. The copies have the same values but different locations in memory from x and y.

- So the function properly swaps a and b, but as soon as the function is closed, those copies are gone
- The solution? Pass the variable itself—not a copy—to the function.
- We do this by passing the swap function a "pointer" to the variable, rather than simply its value
- This is called passing by reference.
- Observe a new version of the function swap(), contained in swap.c:

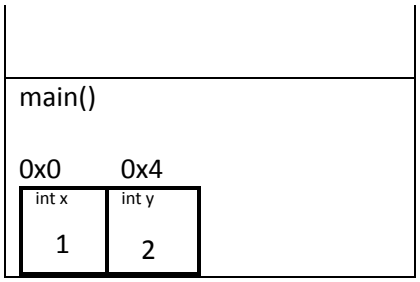
```
void
swap(int *a, int *b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

- int *a and int *b are pointers. They are variables which "point" to the memory locations of a and b.
- This means that, rather than telling swap the values of a and b, we are telling it where in memory to find the two integers. That way, it can access them directly and perform a swap.
- But now that a and b are integer pointer variables, we can't swap their values. This is because their values are actually memory addresses.
- Instead, we want to tell the computer to swap the values located at the memory locations to which a and b point.
- To accomplish this, we use the dereferencing operator *
- When we put * before a variable which is itself a pointer, the resulting meaning is "the value located at the address stored in"
- So tmp = *a means, "go to the address stored in a, fetch that value, and put it in tmp"
- *a = *b means "go the address stored in b, fetch that value, and put it at the address pointed to by b"
- Finally, *b = tmp means, "put the value tmp in the address pointed to by b"
- But if addresses are what we need, how do we pass them to the function?
- Take a look at the main routine from swap.c:

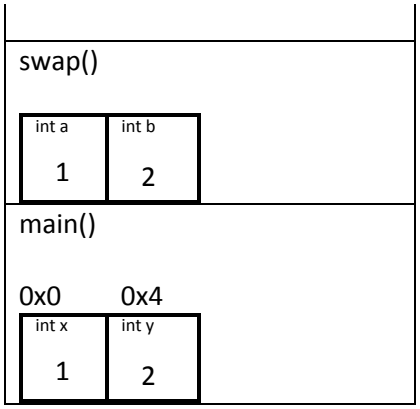
```
int
main(int argc, char * argv[])
{
    int x = 1;
    int y = 2;

    swap(&x, &y);
}
```

- &x means "the address of x"

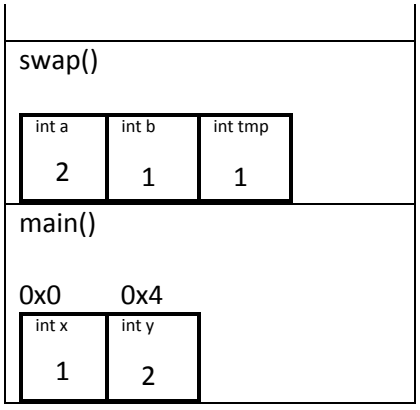
- In this main routine, we pass swap() the addresses of x and y, rather than their values as before
- Recall the stack. At the bottom is a chunk of memory for main. In the chunk of memory are two variables, x and y. Suppose they have addresses 0x0 and 0x4. These variables get the values 1 and 2.



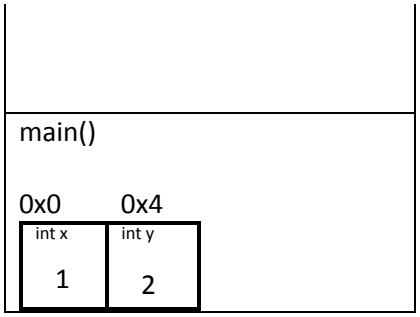
- (Why are the memory addresses separated by 4? Because an int is 32 bits or 4 bytes)
- Suppose we are running buggy3.c (passing by value). When main calls swap, swap gets a new chunk of memory above main. Within this chunk of memory it makes two vararibles called a and b. These variables get the values 1 and 2.



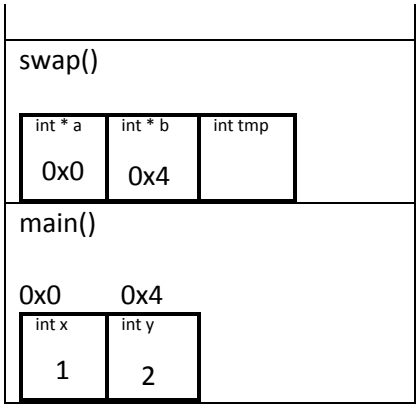
- We perform swap as planned:



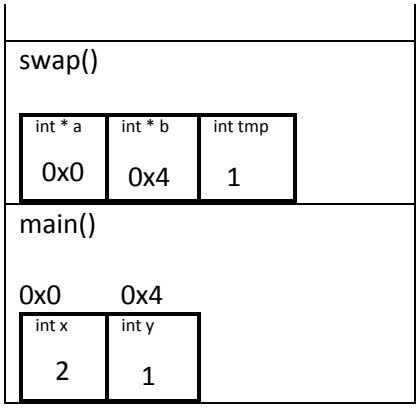
- But when the function closes, it gets popped off the stack and a and b are gone. x and y have their same from before values.



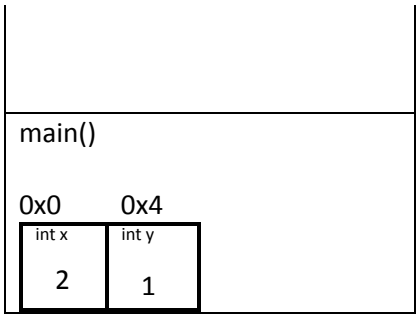
- Now suppose we run swap.c (passing by reference). This time when we call swap, we pass it the addresses of x and y. The variables a and b get these addresses:



- When we perform the swap, we follow those addresses and swap the values contained in them.



- Then, when swap closes, it gets popped off the stack and a and b disappear, but our swap has been achieved:



GDB (30:00-34:00)

- To demonstrate, let's run this through gdb
- As usual, we can print the values of x, y to get 1,2,
- Now, we can also type `print &x` (or, in shorthand, `p &x`) to get the address of x: 0xbfc32220
- (It's a bit bigger than 0x0 because the computer's already stored a bunch of stuff in RAM)

Checking Pointers (34:00-39:30)

- As you've already seen, we can get seg faults if we touch memory that doesn't belong to us
- When we're dealing with pointers, we will get seg faults if we end up with pointers that are invalid
- When the swap function follows the pointers a and b, it is taking a leap of faith that those pointers are valid. If they're not, this could result in a seg fault.
- For instance, suppose we declared x and y as pointers in main. Notice that we can do this using t he following syntax:

```
int * x;
int * y;
```

- Then we pass them to swap by calling swap(x, y). Notice that we don't' pass the address of x and y because x and y are already pointers.
- Now swap gets these pointers, but they haven't been initialized! Who knows what they might contain! Since we didn't initialize them, they might contain addresses to memory locations we don't want accessed. This gives hackers a way to access, alter, and maybe even insert own program into parts of memory.
- So, as a convention, we initialize pointers to NULL when we declare them.
- Then, when we read them into functions, or get them back from functions, we check to make sure they are not equal to NULL.
- So we would declare them in main as

```
Int * x = NULL;
int * y = NULL;
```

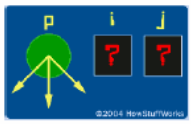
- Then we'd pass them to this improved version of swap from swap1.c that checks to see if the pointers are NULL before trying to access them:

```
void
swap(int *a, int *b)
{

    if (a ! = NULL && b != NULL) {
          int tmp;
          tmp = *a;
          *a = *b;
          *b = tmp;
     }
}
```

Some Pointer Illustrations (39:30-45:30)

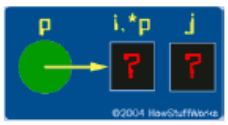
- Check out hwostuffworks.com for some sweet visual representations of pointers. We examine a few here.
- What's going on in this one?

```
int i, j;
int *p;
```



- Essentially, this means that if we write those lines of code, we get two boxes in memory with unknown content, and one location in memory pointing to who-knows-what
- What about this one?

```
p = &i;
```



- This just means we're taking the address of i and storing it in p, thus making p "point" to i.
- Finally, *p = 5 just means follow p's arrow, and put the value 5 there:

```
*p = 5;
```