X Window System

or

How to open windows on your computer

What is a Window System?

Basic Framework for GUI environment

Drawing and moving windows

Interacting with mouse and keyboard

What is the X Window System?

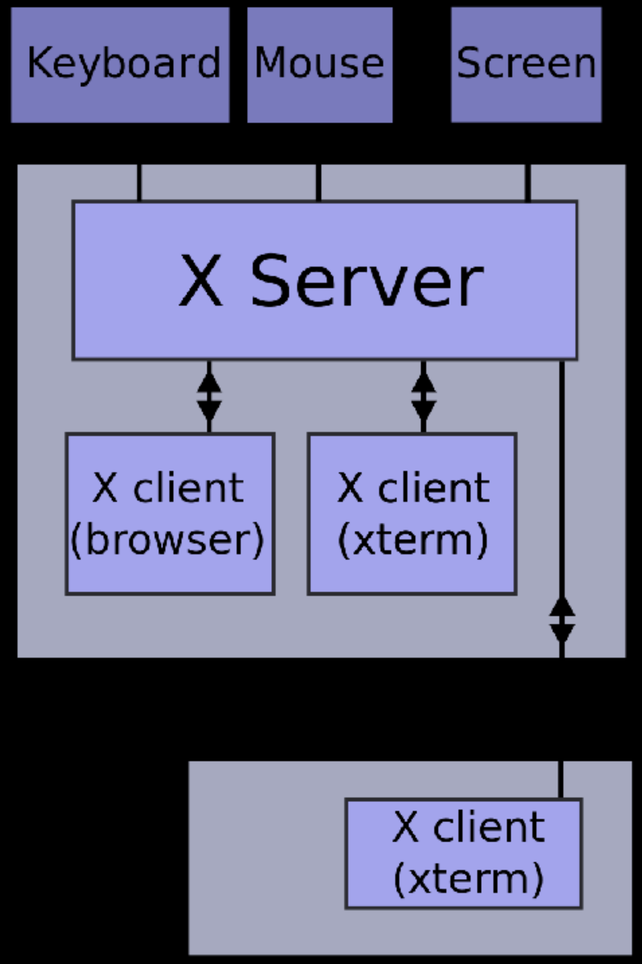
a topic that cannot be done justice in ten minutes

Basics:

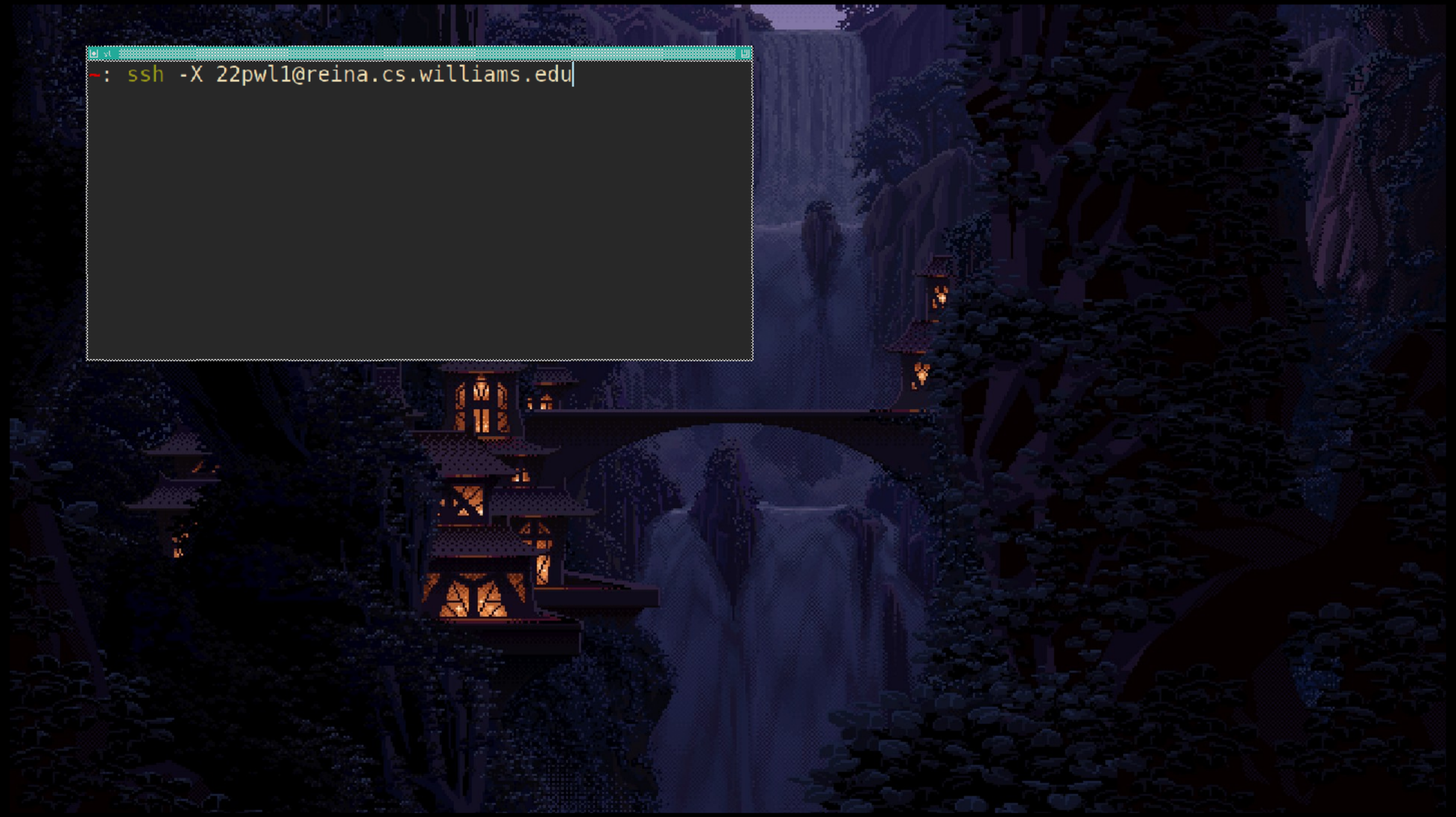
- Originally developed at MIT, 1984
- Primarily used by Unix/Unix-like operating systems
- Open Source
- Dominant windowing system on Linux for decades.
- Recently, starting to be replaced by Wayland

Features / Non-Features

- Basic, low level functionality.
- Tools to make a GUI: not the actual GUI
- Designed to be used over network; completely network transparent
- No native support for audio


Keyboard
Mouse
Screen
X Server
X client
(browser)
X client
(xterm)
X client
(xterm)

Is X really network transparent? can you prove it?


~: ssh -X 22pwl1@reina.cs.williams.edu

```
22pwl1@reina:~$ cd cs432/lab6
22pwl1@reina:~/cs432/lab6$ evince 06-threads.pdf &
[1] 179538
22pwl1@reina:~/cs432/lab6$ emacs user/setjmp.S
```

emacs@reina

File Edit Options Buffers Tools Asm Help

Save Undo

```
/*
* setjmp(struct context *c)
 * setjmp routine stores the current callee saved registers into the structure
* stored at c.  As this routine enters, the register a0 contains the
* address of this structure.  Returns 0.
*/
    .global setjmp
setjmp:
        sd    ra,0(a0)
        sd    sp,8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)
        li    a0,0 # return 0
        ret

/*
    * longjmp(struct context *c, uint64 rval)
* Longjmp routine restores stores the current context at c.
* When this routine returns, it should appear to return from the target
* setjmp, and returns rval.
*/
    .global     longjmp
longjmp:
        ld    ra,0(a0)
```

-:--- setjmp.S Top L1 Git main (Assembler)

For information about GNU Emacs and the GNU system, type C-h C-a.

06-threads.pdf

2 of 4 06-threads.pdf 49.0%

So, what's happening? The C library routine `setjmp` stores the state of the machine in the `target`. When called, `setjmp` always returns 0. Later, the programmer can choose to resurrect the state of the machine at the point of the `setjmp` with a `longjmp`. When the `longjmp` is called, it returns, but not from its own call, but from the call to `setjmp`! To distinguish between the two different returns from `setjmp`, `longjmp` takes a second parameter that describes the return value from `setjmp`. Typically this value is non-zero. Your first job this week is to build the mechanism that makes this work.

A Little Thread Library. The promise of multiprogramming, of course, is to support lots of concurrent executions. Typically, of course, these different processes have very little to do with each other. Indeed, a large part of our semester has been spent thinking carefully how these processes can be *isolated* from each other. Sometimes, however, we would like to have an abstraction where related *threads of execution* run concurrently, trying to collaborate on the solution to some problem. This week we will think about the construction of a very simple library that allows a single monitor process to clone itself in an attempt to perform more than one computation concurrently, or "at the same time".

Our library might support this application, `snap.c`:

```
#include "kernel/types.h"
#include "user/user.h"
#include "user/thread.h"
volatile int counter = 0;  // Why volatile?!?

void snap() {
    for (int i = 0; i < 3; i++) {
        printf("%d: snap!\n",++counter);
        yield();
    }
    fini();
}
void crackle() {
    for (int i = 0; i < 5; i++) {
        printf("%d: crackle!\n",++counter);
        yield();
    }
    fini();
}
void pop() {
    for (int i = 0; i < 4; i++) {
        printf("%d: pop!\n",++counter);
        yield();
    }
    fini();
}
int main(int argc, char **argv) {
    init();         // initialize thread system
    create(snap);   // create runnable threads
    create(pop);
    create(crackle);
    yield();        // let the threads go!
    exit(0);
}
```

Here, the `main` method manages three threads executing the routines `snap`, `crackle`, and `pop`. When the `main` method yields the machine, the threads execute concurrently. Each thread shares the memory of `main`, but each also maintains its own execution context that includes a stack and local variables. One possible output from the above program could be:

```
$ snap
```

How basic is basic?

C library.

<X11/Xlib.h>

How do I start the X window system?

xinit: reads commands from xinitrc

(basically) need some form of window manager on top of X

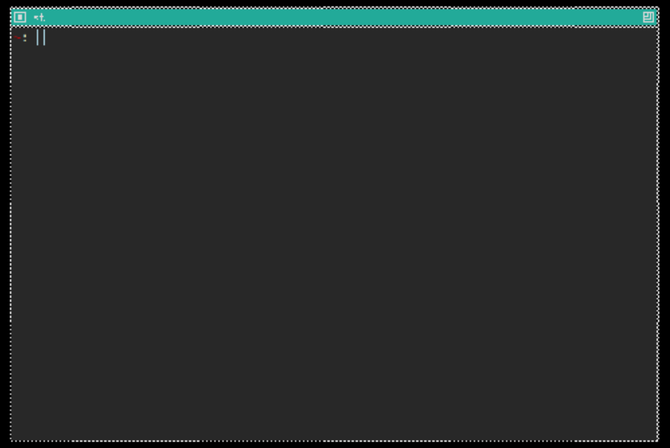
X will not process events on its own. It will just receive them.

The window manager processes events like changing window focus when the mouse moves.

twm is default X window manager, often comes packaged with X.

easy to swap out for other window managers: specify in .xinitrc

Lots of cool window managers written for X: DWM, Xmonad, BSPWM, etc.

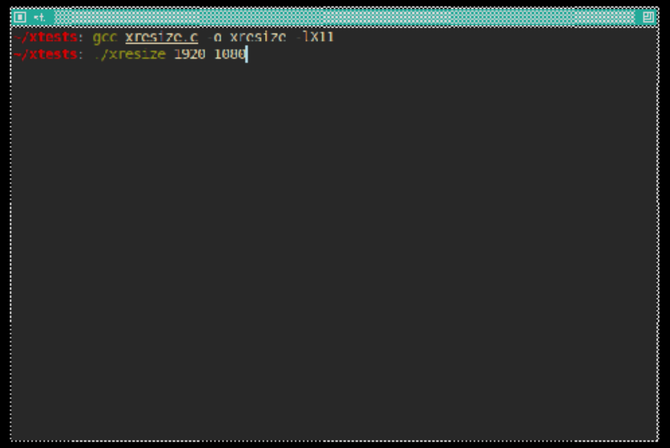


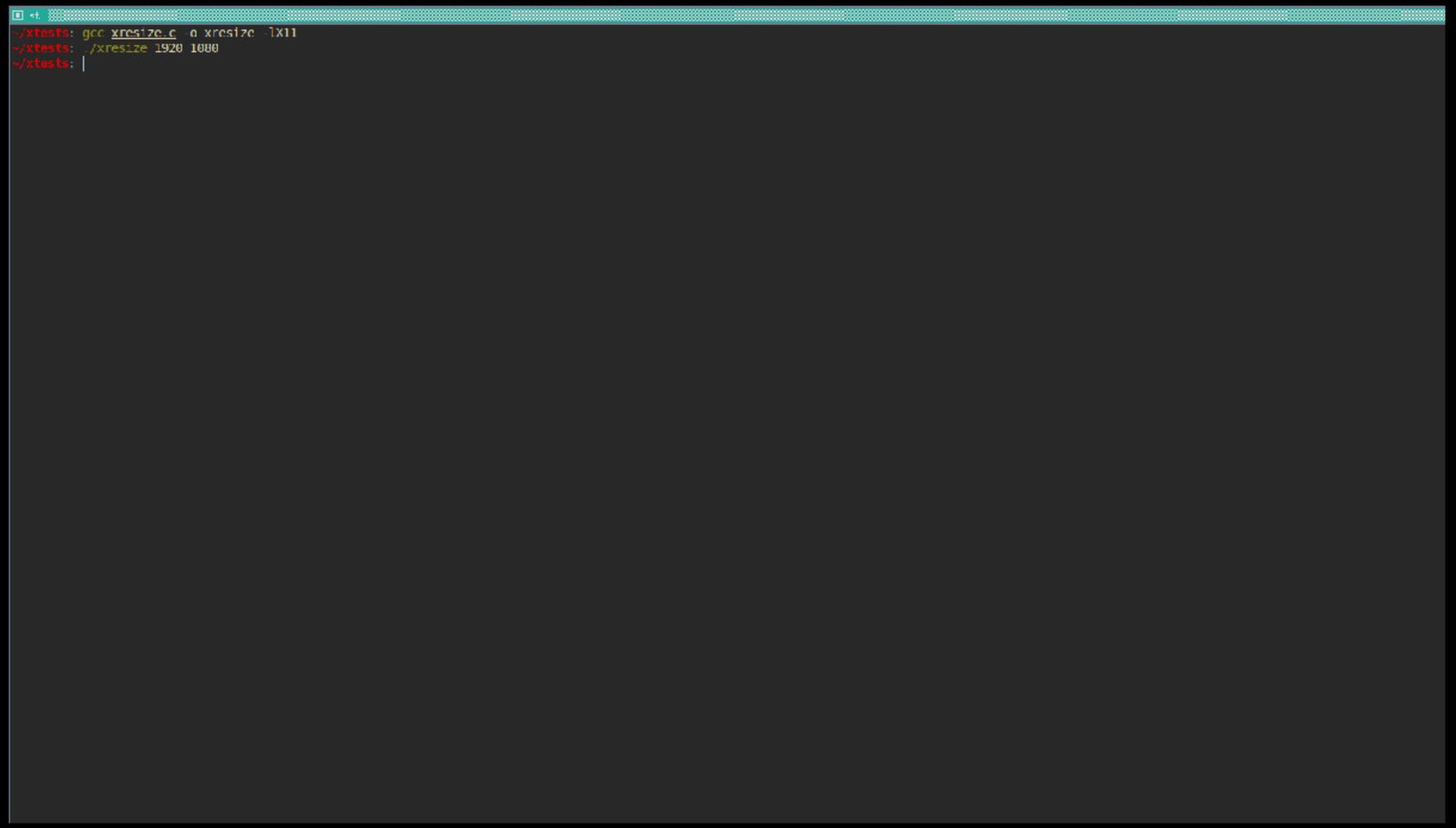
xresize.c

```c
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    Display* display = XOpenDisplay(NULL); //get our display
    int revert;
    Window window;

    XGetInputFocus(display, &window, &revert); //get current window
    XResizeWindow(display,window,atoi(argv[1]),atoi(argv[2])); //resize
    XCloseDisplay(display); //cleanup, I guess?
    return 0;
}
```


~/xtests: gcc xresize.c -o xresize -lX11
~/xtests: ./xresize 1920 1080

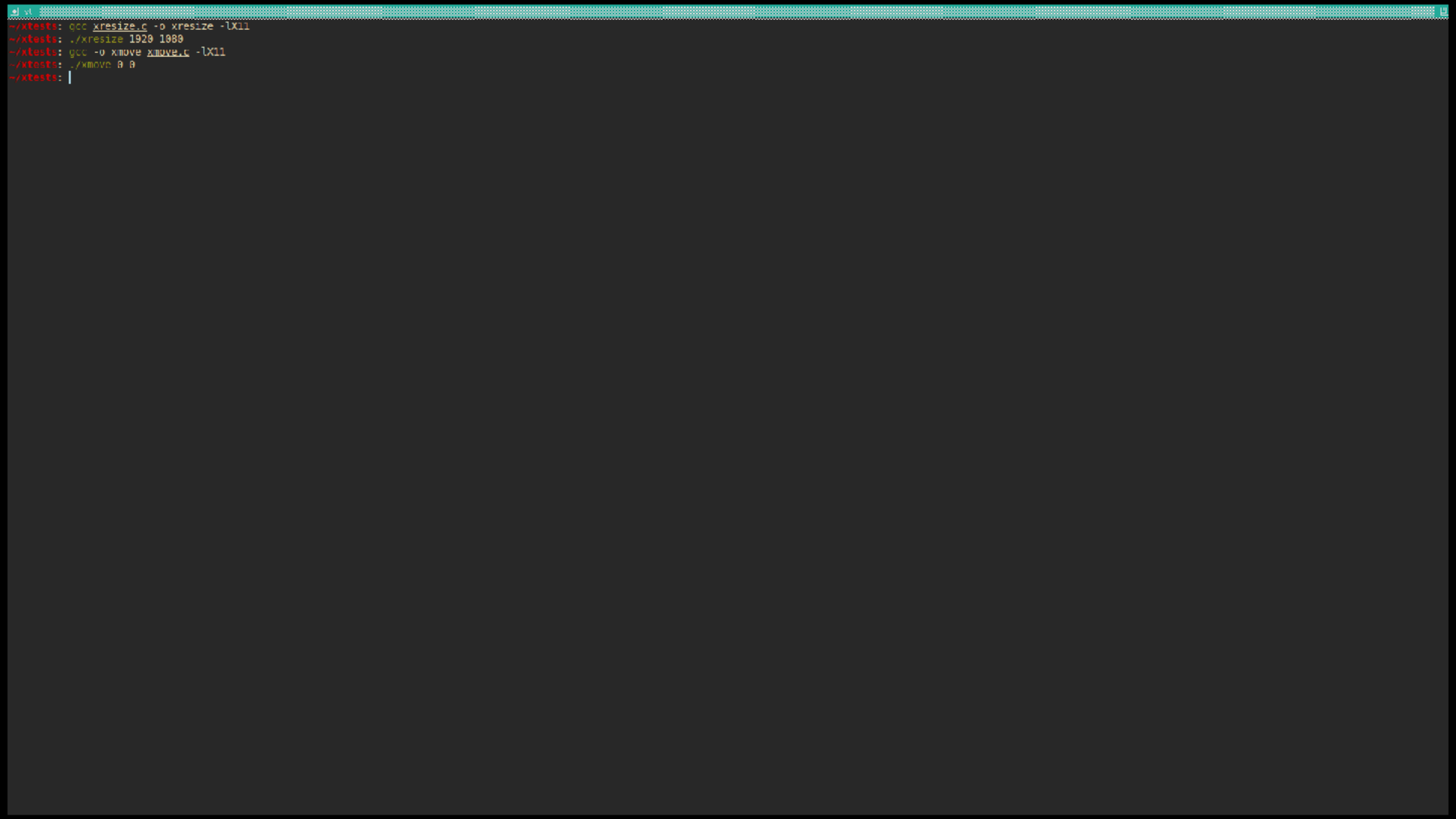

~/xtests: gcc xresize.c -o xresize -lX11
~/xtests: ./xresize 1920 1080
~/xtests:

xmove.c

```c
#include <X11/Xlib.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    Display* display = XOpenDisplay(NULL); //get our display
    int revert;
    Window window;

    XGetInputFocus(display, &window, &revert); //get current window
    XMoveWindow(display,window,atoi(argv[1]),atoi(argv[2])); //move it
    XCloseDisplay(display); //cleanup (?)
    return 0;
}
```


vi
~/xtests: gcc xresize.c -o xresize -lX11
~/xtests: ./xresize 1920 1080
~/xtests: gcc -o xmove xmove.c -lX11
~/xtests: ./xmove 0 0
~/xtests:

This is tedious.

As a user, there is a better way

xdotool: emulate keyboard/mouse input; move/arrange windows

xrandr: arrange monitors

These sorts of tools allow for automating/scripting window arrangements

Which is great!

Desktop environmments are built on top of these low-level C functions.
There's no magic.
If you drag a window to move it on a lab machine, it's probably using X
to detect where the mouse is and using X to change the window's location.

Takeaways:

- Does things that you expect to be done / take for granted on modern computers.
- Easy to overlook how it works.
- But often times how it works is important for security or other reasons.
- Having a basic understanding of how X works lets you automate otherwise tedious tasks.