Ground

- if we have multiple supplies, the grounds may be at a different potential – we'll need to connect them in that case to get a common ground
- on the breadboard, we can connect the power and the ground to the rails along the side, which gives us many places we can get the power or ground from (along the rails line)

Digital Input

Switch

- use a circuit with just a switch in it, connecting an output pin to an input
  - when the circuit's closed, the input pin detects the voltage as a 1

Code

- `#define` defines some text-replacement – the defined name is replaced by its value in the pre-processing step
- all pre-processor instructions start with a # symbol

```
#define IN 8
#define OUT 13

void setup() {
    pinMode(IN, INPUT);
    pinMode(OUT, OUTPUT);
}

void loop() {
    if (digitalRead(IN)) {
        digitalWrite(OUT, HIGH);
    } else {
        digitalWrite(OUT, LOW);
    }
}
```

Weird Circuit Behaviour

- when you wire up the circuit, the input pin retains its charge and the LED stays on, even when the button isn't pressed
  - charge comes from static electricity from the surrounding environment
  - to solve this you can put a connection from the input pin to ground, through a resistor with a highish resistance
    * when the switch is closed, very little current flows through the resistor, since its value is high.
    * when the switch is open, the charge will drain to ground from the input pin, rather than staying there
      · how quickly this happens depends on the resistance of the resistor
    * this is called a pull-down resistor, because we're pulling the voltage down to ground

We can also use a pull-up resistor to do the opposite – connect it to 5V instead of ground, and then the circuit's behaviour is inverted.

This only works because the Arduino's input pins have extremely high impedance (so almost no current flows).

Interestingly, we've now inverted the code-circuit behaviour by changing the circuit, rather than the code. We could also have changed the code. In general, we can either change the circuit or the code to do particular things.

A floating pin is an input pin that doesn't have a well-defined state and could be read as either a 0 or a 1 – this is a source of a large number of errors. We prevented this with a pull-down resistor.

Toggling the LED

Rather than the LED being on when the switch is on, we'd prefer each press of the switch to toggle the LED:

```
#define IN 8
#define OUT 13

// This line is C++, in C typically you have an integer representing true or
// false
boolean ledOn = false;

void setup() {
    pinMode(IN, INPUT);
```

```
    pinMode(OUT, OUTPUT);
    digitalWrite(OUT, LOW);
}

void loop() {
    if (digitalRead(IN)) {
        // Inverts the boolean and writes its value out
        digitalWrite(OUT, (ledOn = !ledOn));
    }
    delay(5);
}
```

Note that assignment statements can be used as expressions in C – the value of the expression is the value of the left-hand operand once the statement has finished:

```
// b gets the value 2, and a gets the value that b got
a = b = 2;
```

Running this code with the circuit, we'll see that sometimes the LED won't switch the way we want. There are two possible reasons for this:

1. If we hold the switch down for longer than one run of the loop, the LED will toggle continuously.
2. Any mechanical switch has a bounce – one press may result in a number of apparent presses

To overcome the first reason we can use a positive-edge trigger, by detecting the transition from 0 to 1, and switch the LED only then.

However, the second problem will still cause the LED not to switch the way we want.

Noisy Switches (Bounce)

A switch press is a coming together of contacts, and because of the mechanical nature of the switch, the contacts may come together and separate several times in quick succession over the course of a single press.

We can address this in hardware or software. In hardware we can use a capacitor – when the switch is pressed the capacitor empties, and must fill again before a new switch press can be detected. The capacitor is acting as a smoothing circuit.

We could also use a latch to fix the problem in hardware. The advantage of this is that no recovery time is needed.

Software Solution

Write an algorithm that determines when the input has finished bouncing, and report a state transition at this point.

Inferior solutions use delays, making assumptions about timings based on the type of switch. The best use timer interrupts (which we'll deal with later).

Delay-based solutions are good enough for us for now.

```
#define IN_PIN 8
#define OUT_PIN 13
#define MAX_BOUNCE_TIME 5

void setup() {
    pinMode(IN_PIN, INPUT);
    pinMode(OUT_PIN, OUTPUT);
}

// functions omitted here

void debouncedRead(int pin, boolean state) {
    if (digitalRead(pin) != state) {
        delay(MAX_BOUNCE_TIME);
        return digitalRead(pin);
    }
    return state;
}

void toggleOutPinWithInPin(int outPin, int inPin, int from, int to) {
    static int currentInputState = LOW;
    static int prevInputState = LOW;
    static int state = LOW;

    currentInputState = debouncedRead(inPin, currentInputState);

    if (prevInputState == from && currentInputState == to) {
        digitalWrite(outPin, (state = !state));
    }

    prevInputState = currentInputState;
}

void loop() {
    toggleOutPinWithInPin(OUT_PIN, IN_PIN, LOW, HIGH);
}
```

In this code, we're assuming the switch won't bounce for more than `MAX_BOUNCE_TIME` –

this won't work for every switch, so the code isn't portable.

C Sidebar

There's no boolean type in C – all non-zero values are true and only zero is false.

An assignment statement is an expression, whose value is the value assigned to the variable.

Static Variables

Normal local variables are created on the stack when you reach their declaration. When the enclosing function is exited, the stack frame is removed from the stack and the variables are lost/destroyed.

Static variables are created on the heap when you reach their declaration. They are not destroyed when the enclosing function is exited, but the variable namespace is restricted to the current function – you can't access them outside the function. Static variables are initialised the first time their declarations are reached, and after that they keep their values between function calls.