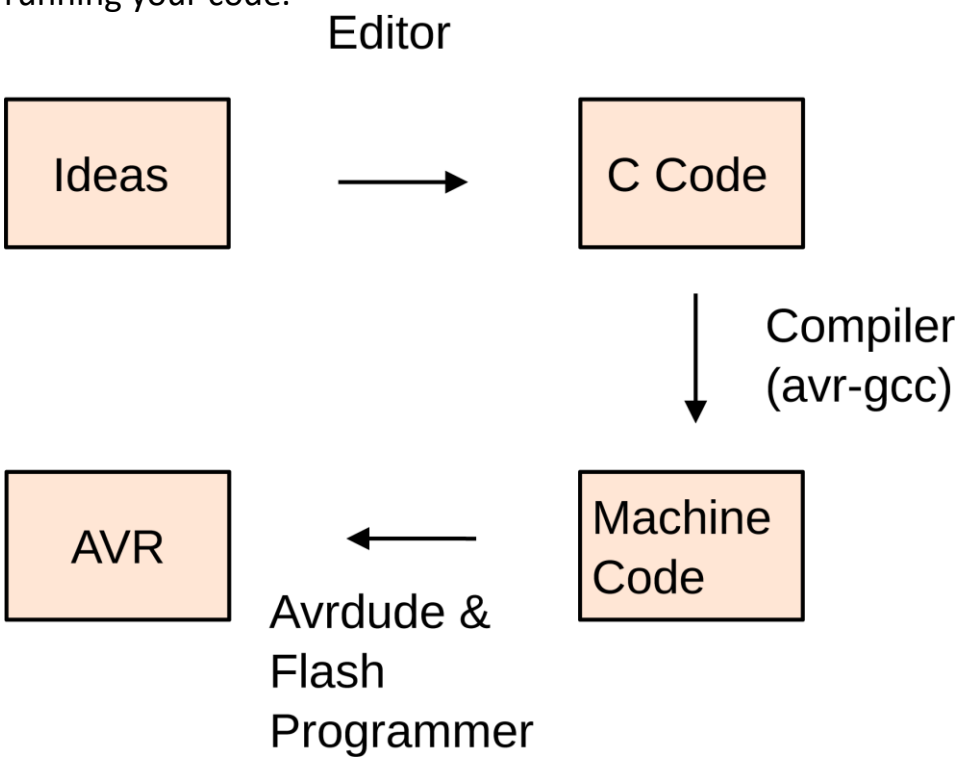
Programming the AVR

Toolchain overview

1. Write your source code in an *editor*.
2. Turn your source code into machine code with a *compiler* (and associated software tools).
3. Using *uploader software* on your big computer and a *hardware flash programmer*, send the machine code to your target AVR chip, which stores the instructions in its nonvolatile flash memory.
4. As soon as the flash programmer is done, the AVR chip resets and starts running your code.



The Arduino as target AVR and as a flash programmer

The first section covers programming the AVR chip that's inside the Arduino using standard C instead of the Arduino dialect.

The second section treats the case where you've already got your AVR chip on a breadboard, and you want to use your Arduino as a hardware flash programmer to transfer the code to the target AVR. Following some simple steps, you can (temporarily and reversibly) use the Arduino as a hardware programmer to flash your code into the bare AVR. And then you can decide to continue using the Arduino IDE to compile and send your code, or you can use any other code editor and the standard AVR development toolchain.

AVR Programming

The Structure of AVR C Code

The code falls into the following rough sections:

```
[preamble & includes]
[possibly some function definitions]
int main(void){
[chip initializations]
[event loop]
while(1) {
[do this stuff forever]
}
return(0);
}
```

Getting Started: Blinking LEDs

```
int main(void) {

  // -------- Inits --------- //
  DDRB |= 0b00100000;          /* Data Direction Register B:



  // ------ Event loop ------ //
  while (1) {
    PORTB = 0b00100000;        /* Turn on sixth LED bit/pin in PORTB */
    _delay_ms(500);                              /* wait */
    PORTB = 0b00000000;        /* Turn off all B pins, including LED */
    _delay_ms(500);                              /* wait */
  }                                    /* End event loop */
  return 0;                      /* This line is never reached */
}
```

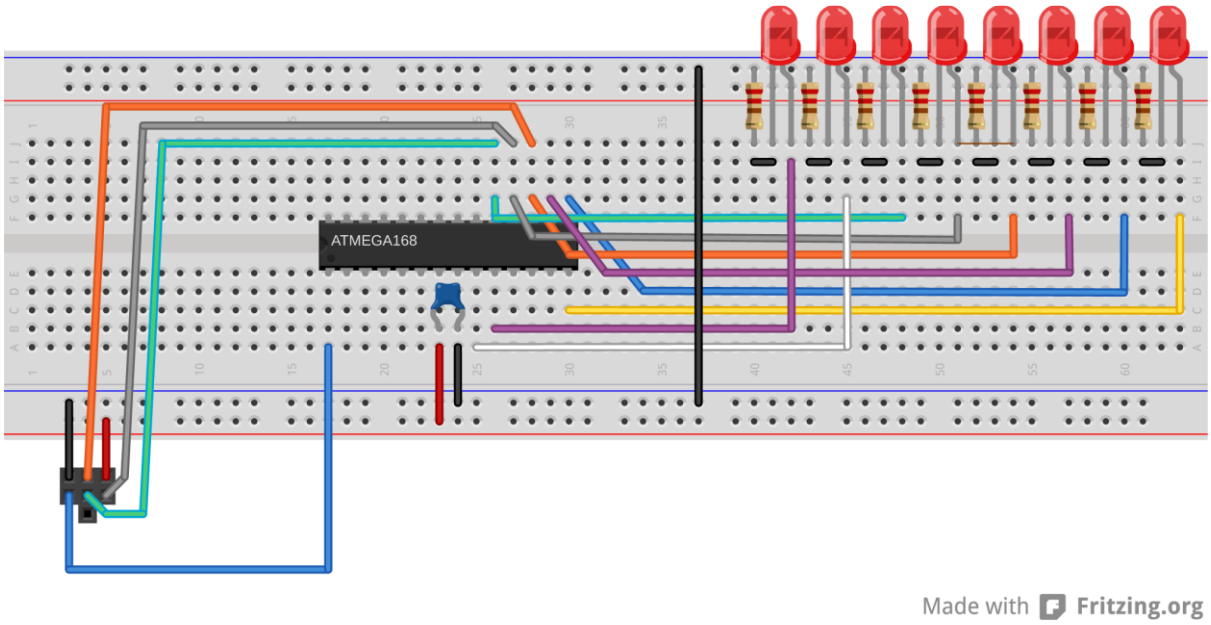
Hardware Registers

DDRx *data-direction registers (port x)*
These registers control whether each pin is configured for input or output—the data direction. After a reset or power-up, the default state is all zeros, corresponding to the pins being configured for input. To enable a pin as output, you write a one to its slot in the DDR.

PORTx *port x data registers*
When the DDRx bits are set to one (output) for a given pin, the PORT register controls whether that pin is set to logic high or low (i.e., the VCC voltage or ground). Switching between these voltage levels could, for instance, turn on and off attached LEDs.
With the DDR configured for input, setting the PORT bits on a pin will control whether it has an internal pull-up resistor attached or whether it's in a "hi-Z" (high-impedance) state, effectively electrically disconnected from the circuit, but still able to sense voltage.

PINx *port x input pins address*
The PIN register addresses are where you read the digital voltage values for each pin that's configured as input. Each PINx memory location is hooked up to a comparator circuit that detects whether the external voltage on that pin is high or low. You can't write to them, so they're not really memory, but you can read from the PINx registers like a normal variable in your code to see which pins have high and low voltages on them.

POV Toy



(pattern printing shall be explained in the session)

```
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
// -------- Inits --------- //
DDRB = 0xff; /* Set up all of LED pins for output */
// ------ Event loop ------ //
while (1) { /* mainloop */
PORTB=(0b00001110);
PORTB=(0b00011000);
PORTB=(0b10111101);
PORTB=(0b01110110);
PORTB=(0b00111100);
PORTB=(0b00111100);
PORTB=(0b00111100);
PORTB=(0b01110110);
PORTB=(0b10111101);
PORTB=(0b00011000);
PORTB=(0b00001110);
PORTB = 0;
_delay_ms(10);
} /* end mainloop */
return (0);
}
```

(this is a random pattern)