C Programming
1DT301

Lecture 2

[OPNOVA]
ENGINEERED INNOVATION

ASM / C Comparison (Blinking LED on an ATmega328P)

```
.org 0x0000
  jmp main

main:
  ;------- Setup stack
  ldi r16, high(RAMEND);
  out SPH,r16
  ldi r16, low(RAMEND)
  out SPL,r16

  ;---------
  sbi DDRB, DDB5 ; Set Pin 5 to OUTPUT

loop:
  sbi PORTB, PB5; LED ON
  rcall Delay200ms
  cbi PORTB, PB5; LED OFF
  rcall Delay200ms
  rjmp loop
```

```
#define F_CPU 16000000UL

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

int main(void)
{
  DDRB = 1 << DDB5;

  while (1)
  {
    PORTB = PORTB | (1 << PB5);
    _delay_ms(200);
    PORTB = PORTB & ~(1 << PB5);
    _delay_ms(200);
  }

  return 1;
}
```

Delay200ms

```
; Generated by delay loop calculator
; at http://www.bretmulvey.com/avrdelay.html
;
; Delay 3 200 000 cycles
; 200ms at 16.0 MHz
Delay200ms:
    ldi  r18, 17
    ldi  r19, 60
    ldi  r20, 204
L1: dec  r20
    brne L1
    dec  r19
    brne L1
    dec  r18
    brne L1
    ret
```

Added by You

push/pop required? Most likely!

Float C -> ASM

```c
int main(void)
{
  float myFloat = 0x1234;

  myFloat = myFloat + 1;

  return 1;
}
```

That call leads to some 200+ lines of Assembly code.

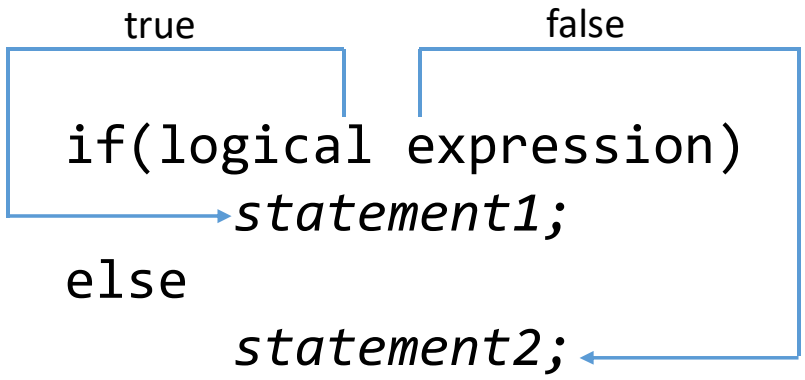
```
int main(void)
{
  80:   cf 93           push    r28
  82:   df 93           push    r29
  84:   00 d0           rcall   .+0             ; 0x86 <main+0x6>
  86:   00 d0           rcall   .+0             ; 0x88 <main+0x8>
  88:   cd b7           in      r28, 0x3d       ; 61
  8a:   de b7           in      r29, 0x3e       ; 62
float myFloat = 0x1234;
  8c:   80 e0           ldi     r24, 0x00       ; 0
  8e:   90 ea           ldi     r25, 0xA0       ; 160
  90:   a1 e9           ldi     r26, 0x91       ; 145
  92:   b5 e4           ldi     r27, 0x45       ; 69
  94:   89 83           std     Y+1, r24        ; 0x01
  96:   9a 83           std     Y+2, r25        ; 0x02
  98:   ab 83           std     Y+3, r26        ; 0x03
  9a:   bc 83           std     Y+4, r27        ; 0x04

        myFloat = myFloat + 1;
  9c:   20 e0           ldi     r18, 0x00       ; 0
  9e:   30 e0           ldi     r19, 0x00       ; 0
  a0:   40 e8           ldi     r20, 0x80       ; 128
  a2:   5f e3           ldi     r21, 0x3F       ; 63
  a4:   69 81           ldd     r22, Y+1        ; 0x01
  a6:   7a 81           ldd     r23, Y+2        ; 0x02
  a8:   8b 81           ldd     r24, Y+3        ; 0x03
  aa:   9c 81           ldd     r25, Y+4        ; 0x04
  ac:   0e 94 68 00     call    0xd0            ; 0xd0 <__addsf3>
  b0:   dc 01           movw    r26, r24
  b2:   cb 01           movw    r24, r22
  b4:   89 83           std     Y+1, r24        ; 0x01
  b6:   9a 83           std     Y+2, r25        ; 0x02
  b8:   ab 83           std     Y+3, r26        ; 0x03
  ba:   bc 83           std     Y+4, r27        ; 0x04

        return 1;
  bc:   81 e0           ldi     r24, 0x01       ; 1
  be:   90 e0           ldi     r25, 0x00       ; 0
}
  c0:   0f 90           pop     r0
  c2:   0f 90           pop     r0
  c4:   0f 90           pop     r0
  c6:   0f 90           pop     r0
  c8:   df 91           pop     r29
  ca:   cf 91           pop     r28
  cc:   08 95           ret
```

Control Flow

Control flow

- Determines in what way your program flows.
- If the sun is shining and the temperature is more than 20 °C then go bathing, otherwise go sleeping at home.

if-else




```
if((sun is shining) && (temp > 20))
        Go bathing;
else
        Go sleeping at home;
```

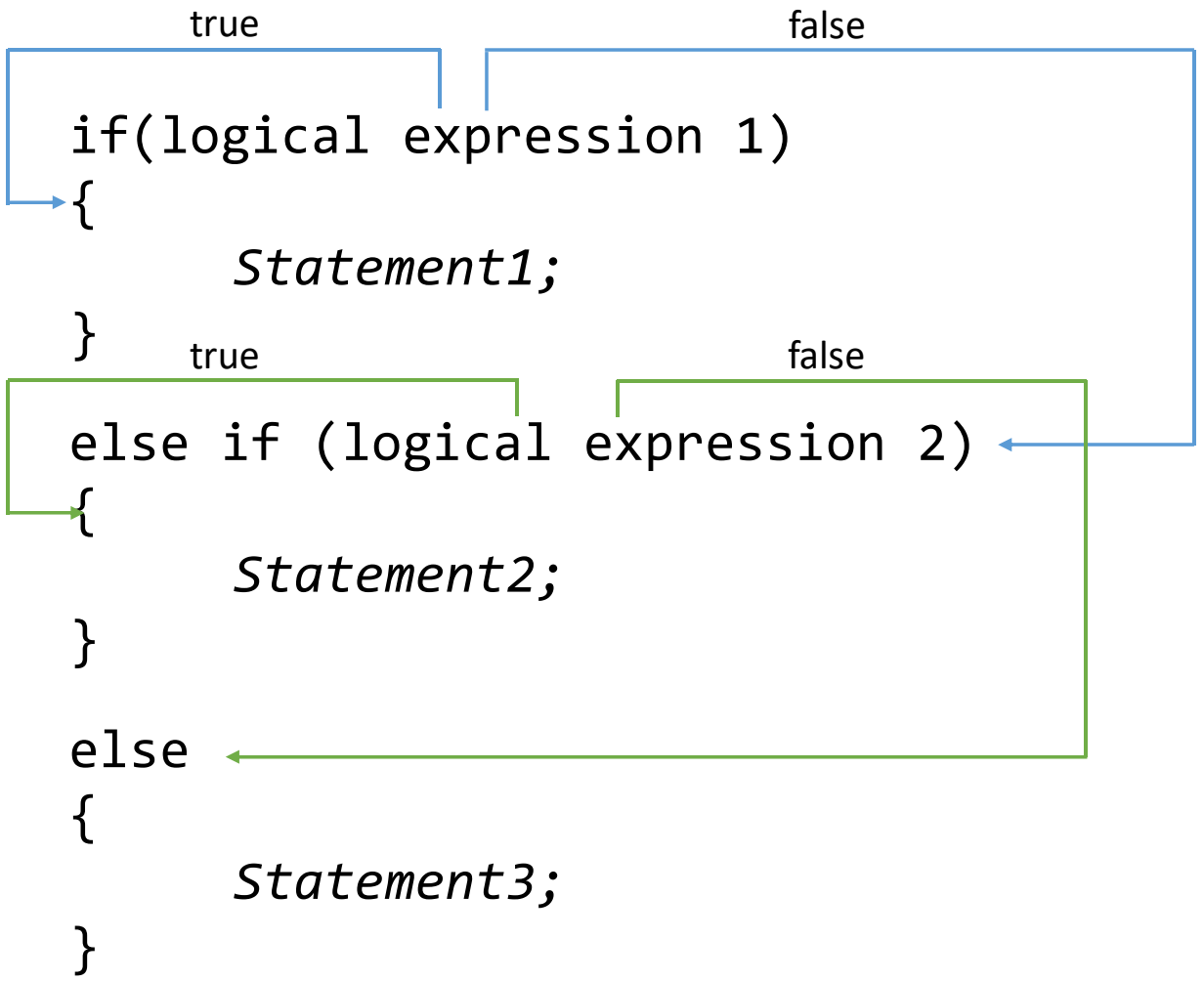
What if more statements are needed ?

```
if(logical expression)
      statement1;
      statement2;
else
      Statement3;
      Statement4;
```

Compiler error!

?

else-if



switch

```
switch(expression)
{
  case constant1:
    break;
  case constant2:
    break;
  case constant_n:
    break;
  default:
}
```

Break can be omitted → new meaning!

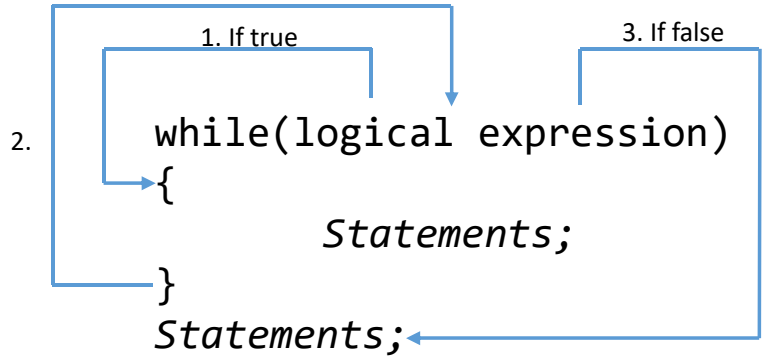
Loops

Loops

- Used for repeating execution.
- There need to be a logical expression telling when to stop repeating. If not, the loop will go on forever.
- while, for, do-while
- goto + labels
- Considered to be bad[1].
- Compare to ASM

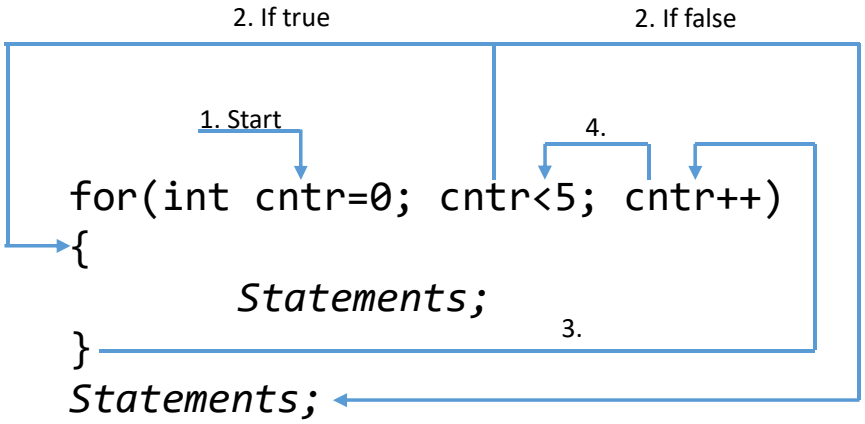
[1]See the classic paper “Go To Considered Harmful” by Edsger W. Dijkstra (http://www.u.arizona.edu/~rubinson/copyright_violations/Go_To_Considered_Harmful.html).

while



Executed zero or more times.

for



for (more formal)

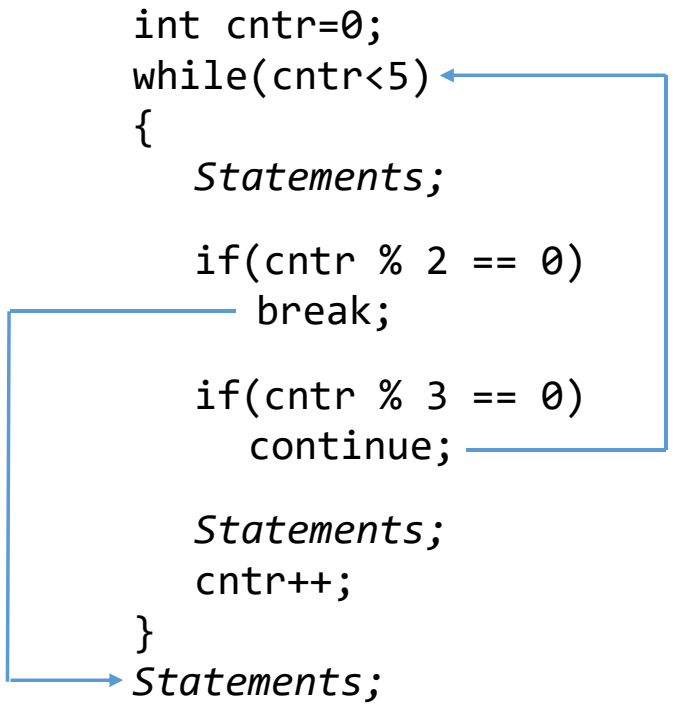
```
for(init expr;logical expr;loop expr)
{
        Statements;
}
Statements;
```

for vs while

```
for(int cntr=0; cntr<5; cntr++)
{
       Statements;
}
Statements;
```

```
int cntr=0;
while(cntr<5)
{
       Statements;
       cntr++;
}
Statements;
```

break/continue



Works with all loop constructs!

Pointers

Pointers

- Variable referring to a memory location.
- Useful when, for instance,
  - Allocating memory
  - Manipulating function arguments.
  - Pointing to different variables
- This is the same thing as in the Assembly language, but the notation is different!

Pointers

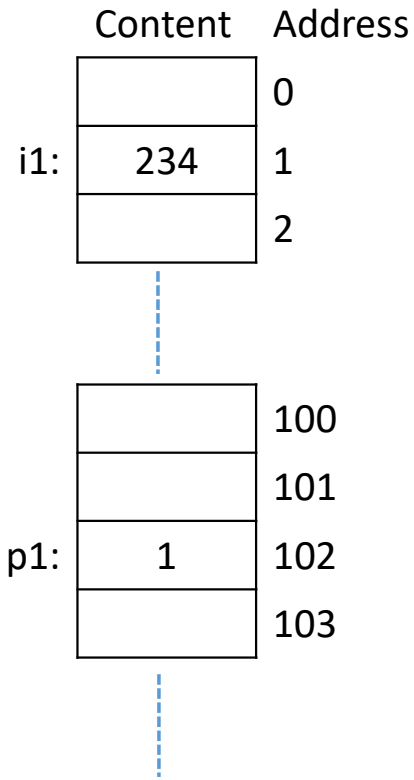
- Declaring
  - `type *name;`
- Getting an address of a variable
  - Put & character in front of the variable.
- A pointers type is important.

Pointers

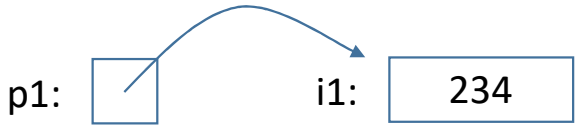
```
1. int i1 = 234;
2. int *p1 = NULL;
3.     p1 = &i1;
```

1. Declare a variable of type int.
2. Declare a pointer variable of type int.
3. Get the address of i1 and assign it to the pointer p1
   (meaning: p1 points at i1 so you can reference i1 through p1)

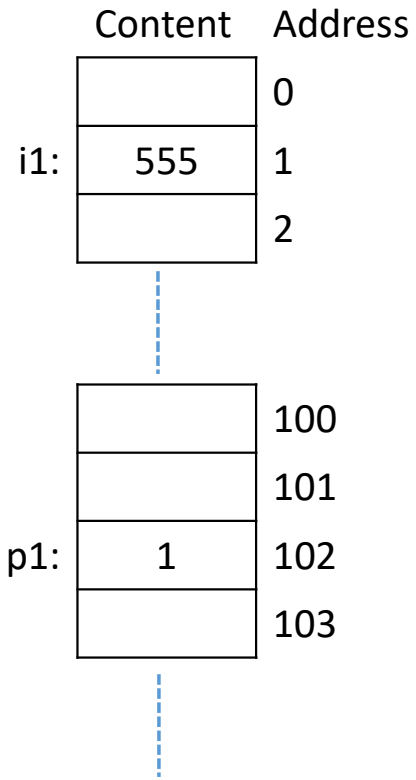
Pointer (memory layout)




```
int i1 = 234;
int *p1 = NULL;
    p1 = &i1;
```

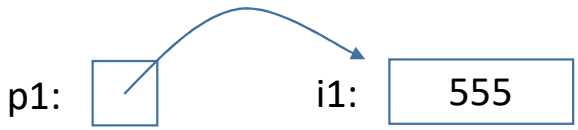


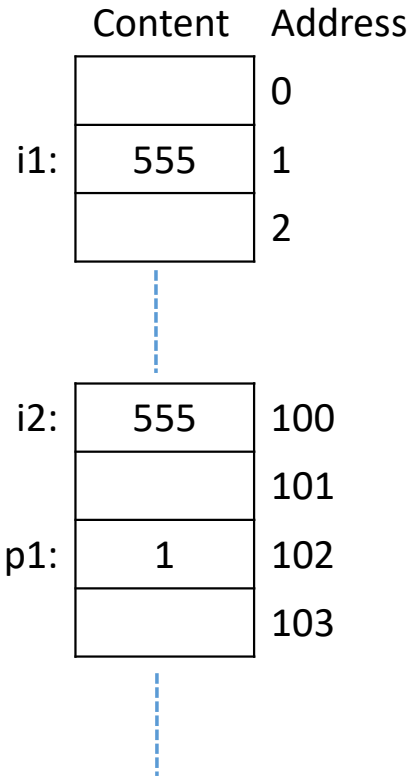
Dereferencing




```
*p1 = 555;
```

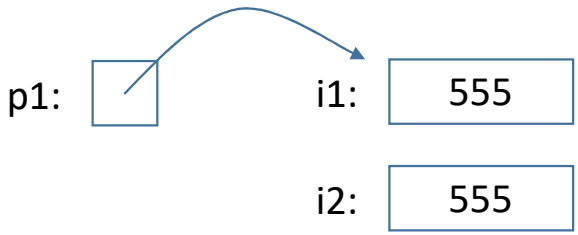
Read as: Wherever p1 points, assign it 555.



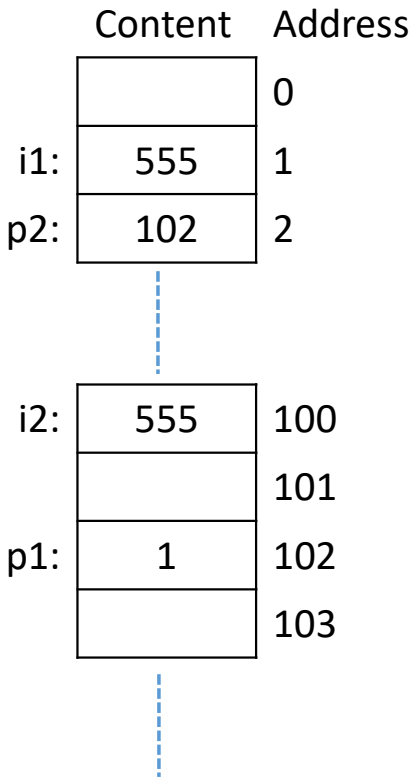



```
int i2 = *pval;
```

Read as: wherever p1 points get that value and assign it to i2.



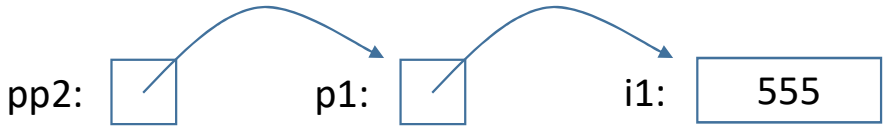
Pointer to a pointer



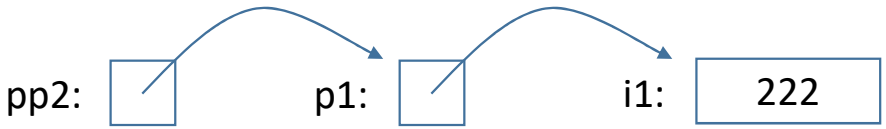

```
int **pp2 = &p1;
```

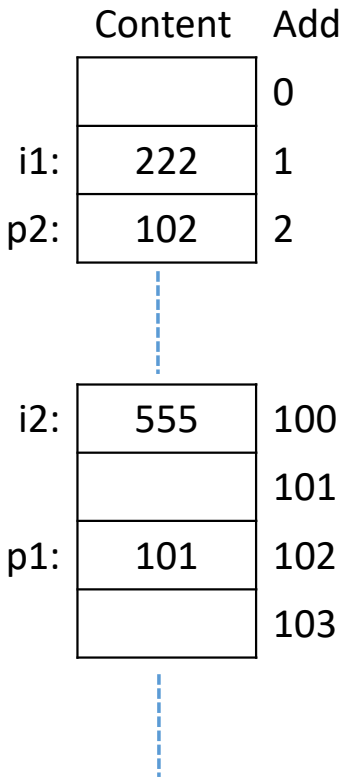
pp2 is a pointer to a pointer.

How are the pointers setup?



```
**pp2 = 222;
```






```
p1 = 101;
```

p1 is referencing memory location 101.

That C program again

Are they dereferenced pointers?

Yes!

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

int main(void)
{
  DDRB = 1 << DDB5;

  while (1)
  {
    PORTB = PORTB | (1 << PB5);
    _delay_ms(200);
    PORTB = PORTB & ~(1 << PB5);
    _delay_ms(200);
  }

  return 1;
}
```

Demystifying DDRB

Digging into **iom2560.h** and **sfr_defs.h** reveals

```
#define DDRB _SFR_IO8(0x04)
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define __SFR_OFFSET 0x20
```

So DDRB turns into `(*(volatile uint8_t *)(0x24))`, that is, a pointer dereferenced to memory address 0x24.

13.4.6 DDRB – Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

(From page 100 in 2549_ATmega2560.pdf (aka *the manual*))

So this explains PORTB and all other I/O memory registers.

2560 Data Memory Map

Address	Content
00-1F	32 Registers (R0-R31)
20-5F	64 I/O Registers (DDRB, PORTB and others are here)
60-1FF	416 External I/O Registers
200-21FF	SRAM
2200-	External SRAM

13.4.6 DDRB – Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

So…

Wherever DDRB points assign…

Wherever PORTB points….

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

int main(void)
{
  DDRB = 1 << DDB5;

  while (1)
  {
    PORTB = PORTB | (1 << PB5);
    _delay_ms(200);
    PORTB = PORTB & ~(1 << PB5);
    _delay_ms(200);
  }

  return 1;
}
```