An introduction to programming
In the beginning...
all we had was machine code

mem[0]=0x23; // load register a with following
mem[1]=0x00;
mem[2]=0xa8; // output a to r0
mem[3]=0x17; // increment a
mem[4]=0xa9; // output a to r1
mem[5]=0x17; // increment a
mem[6]=0xaa; // output a to r2
mem[7]=0x17; // increment a
mem[8]=0xab; // output a to r3
mem[9]=0x17; // increment a
mem[10]=0xac; // output a to r4
mem[11]=0x17; // increment a
mem[12]=0xad; // output a to r5
mem[13]=0x17; // increment a
mem[14]=0xae; // output a to r6
mem[15]=0x17; // increment a
mem[16]=0xaf; // output a to r7
mem[17]=0x17; // increment a
mem[18]=0x04; // jump to first page with following
mem[19]=0x02;
Then along came abstraction, with an assembly language to provide symbolic references for the numeric machine codes.

Main:
```
clrf PORTB ; initialize PORTB
bsf STATUS, RP0 ; Move to bank 1
movlw PORTB_DIR ; value for TRISB
movwf TRISB ; set by defined variable
bcf STATUS, RP0 ; Move to bank 1
movlw MAX_BITS
movwf BIT_COUNT ; sets the bit count to seven
clrf INPUT_BYTE
```

SSTestFall:
```
btfsc PORTB, SS_BIT ; check slave bit, if clear, skip next
goto SSSTestFall ; loop to check again
goto ClockTestFall ; move on
and now we have... compiled languages

Basic

C++

C

Cobol

Forth

Fortran

Java

C#

Ada

Pascal
```c
#include "m_general.h"  // custom macros

#define PIN 4

void init(void);

int x=100;

void main(void)
{
    int i;
    init();                    // initialize the system
    for(i=0; i<x; i++)         // do this 100 times
    {
        toggle(PORTE,PIN);    // toggle PIN
    }
}

void init(void)
{
    set(DDRE,PIN)            // PIN as output
}
```
C is case-sensitive!
white space does not matter
don’t forget the semicolon;
don’t forget the {} 
#define constants
declare variables before use
no magic numbers!

use subroutines
prototype your subroutines

compile and test as you go

comment your code!
please, comment your code...
translating common compiler errors

main.c: In function 'main':
main.c:17: error: 'i' undeclared (first use in this function)
main.c:17: error: (Each undeclared identifier is reported only once)
main.c:17: error: for each function it appears in.)
make: *** [main.o] Error 1

(undeclared variable)

main.c: In function 'main':
main.c:43: error: expected declaration or statement at end of input
make: *** [main.o] Error 1

(missing "}")

main.c: In function 'main':
main.c:19: error: expected ';' before '{}' token
make: *** [main.o] Error 1

(missing ";")

main.c: In function 'main':
main.c:25: warning: implicit declaration of function 'delay_ms'
main.o: In function `main':
main.o: In function `main':
main.c: (.text+0x88): undefined reference to `delay_ms'
make: *** [main.elf] Error 1

(missing subroutine)
preprocessor directives

directives processed before compilation

include other files (generally “header” files with other # defines, function prototypes, etc.)

```
#include <filename.h>        // file in the include path
#include “filename.h”        // file in the current directory
#include “m_general.h”
```

define constants (essentially a find & replace - no semicolon!)

```
#define CONSTANT value
#define ENC_LINES 1024
#define TRUE 1
```
functions and subroutines

functions must be prototyped - either with pre-processor directives, or in a separate header file (preferred for larger projects)

```c

int multiply(int x, int y);

```

functions must return according to the specified type

```c

int multiply(int x, int y);

int multiply(int x, int y){
    int i, answer=0;
    for(i=0;i<x;i++){
        answer += y;
    }
    return answer;
}

```
variables

variables must be declared before they are used!

type variable=initial, variable=initial;

int x;
short y, z;
long foo = 456;
unsigned int a=5, b=6;
char c = ‘b’;

<table>
<thead>
<tr>
<th>type</th>
<th>bits</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>int</td>
<td>16</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32</td>
<td>0</td>
<td>4294967295</td>
</tr>
<tr>
<td>float / double</td>
<td>IEEE32</td>
<td>1.175494E-38</td>
<td>3.402823E+38</td>
</tr>
</tbody>
</table>

ultimately, everything is binary to the CPU
float storage

32 bits

sign \( s \)  exponent \( e \)  mantissa \( m \)

\[ x = (-1)^s \times 2^{(e-127)} \times 1.m \]

floating-point math is

SSSSSSLLLLLLOOOOOOWWWW
### Basic Operators

#### Arithmetic

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td>%</td>
<td>modulus (remainder)</td>
</tr>
</tbody>
</table>

#### Conditional

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
</tbody>
</table>

#### Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>exclusive or</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
</tr>
<tr>
<td>~</td>
<td>one's complement</td>
</tr>
</tbody>
</table>

#### Unary

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>increment</td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
</tr>
</tbody>
</table>

#### Logical

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>not</td>
</tr>
</tbody>
</table>
operator precedence

higher operators will be applied first

parenthesis ( ) []
structure access . ->
unary ! ~ ++ -- - * &
multiply, divide, modulus * / %
add, subtract + -
bit shifts >> <<
inequality < <= >= >
equal, not equal == !=
bitwise AND &
bitwise exclusive OR ^
bitwise OR |
logical AND &&
logical OR ||
ternary conditional ? :
assignment = *= /= %= += -= <<= >>= &= |= ^=
comma ,

(when in doubt, add parentheses!)
iteration

**WHILE**: as long as the expression equals any non-zero value, the directives will be executed repeatedly

```java
while(expression){
    directives;
}
```

**FOR**: as long as the initialized variable is less than the termination value, the directives will be executed repeatedly

```java
int i=0;
for(i=0; i<10; i++){
    directives;
}
```

```java
while(!flag){
    directives;
}
```
**conditionals**

**IF:** if the expression equals any non-zero value, directives will be executed

```java
if(expression){
  expression ? directives : other directives;
  directives;
}
else {
  other directives;
}
```

expressions can be formed using:

<table>
<thead>
<tr>
<th>Conditional</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>!=</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>!</td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
</tr>
</tbody>
</table>

**examples**

- `if(a==b)`
- `if(a!=b)`
- `if(a<b)`
- `if(a&&b)`
- `if((a==5) && (b!=4))`
- `if(!c)`
- `(a==5)?___ : ___ ;`
**advanced conditionals**

**SWITCH**: as long as the expression equals any non-zero value, the directives will be executed repeatedly

```c
switch(variable){
    case condition:
        directives;
        break;
    case condition:
        directives;
        break;
    default:
        directives;
        break;
}
```

```c
int state=0;
while(1){
    switch(state){
        case 0:
            set(PORTD,4);
            state=2;
            break;
        case 1:
            clear(PORTD,4);
            state=0;
            break;
        default:
            state=1;
            break;
    }
}
```
type casting

implied (and often wrong)

```c
int a;
float b = 2.345;
a = b + 1;       // a = 3
```

("=" converts the result to the specified datatype AFTER the operation)

explicit

```c
int a = 2;
float b;
b = a/4;       // b = 0
b = (float)a/4;   // b = 0.5
b = a/4.0;      // b = 0.5
```

```c
unsigned int a = 61000;
unsigned int b = 10000;
long c;
c = a + b;       // c = 5465
// c = (long)a + b;  // c = 71000
```
variable type modifiers

to preserve the value of a variable between successive subroutine calls

```c
static
ISR(TIMER3_COMPA_vect)
{
    static long L_encoder_last=0, R_encoder_last=0;
    int L_velocity_raw, R_velocity_raw;

    // calculate velocity in 10 * ticks (current - previous) per 0.01 sec
    L_velocity_raw = -EGAIN*(L_encoder - L_encoder_last);
    L_encoder_last = L_encoder;
    L_velocity = (float)L_velocity*V_FILTER + (1-V_FILTER)*(float)L_velocity_raw;

    R_velocity_raw = -EGAIN*(R_encoder - R_encoder_last);
    R_encoder_last = R_encoder;
    R_velocity = (float)R_velocity*V_FILTER + (1-V_FILTER)*(float)R_velocity_raw;
}```
variable type modifiers

to alert the compiler that a variable may change outside the routine

volatile

```c
volatile int flag = 0;
char message[3] = {0, 0, 0};

int main(void){
    while(1){
        if(flag){
            toggle(PORTE, 6); // toggle the green LED
            flag = 0;
        }
    }
}

ISR(PCINT0_vect)
{
    if(!check(PINB, 5))
    {
        flag = 1;
        RFreceive(message);
    }
}
```