Programming in C
In the beginning...

there was no compiler... instead, things were written in machine code, which was directly executed by the CPU

```
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 which provided symbolic references for the numeric machine codes

Main:

```assembly
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
clf INPUT_BYTE
```

SSTestFall:

```assembly
btfs PORTB , SS_BIT ; check slave bit, if clear, skip next
goto SSTestFall ; loop to check again
goto ClockTestFall ; move on
```

this is still quite common today!
Now there are two varieties...

a Complex Instruction Set Computer (CISC) combines loading, manipulation, and storage tasks into more complex, processor intensive functions.

a Reduced Instruction Set Computer (RISC) either performs operations directly on registers, or loads and stores data to and from them, which is more memory and software intensive.
Basic program structure

- **Preprocessor directives**
  - `#include <avr/io.h>`  // common defines and macros
  - `#include "maevarm.h"` // custom macros

- **Subroutine prototypes**
  - `void init(void);`

- **Global variables**
  - `int x;`

- **Main function (local variables, directives)**
  - `void main(void){`
    - `int y = 0x03;`
    - `init();`  // initialize the system
    - `while(1){`
      - `set(PORTE,4);`  // set E4 high
    }
  }

- **Subroutine (local variables, directives)**
  - `void init(void){`
    - `set(DDRE,4)`  // E4 as output
  }
Some syntactical reminders

C is case-sensitive!

white space does not matter

don’t forget the semicolon;

comment your code – using either /* comment... */ or //

comment your code...

declare your variables
Preprocessor directives

directives processed before compilation

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

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

defining constants (essentially find & replace – no semicolon!)

```
#define CONSTANT VALUE
#define ENC_LINES 1024
#define EVER (;;)  
#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)

\begin{verbatim}
 type function(type variable, type variable);

    int multiply(int x, int y);

functions must return according to the specified type

    type function(type variable, type variable)
    {
        ...
        return variable;
    } \end{verbatim}
variables must be declared before they are used!

<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

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

32 bits

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

precision for float operations (+, -, *, /) is \(~7\) decimal digits

floating-point math is slow
## 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>multiple</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>

All arithmetic and bitwise operators can be used in assignments.
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!)
Conditional statements

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

```c
if(expression){
    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>not equal</td>
<td>and</td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>less than or equal</td>
<td>or</td>
</tr>
<tr>
<td>&gt;</td>
<td>!</td>
</tr>
<tr>
<td>greater than</td>
<td>not</td>
</tr>
</tbody>
</table>

expressions:

```c
if(a==b)
if(a!=b)
if(a<b)
if(a&&b)
if((a==5)&&(b!=4))
if(!c)
(a==5)?___ : ___ ;
```
Loops and iteration

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

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

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

```plaintext
for(initialization; termination; increment){
    directives;
}
```

```plaintext
int i=0;
while(i<10){
    i++;
    directives;
}
```

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

```plaintext
for(initialization; termination; increment){
    directives;
}
```

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