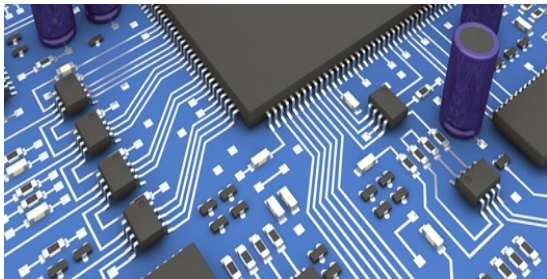


# Aplikace Embedded systémů v Mechatronice



Michal Bastl

# Organizace

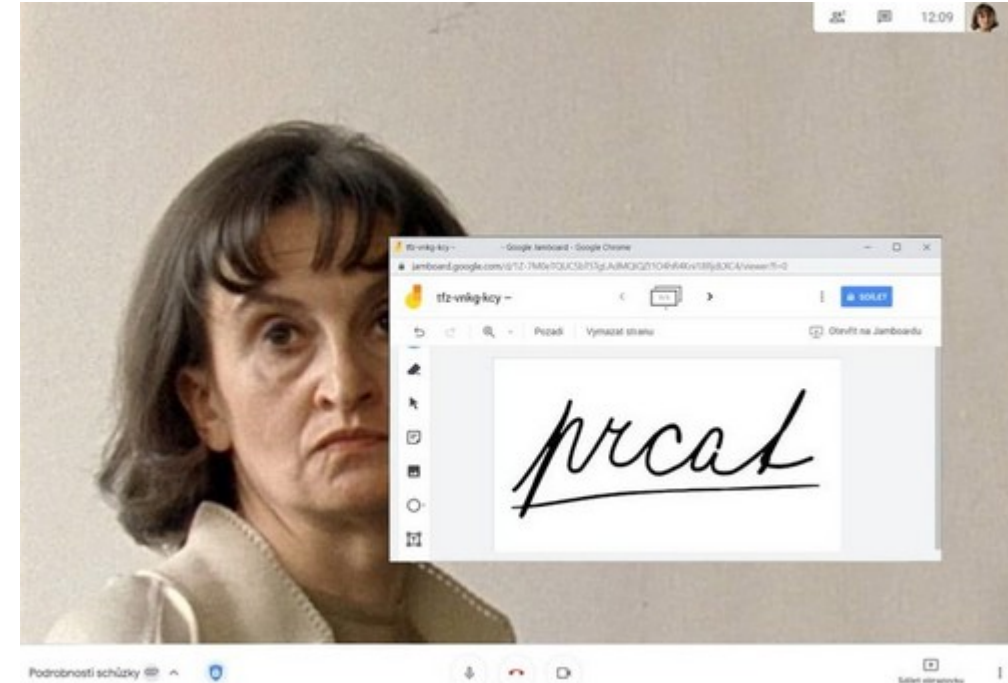
Přednáška: Nepovinná, ale...

Cvičení: Povinné

## Hodnocení předmětu 2021

- 3 domácí úlohy v průběhu semestru 3x5b
- Zápočtové testy 2x15b + 1x20b
- Semestrální projekt 35b

zdroje: github BUT-FME-REV



Velmi pravděpodobně  
distanční formou.

# REV kit

## Součásti:

- Krabice
- Programátor PICKIT3
- 2x USB kabel
- Edukit



# Hlavní rozdíly mezi MATLAB a C:

## C

- Kompilovaný jazyk
- Datové typy musí určit uživatel
- Přímý přístup do paměti
- málo klíčových slov
- Poměrně blízko hardwaru
- Oproti ASM, nebo .HEX je vzniklý kód čitelný (pokud byl programátor slušný : -)
- Je možné natropit chyby, které kompilátor nemůže ověřit
- Indexuje od 0

## MATLAB

- Interpretovaný jazyk
- Datové typy určí interpreter
- Uživatel nemá přímý přístup do paměti
- Mnoho funkcí a klíčových slov
- Propast mezi hardwarem a programátorem je větší
- Je dobře čitelný, vývoj je rychlejší
- Interpreter pracuje s pamětí sám. Nelze udělat zásadní chyby a jejich hledání je jednodušší
- Indexuje od 1



# Číselné soustavy

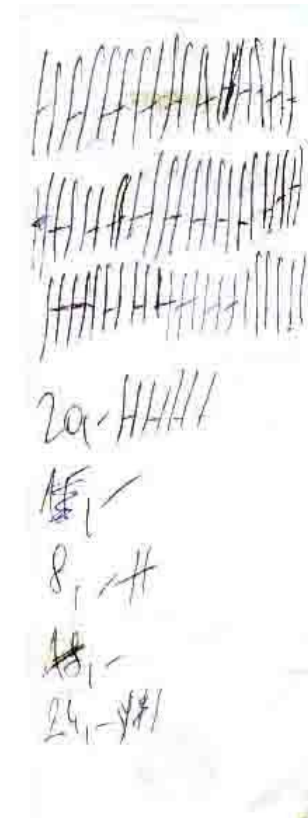
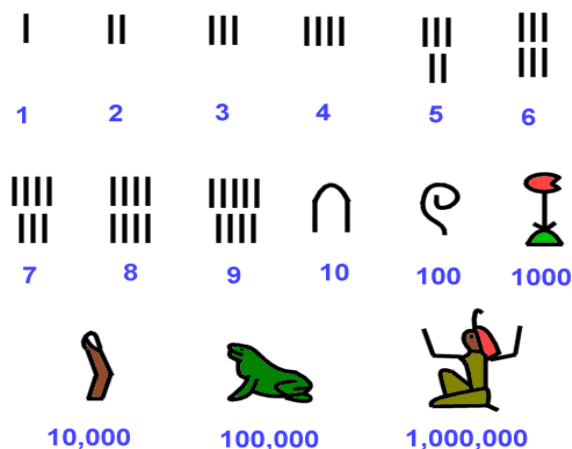
## Poziční:

Klíčovou charakteristikou pozičních soustav je jejich základ. To je obvykle přirozené číslo větší než jedna. Váhy jednotlivých číslic jsou pak mocninami tohoto základu.

Tomuto odpovídá klasická desítková soustava.

# Nepoziční

I II III IV V  
VI VII VIII IX X  
LCDM



# Číselné soustavy

Dekadická:	1052		$10^3 10^2 10^1 10^0$
Binární:	1010	zápis v C 0b1010	$2^3 2^2 2^1 2^0$
Hexadecimální:	A7	zápis v C 0xA7	$16^1 16^0$

Uvnitř MCU se používá pouze binární. Hexadecimální soustava se používá ke zkrácení zápisu.  
Jeden znak Hexadecimální soustavy odpovídá 4 bitům.

Například hodnota 200 v DEC je v BIN 0b11001000  
v HEX 0xC8

# Datové typy v jazyce C

Před použitím musím proměnnou deklarovat i s datovým typem.

Název proměnné nesmí začínat číslicí

Př:

```
char a, b, c;  
unsigned int i=0;  
long counter;
```

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

# Rozsah datových typů je závislý na platformě

Knihovna stdint.h:

uint8\_t, uint16\_t, uint32\_t  
int8\_t, int16\_t, int32\_t

Knihovna stdbool.h:

bool, true, false

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>

int main(void)
{
    uint8_t a,b,c;
    uint16_t d=35000;

    a = 10;
    b = 100;
    c = a + b;

    printf("V c je %d, v d je %d", c,d);

    return(0);
}
```

Ukázka



# Práce s proměnnými

- Programátor určuje datové typy, ale zároveň si musí hlídat jejich rozsah
- Proměnné je možné přetypovat (typ) var
- Povšimněte si různých specifikátorů ve funkci printf() %d, %f, %li...

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main()
{
    int a=10;
    int b=3;

    float c, f;

    c = a/b;
    f = (float)a/b;

    printf("c=%f, f=%f\n", c, f);

    int16_t d = 32768;

    uint16_t e = (uint16_t)d;

    printf("d=%d, e=%d", d, e);
    return 0;
}
```

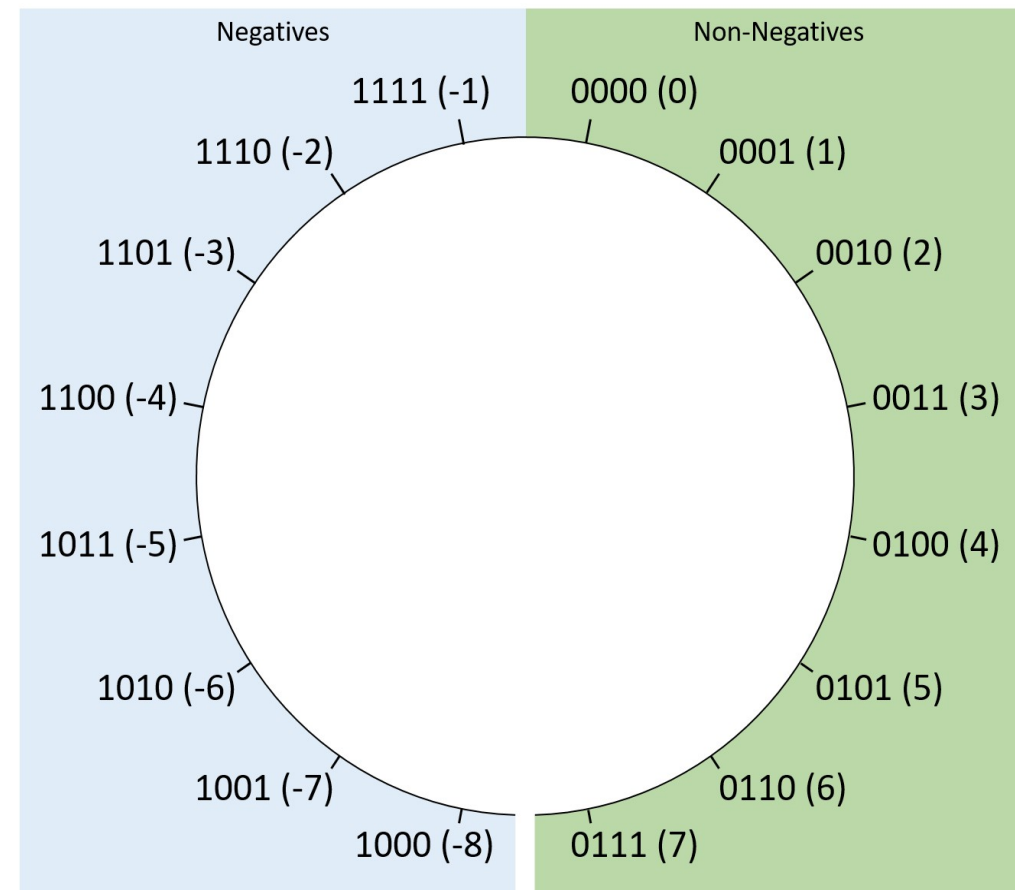
Ukázka

# Reprezentace čísel a dvojkový doplněk

- Realizace znaménka je nejčastěji pomocí tzv. dvojkového doplňku.
- Nevýhodou je asymetričnost (rozsah není pro záporná a nezáporná čísla stejný)

$$\begin{array}{r} 00000000000001010 \quad 10 \\ 11111111111110101 \quad \text{flip all bits} \\ + 00000000000000001 \quad \text{add 1} \\ \hline 11111111111110110 \quad -10 \end{array}$$

$$\begin{array}{r} 11111111111110110 \quad -10 \\ 00000000000001001 \quad \text{flip all bits} \\ + 00000000000000001 \quad \text{add 1} \\ \hline 00000000000001010 \quad 10 \end{array}$$



# operátory

## Využití:

Rychlé násobení a dělení >> bitový posuv:

$A \ll n$  (násobení  $2^n$ ) ;  $A \gg n$  (dělení  $2^n$ )

## Masky:

AND, OR a případně XOR

&		^
10011110	10011110	10011110
<u>00001111</u>	<u>00000001</u>	<u>00001111</u>
00001110	10011111	10010001

Symbol	Operator
&	bitwise AND
	bitwise inclusive OR
^	bitwise XOR (eXclusive OR)
<<	left shift
>>	right shift
~	bitwise NOT (one's complement) (unary)

# operator

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression <code>((c==5) &amp;&amp; (d&gt;5))</code> equals to 0.
	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression <code>((c==5)    (d&gt;5))</code> equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression <code>!(c==5)</code> equals to 0.

Operator	Meaning of Operator	Example
==	Equal to	<code>5 == 3</code> is evaluated to 0
>	Greater than	<code>5 &gt; 3</code> is evaluated to 1
<	Less than	<code>5 &lt; 3</code> is evaluated to 0
!=	Not equal to	<code>5 != 3</code> is evaluated to 1
>=	Greater than or equal to	<code>5 &gt;= 3</code> is evaluated to 1
<=	Less than or equal to	<code>5 &lt;= 3</code> is evaluated to 0

Operator	Example	Same as
=	<code>a = b</code>	<code>a = b</code>
+=	<code>a += b</code>	<code>a = a+b</code>
-=	<code>a -= b</code>	<code>a = a-b</code>
*=	<code>a *= b</code>	<code>a = a*b</code>
/=	<code>a /= b</code>	<code>a = a/b</code>
%=	<code>a %= b</code>	<code>a = a%b</code>



# ASCII tabulka

- Zápis v jazyce C pomocí jednoduchých uvozovek '0'
- Používáme datový typ char 0..255
- V původním rozsahu má 128 znaků (rozšířená pak 255)
- Pro více znaků se používá pole char[x]

Příklad:

- Chci vypisovat číslce 0..9
- ```
char c = 2;
```

Knihovna stdio.h:

```
printf("cislice je %c", c + '0');
```

| Dec | Hx | Oct | Char                        | Dec | Hx | Oct | Html  | Chr   | Dec | Hx | Oct | Html  | Chr | Dec | Hx | Oct | Html   | Chr |
|-----|----|-----|-----------------------------|-----|----|-----|-------|-------|-----|----|-----|-------|-----|-----|----|-----|--------|-----|
| 0   | 0  | 000 | NUL (null)                  | 32  | 20 | 040 | &#32; | Space | 64  | 40 | 100 | &#64; | @   | 96  | 60 | 140 | &#96;  | `   |
| 1   | 1  | 001 | SOH (start of heading)      | 33  | 21 | 041 | &#33; | !     | 65  | 41 | 101 | &#65; | A   | 97  | 61 | 141 | &#97;  | a   |
| 2   | 2  | 002 | STX (start of text)         | 34  | 22 | 042 | &#34; | "     | 66  | 42 | 102 | &#66; | B   | 98  | 62 | 142 | &#98;  | b   |
| 3   | 3  | 003 | ETX (end of text)           | 35  | 23 | 043 | &#35; | #     | 67  | 43 | 103 | &#67; | C   | 99  | 63 | 143 | &#99;  | c   |
| 4   | 4  | 004 | EOT (end of transmission)   | 36  | 24 | 044 | &#36; | \$    | 68  | 44 | 104 | &#68; | D   | 100 | 64 | 144 | &#100; | d   |
| 5   | 5  | 005 | ENQ (enquiry)               | 37  | 25 | 045 | &#37; | %     | 69  | 45 | 105 | &#69; | E   | 101 | 65 | 145 | &#101; | e   |
| 6   | 6  | 006 | ACK (acknowledge)           | 38  | 26 | 046 | &#38; | &     | 70  | 46 | 106 | &#70; | F   | 102 | 66 | 146 | &#102; | f   |
| 7   | 7  | 007 | BEL (bell)                  | 39  | 27 | 047 | &#39; | '     | 71  | 47 | 107 | &#71; | G   | 103 | 67 | 147 | &#103; | g   |
| 8   | 8  | 010 | BS (backspace)              | 40  | 28 | 050 | &#40; | (     | 72  | 48 | 110 | &#72; | H   | 104 | 68 | 150 | &#104; | h   |
| 9   | 9  | 011 | TAB (horizontal tab)        | 41  | 29 | 051 | &#41; | )     | 73  | 49 | 111 | &#73; | I   | 105 | 69 | 151 | &#105; | i   |
| 10  | A  | 012 | LF (NL line feed, new line) | 42  | 2A | 052 | &#42; | *     | 74  | 4A | 112 | &#74; | J   | 106 | 6A | 152 | &#106; | j   |
| 11  | B  | 013 | VT (vertical tab)           | 43  | 2B | 053 | &#43; | +     | 75  | 4B | 113 | &#75; | K   | 107 | 6B | 153 | &#107; | k   |
| 12  | C  | 014 | FF (NP form feed, new page) | 44  | 2C | 054 | &#44; | ,     | 76  | 4C | 114 | &#76; | L   | 108 | 6C | 154 | &#108; | l   |
| 13  | D  | 015 | CR (carriage return)        | 45  | 2D | 055 | &#45; | -     | 77  | 4D | 115 | &#77; | M   | 109 | 6D | 155 | &#109; | m   |
| 14  | E  | 016 | SO (shift out)              | 46  | 2E | 056 | &#46; | .     | 78  | 4E | 116 | &#78; | N   | 110 | 6E | 156 | &#110; | n   |
| 15  | F  | 017 | SI (shift in)               | 47  | 2F | 057 | &#47; | /     | 79  | 4F | 117 | &#79; | O   | 111 | 6F | 157 | &#111; | o   |
| 16  | 10 | 020 | DLE (data link escape)      | 48  | 30 | 060 | &#48; | 0     | 80  | 50 | 120 | &#80; | P   | 112 | 70 | 160 | &#112; | p   |
| 17  | 11 | 021 | DC1 (device control 1)      | 49  | 31 | 061 | &#49; | 1     | 81  | 51 | 121 | &#81; | Q   | 113 | 71 | 161 | &#113; | q   |
| 18  | 12 | 022 | DC2 (device control 2)      | 50  | 32 | 062 | &#50; | 2     | 82  | 52 | 122 | &#82; | R   | 114 | 72 | 162 | &#114; | r   |
| 19  | 13 | 023 | DC3 (device control 3)      | 51  | 33 | 063 | &#51; | 3     | 83  | 53 | 123 | &#83; | S   | 115 | 73 | 163 | &#115; | s   |
| 20  | 14 | 024 | DC4 (device control 4)      | 52  | 34 | 064 | &#52; | 4     | 84  | 54 | 124 | &#84; | T   | 116 | 74 | 164 | &#116; | t   |
| 21  | 15 | 025 | NAK (negative acknowledge)  | 53  | 35 | 065 | &#53; | 5     | 85  | 55 | 125 | &#85; | U   | 117 | 75 | 165 | &#117; | u   |
| 22  | 16 | 026 | SYN (synchronous idle)      | 54  | 36 | 066 | &#54; | 6     | 86  | 56 | 126 | &#86; | V   | 118 | 76 | 166 | &#118; | v   |
| 23  | 17 | 027 | ETB (end of trans. block)   | 55  | 37 | 067 | &#55; | 7     | 87  | 57 | 127 | &#87; | W   | 119 | 77 | 167 | &#119; | w   |
| 24  | 18 | 030 | CAN (cancel)                | 56  | 38 | 070 | &#56; | 8     | 88  | 58 | 130 | &#88; | X   | 120 | 78 | 170 | &#120; | x   |
| 25  | 19 | 031 | EM (end of medium)          | 57  | 39 | 071 | &#57; | 9     | 89  | 59 | 131 | &#89; | Y   | 121 | 79 | 171 | &#121; | y   |
| 26  | 1A | 032 | SUB (substitute)            | 58  | 3A | 072 | &#58; | :     | 90  | 5A | 132 | &#90; | Z   | 122 | 7A | 172 | &#122; | z   |
| 27  | 1B | 033 | ESC (escape)                | 59  | 3B | 073 | &#59; | ;     | 91  | 5B | 133 | &#91; | [   | 123 | 7B | 173 | &#123; | {   |
| 28  | 1C | 034 | FS (file separator)         | 60  | 3C | 074 | &#60; | <     | 92  | 5C | 134 | &#92; | \   | 124 | 7C | 174 | &#124; |     |
| 29  | 1D | 035 | GS (group separator)        | 61  | 3D | 075 | &#61; | =     | 93  | 5D | 135 | &#93; | ]   | 125 | 7D | 175 | &#125; | }   |
| 30  | 1E | 036 | RS (record separator)       | 62  | 3E | 076 | &#62; | >     | 94  | 5E | 136 | &#94; | ^   | 126 | 7E | 176 | &#126; | ~   |
| 31  | 1F | 037 | US (unit separator)         | 63  | 3F | 077 | &#63; | ?     | 95  | 5F | 137 | &#95; | _   | 127 | 7F | 177 | &#127; | DEL |

# podminky if..else

```
//příklad podmínky  
if...else
```

```
int a = 10;  
  
if (a == 10){  
    //function1  
}  
else{  
    //function2  
}
```

```
//příklad podmínky  
if...else
```

```
int a = 10;  
  
if (a == 10){  
    //function1  
}  
else if(a < 10){  
    //function2  
}  
else{  
    //function3  
}
```



# switch

- Program se řídí jednou proměnnou
- Pozor na ukončení příkazem break. Jinak by se vyhodnocovalo dále
- Default není nutně povinná
- Řídící proměnná nemůže být float nebo double

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char c;
    printf("Zadej znak:");
    c = getche();
    printf("\n");

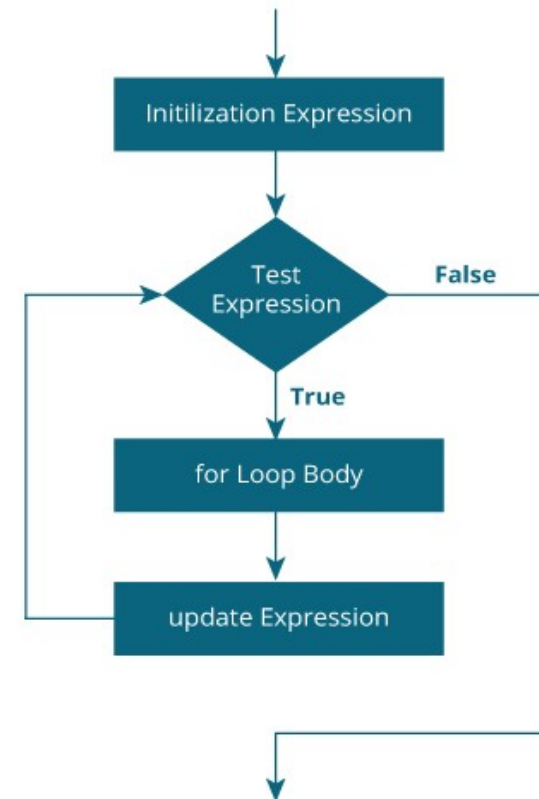
    switch (c)
    {
        case 'A':
            printf("Zadals A");
            break;
        case 'B':
            printf("Zadals B");
            break;
        case 'C':
            printf("Zadals C");
            break;
        default:
            break;
    }
    return 0;
}
```

# for smyčka

//příklad for smyčky

```
int i;  
  
for (i=0; i < 10; i++){  
    printf( "Ahoj svete" );  
}
```

For smyčka se používá tehdy, kdy znám dopředu počet cyklů, které chci provést.



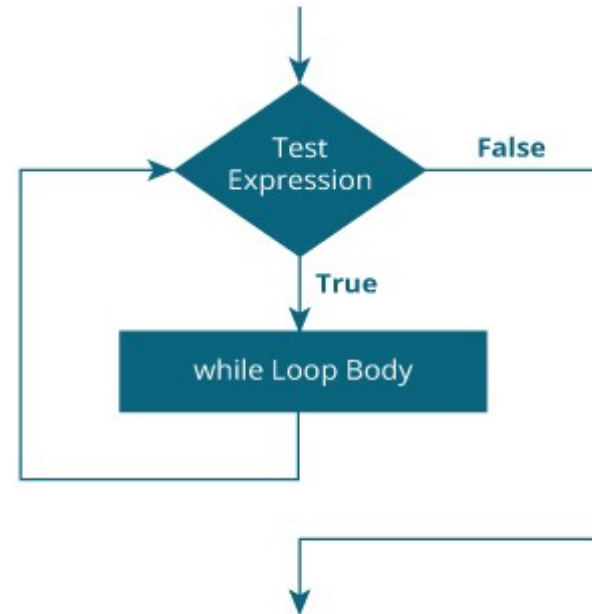
# while smyčka

//příklad while  
smyčky

```
int a = 10;
```

```
while(a <= 100){  
    a = a * 10;  
}
```

While smyčka funguje jinak, jednoduše opakuje blok programu v jeho těle dokud platí podmínka



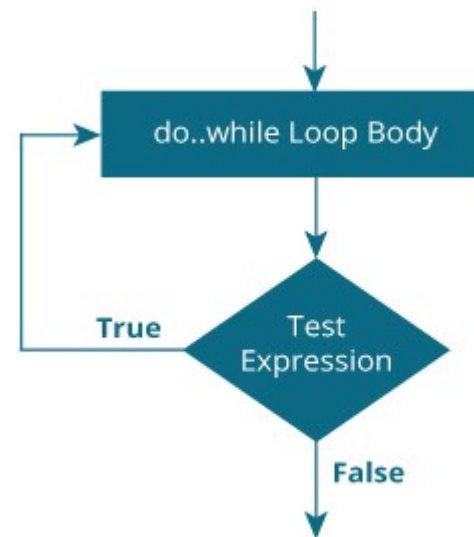
# do-while smyčka

- Stejně jako while, jen se provede program a teprve potom se testuje podmínka
- Provede se tedy min. jednou

```
#include <stdio.h>
#include <stdlib.h>


int main()
{
    do{
        printf("Provede se!");
    } while(0);

    return 0;
}
```




# break, continue

```
while(test){  
  // dalsi kod  
  if(vyraz){  
    break;  
  }  
  // dalsi kod  
}
```



A diagram illustrating the effect of the `break` statement. A black line starts from the right side of the `break;` statement, extends horizontally to the right, then vertically downwards, and finally horizontally to the left, ending with an arrowhead pointing to the left side of the closing curly brace of the `while` loop, indicating an immediate exit from the loop.

```
while(test){  
  // dalsi kod  
  if(vyraz){  
    continue;  
  }  
  // dalsi kod  
}
```



A diagram illustrating the effect of the `continue` statement. A black line starts from the right side of the `continue;` statement, extends horizontally to the right, then vertically upwards, and finally horizontally to the left, ending with an arrowhead pointing to the left side of the opening curly brace of the `while` loop, indicating that the rest of the current iteration is skipped and the loop starts over.

# Funkce v C

- Funkce nemusí vracet hodnotu slovo void, jinak může vracet datové typy např. int, char apod... klíčové slovo return.
- Funkce může přijímat parametry (int a, char, b...) nebo žádné nemá (void)
- Překladač před použitím musí funkci „znát“, případně ví, že funkce někde existuje. Je definovaná .
- Používáme prototyp funkce před prvním použitím. Deklarujeme ji.

```
void putchar(char c);      // prototyp pozor na ;
```

```
//definice funkce
```

```
void putchar(char c){  
    bufferToSend = c;  
}
```



# Čtyři verze funkce

Funkce nic nevrací a nepřijímá:

```
void fun(void)
```

Funkce nic nevrací a přijímá parametry:

```
void fun(int a, int b)
```

Funkce vrací a nepřijímá parametry:

```
int fun(void)
```

Funkce vrací a přijímá parametry:

```
char fun(char a, int b)
```

# Funkce v C

## Deklarace vs. Definice

```
//deklarace (prototyp)
int soucet(int a, int b);
//main
int main(void){
    int c;
    c = soucet(10, 5);
    return 0;
}
//definice
int soucet(int a, int b){
    return a + b;
}
```

```
//definice
int soucet(int a, int b){
    return a + b;
}
//main
int main(void){
    int c;
    c = soucet(10, 5);
    return 0;
}
```

# Ukázka v C

```
#include <stdio.h>

// prototypy
void tisk(void);
void tisk_cisla(int cislo);

int main(void){
    char cislo;

    tisk();                //pouziti funkce

    printf("Zadej cislo:");
    cislo = getche() - '0';
    printf("\n");
    tisk_cisla(cislo);      //pouziti funkce

    return 0;
}

// definice
void tisk_cisla(int a){
    printf("Cislo je %d\n", a);
}

//definice
void tisk(void){
    printf("Tisk z funkce\n");
}
```

```
#include <stdio.h>

// prototypy
int smysl_zivota(void);
int pricti_deset(int cislo);

int main(void){
    char cislo;

    printf("Smysl zivota: %d\n", smysl_zivota());
    printf("Zadej cislo:");
    cislo = getche() - '0';
    printf("\n");
    printf("Cislo +10 je: %d\n", pricti_deset(cislo));

    return 0;
}

// definice
int smysl_zivota(void){
    return 42;
}

//definice
int pricti_deset(int cislo){
    return cislo + 10;
}
```

# Rekurze

- Rekurze znamená volání sebe sama v těle funkce
- Pomocí rekurze lze jednoduše vyřešit určité úlohy
- Některé problémy jsou z podstaty rekurzivní součet čísel od 1..n, faktoriál apod.
- Rekurze vede k efektivnímu zápisu, ale je náročnější na zdroje
- Stack overflow
- Pozor na podmínku ukončení rekurze

```
#include <stdio.h>

// prototypy
int sum(int n);

int main(void){
    char cislo;
    int suma;

    printf("Zadej cislo:");
    cislo = getch() - '0';
    printf("\n");
    suma = sum(cislo);    //pouziti funkce

    printf("Suma je: %d", suma);
    return 0;
}

int sum(int n){
    if(n != 0){
        return n + sum(n-1);
    }
    return 0;
}
```

```
int main() {
    ... ..
    result = sum(number);
    ... ..
}

int sum(int n) {
    if (n != 0)
        return n + sum(n-1);
    else
        return n;
}

int sum(int n) {
    if (n != 0)
        return n + sum(n-1);
    else
        return n;
}

int sum(int n) {
    if (n != 0)
        return n + sum(n-1);
    else
        return n;
}

int sum(int n) {
    if (n != 0)
        return n + sum(n-1);
    else
        return n;
}
```

3+3 = 6  
is returned

2+1 = 3  
is returned

1+0 = 1  
is returned

0  
is returned

Ukázka

# Platnost proměnných

- Proměnné v C mají určitou platnost
- Proměnné použité ve funkci existují pouze při volání funkce jsou lokální
- Při volání funkce se hodnota proměnné zkopíruje
- Globální proměnnou zavedu někde v globálním prostoru vně funkce main()
- Globální proměnné bych neměl nadužívat
- V příkladu je  $a = 10$  pouze v těle funkce

```
#include <stdio.h>

//prototyp
void moje_funkce(int a);

int main(void){

    int a=0;

    moje_funkce(a);
    printf("a je: %d\n", a);

    return 0;
}

// definice
void moje_funkce(int a){
    a = a + 10;
    printf("a je: %d\n", a);
}
```

# Platnost proměnných

- Proměnné v C mají určitou platnost
- Proměnné použité ve funkci existují pouze při volání funkce jsou lokální
- Při volání funkce se hodnota proměnné zkopíruje
- Globální proměnnou zavedu někde v globálním prostoru vně funkce main()
- Globální proměnné bych neměl nadužívat
- V příkladu bude hodnota a = 10 jak ve funkci, tak v main

```
#include <stdio.h>

// globalni proměnná
int g_a;
//prototyp
void moje_funkce(void);

int main(void){

    moje_funkce();
    printf("a je: %d\n", g_a);

    return 0;
}

// definice
void moje_funkce(void){
    g_a += 10;
    printf("a je: %d\n", g_a);
}
```



# Proměnné mohu zakládat i v telě funkce

- V tomto případě proměnná sum přestane existovat po opoštění funkce
- Pokud chci proměnnou zachovávat musím použít klíčové slovo static
- Proměnná static je vlastně globální proměnná (existuje po celou dobu programu)
- Nemohu k ní však přistupovat z dalších funkcí

```
#include <stdio.h>

// prototyp
int suma(int a, int b, int c);

int main()
{
    int a=1, b=5, c=10;

    printf("Suma1 je: %d\n", suma(a, b, c));

    printf("Suma2 je: %d\n", suma(a, b, c));

    printf("Suma3 je: %d\n", suma(a, b, c));

    return 0;
}

int suma(int a, int b, int c){
    static int pocet=0;
    int sum;

    sum = a + b + c + pocet;
    pocet++;
    return sum;
}
```

# Globální/lokální/const

- V tomto případě proměnná sum přestane existovat po opoštění funkce
- Pokud chci proměnnou zachovávat musím použít klíčové slovo static
- Proměnná static je vlastně globální proměnná (existuje po celou dobu programu)
- Nemohu k ní však přistupovat z dalších funkcí

```
#include <stdio.h>

// prototyp
int suma(int a, int b, int c);

int main()
{
    int a=1, b=5, c=10;

    printf("Suma1 je: %d\n", suma(a, b, c));

    printf("Suma2 je: %d\n", suma(a, b, c));

    printf("Suma3 je: %d\n", suma(a, b, c));

    return 0;
}

int suma(int a, int b, int c){
    static int pocet=0;
    int sum;

    sum = a + b + c + pocet;
    pocet++;
    return sum;
}
```

# Překlad kódu

- Preprocessing

Základní příprava zpracuje direktivy preprocesoru a odstraní komentáře.

- Compiling

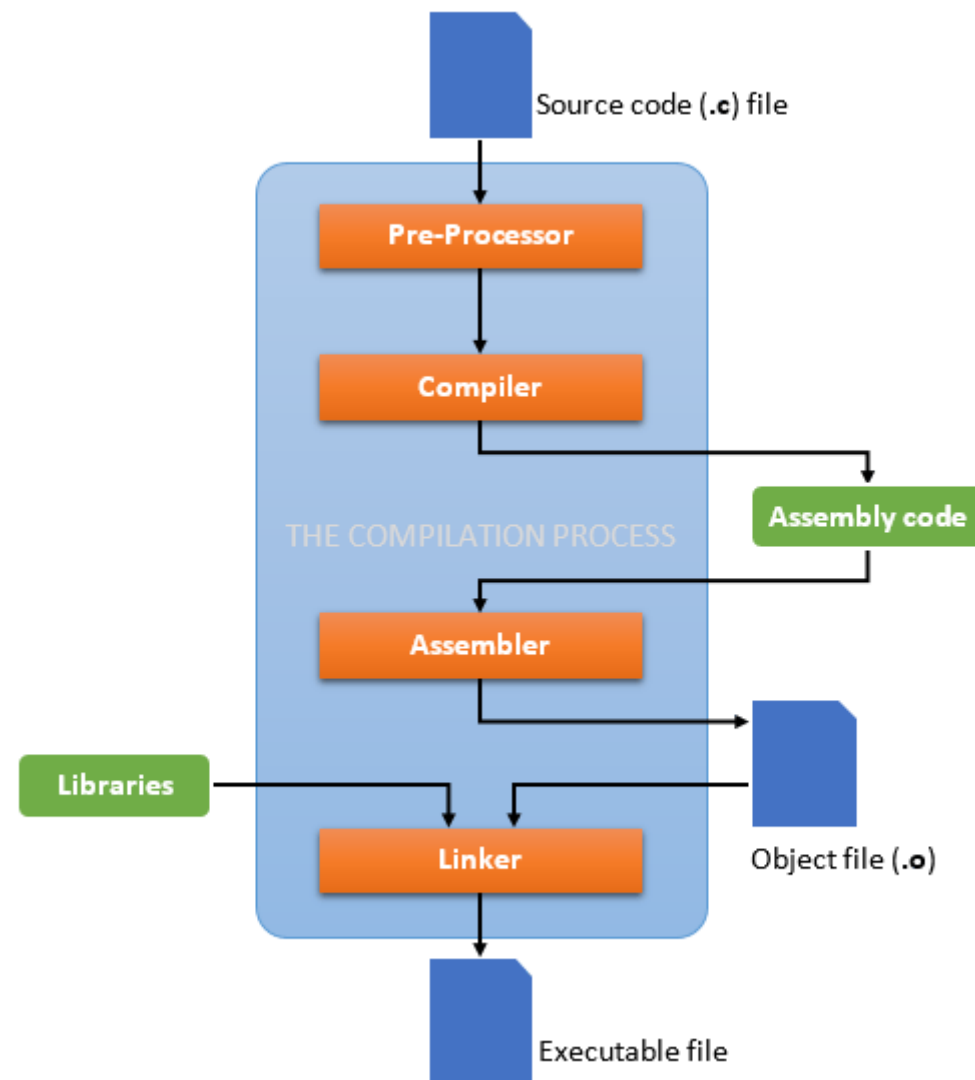
Vezme výstup preprocesoru a převede C na assembly.

- Assembly

Sestaví strojový kód tzv object .o kód

- Linking

Linker sestaví vše object kódy knihovny do jednoho. Určí jak vše bude v paměti. Ovlivňuje Linker file.



# Preprocesor makra

```
#define MAX 1000
#define PI 3.14159
#define TWO_PI (2 * PI)
#define AND &&
```

```
#include <stdio.h>
#include <stdlib.h>

#define PI 3.1415F
#define ADD(x,y) (x + y)

int main()
{
    int polomer, a, b;

    printf("Vloz cislo:");
    polomer = (int)(getche() - '0');
    printf("\n");
    float obsah = PI*polomer*polomer;
    printf("Obsah je: %f\n", obsah);

    printf("Vloz cislo a:");
    a = (int)(getche() - '0');
    printf("\n");

    printf("Vloz cislo b:");
    b = (int)(getche() - '0');
    printf("\n");

    printf("Soucet a+b je:%d", ADD(a,b));

    return 0;
}
```

# Podmíněný překlad

- Celé celky kódu mohu z překladu vyloučit preprocesorem

```
#ifndef  
#if  
#endif  
#endif
```

```
#include <stdio.h>
```

```
#define NOT_IMPLMENTED 0
```

```
int main(void)  
{  
#if NOT_IMPLMENTED  
    printf("Nevytisknes\n\r");  
#endif  
  
    return 0;  
}
```

# Vytvoření knihovny

K vytvoření knihovny potřebuji tzv. hlavičkový soubor a skript, kde mám své funkce, případně datové typy atd...

- Vytvoříme knihovnu a zavedeme funkce pro součet a odečet dvou celočíselných proměnných.
- `#ifndef` zabraňuje vícenásobnému vložení téhož kódu. Prostředí Vám doplní do .h souboru automaticky

Soubor MyMath.h

```
#ifndef MYMATH_H
#define MYMATH_H

int soucet(int a, int b);
int odecet(int a, int b);

#endif
```

Soubor MyMath.c

```
#include "MyMath.h"

int soucet(int a, int b){
    return a + b;
}

int odecet(int a, int b){
    return a - b;
}
```



# Kompilace pomocí GCC

- `gcc -E main.c` proběhne pouze preprocesor
- `gcc -S main.c` assembler
- `gcc -c main.c` object code
- `gcc main.c` komplet i s linkerem na `w10 a.exe`
- `gcc main.c -o program` vznikne `program.exe`

# Make

```
# Projekt:   Pokus komentar
```

```
CC=gcc
```

```
CFLAGS= -std=c99
```

```
main: main.c
```

```
    $(CC) $(CFLAGS) main.c lib/common.c -o program
```

Pak jen volám „make“