

# Aplikace Embedded systémů v Mechatronice



Michal Bastl

# Opakování

- Vytvoření projektu
  - Kompilace a nahrání
  - Debugging
  - Základní nastavení kompilátoru
- 
- Soustavy
  - Binární operátory

# 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(){
    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(){
    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 = getch() - '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 = getch() - '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>

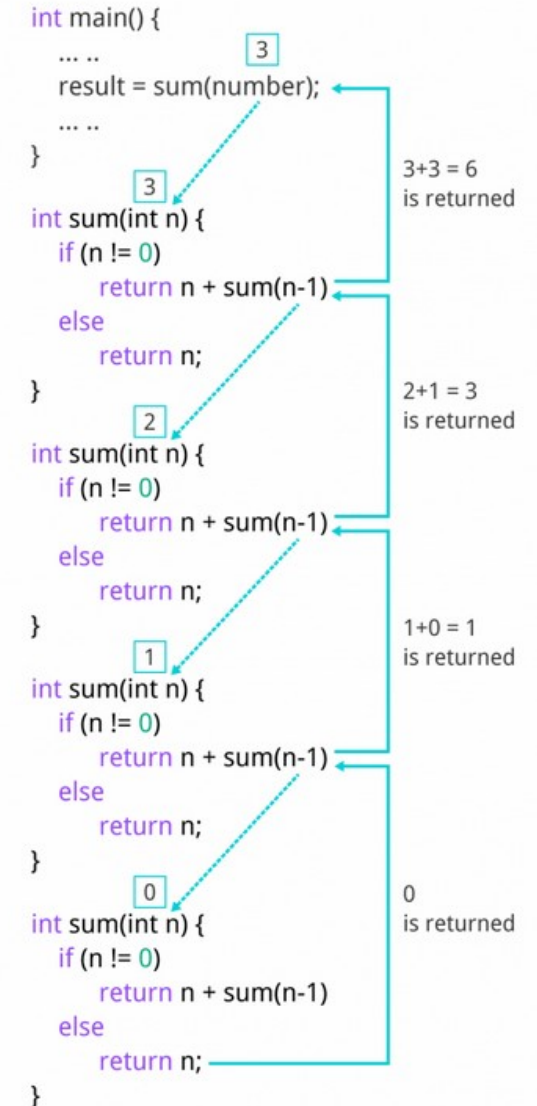
// prototypy
int sum(int n);

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;
}
```



# 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;
}
```

# 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;
}
```

# Pointery/ukazatele

- Ukazatele v jazyce C slouží k přístupu do paměti a manipulaci s adresou.
- Celá věc v C funguje tak, že existují speciální proměnné, které uchovávají adresu v paměti.
- V C můžete pointer vytvořit příkazem `typ* proměnná`
- právě znak `*` určuje, že se bude jednat o ukazatel na příslušný datový typ
- pokud chci získat adresu proměnné používám referenční operátor `&`
- dereferenční operátor `*` slouží k získání hodnoty uložené na adrese

ADD	VAL
0x00	10
0xff	0x00



# Pointery/ukazatele

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

```
int main()
```

```
{ int c;
```

```
  int* p_c;
```

```
  c = 10;
```

```
  p_c = &c;
```

```
  printf("Na adrese 0x%p je hodnota%d\n",p_c,*p_c);
```

```
  return 0;
```

```
}
```

operátor reference &c vrací adresu paměti

operátor dereference \*p\_c vrací hodnotu uloženou na adrese  
symbol \*p\_c slouží současně pro deklaraci pointeru

>>Na adrese 0x0060FF08 je hodnota 10

# Pointery-předání funkci

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

void prohod(int* a, int* b);

int main(){
    int jedna;
    int dva;


    jedna = 1;
    dva = 2;
    prohod(&jedna, &dva);
    printf("jedna = %d; dva = %d\n", jedna, dva);
    return 0;
}

void prohod(int* a, int* b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

>>jedna = 2; dva = 1

operátor reference &c vrací adresu paměti  
operátor dereference \*p\_c vrací hodnotu  
uloženou na adrese  
symbol \*p\_c slouží současně pro deklaraci  
pointeru

```
void prohod(int a, int b){
    int tmp = a;
    a = b;
    b = tmp;
}
```

 NEFUNGUJE!!

# Pointer vs. pole

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

void uloz_do_pole(int pole[], int index, int cislo);

int main() {

    int cisla[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    printf("%d\n", cisla[7]);

    uloz_do_pole(cisla, 7, 3);

    printf("%d\n", cisla[7]);

    return 0;
}

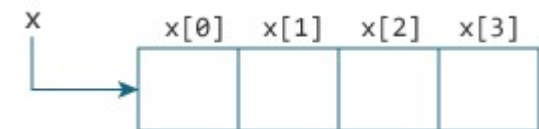
void uloz_do_pole(int pole[], int index, int cislo){
    pole[index] = cislo;
}

>>7
>>3
```

Pole a pointery spolu v C souvisí. Pokud předám funkci pole, provádím to vždy referencí. Proto změny, které ve funkci provedu, v poli zůstanou zachovány. Toto předání referencí proběhne u pole vždy.

Pole v C je ukazatel na místo v paměti, kde pole začíná.

Proto:  
cisla[1] a \*cisla + 1 vrací stejný výsledek



# Řetězce = pole znaků

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

```
int main() {
```

```
    char abc[] = "Pointery jsou fajn!";
    char* p_abc = abc;
```

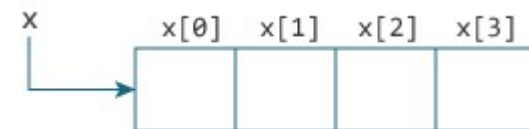
```
    while(*p_abc != '\0'){
        printf("%c", *p_abc);
        p_abc++;
    }
```

```
    return 0;
}
```

Řetězec znaků končí vždy nulovým znakem `\0`. Pro uložení slova Ahoj potřebuji tedy 5 pozic!

Pole v C je ukazatel na místo v paměti, kde pole začíná.

Proto:  
`cisla[1]` a `*cisla + 1` vrací stejný výsledek





# Aritmetika pointerů

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

int main() {

    int16_t pole[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int16_t *p_prvni, *p_posledni;
    p_prvni = pole;
    p_posledni = pole + 9;

    if(p_posledni > p_prvni){

        printf("Adresa %d \n", p_prvni);
        printf("Adresa %d \n", p_posledni);
        printf("Prvni %d \n", *p_prvni);
        printf("Posledni %d \n", *p_posledni);
        printf("Vysledek %d \n", p_posledni - p_prvni);

    }

    return 0;
}
```

>>Adresa 6356724  
>>Adresa 6356742  
>>Prvni 1  
>>Posledni 10  
>>Vysledek 9

S pointery jde počítat. Lze k nim přičítat celá čísla. Lze je mezi sebou porovnávat a také přičítat a odčítat mezi sebou. Smysluplné výsledky dostaneme například pokud máme dva ukazatele v jednom bloku paměti. Je třeba mít na paměti, že dochází ke srovnávání adres a tedy porovnání v příkladu `p_posledni > p_prvni` říká, že `p_posledni` je „dále“ v bloku paměti. Rozdíl v příkladu je devět bloků příslušného datového typu. Tedy dle adres 18 bajtů. Kód `p_prvni++` tedy posune ukazatel o dva bajty. Hodnotu do které se ukládá `int16`.

# typedef – uživatelské datové typy

V jazyce C je možné vytvořit uživatelský datový typ používá se klíčového slova typedef

Příklad samozřejmě nemá valný smysl, tato možnost se s výhodou používá např. právě při tvorbě struktur v C

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

```
int main() {

    typedef int mujInt;

    mujInt a, b;

    a = 10;
    b = 20;

    mujInt c = a + b;

    printf("%d", c);

    return 0;
}
```

# Struktury

Struktura je zjednodušeně datový typ, do které uzavřeme další datové typy, které s tímto typem nějak abstraktně souvisí. Například každý uživatel má jméno, věk atd.

Struktura může uchovávat různé datové typy a pole.

Strukturu lze vytvořit různým zápisem, ale vřele doporučujeme držet se tohoto zápisu a vytvořit strukturu jako nový datový typ. V příkladu je umístěna do globálního prostoru.

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

typedef struct{
    char  jmeno[25];
    int   vek;
    int   vyska;
} clovek;

int main() {

    clovek Petr = {"Petr Novak", 25, 178};
    clovek Michal;

    Michal.vek = 16;
    Michal.vyska = 193;

    strcpy(Michal.jmeno, "Michal Novak");

    printf("Petr ma %d let\n", Petr.vek);
    printf("Michal se jmenuje %s", Michal.jmeno);

    return 0;
}
```

>>Petr ma 25 let  
>>Michal se jmenuje Michal Novak