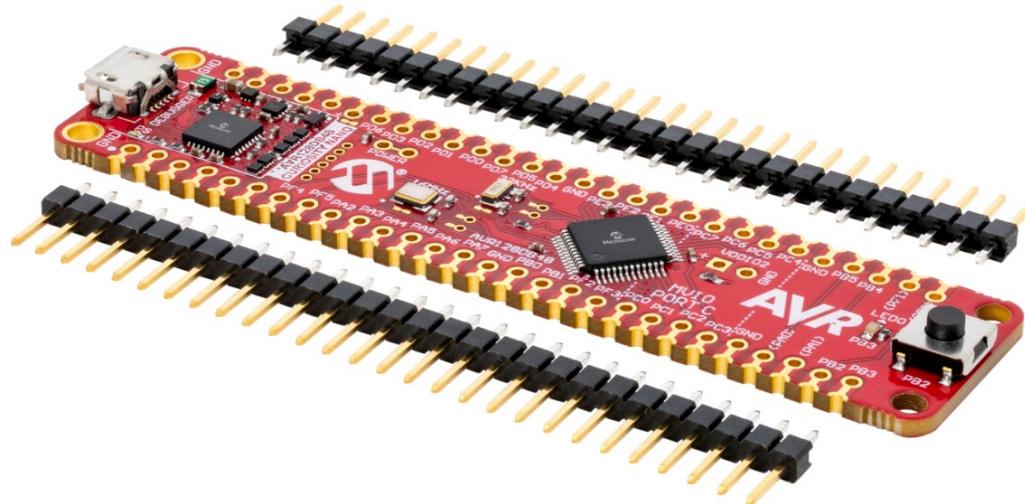
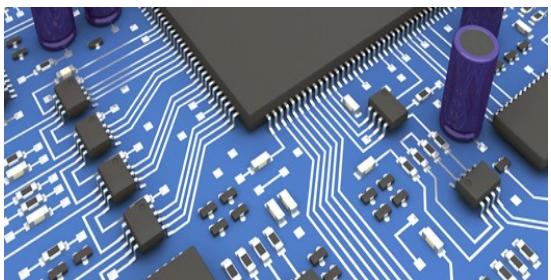


Aplikace Embedded systémů v Mechatronice



Michal Bastl

Překlad kódu

- **Preprocessing**

Základní příprava zpracuje direktivy preprocesoru a odstraní komentáře.

- **Compiling**

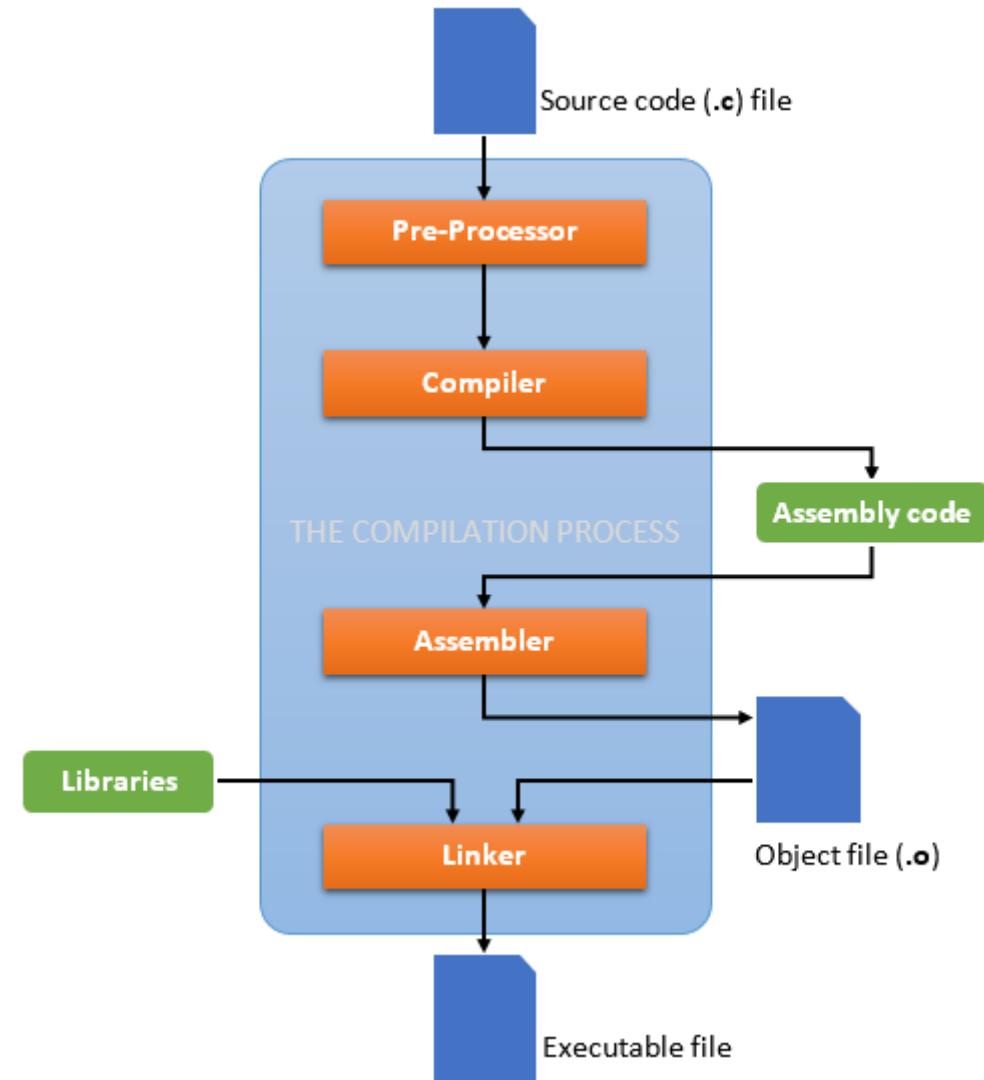
Vezme výstup preprocesoru a převede C na assembly code.

- **Assembler**

Vytvoří tzv object .o kód (relocatable)

- **Linking**

Linker sestaví vše (object kódy knihovny) do jednoho. Určí jak vše bude v paměti. Ovlivňuje Linker file. Vystup: .exe .hex .bin



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
```

```
#endif
```

```
#if
```

```
#endif
```

```
#include <stdio.h>  
  
#define NOT_IMPLEMENTED 0  
  
int main(void)  
{  
#if NOT_IMPLEMENTED  
    printf("Nevytisknes\r\n");  
#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;
}
```

Ukázka

Kompilace pomocí GCC

gcc -E main.c proběhne pouze preprocesor

gcc -S main.c assembler

gcc -c main.c object code

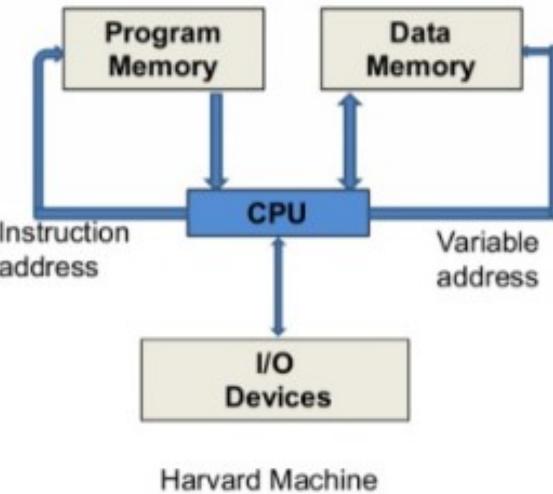
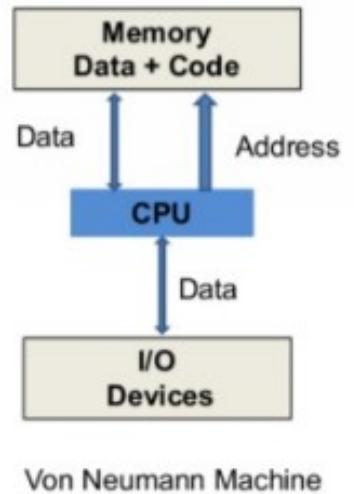
gcc main.c komplet i s linkerem na w10 a.exe

gcc main.c -o program vznikne program.exe

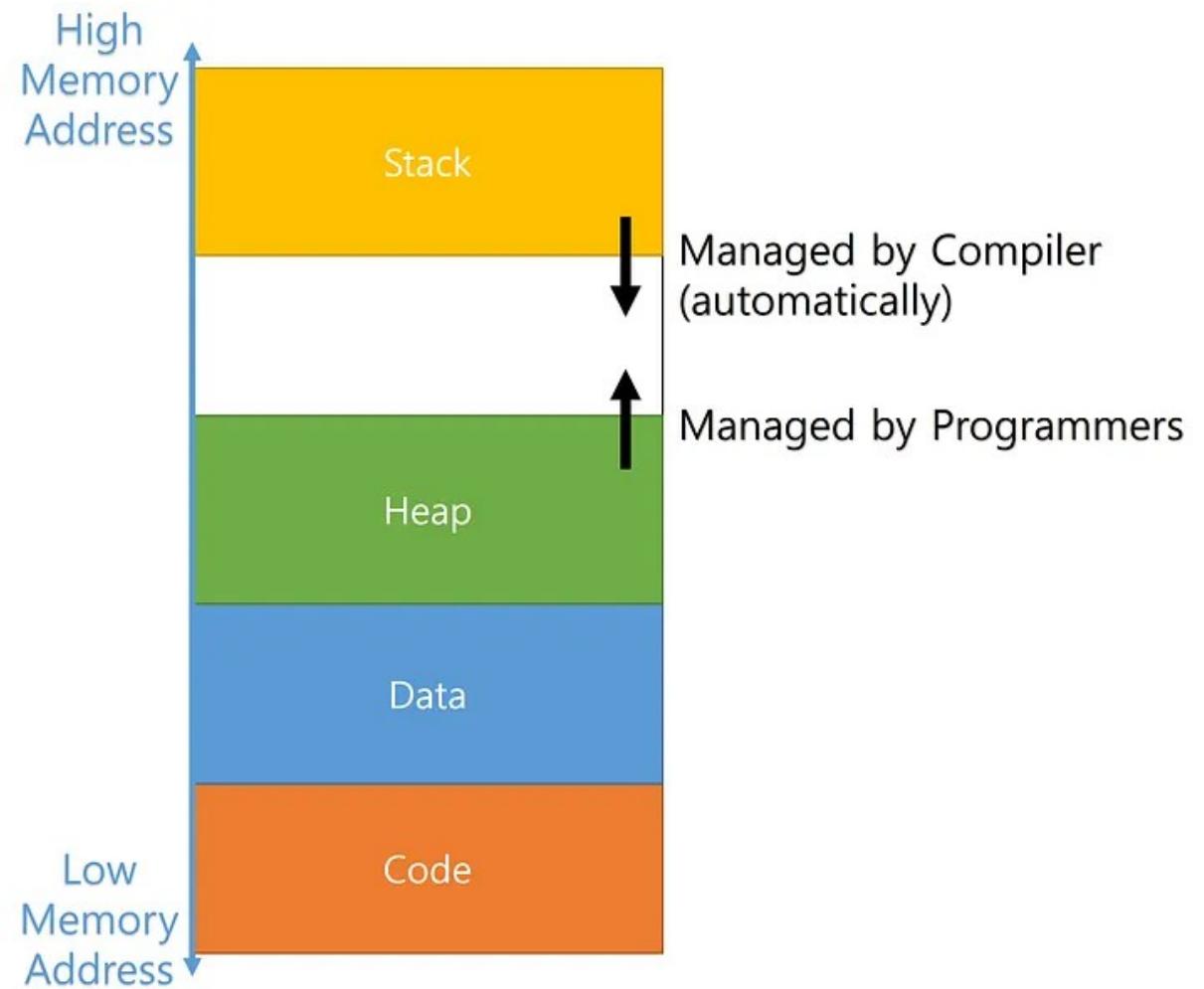
Paměť a její organizace

Desktop PC: Von Neumann

Embedded: Harvardska (jsou i vyjimky)



- Paměť je tvořena adresovatelnými bloky
- Int32 zabírá 4 bajty



Paměť a její organizace

```
#include <stdlib.h>

// --- GLOBÁLNÍ DATA ---
int global_data = 100;

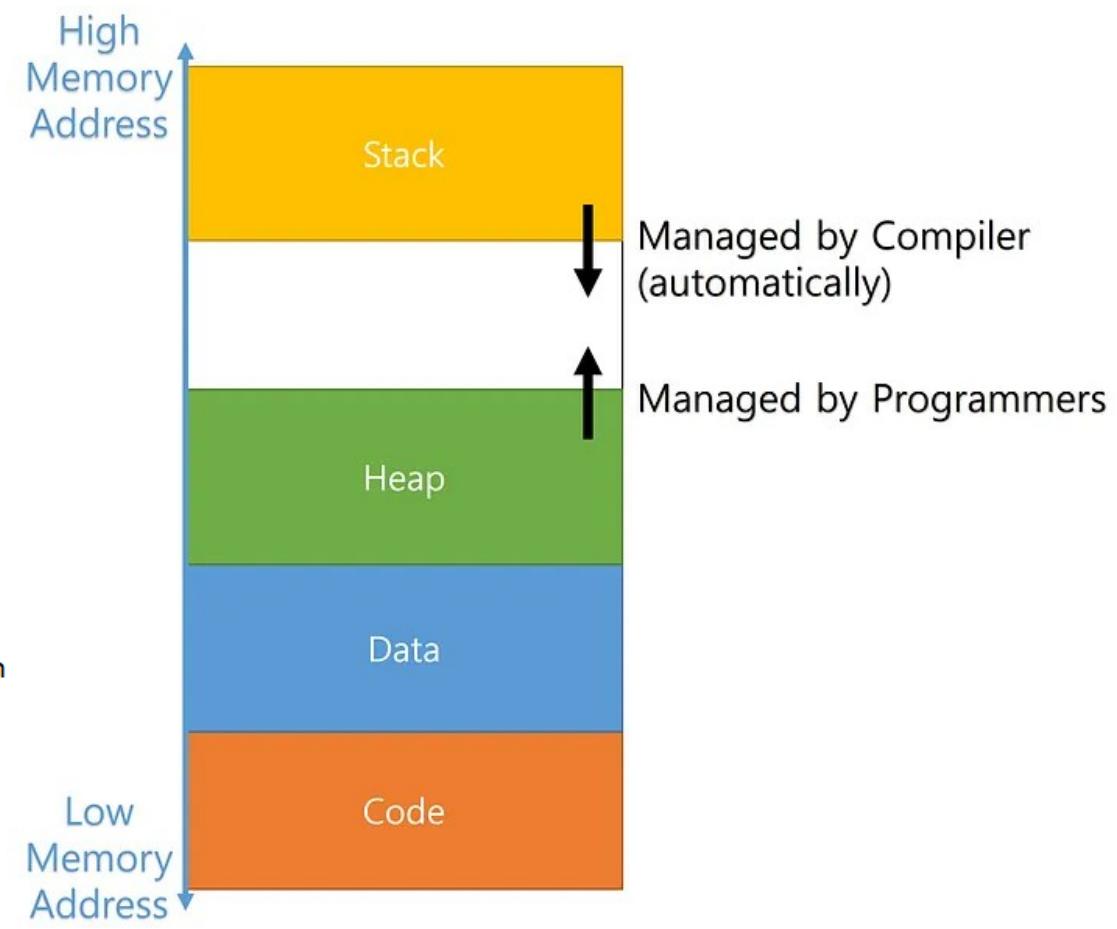
void function_example(int arg_f) {          // STACK (Argument)
    static int static_f = 10;                 // GLOBÁLNÍ DATA
    int local_f = 20;                        // STACK (Lokální)

    int *ptr_f = malloc(sizeof(int));        // ptr_f: STACK, *ptr_f: HEAP
    if (ptr_f) {
        *ptr_f = arg_f + local_f;
        free(ptr_f);
    }
}

int main(int argc, char **argv) {           // STACK (Argumenty)
    // --- STACK (Lokální data v main) ---
    int main_local = 50;
    int array_on_stack[10];

    // --- HEAP (Dynamická data v main) ---
    int *main_heap = malloc(100 * sizeof(int)); // main_heap: STACK, *main_h
    if (main_heap) {
        function_example(main_local);
        free(main_heap);
    }

    return 0;
}
```



Endianita

Little-endian:

- Na nejnižším místě v paměti je nejméně významný bajt

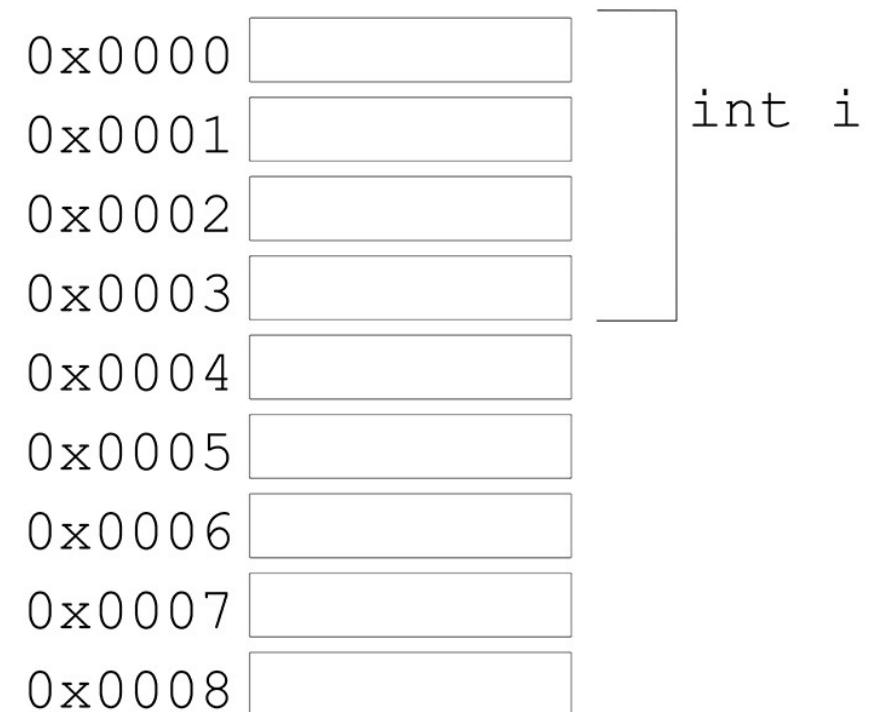
Např. 32bitové číslo `0x4A3B2C1D` se na adresu `100` uloží takto:

	100	101	102	103	
...	1D	2C	3B	4A	...

Big-endian:

- Na nejnižším místě v paměti je nejvíce významný bajt

	100	101	102	103	
...	4A	3B	2C	1D	...



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

Pointery/ukazatele

```
#include <stdio.h>

int main()
{ int c;
  int* p_c;
  int* p_m;

  c = 10;
  p_c = &c;
  p_m = &c;

  printf("Na adrese 0x%p je hodnota: %d\n",p_c,*p_c);
  printf("Na adrese 0x%p je hodnota: %d\n",p_m,*p_m);
  return 0;
}

>>Na adrese 0x0060FF08 je hodnota 10
```

operátor reference &c vrací adresu
operátor dereference *p_c vrací
hodnotu uloženou na adresu
symbol *p_c slouží současně pro
deklaraci pointeru

Pointery 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;
}

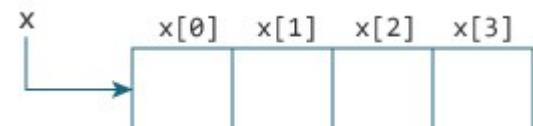
>>7
>>3
```

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

Pole je de facto konstantní pointer

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

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



Ukázka

Řetězce = pole znaků

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

int main() {

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

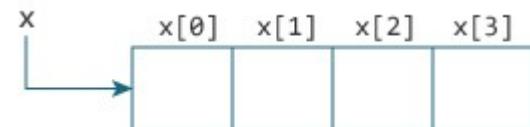
    while(*p_abc != '\0'){
        printf("%c", *p_abc);
        p_abc++;
    }

    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.
Je to konstantní pointer.

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



Ukázka

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`.

Ukázka

typedef – uživatelské datové typy

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

Příklad je jen ilustrativní, tato možnost
se s výhodou používá např. právě při
tvorbě struktur v C

```
#include <stdio.h>
#include <stdlib.h>
typedef unsigned char U8;

int main() {
    U8 a, b;
    a = 10;
    b = 20;

    U8 c = a + b;
    printf("%d", c);

    return 0;
}
```

Pointer a funkce (predani pointeru)

```
#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!!

Ukázka

Pointer a funkce (vraceni pointeru)

Funkce může vracet pointer ve smyslu ukazatele do poměti (hodnota adresy)

- Přiložený kód nastaví pointer na začátek pole
- Pokud je hodnota větší než kam ukazuje pointer změním adresu
- Vrátím ukazatel na nejvyšší prvek

char *max(char *pole, char n);

```
#include <stdio.h>

char *max(char *pole, char n);

int main(void){

    char pole[5] = {1, 15, 3, 10, 3};
    char *p_max = max(pole, 5);

    printf("Max prvek je: %d", *p_max);

    return 0;
}

char *max(char *pole, char n){
    char i;
    char *p_max = pole;

    for(i=1; i<n; i++){
        if(pole[i]>*p_max){
            p_max = &pole[i];
        }
    }
    return p_max;
}
```

Ukázka

Pointer a funkce (pointer na funkci)

- Mohu vytvářet i pointer na funkce
- Název funkce slouží k předání adresy
- Pointer na funkci zná zda funkce příjme, nebo vrací parametry
- Funkci na kterou pointer ukazuje mohu měnit
- Princip registrace callbacku

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

void fun1(void){
    printf("fun1\n");
}

void fun2(void){
    printf("fun2\n");
}

int main(void) {
    void (*call)(void); // funkce
    call= &fun1; // mohu pouzit symbol adresy
    (*call)();
    call = fun2; // take funguje
    call(); // take možnost
    return 0;
}
```

Ukázka

Pole pointerů na funkci

Z pointerů na funkce lze vytvořit pole

Toto může být výhodné, pokud potřebuji proiterovat více funkcí

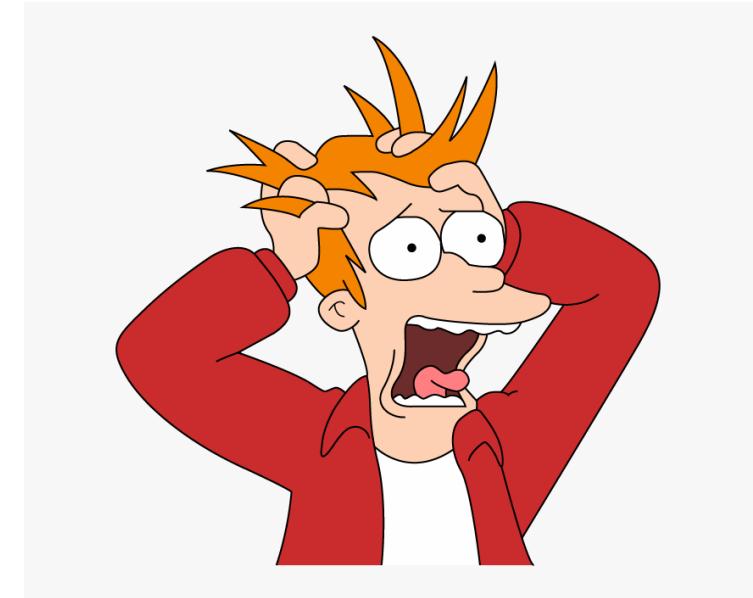
`void (*fun_ptr_arr[])(int, int)`

```
#include <stdio.h>
void add(int a, int b)
{
    printf("Add %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Sub %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Mul %d\n", a*b);
}

int main()
{
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned a = 5, b = 5;

    char i;
    for(i=0; i<=2; i++){
        (*fun_ptr_arr[i])(a, b);
    }

    return 0;
}
```



Ukázka

Void pointer

- Void pointer ukazuje na místo v paměti
- Nemá specifikovaný datový typ
- Z tohoto důvodu nelze dereferencovat
- Musím uvést typ, se kterým chci pracovat
- `void* p`, vrací některé funkce např. `malloc()`

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

int main(void){
    int16_t a;
    void* p_v = &a; // void pointer nelze dereferencovat

    a = 1795;

    //printf("a: %d\n", *p_v);

    printf("a: %d\n", a);
    printf("Prvni: %d a druhý: %d bajt \n", *(char*)p_v, *(char*)(p_v+1));

    return 0;
}
```

Ukázka