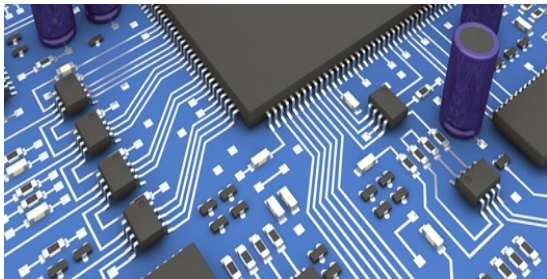


Aplikace Embedded systémů v Mechatronice



Michal Bastl

Překlad kódu

- Preprocessing

Základní příprava zpracuje direktivy preprocesoru a odstraní komentáře.

- Compiling

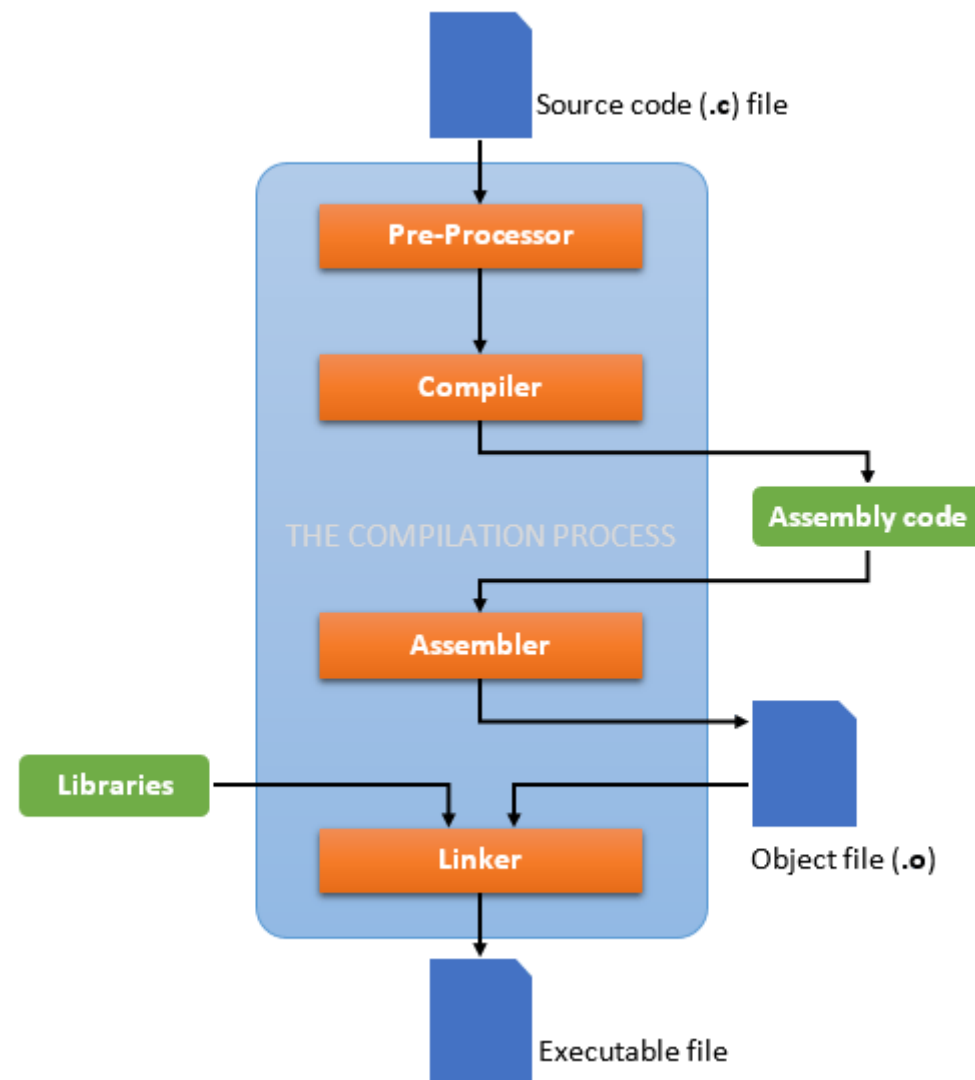
Vezme výstup preprocesoru a převede C na assembly code.

- 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. Výstup: .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 3.14159
#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_IMPLMENTED 0
```

```
int main(void)
```

```
{
```

```
#if NOT_IMPLMENTED
```

```
    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

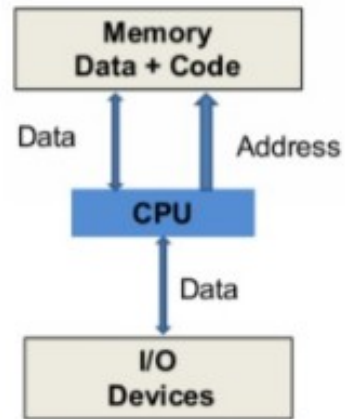
Kompilace pomocí GCC

- `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`

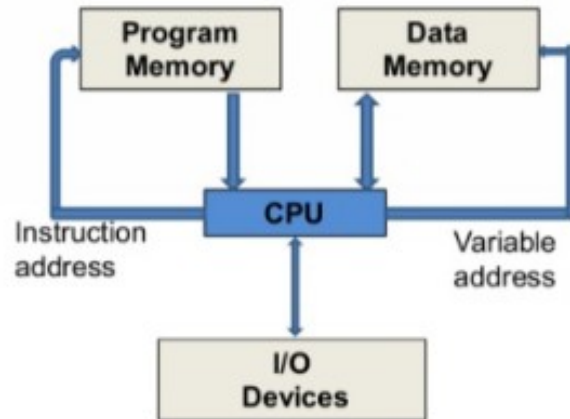
Paměť a její organizace

Desktop PC: Von Neumann

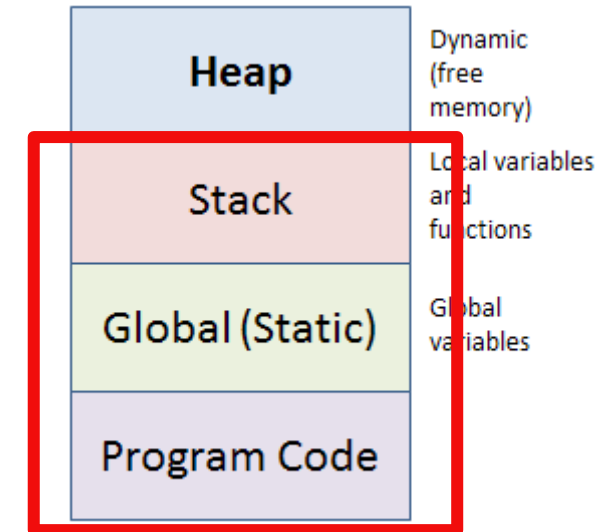
Embedded: Harvardska (jsou i vyjimky)



Von Neumann Machine



Harvard Machine



0x0000	<input type="text"/>	int i
0x0001	<input type="text"/>	
0x0002	<input type="text"/>	
0x0003	<input type="text"/>	
0x0004	<input type="text"/>	
0x0005	<input type="text"/>	
0x0006	<input type="text"/>	
0x0007	<input type="text"/>	
0x0008	<input type="text"/>	

- Paměť je tvořena adresovatelnými bloky
- Int32 zabírá 4 bajty

Endianita

Little-endian:

- Na nejnižším místě v paměti je nejméně významný bajt

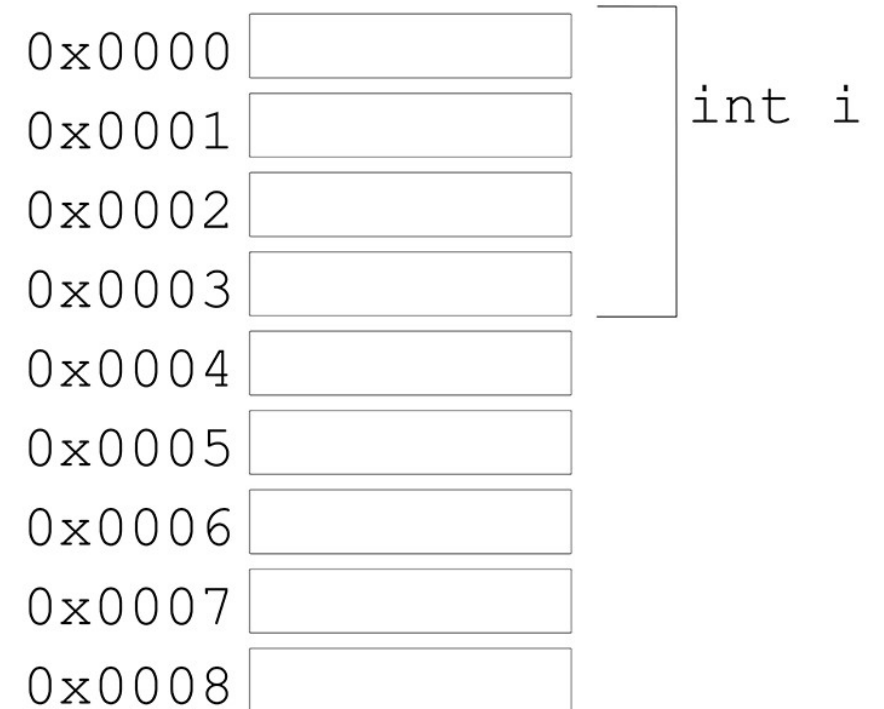
Např. 32bitové číslo `0x4A3B2C1D` se na adresu `100` uloží takto:

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

Big-endian:

- Na nejnižším místě v paměti je nejvíce významný bajt

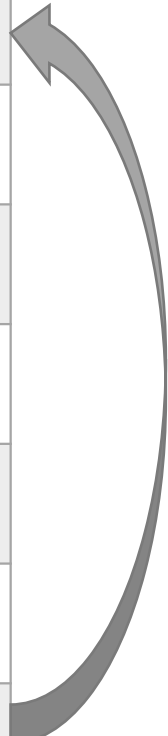
	100	101	102	103	
...	4A	3B	2C	1D	...



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

ADD	VAL
0x03	10
0xff	0x03



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 adrese
symbol *p_c slouží současně pro
deklaraci pointeru

Pointer vs. pole

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

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

int main() {

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

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

    uloz_do_pole(ciska, 7, 3);

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

    return 0;
}

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

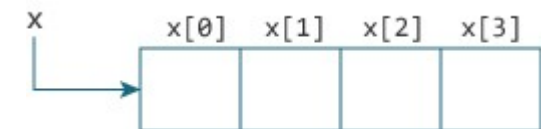
>>7
>>3
```

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 je de facto konstantní pointer

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

Proto:
ciska[1] a *ciska + 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++;
    }
```

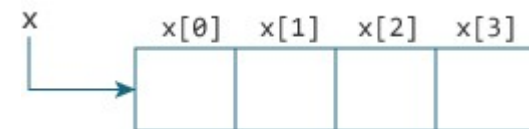
```
    return 0;
```

```
}
```

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`.

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!!

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 přijme, 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)();             // mohu volat se symbolem dereference

    call = fun2;           // take funguje

    call();               // take možnost

    return 0;
}
```

Ukázka

Callback

- Při využívání externích knihoven a driverů
- Může se po uživatelovi chtít předat (zaregistrovat) ukazatel na svoji funkci
- Příklad je callback na interrupt funkci v knihovnách MCC konfigurátoru

```
void my_fun(void);

int main(void) {
    register_callback(my_fun);

    callback();

    return 0;
}
```

```
#include "mcc_generated_files/mcc.h"
```

```
void myTimer4ISR(void);
```

```
Int main(){
    SYSTEM_Initialize();
```

```
    TMR4_SetInterruptHandler (myTimer4ISR);
```

```
    while (1)
    {
        // Add your application code
    }

}

void myTimer4ISR(void){
    IO_LED_D7_Toggle(); //Control LED
}
```

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 druhy: %d bajt \n", *(char*)p_v, *(char*)(p_v+1));

    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

Ukázka

Struktury - bitova pole

U proměnné ve struktuře je možné určit rozsah bitech. Může se tak šetřit místem a nebo využívat omezený rozsah takové proměnné.

Počet bitů se uvádí za dvojtečku. Tato velikost se pak v rámci programu dodržuje.

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

typedef struct{
    uint8_t u1:4;
    uint8_t u2:4;
}bitf_t;

void main(void){

    bitf_t my_st;

    my_st.u1 = 1;

    int i;
    for(i=0; i<20;i++){
        printf("%d\n", my_st.u1++);
    }

}
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
1
2
3
4

Ukázka

Union

Union zabere v paměti pouze tolik paměti jako na místo nejnáročnější proměnná. Tyto proměnné se překrývají.

```
typedef union{  
    int a;  
    uint8_t b[4];  
}my_un;
```

```
typedef union{  
  
    struct{  
        uint8_t b0 :1;  
        uint8_t b1 :1;  
        uint8_t b2 :1;  
        uint8_t b3 :1;  
  
        uint8_t b4 :1;  
        uint8_t b5 :1;  
        uint8_t b6 :1;  
        uint8_t b7 :1;  
    };  
  
    uint8_t all;  
  
}reg_t;
```

Enum – výčtový typ

Enum je datový typ, který sdružuje konstanty. Může být argumentem funkcí i jako návratová hodnota. Jedná se o datový typ int pro příslušnou platformu. Používá se často jako tzv. flags, jak je demonstrováno v příkladu.

```
typedef enum{
    OK=0,
    NOK,
    ERR,
}STATUS;

STATUS fun(int a){
    if(a == 0){
        return OK;
    }
    else if(a > 0){
        return NOK;
    }
    else{
        return ERR;
    }
};
```

```
#include <stdio.h>

STATUS fun(int a);

void main(void){

    switch(fun(-1)){
        case OK:
            printf("OK");
            break;
        case NOK:
            printf("NOK");
            break;
        case ERR:
            printf("ERR");
            break;
    }

    printf("\n%d", ERR);
}
```


Dynamická alokace

Funkce:

- malloc()
- calloc()
- free()
- realloc()
- Dynamická alokace se provádí na haldě Heap
- Ukážeme si malloc a free
- Funkci malloc si řeknu o alokaci jasně určené velikosti paměti v bajtech
- Funkce malloc vrátí pointer na začátek alokované paměti void*
- Pokud se alokace nezdaří - NULL

```
int main()
{

    int* ptr;
    int n, i;

    n = 5;

    ptr = (int*)malloc(n * sizeof(int));

    //vždy kontrola
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        return -1;
    }
```