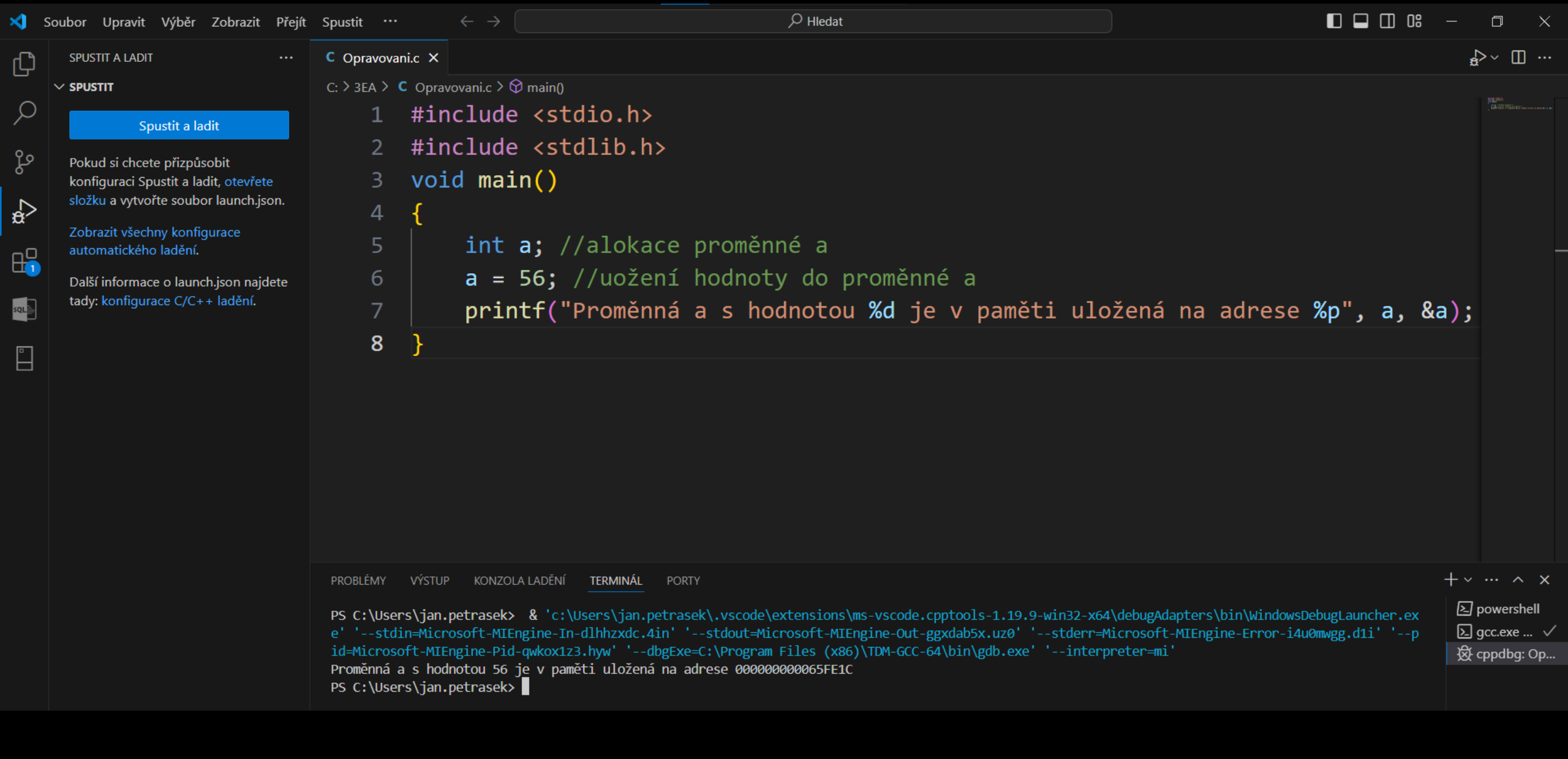
Ukazatele v jazyce C

Petrásek Jan

Proměnná

- Je místo v paměti pro uložení hodnoty určitého datového typu
- Adresa v paměti je hexadecimální číslo a ukazuje na 1 B
- Při alokaci paměti dochází k tomu, že aplikace vyžaduje určité místo pro uložení proměnné, od OS dostane adresu
- Adresu proměnné v paměti získáme referenčním operátorem &
- Pro výpis adresy do konzole využijeme ve formátovacím řetězci %p, což ji vypíše v šestnáctkové soustavě
- Vrácená adresa je vždy adresou začátku alokace, konec pak závisí na datovém typu

Získání adresy proměnné – příklad



Ukazatele (pointry)

- Ukazatel je proměnná, jejíž hodnotou je adresa někam do paměti
- Jazyk s ukazatelem nepracuje jako s hodnotou, ale opravdu jako s adresou
- Pokud ukazatel vypíšeme, obvykle vypisujeme hodnotu na kterou ukazuje, nikoliv adresu
- Dobrou praxí (bude vyžadováno!), je prefix názvu ve tvaru p_
- Před název pointru píšeme derefenční operátor *
- Datový typ ukazatele musí být shodný s proměnnou na kterou ukazuje
- Dereferenční operátor se využívá vždy, když se pracuje s hodnotou v paměti, nikoli s adresou

Ukazatele (pointry)

```c
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int a; //alokace proměnné a
    int *p_a; //alokace ukazatele
    a = 56; //uožení hodnoty do proměnné a
    p_a = &a; //uložení adresy proměnné a do ukazatele p_a
    printf("Proměnná a s hodnotou %d je v paměti uložená na adrese %p", a, p_a);
    *p_a = 2; //zápis hodnoty 2 na adresu v ukazateli
    printf("\nUkazatel p_a má hodnotu %d a ukazuje na hodnotu %d", p_a, *p_a);
}
```

PROBLÉMY VÝSTUP KONZOLA LADĚNÍ TERMINÁL PORTY

```
PS C:\Users\jan.petrasek>  & 'c:\Users\jan.petrasek\.vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-oez0ypnz.ut5' '--stdout=Microsoft-MIEngine-Out-xty12jsz.zk3' '--stderr=Microsoft-MIEngine-Error-z2bhmtzr.kvs' '--pid=Microsoft-MIEngine-Pid-onnn1sdt.w3s' '--dbgExe=C:\Program Files (x86)\TDM-GCC-64\bin\gdb.exe' '--interpreter=mi'
Proměnná a s hodnotou 56 je v paměti uložená na adrese 000000000065FE14
Ukazatel p_a má hodnotu 6684180 a ukazuje na hodnotu 2
PS C:\Users\jan.petrasek>
```

gcc.exe ... ✓
cppdbg: Op...
cppdbg: Op...

Ukazatele (pointry)

Stav paměti k příkladu z minulého snímku:

	0x65FE14	0x65FE24	0x65FE34	0x65FE44	
……	0000	0000	0000	0010	……

Desítkově:

	6684180	6684196	6684212	6684228	
……	0	0	0	2	……

Předávání referencí

- Slouží k předávání hodnot do / z funkcí
- Předávání hodnotou

```
4  void prohod(int a, int b)
5  {
6      int pomocna = a; //ulozeni hodnoty z a do pomocna
7      a = b;  //pzápis hodnoty z b do a
8      b = pomocna; //uložení původní hodnoty a do b
9  }
10 //tento kód nefunguje!!
11 void main() {
12     int cislo1 = 15; //inicializace proměnné
13     int cislo2 = 8; //inicializace proměnné
14     prohod(cislo1, cislo2); //volání funkce s předáním parametrů hodnotou
15     //hodnoty proměnných cislo1 a cislo2 se zkopírují do proměnných a a b
16     printf("V cislo1 je číslo %d a v cislo2 je číslo %d.", cislo1, cislo2);
17 }
```

```
PROBLÉMY  VÝSTUP  KONZOLA LADĚNÍ  TERMINÁL  PORTY

PS C:\Users\jan.petrasek>  & 'c:\Users\jan.petrasek\.vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-ncvesn0q.zeu' '--stdout=Microsoft-MIEngine-Out-r1kpnovq.hgv' '--stderr=Microsoft-MIEngine-Error-53earldx.izj' '--pid=Microsoft-MIEngine-Pid-vi442v3i.aaa' '--dbgExe=C:\Program Files (x86)\TDM-GCC-64\bin\gdb.exe' '--interpreter=mi'
V cislo1 je číslo 15 a v cislo2 je číslo 8.
PS C:\Users\jan.petrasek>

gcc.exe ...
cppdbg: Op...
cppdbg: Op...
```

Předávání referencí

- Libovolnou proměnnou můžeme předat referencí
- Funkce bude jako parametry přijímat pointry

```
4  void prohod(int *p_a, int *p_b)
5  {
6      int pomocna = *p_a; //ulozeni hodnoty referované ukazatelem p_a do pomocna
7      *p_a = *p_b;  //zápis hodnoty z adresy p_b na adresu p_a
8      *p_b = pomocna; //uložení původní hodnoty z adresy p_a na adresu p_b
9  }
10 //tento kód NFUNGUJE!!
11 void main() {
12     int cislo1 = 15; //inicializace proměnné
13     int cislo2 = 8; //inicializace proměnné
14     prohod(&cislo1, &cislo2); //volání funkce s použitím referenčního operátoru
15     printf("V cislo1 je číslo %d a v cislo2 je číslo %d.", cislo1, cislo2);
16 }
```

```
PROBLÉMY  VÝSTUP  KONZOLA LADĚNÍ  TERMINÁL  PORTY

PS C:\Users\jan.petrasek>  & 'c:\Users\jan.petrasek\.vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-p2bs2wiu.qld' '--stdout=Microsoft-MIEngine-Out-ajh5qvw3.1y4' '--stderr=Microsoft-MIEngine-Error-hb1vkqn5.pck' '--pid=Microsoft-MIEngine-Pid-scknicr5.0qw' '--dbgExe=C:\Program Files (x86)\TDM-GCC-64\bin\gdb.exe' '--interpreter=mi'
V cislo1 je číslo 8 a v cislo2 je číslo 15.
PS C:\Users\jan.petrasek>

gcc.exe ...
cppdbg: Op...
```

Předávání pole

- Pole je vždy automaticky předáváno referencí
- Název pole je vždy ukazatelem

```
3  #define DELKA 10
4
5  void napln_pole(int pole[])
6  {
7      for (int i = 0; i < DELKA; i++)
8      {
9          pole[i] = i + 1; //naplní pole hodnotami 1 až 10
10     }
11 }
12
13 void main() {
14     int cisla[DELKA]; //deklaruje datovou strukturu pole typu int o 10 prvcích
15     napln_pole(cisla); //zavolá funkce napln_pole a předá refeneci na pole
16     printf("%d", cisla[5]); // Vypíše číslo 6
17 }
```

```
PROBLÉMY  VÝSTUP  KONZOLA LADĚNÍ  TERMINÁL  PORTY

PS C:\Users\jan.petrasek> & 'c:\Users\jan.petrasek\.vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-eqjvy4ow.w3u' '--stdout=Microsoft-MIEngine-Out-ahgnrkm3.shp' '--stderr=Microsoft-MIEngine-Error-sre3yegi.zgy' '--pid=Microsoft-MIEngine-Pid-dquqloj1.daa' '--dbgExe=C:\Program Files (x86)\TDM-GCC-64\bin\gdb.exe' '--interpreter=mi'
6
PS C:\Users\jan.petrasek>

gcc.exe ...
cppdbg: Op...
```

NULL

- Pokud je hodnota ukazatele null, jde o prázdný ukazatel
- Dobré je, čerstvě alokovaný ukazatel inicializovat na NULL, později do něj přiřadit adresu

```c
void main()
{
    int a; //alokace proměnné a
    int *p_a; //alokace ukazatele
    a = 56; //uožení hodnoty do proměnné a
    printf("Ukazatel obsahuje adresu %p", p_a);
    p_a = &a; //uložení adresy proměnné a do ukazatele p_a
    printf("\nDo ukazatele byla vlozena adresa %p", p_a);
    p_a = NULL; //vymazání adresy z ukazatele
    printf("\nUkazatel obsahuje adresu %p", p_a);
}
```

```
PROBLÉMY  VÝSTUP  KONZOLA LADĚNÍ  TERMINÁL  PORTY

PS C:\Users\jan.petrasek>  & 'c:\Users\jan.petrasek\.vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters
e' '--stdin=Microsoft-MIEngine-In-omxchnuf.2i5' '--stdout=Microsoft-MIEngine-Out-myvnukcr.kux' '--stderr=Microsoft-MIEng
id=Microsoft-MIEngine-Pid-rc50aeqh.wzf' '--dbgExe=C:\Program Files (x86)\TDM-GCC-64\bin\gdb.exe' '--interpreter=mi'
Ukazatel obsahuje adresu 0000000000000010
Do ukazatele byla vlozena adresa 000000000065FE14
Ukazatel obsahuje adresu 0000000000000000
PS C:\Users\jan.petrasek>
```

Klíčové slovo const

- Nastavuje něco na konstantu, tedy na neměnnou hodnotu
- Const int i = 10; //proměnná i je konstantou rovnou 10
- Const int *p_c; //nastavuje cílovou hodnotu na konstantu, proměnná na kterou ukazuje jde měnit
- Int const *p_c; //adresa, na kterou ukazatel ukazuje je neměnná, číselná hodnota na dané adrese lze měnit
- Konstantním ukazatelem je pole, název pole je ukazatelem na jeho počátek
- Const int const *p_c; //adresa, na kterou ukazatel ukazuje je neměnná, stejně tak cílová hodnota

Klíčové slovo const

```
//KONSTANTY

const int i = 10;



//ukazatel na int
//muzeme menit obsah ukazatele (adresu)   p_i = &i;
//muzeme menit hodnotu na kterou ukazuje ukazatel *p_i = 20;
int *p_i;

//ukazatel na konstantni int
//muzeme menit obsah ukazatele (adresu) p_k = &k;
//NEMUZEME menit hodnotu na kterou ukazuje ukazatel *p_k = 40;
const int *p_k;

//konstantni ukazatel na int
//NEMUZEME menit obsah ukazatele (adresu) p_c = &c;
//muzeme menit hodnotu na kterou ukazuje ukazatel *p_c = 70;
//v podstate to same je pole
int const *p_c;

//konstantni ukazatel na konstantni int
//NEMUZEME menit obsah ukazatele (adresu) p_b = &b;
//NEMUZEME menit hodnotu na kterou ukazuje ukazatel *p_b = 90;
const int const *p_b;
```

Alokace paměti

Staticky alokovaná paměť

- Když se náš program překládá, může překladač ve velkém množství případů jednoduše zjistit, kolik paměti bude při běhu programu třeba.
- Když si vytvoříme proměnnou typu int, Céčko ví, že na ni má vyhradit 32 bitů atd.
- Pokud není při běhu programu třeba žádná data přidávat, s touto automatickou alokací si bohatě vystačíme.
- To je v podstatě způsob, jakým jsme programovali doposud.

Dynamicky alokovaná paměť v zásobníku

- Týká se lokálních proměnných funkcí
- Vše se ovšem děje zas plně automaticky
- Jakmile funkci zavoláme, Céčko si řekne o paměť a jakmile funkce skončí, tato paměť se uvolní
- Toto je důvod, proč funkce nemůže vracet pole v ní vytvořené

Dynamicky alokovaná paměť na haldě

- Nastupuje v situaci, kdy nevíme, kolika položkovou datovou strukturu budeme potřebovat
- Pokud si programátor říká o paměť sám, bude vždy přidělena z haldy
- Těžištěm pro práci s dynamickou pamětí jsou funkce malloc() a free()
- Pro změnu velikosti alokované paměti slouží funkce realloc()
- Pro alokaci souvislého bloku paměti s jejím vynulováním pak slouží funkce callock()

Dynamicky alokovaná paměť na haldě – maloc()

- Řekne operačnímu systému o libovolné množství paměti (kolik jí potřebujeme uvedeme do parametru funkce v bajtech).
- Funkce vrátí pointer na první adresu, kde začíná naše nová paměť.
- Aby byla funkce malloc() univerzální, vrací pointer na typ void.
- Její výsledek bychom měli vždy přetypovat na takový pointer, který potřebujeme
- Při nezdařené alokaci vrátí maloc() hodnotu NULL
- Pro zjištění velikosti datového typu budeme vždy využívat funkci sizeof()
- Sizeof() je při kompilace nahrazena adekvátními konstantami

Dynamicky alokovaná paměť na haldě – free()

- Po každém zavolání funkce malloc() musí někdy (třeba až na konci programu) následovat zavolání funkce free(), která paměť označí opět jako volnou.
- Tato paměť je plně v naší režii a nikdo jiný než my ji za nás neuvolní.
- Jakmile přestaneme nějakou dynamicky alokovanou paměť potřebovat, musíme ji ihned uvolnit!
- Každý mallock() či callock() musí mít odpovídající free()!
- Paměť uvolňujeme v opačném pořadí, nežli jsme ji alokovali!

Dynamicky alokovaná paměť na haldě

```c
int main()
{
    int velikost; //určuje počet jednotek, pro které se bude alokovat paměť
    do //ošetření vstupu…
    } while (1); //konec osetreni vstupu
    float *p_i;
    printf("Pokousim se alokovat prostor pro %d cisel...", velikost);
    //dynamická alokace paměti na haldě
    p_i = (float *) malloc(sizeof(float)*velikost);
    //kontrola úspěšnosti alokace paměti
    if (p_i == NULL) //výpis chyby alokace a ukončení programu…
    //uvolnění paměti
    printf("\nUvolnuji uspesne alokovanou pamet...");
    free(p_i); //samostné uvolnění paměti z haldy
    p_i = NULL; //vymazání ukazatele
    return 0;
}
```

PROBLÉMY VÝSTUP KONZOLA LADĚNÍ TERMINÁL PORTY

```
PS C:\Users\jan.petrasek>  & 'c:\Users\jan.petrasek\.vscode\extensions\ms-vscode.cpptools-1.19.9-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-uednth2f.kni' '--stdout=Microsoft-MIEngine-Out-evjzgp2x.ml2' '--stderr=Microsoft-MIEngine-Error-fc4d554a.jf4' '--pid=Microsoft-MIEngine-Pid-iefdofcw.3rz' '--dbgExe=C:\Program Files (x86)\TDM-GCC-64\bin\gdb.exe' '--interpreter=mi'
Zadej pocet cisel: 30
Pokousim se alokovat prostor pro 30 cisel...
Uvolnuji uspesne alokovanou pamet...
PS C:\Users\jan.petrasek>
```

Dynamicky alokovaná paměť na haldě – callock()

- Alokuje souvislý blok paměti na haldě, který vynuluje
- Parametry
  - Počet prvků, které se mají alokovat
  - Velikost prvku v bytech

Dynamicky alokovaná paměť na haldě – callock()

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

int main()
{
  int *b; //ukazatel na int
  b = (int *)calloc(200, sizeof(int)); //vytvoří pole o 200 intech
  if (b == NULL)
  {
    printf("Nedostatek paměti");
    return (-1);
  }
  else
  {
    printf("Úspěšně alokováno");
  }
  free(b);
  return 0;
}
```

Dynamicky alokovaná paměť na haldě – reallock()

- Slouží ke změně velikosti dynamicky alokované paměti
- 1. parametr je začátek dynamické paměti, která být editována
- 2. parametr je nová velikost alokované paměti
- Je potřeba pomocný ukazatel
- Realokovaná paměť se adresuje pomocí pomocného ukazatele
- V případě úspěchu se pomocný ukazatel přehraje do původního ukazatele
- Přímé adresování původním ukazatelem vede při neúspěchu ke ztrátě dat

Dynamicky alokovaná paměť na haldě – reallock()

```
4   int main()
5   {
6     int *b; //uakzatel na int
7     b = (int *)calloc(10, sizeof(int)); //vynulované dynamické pole pro 10 intů
8   ›   if (b == NULL)···
13  ›   for (int i = 0; i < 10; i++) //naplnění hodnotami 0 až 90 po desítkách···
17    int *temp; //pomocný ukazatel
18    temp = (int *) realloc(b, sizeof(int) * 11); //realokace pole, které je o 1 po
19    if (temp == NULL)   //validace úspěčné realokace
20    {
21      printf("Nedostatek paměti");
22      free(b);
23      return (-1);
24    }
25    else
26    {
27      b = temp; //b ukazuje na nově realokovanou paměť
28    }
29    b[10] = 100; //zápis do nově aplokované části pole b
30  ›   for (int i = 0; i < 11; i++) //výpis hodnot ze zvětšeného pole···
34    free(b); //uvolnění pole b z paměti
35    b = NULL; //zrušení ukazatele b
```

Časté chyby – pointry – neuvolnění paměti

- Pokud jednou zapomeneme uvolnit nějakou paměť, tak se v zásadě nic nestane – platí pro alokaci v main() a pro práci na OS, nikoliv mikrokontrolery.
- Problém je to v případě, kdy paměť zapomeneme uvolnit uvnitř nějaké funkce, která se za běhu programu volá několikrát
- Nejhorší situace je, když paměť zapomeneme uvolnit v nějakém cyklu.
- Paměť nám při této chybě samozřejmě za nějakou dobu dojde, aplikace spadne a uživatel přijde o data

Časté chyby – pointry - Překročení hranic paměti

- Stejně jako u polí, ani u pointerů nikdo nehlídá co do této paměti ukládáme.
- Pokud uložíme něco většího, než kolik místa máme vyhrazeno, nabouráme paměť jiné části aplikace.
- Tato chyba se může projevit naprosto kdekoli a pravděpodobně ji budeme velmi dlouho hledat.
- Následná chyba totiž nijaklogicky nesouvisí s místem v programu, kde nám paměť přetekla.
- Může se najednou rozbít jakákoli část aplikace, protože do ní tato paměť vytekla

Časté chyby – pointry - Práce s uvolněnou pamětí

- Může se nám stát, ženějakou paměť uvolníme a poté se na tuto adresu pokusíme znovu něco zapsat.
- V tu chvíli však zapisujeme opět na paměť, která nám nepatří, následky viz. Překročení hranic paměti
- Proto je dobré uložit do ukazatele po uvolnění jeho paměti hodnotu NULL, abychom se této chybě vyvarovali.

Poíntrová matematíka

Přičítání / odečítání celého čísla k pointru

- Přičtením celého čísla k ukazateli se posouváme o daný násobek bitové délky datového typu ukazatele v paměti
- Př.: přičtením 1 k ukazateli datového typu int se v paměti posouváme o 1 int dále
- Pokud máme 2 ukazatele, které ukazují na stejný blok paměti, můžeme jejich hodnoty odečíst. Pokud bude ale např. jeden ukazatel ukazovat na začátek dynamického pole intů a druhý bude ukazovat např. na pátý prvek tohoto pole, získáme odečtením ukazatelů číslo 4
- Porovnáním ukazatelů obvyklými operátory získáme informace o vzájemné pozici prvků v paměti

Přičítání / odečítání celého čísla k pointru

```c
void main()
{
    int *p_i, *p_paty;
    // Alokace 100 krát velikosti intu
    p_i = (int *) malloc(sizeof(int) * 100); //dynamická alokace paměti na halde
    if (p_i == NULL)
    {
        printf("Nedostatek paměti.\n");
        exit(1);
    }
    printf("Alokovana pamet zacina na adrese %p", p_i);
    p_paty = p_i + 4; // Výpočet adresy pátého prvku
    printf("\n5. prvek alokovane pameti lezi na adrese %p", p_i);
    *p_paty = 56; // Uložení hodnoty na pátý prvek
    printf("\n5. prvek alokovane pameti obsahuje hodnotu %d", *p_paty);
    free(p_i); // Uvolnění paměti
    p_i = NULL; // Vymazání ukazatele
}
```

PROBLÉMY VÝSTUP KONZOLA LADĚNÍ TERMINÁL PORTY

```
id=Microsoft-MIEngine-Pid-nl3tu20y.c34' '--dbgExe=C:\Program Files (x86)\TDM-GCC-64\bin\gdb.exe' '--interpreter=mi'
Alokovana pamet zacina na adrese 00000000000C1530
5. prvek alokovane pameti lezi na adrese 00000000000C1530
5. prvek alokovane pameti obsahuje hodnotu 56
PS C:\Users\jan.petrasek>
```

gcc.exe ...
cppdbg: Op...

Řádek 17, sloupec 48 Mezery: 4 UTF-8 CRLF {} C Win32

Ukazatele a pole

- Pole je souvislý blok paměti
- K prvkům alokované paměti můžeme přistupovat pomocí hranatých závorek stejně, jako v případě pole
-
```
p_i = (int *) malloc(sizeof(int) * 100);
for(int i = 0; i < 100; i++)
{
    p_i[i] = 0; //zápis nuly do daného bloku paměti
}
```
- Počet prvků v pameti nelze zpětně určit

Dynamické textové řetězce

Vytvoření statického řetězce

```
char jmeno[21];
printf("Zadej jméno: ");
scanf(" %20s[^\n]", jmeno);
printf("Jmenuješ se %s", jmeno);
```

- Maximální délka textu je tedy 20 znaků, 21. znak je ukončení řetězce (\0)
- V paměti je alokováno místo na 21 znaků bez ohledu na to, kolik jich reálně potřebujeme

Vytvoření dynamického řetězce

- Vyžaduje vytvoření pomocného statického řetězce
- Scanf(), případně scanf_s() načte do statického řetězce vstup
- Překopírováním využité části pomocného řetězce do dynamické struktury vytvoříme dynamický řetězec o délce vstupu
- Výhodou je minimalizace obsazení paměti díky opakovanému užití pomocného statického řetězce

Vytvoření dynamického řetězce

```c
#include <string.h>

void main()
{
    char buffer[101]; //pomocný statický řetězec
    printf("Zadej jméno: ");
    scanf(" %100s[^\n]", buffer); // Načtení jména do pomocné paměti
    // Vytvoření dynamického řetězce
    char* jmeno = (char *) malloc(strlen(buffer) + 1);
    strcpy(jmeno, buffer); // Nastavení hodnoty
    buffer[0] = '\0'; //vymazání vstupu
    printf("Zadej prijmeni: ");
    scanf(" %100s[^\n]", buffer); //opakovane pouziti bufferu
    char* prijmeni = (char *) malloc(strlen(buffer) + 1); // Vytvoření dynamické
    strcpy(prijmeni, buffer); // Nastavení hodnoty
    printf("Jmenuješ se %s %s", jmeno, prijmeni);
    free(jmeno); // Uvolnění paměti
}
```

PROBLÉMY VÝSTUP KONZOLA LADĚNÍ TERMINÁL PORTY

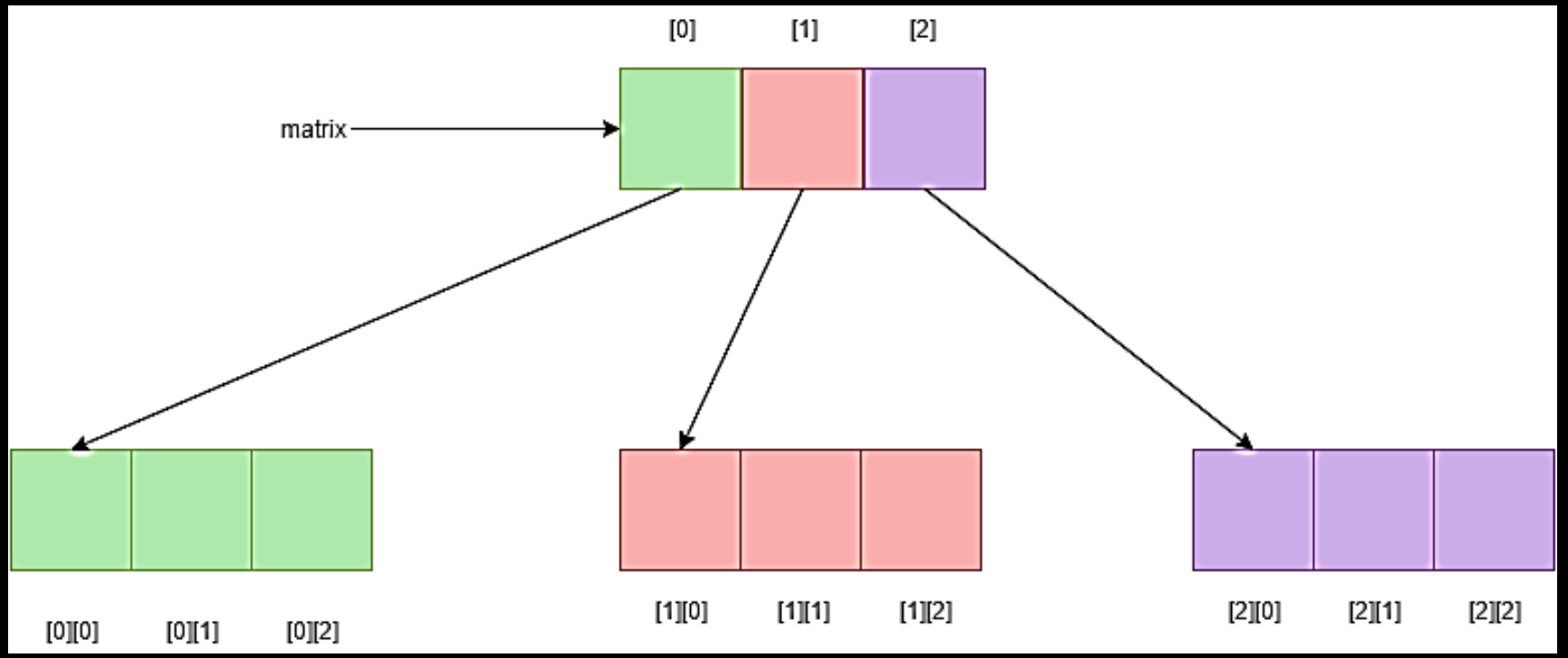
```
Zadej jméno: Jan
Zadej prijmeni: Petrasek
Jmenuješ se Jan Petrasek
PS C:\Users\jan.petrasek>
```

powershell
gcc.exe ...
cppdbg: Op...

Dynamické dvourozměrné pole

Problematika dynamických 2D polí

- Asi nejvěrnější je technika ukazatele na ukazatele
  - ** značí, že jde o ukazatel na ukazatele
  - Pole je v paměti jako sada 1D polí
  - Ukazatele na jednotlivá 1D pole jsou v 1D poli ukazatelů
  - Ukazatel na pole ukazuje na pole s ukazateli



Problematika dynamických 2D polí

```
4  int main()
5  {
6    //technika ukazatel na ukazatele
7    //dynamické 2D pole pro celá čísla (3x5)
8    int **matice; //definuje ukazatel na ukazatele na int
9    matice = (int **)malloc(sizeof(int *) * 3); //alokuje paměť pro 3 ukazatele na
10   if (matice == NULL)
11   {
12     printf("Nedostatek pameti");
13     return (-1);
14   }
15   for (int i = 0; i < 3; i++) //projde sadu ukazatelů
16   {
17     matice[i] = (int *)calloc(5, sizeof(int)); //dynamická paměti pro 5 intů
18     if (matice[i] == NULL)
19     {
20       printf("Nedostatek pameti");
21       return (-1);
22     }
23   }
24   //nyní je hotová dynamická alokace paměti a s polem pracujeme jakoí se statick
```

Uvolnění paměti 2D pole

- Paměť je potřeba uvolnit v opačném pořadí, nežli byla alokována
- Příklad z minulého snímku bude paměť uvolňovat následovně:

```
for (int i = 0; i < 3; i++)
{
        free(matice[i]); //uvolnění jednotlivých řádků
}
free(matice); //uvolnění ukazatele na ukazatele
```

Simulace 2D pole 1D polem

- Vytvoří se pole o délce radky*sloupce
- Dvourozměrnost simulujeme prací s indexy
- Příklad uložení pole 3x3 znaky v paměti

data		a	b	c	d	e	f	g	h	i	
index	.....	0	1	2	3	4	5	6	7	8	....
simulace		0, 0	0, 1	0, 2	1, 0	1, 1	1, 2	2, 1	2, 2	2, 3	

```
for (int i = 0; i < rady; i++)    //prochází řádky
  {
    for (int j = 0; j < sloupce; j++) //prochází sloupce
    {
        Pole[i * sloupce +j] //přistoupí na skutečný index 1D pole
```