

PB071 – Programování v jazyce C

Jaro 2014

Argumenty main(), Typový systém,
Dynamická alokace

Argumenty funkce main()

Motivace

- *Jak může program přijímat vstupní data/info?*
- Standardní vstup (klávesnice nebo přesměrování)
- Otevření souboru (disk, stdin...)
- Sdílená paměť (sdílená adresa, virtuální soubor)
- Příkazová řádka (argumenty při spuštění)
- Zprávy (message queue)
- Síťová komunikace (sokety)
- Přerušování, semaforey...

Argumenty funkce `main`

- Funkce `main` může mít tři verze:
 - `int main(void) ;`
 - bez parametrů
 - `int main(int argc, char *argv[]) ;`
 - parametry předané programu při spuštění
 - `**argv == *argv[]`
 - `int main(int argc, char **argv, char **envp) ;`
 - navíc proměnné prostředí,
- `binary.exe -test -param1 hello "hello world"`
- `argc` obsahuje počet parametrů
 - vždy alespoň jeden – cesta ke spuštěnému programu
- `argv[]` je pole řetězců, každý obsahuje jeden parametr
 - `argv[0]` ... cesta k programu
 - `argv[1]` ... první parametr

Parametry funkce `main` - ukázka

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[], char *envp[]) {
    if (argc == 1) {
        printf("No extra parameters given\n");
    }
    else {
        for (int i = 0; i < argc; i++) {
            printf("%d. parameter: '%s'\n", i, argv[i]);

            if (strcmp(argv[i], "-param1") == 0) {
                printf("    Parameter '-param1' detected!\n");
            }
        }
    }
    // Print environmental variables. Number is not given,
    // but envp ends with NULL (==0) pointer
    printf("\nEnvironment parameters:\n");
    int i = 0;
    while (envp[i] != NULL) printf("%s\n", envp[i++]);

    return EXIT_SUCCESS;
}
```

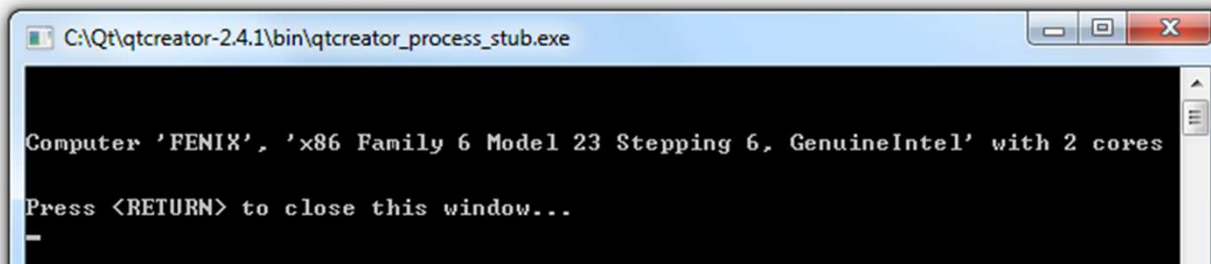
zpracování
parametrů
spuštění

zpracování
proměnných
prostředí

Demo: Využití proměnných prostředí

- Jednotlivé položky ve formátu NAZEV=hodnota
 - `PROCESSOR_ARCHITECTURE=x86`
 - lze manuálně parsovat řetězec
 - lze využít funkci `getenv()` funkce v `stdlib.h`
 - poslední položka obsahuje ukazatel na `NULL`
- Řetězce z `envp` nemodifikujte
 - jsou alokovány v paměti příslušné systému, nikoli našemu programu

```
int main(int argc, char *argv[], char *envp[]) {  
    const char* host = getenv("COMPUTERNAME");  
    const char* cpu = getenv("PROCESSOR_IDENTIFIER");  
    const char* cores = getenv("NUMBER_OF_PROCESSORS");  
    printf("Computer '%s', '%s' with %s cores\n\n\n", host, cpu, cores);  
    return EXIT_SUCCESS;  
}
```



Typový systém

Typový systém - motivace

- Celé znaménkové číslo se reprezentuje nejčastěji v dvojkovém doplňkovém kódu

iVal1 = 5 (dvojkový doplňkový)

000000000000000000000000000000000000101

```
int iVal1 = 5;
```

- Číslo 5, uložené jako reálné číslo (IEEE 754)

fVal1 = 5 (IEEE 754)

01000000101000000000000000000000

```
float fVal1 = 5;
```

- Paměť `iVal1` interpretovaná jako reálné číslo

- 7.006492321624085⁻⁴⁵

- Procesor musí vědět, jak paměť interpretovat!

- datový typ části paměti

Typový systém obecně

- Co je typový systém
 - každá hodnota je na nejnižší úrovni reprezentována jako sekvence bitů
 - každá hodnota během výpočtu má přiřazen svůj typ
 - typ hodnoty dává sekvenci bitů význam – jak se má interpretovat
- Jsou definována pravidla
 - jak se mohou měnit typy hodnot
 - které typy mohou být použity danou operací
- C je staticky typovaný systém
 - typ kontrolován během překladu

Typový systém zajišťuje

- Aby nebylo nutné přemýšlet na úrovni bitů
 - podpora abstrakce (celé číslo, ne sekvence bitů)
- Aby se neprováděla operace nad neočekávaným typem hodnoty
 - jaký má význam dělení řetězců?
- Typ proměnných může být kontrolován
 - překladačem během kompilace – staticky typovaný systém, konzervativnější
 - běhovým prostředím – dynamicky typovaný systém
- Aby bylo možné lépe optimalizovat

Typová kontrola překladačem

1. Určí se typ výrazu (dle typů literálů)
2. Pokud je použit operátor, zkontroluje se typ operandů resp. funkčních argumentů
3. Pokud daný operátor není definován pro určený typ, provede se pokus o automatickou konverzi
 - hierarchie typů umožňujících automatickou konverzi
 - např. `int` → `float` ANO, `int` → `float*` NE
4. Pokud nelze provést automatickou konverzi, ohlásí překladač chybu
 - *error: invalid conversion from 'int*' to 'int'*
5. Je zajištěno, že operace se provedou jen nad hodnotami povolených typů (POZOR: neznamená validní výsledek!)

```
int iValue;  
int iValue2 = &iValue;
```

Konverze typů - přetypování

- Automatická (implicitní) konverze proběhne bez dodatečného příkazu programátora
 - anglicky *type coercion*
 - `float` `realVal` = `10`; // 10 je přetypováno z `int` na `float`
- Explicitní konverzi vynucuje programátor
 - syntaxe explicitního přetypování: `(nový_typ)` výraz
 - anglicky *type conversion, typecasting*
 - `float` `value` = `(float)` `5` / `9`; // 5 přetypováno z `int` na `float`
 - pozor na: `float` `value` = `(float)` (`5` / `9`);
 - 5 i 9 jsou defaultně typu `int`
 - výraz `5/9` je tedy vyhodnocen jako celočíselné dělení → 0
 - 0 je přetypována z `int` na `float` a uložena do `value`

Automatická typová konverze

- Automatická typová konverze může nastat při:

1. Vyhodnocení výrazu

```
int value = 9.5;
```

2. Předání argumentů funkci

```
void foo(float param);  
foo(5);
```

3. Návrátové hodnotě funkce

```
double foo() { return 5; }
```

Zarovnání dat v paměti

- Proměnné daného typu mohou vyžadovat zarovnání v paměti (závislé na architektuře)
 - char na každé adrese
 - int např. na násobcích 4
 - float např. na násobcích 8
- Paměťový aliasing
 - na stejné místo v paměti ukazatel s různým typem
- Pokus o přístup na nezarovnanou paměť způsobuje pád
- Striktnější varianta: Strict aliasing
 - žádné dvě proměnné různého typu neukazují na stejné místo
 - zavedeno z důvodu možnosti optimalizace
 - standard vyžaduje, ale lze vypnout (na rozdíl od obecného aliasingu)
- http://en.wikipedia.org/wiki/Aliasing_%28computing%29

Způsob provedení konverze

- Konverze může proběhnout na **bitové** nebo **sémantické** úrovni
- **Bitová úroveň** vezme bitovou reprezentaci původní hodnoty a interpretuje ji jako hodnotu v novém typu

- pole **float** jako pole **char**
- **float** jako **int**
- ...

```
float fArray[10];  
char* cArray = (char*) fArray;
```

Pozn.: výše uvedený kód nesplňuje požadavek na strict aliasing a není dle standardu validní (viz. dříve)

- **Sémantická úroveň** “vyčte” hodnotu původního typu a “uloží” ji do nového typu
 - např. **int** a **float** jsou v paměti realizovány výrazně odlišně
 - dvojkový doplňkový kód pro int vs. IEEE 754 pro float
 - **float** fValue = 5.4; **int** iValue = fValue;
 - 5.4 je typu **double** → konverze na **float** (5.4, bez ztráty)
 - fValue → konverze na **int** (5, ztráta)
 - výrazný rozdíl oproti bitové konverzi

Ukázky konverzí

```
float fVal = 5.3;
```

```
char cVal = fVal;
```

```
float fArray[10];
```

```
char* cArray = (char*) fArray;
```

```
4      float fVal = 5.3;
```

```
0x00401352 <+14>:      mov     $0x40a9999a,%eax
```

```
0x00401357 <+19>:      mov     %eax,0x4c(%esp)
```

```
5      char cVal = fVal;
```

```
0x0040135b <+23>:      flds     0x4c(%esp)
```

```
0x0040135f <+27>:      fnstcw  0xe(%esp)
```

```
0x00401363 <+31>:      mov     0xe(%esp),%ax
```

```
0x00401368 <+36>:      mov     $0xc,%ah
```

```
0x0040136a <+38>:      mov     %ax,0xc(%esp)
```

```
0x0040136f <+43>:      fldcw   0xc(%esp)
```

```
0x00401373 <+47>:      fistp   0xa(%esp)
```

```
0x00401377 <+51>:      fldcw   0xe(%esp)
```

```
0x0040137b <+55>:      mov     0xa(%esp),%ax
```

```
0x00401380 <+60>:      mov     %al,0x4b(%esp)
```

sémantická konverze

```
6
```

```
7      float fArray[10];
```

```
8      char* cArray = (char*) fArray;
```

```
0x00401384 <+64>:      lea     0x1c(%esp),%eax
```

```
0x00401388 <+68>:      mov     %eax,0x44(%esp)
```

bitová konverze

(Ne)ztrátovost typové konverze

- Bitová konverze není ztrátová
 - neměníme obsah paměti, jen způsob její interpretace
 - lze zase změnit typ “zpět”
 - *(platí jen dokud do paměti nezapíšeme)*
- Sémantická typová konverze může být ztrátová
 - uložení větší hodnoty do menšího typu (**int** a **char**)
 - ztráta znaménka (**int** do **unsigned int**)
 - ztráta přesnosti (**float** do **int**)
 - reprezentace čísel (**16777217** jako **float**)

Ukázka ztrátovosti při typové konverzi

```
int iValue    = 16777217;
float fValue = 16777217.0;
printf("The integer is: %i\n", iValue);
printf("The float is:    %f\n", fValue);
printf("Their equality: %i\n", iValue == fValue);
```

```
The integer is: 16777217
The float is: 16777216.000000
Their equality: 0
```

```
int iValue    = 16777217;
double fValue = 16777217.0;
printf("The integer is: %i\n", iValue);
printf("The double is:   %f\n", fValue);
printf("Their equality: %i\n", iValue == fValue);
```

```
The integer is: 16777217
The double is: 16777217.000000
Their equality: 1
```

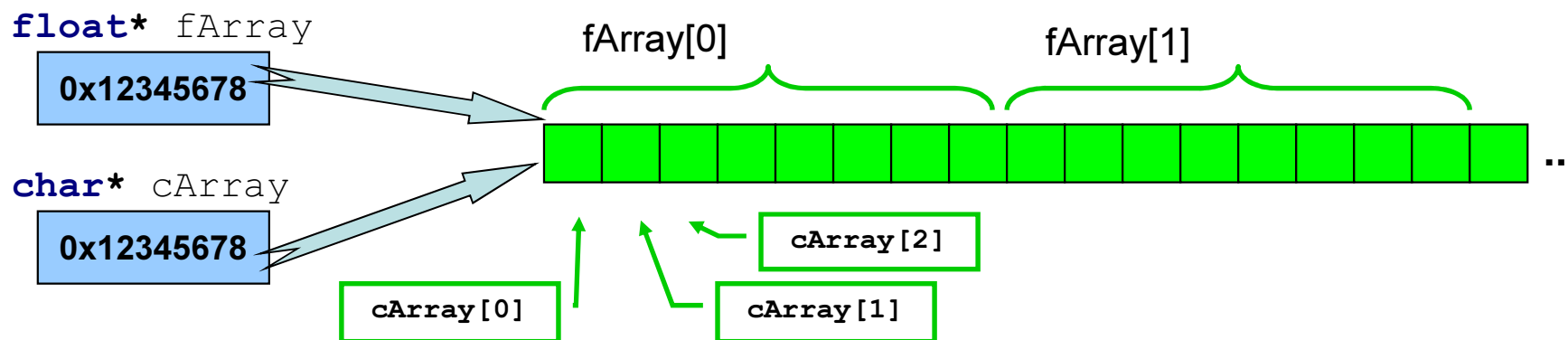
Příklad z http://en.wikipedia.org/wiki/Type_conversion

Přetypování ukazatelů

```
float fArray[10];  
char* cArray = (char*) fArray;
```

Pozn.: problém se strict aliasing

- Ukazatele lze přetypovat
 - bitové přetypování, mění se interpretace, nikoli hodnota
- Po přetypování jsou data na odkazované adrese interpretována jinak
 - závislost na bitové reprezentaci datových typů



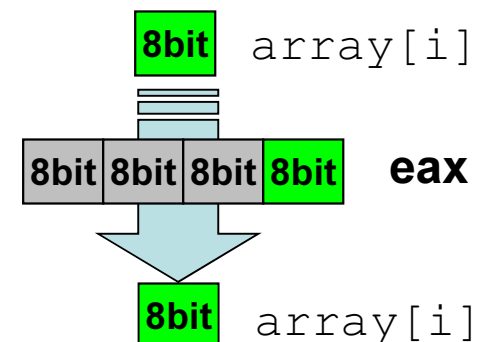
- Pozor na požadavek standardu na “strict aliasing”
 - nesmí existovat dva ukazatele různého typu na stejnou paměť
 - ukazatele `fArray` a `cArray` by neměli existovat současně
 - omezuje možnost optimalizace překladačem

Přetypování ukazatelů – void*

- Některé funkce mohou pracovat i bez znalosti datového typu
 - např. nastavení všech bitů na konkrétní hodnotu
 - např. bitové operace (^, & ...)
- Je zbytečné definovat separátní funkci pro každý datový typ
 - namísto specifického typu ukazatele se použije void*

```
void* memset(void* ptr, int value, size_t num);  
int array[100];  
memset(array, 0, sizeof(array));
```

Přetypování ukazatelů - využití



- Některé operace fungují na úrovni bitů
 - např. XOR, AND...
 - instrukce procesoru u x86 pracují na úrovni 32 bitů
 - při práci na úrovni např. char (8 bitů) se nevyužívá celý potenciál šířky registru (32 bitů)

```
unsigned char array[80];  
// Will execute 80 iteration  
for (int i = 0; i < sizeof(array); i++) array[i] ^= 0x55;  
  
// Will execute only 20 iteration (x86)  
for (int i = 0; i < (sizeof(array) / sizeof(int)); i++) {  
    ((unsigned int*) array)[i] ^= 0x55555555;  
}
```

POZOR, bude fungovat jen pro
sizeof(int) == 4



Proč? Jak udělat funkční obecně?

Vhodnost použití přetypování

- Typová kontrola výrazně snižuje riziko chyby
 - nedovolí nám dělat nekorektní operace
 - klíčová vlastnost moderních jazyků!
- Explicitní přetypování obchází typový systém
 - programátor musí využít korektně, jinak problém
- Snažte se obejít bez typové konverze
 - automatické i explicitní
- Nepiště kód závisející na automatické konverzi
 - vyžaduje po čtenáři znalost možných konverzí
- Preferujte explicitní konverzi před implicitní
 - je čitelnější a jednoznačnější

Dynamická alokace

Dynamická alokace - motivace

- U statické alokace je velikost nebo počet známa v době překladu
 - `int array[100];`
 - `int a, b, c;`
- Často ale tyto informace nejsou známy
 - např. databáze se rozšiřuje
 - např. textový popis má typicky 20 znaků, ale může mít i 1000
 - neefektivní mít vždy alokováno 1000 znaků
- Dynamická alokace umožňuje vytvořit datovou položku (proměnnou, pole, strukturu) až za běhu

Statická vs. dynamická alokace

- Statická alokace alokuje na zásobníku (**stack**)
 - automaticky se uvolňuje po dokončení bloku kódu
 - typicky konec funkce, konec cyklu for...
 - výrazně rychlejší (obsah zásobníku je typicky v cache)
 - využití pro krátkodobé proměnné/objekty
 - `int array[100];`
- Dynamická alokace na haldě (**heap**)
 - zůstává do explicitního uvolnění (nebo konce aplikace)
 - využití pro dlouhodobé paměťové entity
 - `int* array = malloc(variable_len*sizeof(int));`
- Specialitou je alokace polí s variabilní délkou (od C99)
 - délka není známa v době překladu
 - ale alokuje se na zásobníku a automaticky odstraňuje
 - `int array[variable_length];`

Organizace paměti

- Instrukce (program)
 - nemění se
- Statická data (static)
 - většina se nemění, jsou přímo v binárce
 - globální proměnné (mění se)
- Zásobník (stack)
 - mění se pro každou funkci
 - lokální proměnné
- Halda (heap)
 - mění se při každém malloc, free
 - dynamicky alokované prom.

Dynamická alokace

Statická alokace,
Pole s proměnnou délkou (VLA)

Celková paměť programu

Instrukce

```
...  
push %ebp 0x00401345  
mov %esp,%ebp 0x00401347  
sub $0x10,%esp 0x0040134d  
call 0x415780  
...
```

Statická a glob. data

```
"Hello",  
"World"  
{0xde 0xad 0xbe 0xef}
```

Zásobník

```
lokalniProm1  
lokalniProm2
```

Halda

```
dynAlokace1  
dynAlokace2
```

Funkce pro dynamickou alokaci paměti

- `#include <stdlib.h>`
- **`void*`** `malloc(size_t n)`
 - alokuje paměť o zadaném počtu bajtů
 - jeden souvislý blok (jako pole)
 - na haldě (heap)
 - paměť není inicializována („smetí“)
 - pokud se nezdaří, tak vrátí NULL
- **`void*`** `calloc(size_t n, size_t sizeItem)`
 - obdobné jako malloc
 - ale alokuje $n * \text{sizeItem}$
 - inicializuje paměť na 0

Změna velikosti alokované paměti

- **void** * realloc (**void** * ptr, size_t size);
 - pokud je ptr == NULL, tak stejně jako malloc()
 - pokud je ptr != NULL, tak změni velikost alokovaného paměťového bloku na hodnotu size
 - pokud size == 0, tak stejně jako free()
- Obsah původního paměťového bloku je zachován
 - pokud je nová velikost větší než stará
 - jinak je zkrácen
- Při zvětšení je dodatečná paměť neinicializovaná
 - stejně jako při malloc()

Uvolňování alokované paměti

- **void free(void*)**

- uvolní na haldě alokovanou paměť
- musí být hodnota ukazatele vrácená předchozím malloc()
 - nelze uvolnit jen část paměti (např. od ukazatele modifikovaného pomocí ukazatelové aritmetiky)

- Při přístupu na paměť po zavolání free(paměť)

- paměť již může být přidělena jiné proměnné

- Pozor, free() nemaže obsah paměti

- citlivá data dobré předem smazat (klíče, osobní údaje)

Rychlé nastavování paměti – memset()

- **void*** memset(**void** * ptr, **int** value, size_t num);
- Nastaví paměť na zadanou hodnotu
- Výrazně rychlejší než cyklus for pro inicializaci
- Pracuje na úrovni bajtů
 - časté využití v kombinaci se sizeof()

```
void* memset(void* ptr, int value, size_t num);  
int array[100];  
memset(array, 0, sizeof(array));
```

Dynamická alokace - shrnutí

1. Alokuje potřebný počet bajtů
 - (musíme správně vypočítat)
 - typicky používáme `počet_prvků * sizeof(prvek)`
2. Uložíme adresu paměti do ukazatele s požadovaným typem
 - `int* array = malloc(100 * sizeof(int));`
3. Po konci práce uvolníme
 - `free(array);`

Ilustrace statické vs. dynamické alokace

```
void foo() {  
    char array1[10];  
    char* array2 = malloc(10);  
    array1[2] = 1;  
    array2[2] = 1;  
}
```

```
int main() {  
    foo();  
}
```

foo();

return 0;

}

Neuvolněná paměť,
memory leaks

array1
01010111110010101

array2
0x12345678

malloc()
0101001100100101

array1
0101010100100101

array2
0x87654321

malloc()
10101011001010110101

Zásobník

Halda

- Paměťové bloky na haldě zůstávají do zavolání free()

Memory leaks

- Dynamicky alokovaná paměť musí být uvolněna
 - dealokace musí být explicitně provedena vývojářem
 - (C nemá Garbage collector)
- Valgrind – nástroj pro detekci memory leaks (mimo jiné)
 - `valgrind -v --leak-check=full testovaný_program`
 - lze využít např. Eclipse Valgrind plugin
 - není funkční port pro Windows ☹
- Microsoft Visual Studio
 - automaticky zobrazuje detekované memory leaks v debug režimu
- Detekované *memory leaks* ihned odstraňujte
 - stejně jako v případě warningu
 - nevšimnete si jinak nově vzniklých
 - obtížně dohledáte místo alokace

Memory leaks – demo

- Násobná alokace do stejného ukazatele
- Dealokace nad modifikovaným ukazatelem

```
#include <stdlib.h>
int main(int argc, char* argv[]) {
    int* pArray = NULL;
    const int arrayLen = 25;

    pArray = malloc(arrayLen * sizeof(int));
    if ((argc == 2) && atoi(argv[1]) == 1) {
        pArray = malloc(arrayLen * sizeof(int));
    }
    if (pArray) { free(pArray); pArray = NULL; }

    int* pArray2 = 0;
    pArray2 = malloc(arrayLen * sizeof(int));
    pArray2 += 20; // chyba
    if (pArray2) { free(pArray2); pArray2 = NULL; }

    return 0;
}
```

násobná alokace

dealokace s modifikovaným ukazatelem

Předávání hodnotou a hodnotou ukazatele

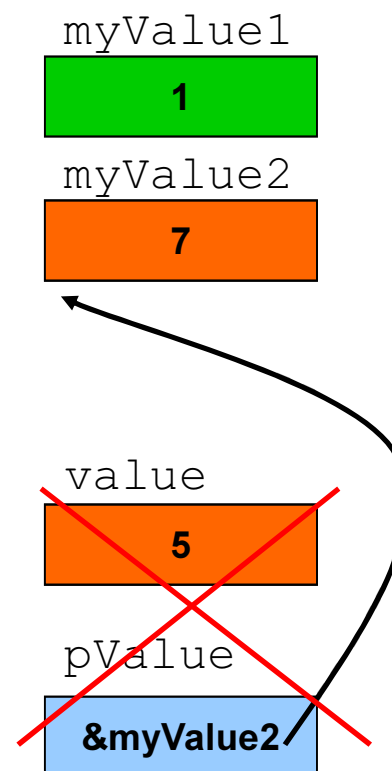
Zásobník

```
int main() {
    int myValue1 = 1;
    int myValue2 = 1;

    valuePassingDemo(myValue1, &myValue2);

    return 0;
}

void valuePassingDemo(int value, int* pValue) {
    value = 5;
    *pValue = 7;
}
```



- Proměnná value i pValue zaniká, ale zápis do myValue2 zůstává

Předávání alokovaného pole z funkce

Neuvolněná paměť

- Jak předat pole dyn. alokované ve funkci?
 - ukazatel na ukazatel

```
int main() {  
    int* pMyPointer = NULL;  
    allocateArrayDemo(pMyPointer, &pMyPointer);  
    free(pMyPointer);  
  
    return 0;  
}  
  
void allocateArrayDemo(int* pArray, int** ppArray) {  
    // address will be lost, memory leak  
    pArray = malloc(10 * sizeof(int));  
    // warning: integer from pointer - BAD  
    *pArray = malloc(10 * sizeof(int));  
    // OK  
    *ppArray = malloc(10 * sizeof(int));  
}
```

Zásobník

Halda

pMyPointer

0x4733487

01010100011

pArray

0x3287643

01010100011

ppArray

&pMyPointer

Dynamická alokace vícedimenz. polí

- 2D pole je pole ukazatelů na 1D pole
- 3D pole je pole ukazatelů na pole ukazatelů na 1D pole

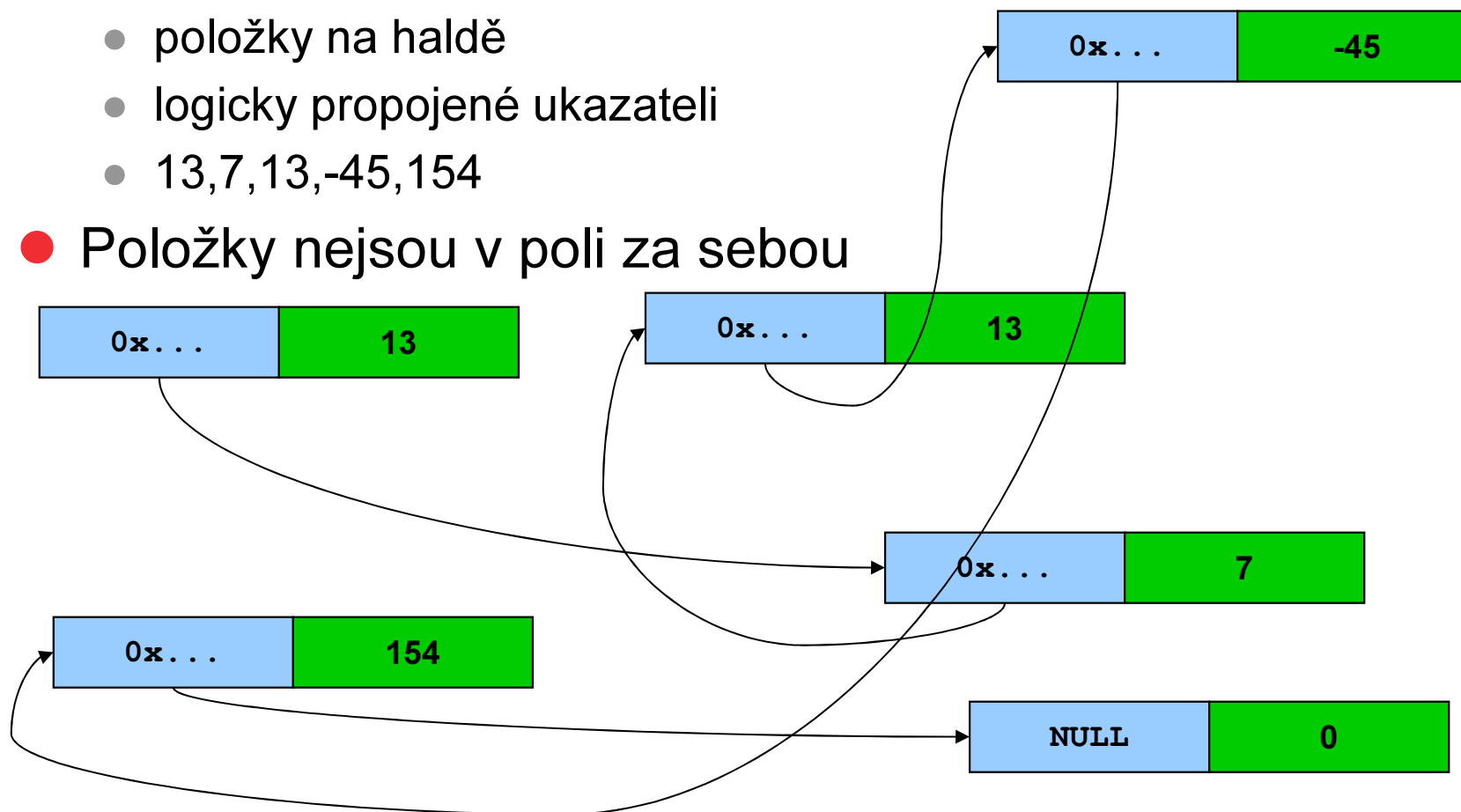
```
void allocate2DArray() {  
    // 100 items array with elements int*  
    const int LEN = 100;  
    int** array2D = malloc(LEN * sizeof(int*));  
    for (int i = 0; i < LEN; i++) {  
        // allocated 1D array on position array2D[i]  
        // length is i+1 * 10  
        array2D[i] = malloc((i+1) * 10 * sizeof(int));  
    }  
    array2D[10][20] = 1;  
  
    // You need to free memory properly  
    for (int i = 0; i < LEN; i++) {  
        free(array2D[i]); array2D[i] = NULL;  
    }  
    free(array2D); array2D = NULL;  
}
```

Dynamická alokace – zřetěžený seznam

- Struktura s „neomezenou“ velikostí

- položky na haldě
- logicky propojené ukazateli
- 13,7,13,-45,154

- Položky nejsou v poli za sebou



Dynamická alokace - poznámky

- Při dealokaci nastavte proměnnou zpět na NULL
 - opakovaná dealokace nezpůsobí pád
 - validitu ukazatele nelze testovat, ale hodnotu NULL ano
- Dynamická alokace je důležitý koncept
 - je nutné znát a rozumět práci s ukazateli
- Dynamicky alokovanou paměť přiřazujeme do ukazatele
 - sizeof(pointer) vrátí velikost ukazatele, nikoli velikost pole!
- Dynamická (de-)alokace nechává paměť v původním stavu
 - neinicializovaná paměť & zapomenuté klíče

Alokace paměti - typy

1. Paměť alokovaná v době překladač s pevnou délkou ve statické sekci
 - typicky konstanty, řetězce, konstantní pole
 - `const char* hello = "Hello World";`
 - délka známa v době překladač
 - alokováno ve statické sekci programu (lze nalézt v nespuštěném programu)
2. Paměť alokovaná za běhu na zásobníku, délka známa v době překladač
 - lokální proměnné, lokální pole
 - paměť je alokována a dealokována automaticky
 - `int array[10];`
3. Paměť alokovaná za běhu na zásobníku, délka není známa v době překladač
 - variable length array, od C99
 - paměť je alokována a dealokována automaticky
 - `int array[userGivenLength];`
4. Paměť alokovaná za běhu na haldě, délka není známa v době překladač
 - alokace i dealokace explicitně prostřednictvím funkcí malloc a free
 - programátor musí hlídat uvolnění, jinak memory leak
 - `int* array=malloc(userGivenLength*sizeof(int)); free(array);`

Shrnutí

- Argumenty funkce `main()` umožňují přijmout informace z příkazové řádky
- Explicitní vs. Implicitní typové konverze
- Dynamická alokace je velmi důležitý praktický koncept
 - Většina paměti je typicky dynamicky alokovaná
 - V C je programátor odpovědný za uvolnění paměti
- Dynamicky alokované struktury jsou typicky náročnější na ladění
 - Defensivní programování, výpisy, debugger