PB071 – Principy nízkoúrovňového programování

Dynamická alokace

Slidy pro komentáře (děkuji!):
https://drive.google.com/file/d/1wJqW1sPRj63hIeIN_TTfBmFFUfnul9xd/view?usp=drive_link

Dynamická alokace

Dynamická alokace - motivace

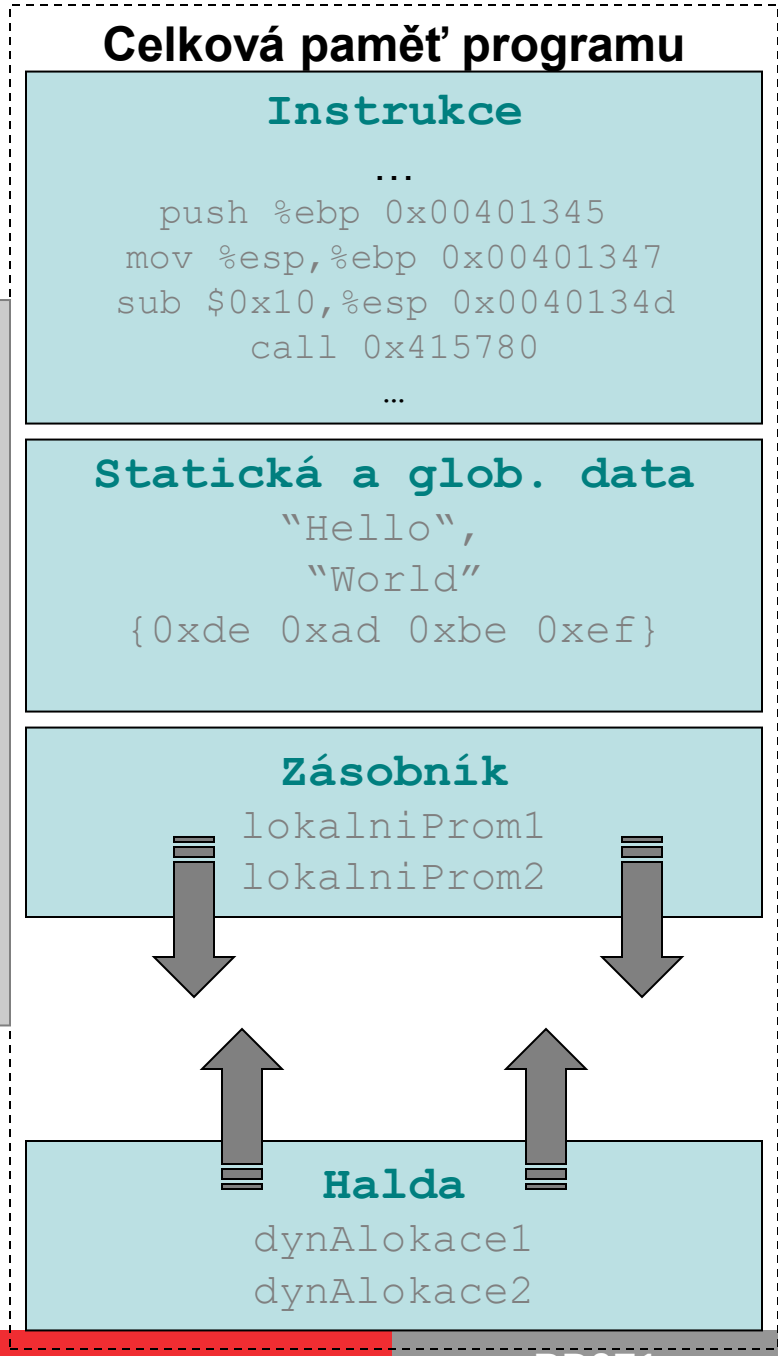
- U statické alokace je velikost nebo počet známa v době překladu
  - `int array[100];`
  - `int a, b, c;`
- Často ale tyto informace nejsou známy
  - např. databáze se rozšiřuje
  - např. textový popisek má typicky 20 znaků, ale může mít i 1000
    - neefektivní mít vždy alokováno maxiumum 1000 znaků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

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.

Statická alokace, Pole s proměnnou délkou (VLA)

Dynamická alokace

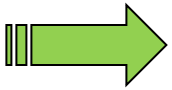


Jakým směrem roste zásobník a halda?

- „Učebnicový“ růst zásobníku je „zdola“ „nahoru“
  - Co to ale znamená pro adresy uložení proměnných?
  - Např. budou lokální proměnné main() na nižších nebo vyšších adresách než proměnné funkce foo() volané z main?
    - Rozložení lokálních proměnných v rámci dané funkce je čistě na překladači
  - Je přesnější se bavit o růstu vhledem k adresám
- Záleží na platformě a překladači
  - Pro danou platformu je konzistentní (i přesto mohou být vzácné výjimky)
  - Růst směrem dolů je častější
    - main() bude na vyšší adrese než foo()
    - x86, PowerPC, MIPS, SPARC, EE, Cell SPU architektury
  - https://stackoverflow.com/questions/1677415/does-stack-grow-upward-or-downward
- Lze snadno zjistit (porovnání adres), vyzkoušejte si
  - Lokální proměnná v main a následně v zavolané fukci foo
  - Adresa získaná pomocí malloc() + globální proměnná + adresa funkce

Statická vs. dynamická alokace

- Statická alokace alokuje na zásobníku (stack)
  - automaticky se „uvolňuje” po dokončení bloku kódu
    - typicky konec funkce, konec cyklu for…
    - (paměť není explicitně uvolněna, znovu se použije po změně %esp)
  - výrazně rychlejší
    - Na zásobníku nevzniká fragmentace, snadné „uvolnění“ paměti
    - Paměť u vrcholu zásobníku je typicky v cache
  - využití pro krátkodobé proměnné/objekty
  - `int array[100];`
- Dynamická alokace na haldě (heap)



  - zůstává do explicitního uvolnění (nebo typicky do 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];`
  - Od C11 nepovinné (__STDC_NO_VLA__), (spíše nepoužívejte)

** C negarantuje po ukončení aplikace uvolnění paměti (záleží na OS)*

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“ v paměti)
  - pokud se nezdaří, tak vrací NULL
- **void*** calloc**(**size_t n**,** size_t sizeItem**)**
  - obdobné jako malloc
  - ale alokuje n * sizeItem
  - + inicializuje paměť na 0

Uvolňování alokované paměti

- **void** free**(void*)**
  - uvolní na haldě alokovanou paměť
  - musí být ukazatel vrácený předchozím malloc(), calloc() nebo realloc()
    - nelze uvolnit jen část paměti (např. od ukazatele modifikovaného pomocí ukazatelové aritmetiky)
- Pozor na přístup na paměť po zavolání free(paměť)
  - paměť již může být přidělena jiné proměnné
  - Tzv. use-after-free zranitelnost (exploits)
- Pozor, free() nemaže obsah paměti
  - citlivá data dobré předem smazat (klíče, osobní údaje)
- Je vhodné nastavit uvolněný ukazatel na NULL
  - Opakované free je korektní, pokud je argument NULL
  - Přístup na adresu je sice stále nevalidní, ale nepřepisuje používaná data

Změna velikosti alokované paměti

- **void** * realloc **(** **void** * ptr**,** size_t size **);**
  - pokud je ptr == NULL, tak stejně jako malloc()
  - pokud je ptr != NULL, tak změní velikost alokovaného paměťového bloku na hodnotu size
  - pokud size == 0, tak v POSIXu[1] 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(), obsahuje předchozí „smetí“

[1] Garantováno v POSIXu, ve standardu C není specifikované (=> závisí na překladači)

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()
  - pozor na sizeof() nad ukazatelem z malloc!

```
void* memset(void* ptr, int value, size_t num);


int array[100];
memset(array, 0, 100 * sizeof(int));
```

Dynamická alokace - shrnutí

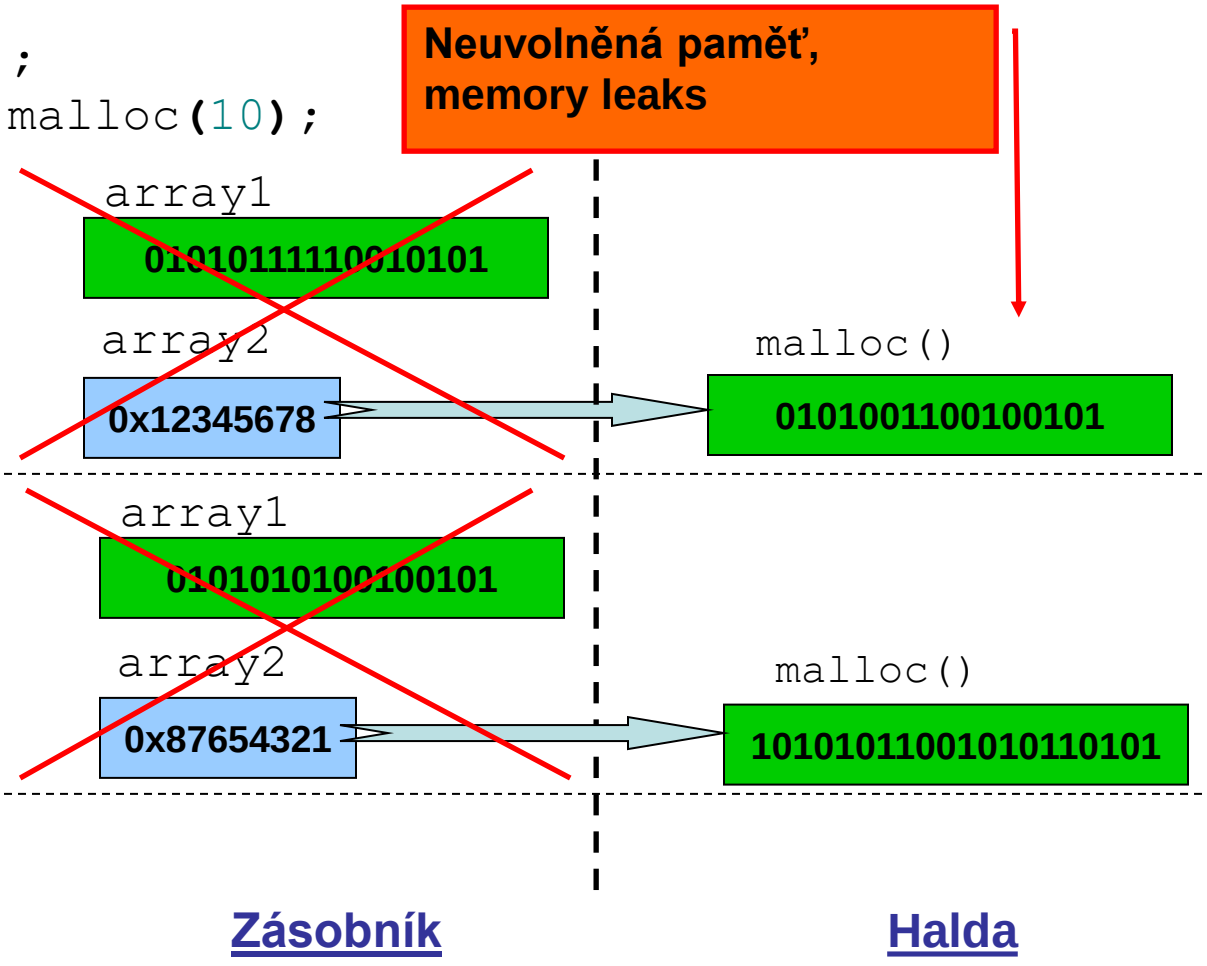
1. Alokuje potřebný počet bajtů
   - (musíme správně vypočíst)
   - typicky používáme `počet_prvků*sizeof(prvek)`
2. Uložíme adresu paměti do ukazatele s požadovaným typem
   - `int* array = malloc(100 * sizeof(int));`
3. Po konci práce uvolníme
   - `free(array);`

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;
}
```



- 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
  - Na Windows 10 lze využít Linux subsystem pro spuštění Valgrindu
    - http://www.albertgao.xyz/2016/09/28/how-to-use-valgrind-on-windows/
- Na Windows použijte Dr. Memory
  - https://drmemory.org/
- 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

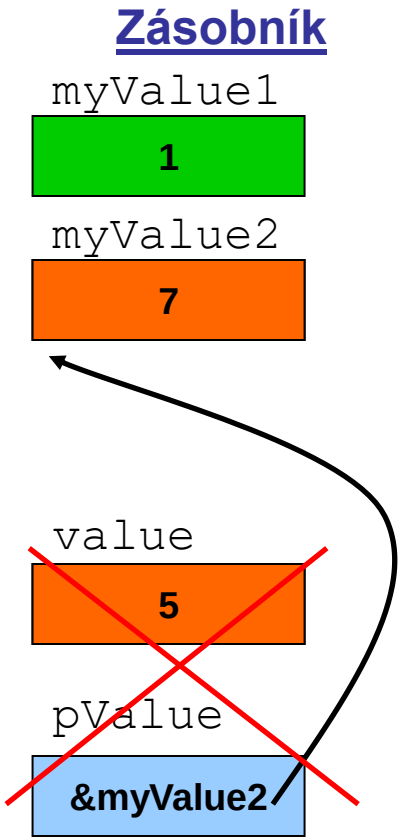
Předávání hodnotou a hodnotou ukazatele

```
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

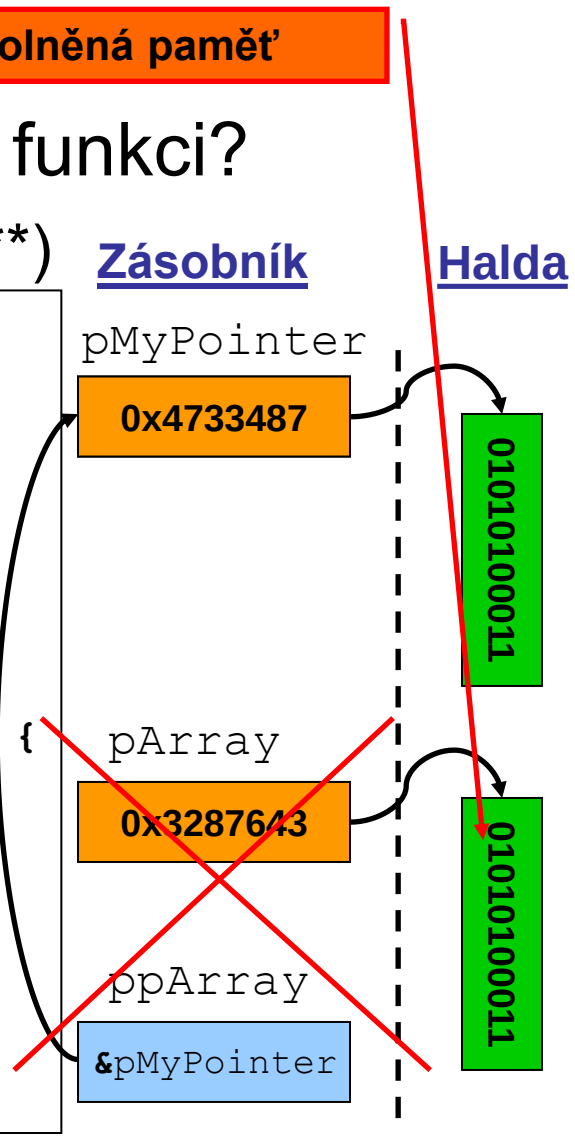
Předávání alokovaného pole z funkce

- Jak předat pole dyn. alokované ve funkci?
  - ukazatel na ukazatel (schránka typu **)

```
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));
}
```



Předávání alokovaného pole z funkce

- 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));
}
```

Dynamická alokace vícedimenz. polí

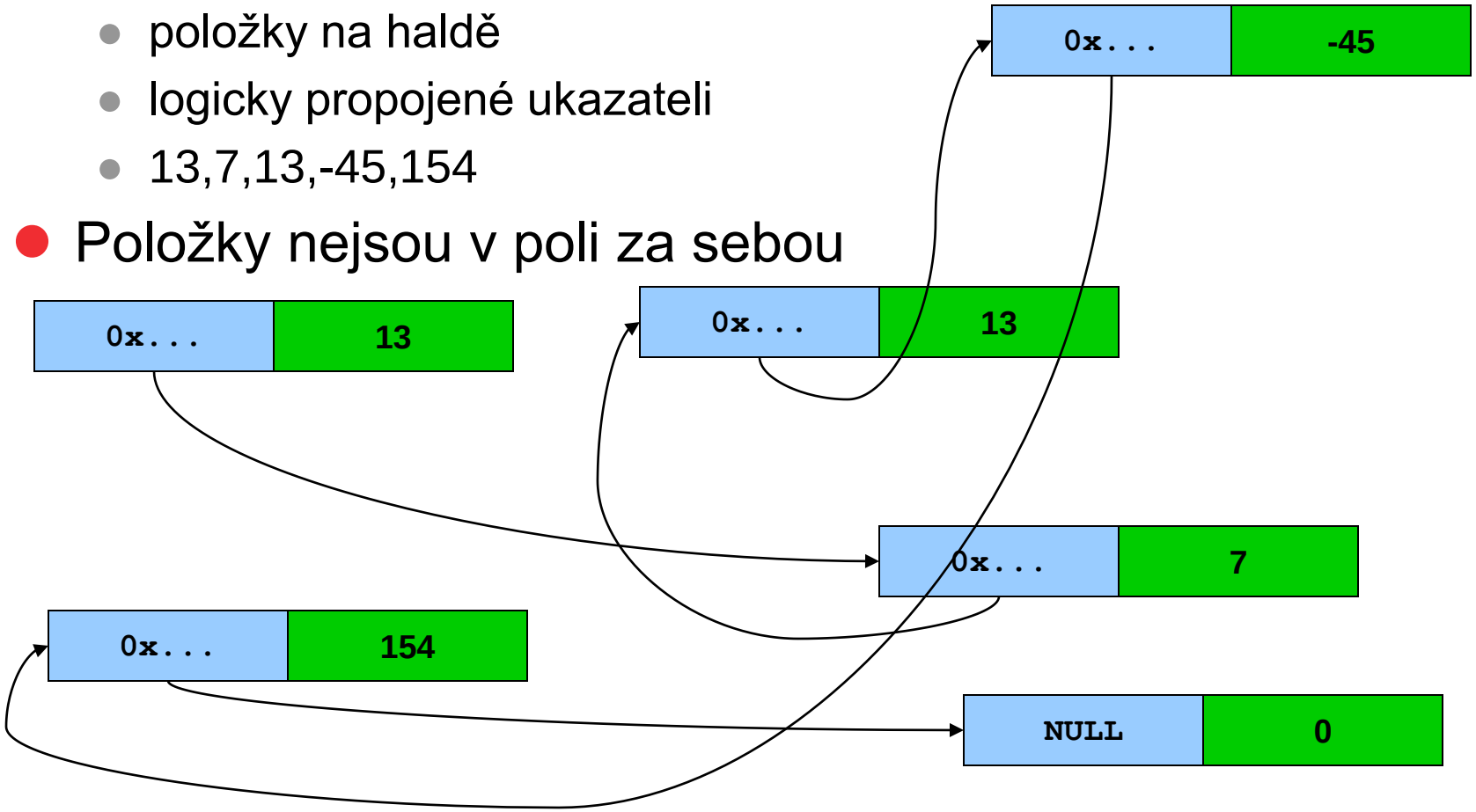
- 2D pole je pole ukazatelů na 1D pole
- 3D pole je pole ukazatelů na pole ukazatelů na 1D pole

```
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 – zřetězený 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) vrací velikost ukazatele, nikoli velikost pole!
- Dynamická (de-)alokace nechává paměť v původním stavu
  - neinicializovaná paměť & zapomenuté sensitivní infomace (hesla, klíče...)

Alokace paměti - typy

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

https://www.fi.muni.cz/pb071/man/#valgrind

VALGRIND

Valgrind http://www.valgrind.org/

- Sada více nástrojů (**`valgrind --tool=<toolname>`**)
- Memcheck – nástroj pro analýzu paměti
  - Nejčastěji používaný nástroj
  - Nahradí standardní paměťový alokátor C vlastní implementací, která kontroluje memory leaks, zápis za konec pole…
  - Nekorektní ukazatele, neuzavřené soubory, neinicializované proměnné
  - http://www.valgrind.org/docs/manual/mc-manual.html
- Massif – analyzátor použití haldy
- Hellgrind – detekce problému s více souběžnými vlákny
- Callgrind – generování grafu volání
- ...

Valgrind – hlavní přepínače

- Kompilujte s ladícími symboly
  - `gcc -std=c99 -Wall -g -o program program.c`
  - Valgrind později může zobrazit více kontextových informací (např. jméno problematické funkce)
- Spusťte cílový program skrze Valgrind
  - `valgrind <options> ./program`
  - Lze předat i argumenty programu
    - `valgrind -v --leak-check=full ./program arg1 arg2`
- Zobrazit neuzavřené popisovače souborů (file descriptors)
  - `--track-fds=yes`
- Pokud chcete sledovat i pod-procesy
  - `--trace-children=yes`
  - (Potřebné pro vícevláknové programy)

Valgrind – hlavní přepínače

- Detailní report o memory leaks checks
  - `--leak-check=full`
- Memory leaks
  - *Definitely lost*: paměť je přímo ztracena (neexistuje žádný ukazatel)
  - *Indirectly lost*: odkazují pouze ukazatele z již ztracené paměti
  - *Possibly lost:* adresa může někde existovat, ale Valgrind si není jistý (typicky ale jde o reálný leak)

Memcheck – neinicializované proměnné

- Detekce použití neinicializovaných promněných
  - `--undef-value-errors=yes` (default)
- Zjistit odkud neinicializovaná proměnná pochází
  - `--track-origins=yes`
  - pozor, zpomalí výrazně běh aplikace

Memcheck – nevalidní čtení a zápisy

- Zápis mimo alokovanou paměť („buffer overflow“)
- Pozor, pouze pro paměť na haldě (malloc())
- Nedetekuje problémy na zásobníku a globální proměnné
  - https://en.wikipedia.org/wiki/Valgrind#Limitations_of_Memcheck
- Zápis do již de-alokované paměti (po free())
  - Valgrind se snaží odložit vlastní dealokaci co nejdéle, aby dokázal detekovat následné (nekorektní) zápisy/čtení do této paměti

Ukázka problému a interpretace Valgrindu

```c
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    int notInit;
    if (notInit == 1) printf("hmm");
    int *leak = malloc(sizeof(int));
    leak[2] = 0;
    return 0;
}
```

- gcc -g -o leak leak.c
- valgrind --leak-check=full --show-reachable=yes ./leak

```
/home/xsvenda: valgrind --leak-check=full --show-reachable=yes ./leak
==57063== Memcheck, a memory error detector
==57063== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==57063== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==57063== Command: ./leak
==57063==
==57063== Conditional jump or move depends on uninitialised value(s)
==57063==    at 0x400569: main (leak.c:6)
==57063==
==57063== Invalid write of size 4
==57063==    at 0x400590: main (leak.c:8)
==57063==  Address 0x5203048 is 4 bytes after a block of size 4 alloc'd
==57063==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==57063==    by 0x400583: main (leak.c:7)
==57063==
==57063==
==57063== HEAP SUMMARY:
==57063==     in use at exit: 4 bytes in 1 blocks
==57063==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==57063==
==57063== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==57063==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==57063==    by 0x400583: main (leak.c:7)
==57063==
==57063== LEAK SUMMARY:
==57063==    definitely lost: 4 bytes in 1 blocks
==57063==    indirectly lost: 0 bytes in 0 blocks
==57063==      possibly lost: 0 bytes in 0 blocks
==57063==    still reachable: 0 bytes in 0 blocks
==57063==         suppressed: 0 bytes in 0 blocks
==57063==
==57063== For counts of detected and suppressed errors, rerun with: -v
==57063== Use --track-origins=yes to see where uninitialised values come from
==57063== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

Použití neinicializované proměnné (řádek 6, leak.c)

```
1: #include <stdlib.h>
2: #include <stdio.h>
3: int main(void)
4: {
5:   int notInit;
6:   if (notInit == 1) printf("hmm");
7:   int *leak = malloc(sizeof(int));
8:   leak[2] = 0;
9:   return 0;
}
```

Nekorektní zápis o délce 4 bajty, alokováno na řádku 7 v souboru leak.c

Detekován memory leak o délce 4 bajty

Místo alokování (řádek 7, leak.c)

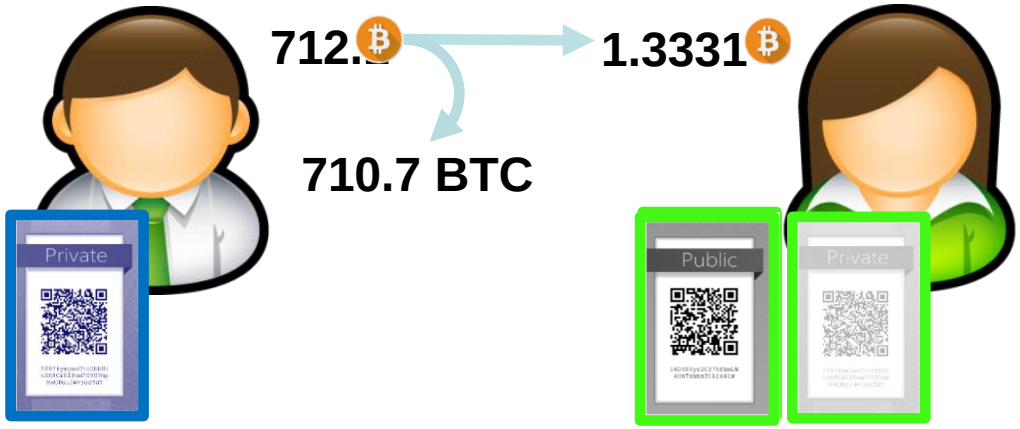
Pokud chceme znát místo neinicializované proměnné, přidáme přepínač --track-origins=yes

DÍKY ZA VÁŠ ČAS A V PONDĚLÍ ZASE NA VIDĚNOU

Bonus ☺

CTxIn

CTxOut
CTxOut



Jak vydělat OBROVSKÉ peníze (přetečením datového typu)

- Bitcoin block 74638 (15. Srpen 2010)

Coinbase transakce s odměnou pro minera

CBlock(hash=000000000790ab3, ver=1, hashPrevBlock=0000000006
nTime=1281891957, nBits=1c00800e, nNonce=28192719, vtx=2)
CTransaction(=1, vout.size=1, nLockTime=
CTxIn(COut 0e80001c028f00)
CTxOut(nVal ey=0x4F4BA55D1580F8C3A8
CTransaction(hash=1d5e51, ver=1, vin.size=1, vout.size=2, nLockTime=
CTxIn(COutPoint(237fe8, 0), scriptSig=0xA87C02384E1F184B79C6AC)
CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_HA
CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_HA
vMerkleTree: 012cd8 1d5e51 618eba

Vstupní transakce (s 0.5BTC)
https://blockexplorer.com/tx/237fe8348fc77ace1104993
1058abb034c99698c7fe99b1cc022b1365a705d39

2 výstupní transakce (každá na $9*10^{10}$ BTC!!!)

Block hash: 0000 bddeb97c67a6f
Transaction hash e9598ad67e7408077a

Mělo být zamítnuto těžaři protože hodnota výstupů >> hodnota vstupů. Ale nebyla…

Rozbor chyby

- Připomenutí: částka v Bitcoinu je reprezentována jako celé číslo s fixní desetinnou částkou, 8 bajtů (INT64)
  - 0.5 BTC == 0.5 * 10^8 => 50000000
- CTxOut value:92233720368.54275808 BTC
  = 0x7fffffffffff85ee0
- INT64_MAX = 0x7fffffffffffffff
- Součet 2 CTx = 0xffffffffffff0bdc0 (přetečení)
  = -1000000_{10} = -0.01BTC
- Rozdíl mezi součtem vstupů a výstupů je interpretován jako poplatek pro minera
- Nedokonalá kontrola podtečení
  - Je součet výstupů menší než vstup? (Ano, -0.01 je menší než 0.5)
  - Dostává miner nenulový poplatek? (Ano, 0.51 BTC)

Fix – kontrola přetečení maximální hodnoty

https://github.com/bitcoin/bitcoin/commit/d4c6b90ca3f9b47adb1b2724a0c3514f80635c84#diff-118fcbaaba162ba17933c7893247df3aR1013

11 main.h — View

```
@@ -18,6 +18,7 @@ static const unsigned int MAX_SIZE = 0x02000000;
 static const unsigned int MAX_BLOCK_SIZE = 1000000;
 static const int64 COIN = 100000000;
 static const int64 CENT = 1000000;
+static const int64 MAX_MONEY = 21000000 * COIN;
 static const int COINBASE_MATURITY = 100;

 static const CBigNum bnProofOfWorkLimit(~uint256(0) >> 32);
@@ -471,10 +472,18 @@ class CTransaction
         if (vin.empty() || vout.empty())
             return error("CTransaction::CheckTransaction() : vin or vout empty");

-        // Check for negative values
+        // Check for negative or overflow output values
+        int64 nValueOut = 0;
         foreach(const CTxOut& txout, vout)
+        {
             if (txout.nValue < 0)
                 return error("CTransaction::CheckTransaction() : txout.nValue negative");
+            if (txout.nValue > MAX_MONEY)
+                return error("CTransaction::CheckTransaction() : txout.nValue too high");
+            nValueOut += txout.nValue;
+            if (nValueOut > MAX_MONEY)
+                return error("CTransaction::CheckTransaction() : txout total too high");
+        }

         if (IsCoinBase())
         {
```