PB071 – Principy nízkoúrovňového programování

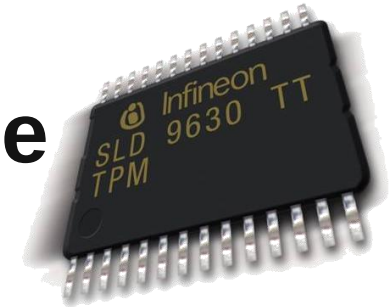
Pole, vícerozměrné pole, ukazatelová aritmetika, typový systém

Slidy pro komentáře (děkuji!):
https://drive.google.com/file/d/1qLh9WTTrF3nhVpLRrObhrqhCzcpZVk36/view?usp=sharing

Obsah přednášky

- Jak pracovat efektivně s více stejnými objekty stejného typu
  - Jednorozměrné pole
- Jak využít kreativně ukazatele s poli
  - Ukazatelová aritmetika
  - (A mít něco co Java a Python nemá ☺)
- Proč je explicitní uvedení typu proměnné a jeho kontrola důležitá
  - Typový systém

Trusted Platform Module



TPM

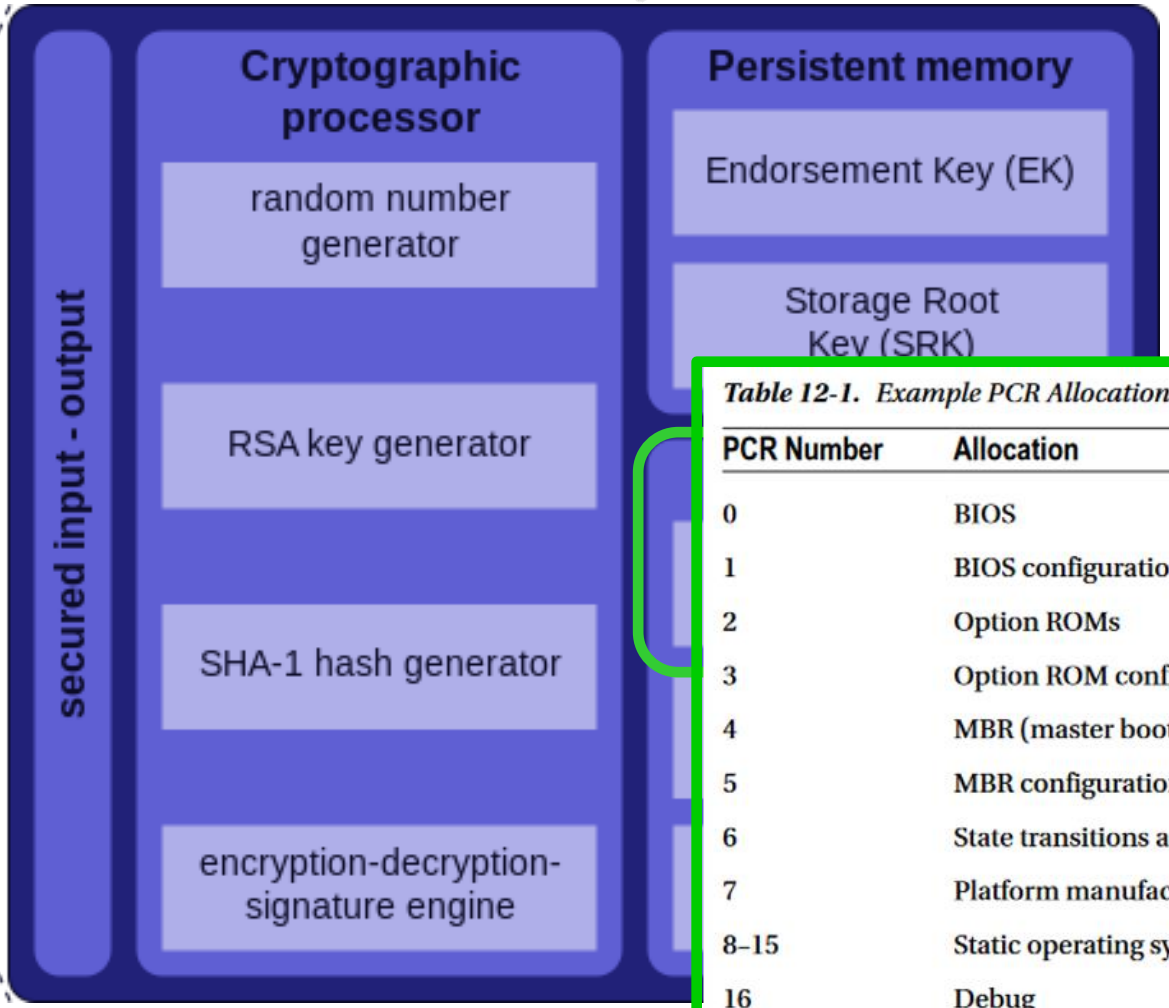
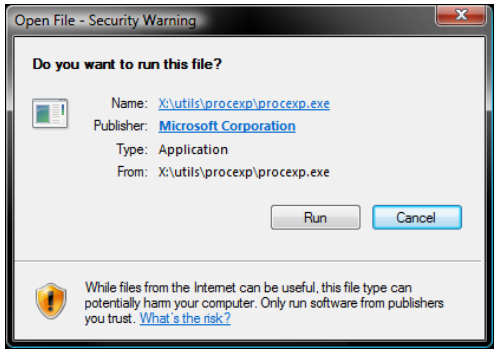


Table 12-1. Example PCR Allocation

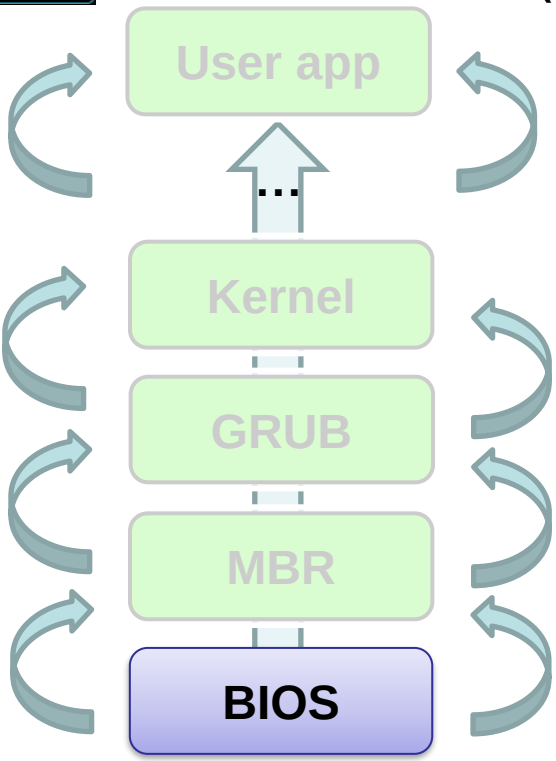
PCR Number	Allocation
0	BIOS
1	BIOS configuration
2	Option ROMs
3	Option ROM configuration
4	MBR (master boot record)
5	MBR configuration
6	State transitions and wake events
7	Platform manufacturer specific measu
8–15	Static operating system
16	Debug
23	Application support

“Verifikovaný” boot “Měřený” boot



1. Hodnota PCR musí být chráněna (malware by jinak přepsal na falešnou)
2. Hodnota PCR může být podepsána (důkaz, že PCR nabyl danou hodnotu)

PCR = H(…H(H(0|H(MBR))|H(GRUB)…H(User app))



VERIFY (RSA)

MEASURE: PCR = H(PCR | H(User app))

VERIFY (RSA)

MEASURE: PCR = H(PCR | H(Kernel))

VERIFY (RSA)

MEASURE: PCR = H(PCR | H(GRUB))

VERIFY (RSA)

MEASURE: PCR = H(PCR | H(MBR))

RESET: PCR = 0

https://crocs.fi.muni.cz/tpm

Prosba o pomoc v experimentu

- Sbíráme jen veřejná/testovací data!
- Pro uživatele MS Windows

  - Dlouhodobý sběr PCR registrů a jejich vývoj v čase
  - Stáhněte TPM_PCR.exe v0.1.8 z https://github.com/petrs/TPM_PCR/releases
  - Uložte do adresáře (např. Documents\TPM), spusťte, potvrďte Y(es)
  - 1x denně (19:00) uloží PCR do souboru (nechte běžet)
  - Po spuštění zašlete **`PCR_measurements.zip`** na svenda@fi.muni.cz
- Pro uživatele Linux i Windows

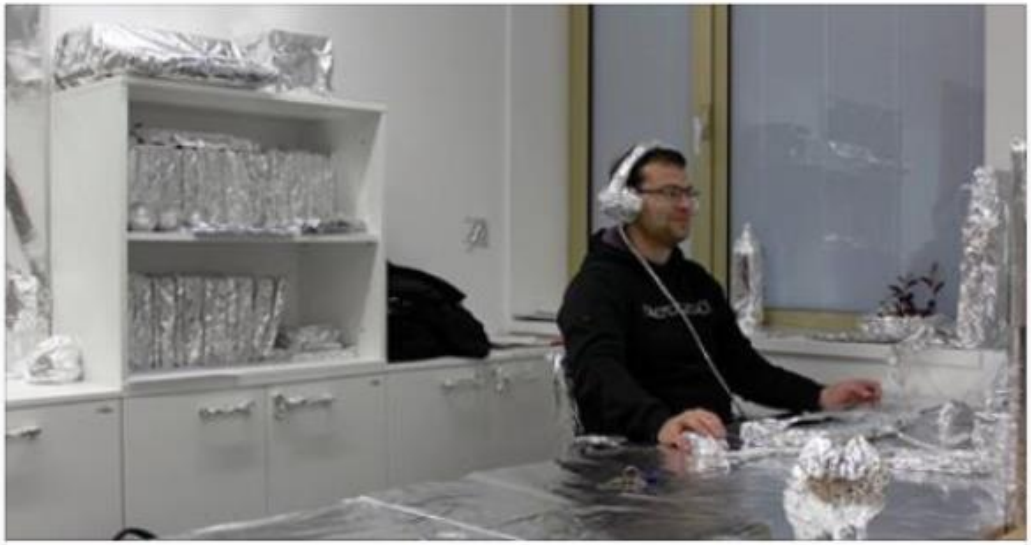
  - Analýza rychlosti TPM čipu a testovacích klíčů
  - Vytvoříte si bootovací USB (Fedora + nástroj tpmalgest)
  - Spustíte a necháte běžet (přes noc)
  - Po dokončení zašlete **`algtest_xxx.zip`** na svenda@fi.muni.cz
  - Detailní popis na https://crocs.fi.muni.cz/tpm

Petr attached FB_post_TPM_20161220.png to this card Dec 29, 2023 at 2:49 PM

Fakulta informatiky - FI MUNI
14 hrs ·

Petr Švenda, učitel a výzkumník na #fimuni, se na vás chce obrátit s prosbou o pomoc ve výzkumu bezpečnosti kryptografického hardware. Detaily a postup najdete na https://crocs.fi.muni.cz/public/research/tpm. Tento dobrý skutek vám nezabere víc jak 2 minuty, takže se na to můžete hned vrhnout 🙂

Máme zájem o co nejvíce různých výsledků - neváhejte tedy prosím přeposlat tuto prosbu dalším lidem. Děkujeme!



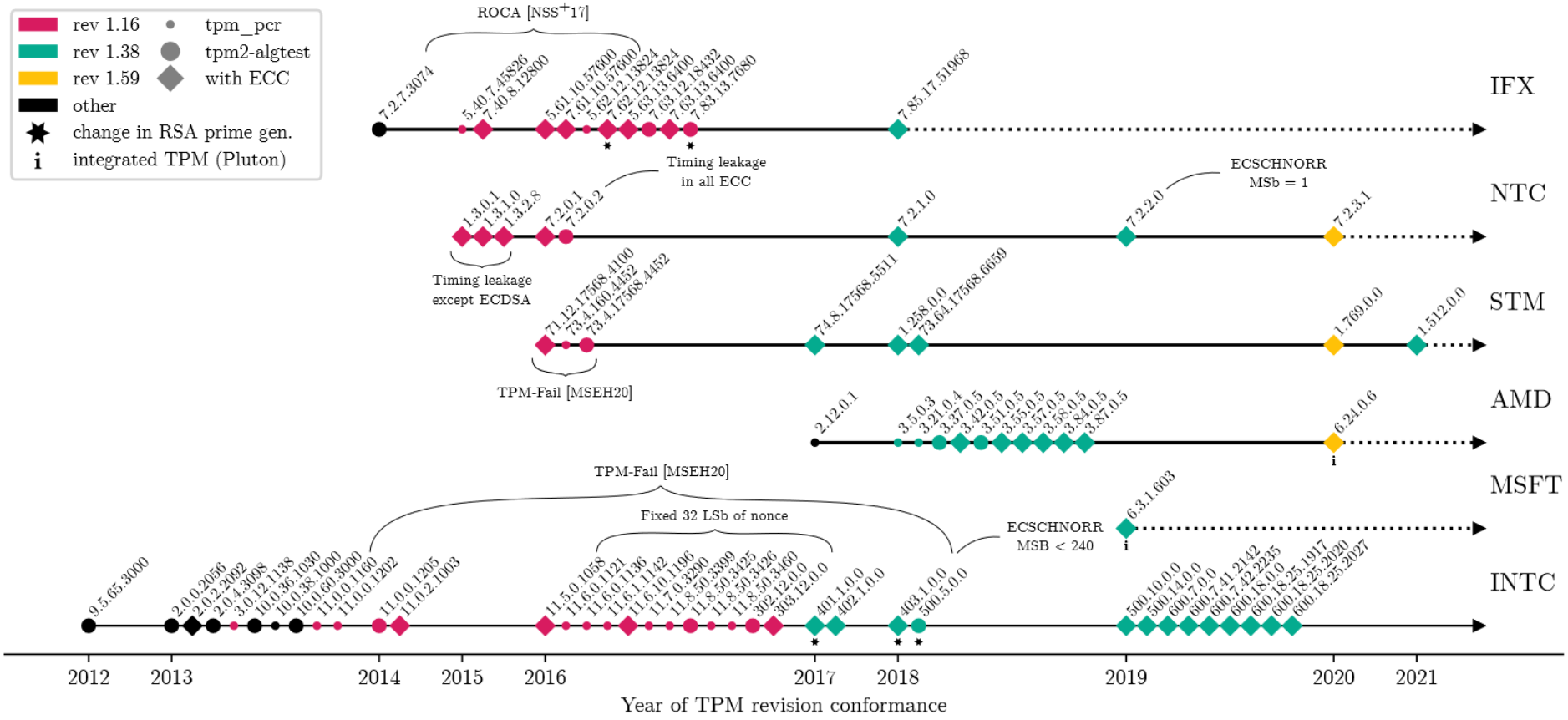
Trusted Platform Module (TPM) [CRoCS wiki]

Trusted Platform Module (TPM) V rámci laboratoře CRoCS na Fakultě Informatiky provádíme výzkum bezpečnosti kryptografického hardware TPM (Trusted Platform Module), který je nyní umisťován do většiny notebooku a...

CROCS.FI.MUNI.CZ

TPMScan paper 2024 (po 7 letech sběru)

- https://crocs.fi.muni.cz/_media/publications/pdf/2024-ches-tpmscan.pdf



https://crocs.fi.muni.cz/tpm

Prosba o pomoc v experimentu

- Sbíráme jen veřejná/testovací data!
- Pro uživatele MS Windows

  - Dlouhodobý sběr PCR registrů a jejich vývoj v čase
  - Stáhněte TPM_PCR.exe v0.1.8 z https://github.com/petrs/TPM_PCR/releases
  - Uložte do adresáře (např. Documents\TPM), spusťte, potvrďte Y(es)
  - 1x denně (19:00) uloží PCR do souboru (nechte běžet)
  - Po spuštění zašlete **`PCR_measurements.zip`** na svenda@fi.muni.cz
- Pro uživatele Linux i Windows

  - Analýza rychlosti TPM čipu a testovacích klíčů
  - Vytvoříte si bootovací USB (Fedora + nástroj tpmalgest)
  - Spustíte a necháte běžet (přes noc)
  - Po dokončení zašlete **`algtest_xxx.zip`** na svenda@fi.muni.cz
  - Detailní popis na https://crocs.fi.muni.cz/tpm

Práce s poli

Jednorozměrné pole

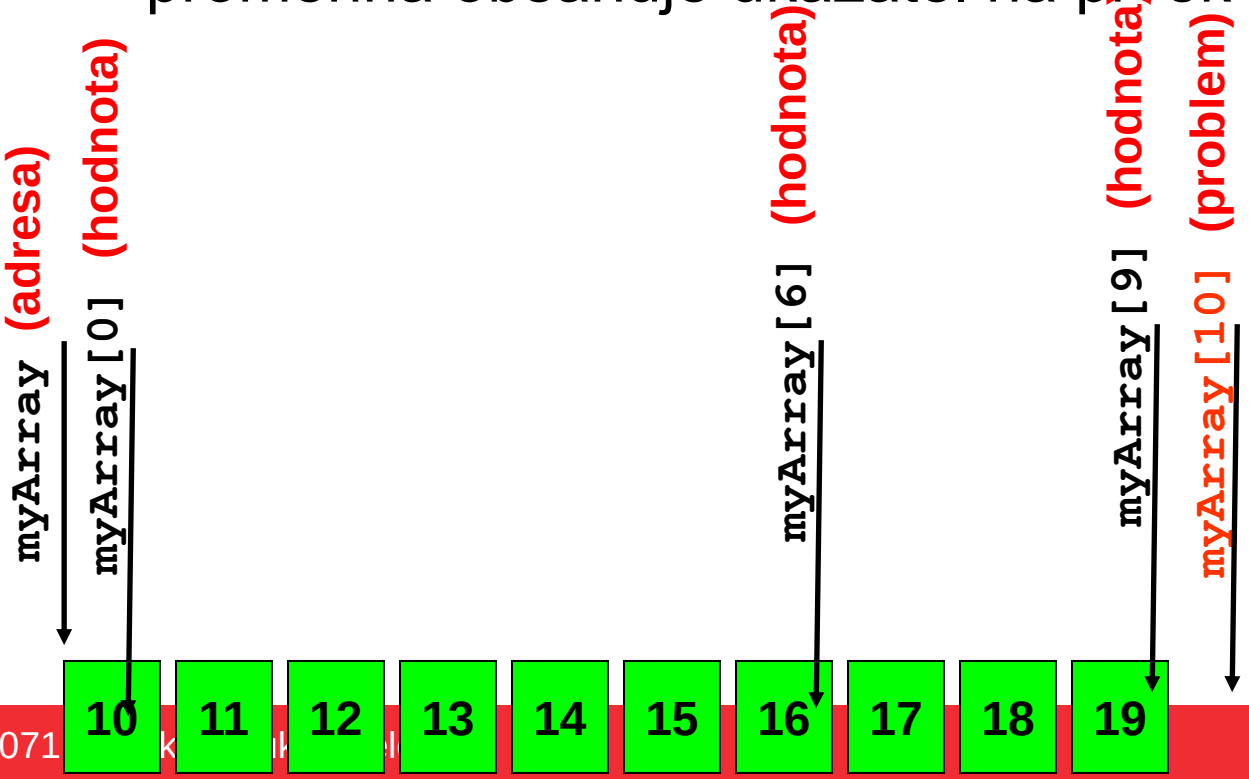
Jednorozměrné pole

- Jednorozměrné pole lze implementovat pomocí ukazatele na souvislou oblast v paměti
  - jednotlivé prvky pole jsou v paměti za sebou
  - první prvek umístěn na paměťové pozici uchovávané ukazatelem
- Prvky pole jsou v paměti kontinuálně za sebou
- Deklarace: `datový_typ jméno_proměnné[velikost];`
  - velikost udává počet prvků pole

```
int myArray[10];
for (int i = 0; i < 10; i++) {myArray[i] = i+10;}
```

Jednorozměrné pole

- Syntaxe přístupu: `jméno_proměnné[pozice];`
  - na prvek pole se přistupuje pomocí operátoru `[]`
  - indexuje se od 0, tedy n-tý prvek je na pozici `[n-1]`
  - proměnná obsahuje ukazatel na prvek `[0]`



Zjištění velikosti pole

- Jak zjistit, kolik prvků se nachází v poli?
- Jak zjistit, kolik bajtů je potřeba na uložení pole?
- Jazyk C obecně neuchovává velikost pole
  - Např. Java uchovává prostřednictvím **`pole.length`**
- Velikost pole si proto musíme pamatovat
  - Dodatečná proměnná
- (V některých případech lze využít operátor **`sizeof`**)
  - Pozor, ne u ukazatelů - vrátí velikost ukazatele, ne pole
  - Pozor, funguje jen u pole deklarovaného s pevnou velikostí
  - Pozor, nefunguje u pole předaného do funkce
  - Nespoléhejte, pamatujte si velikost v separátní proměnné.

Využití operátoru sizeof()

- `sizeof(pole)`
  - velikost paměti obsazené polem v bajtech
  - funguje jen pro statická pole, jinak velikost ukazatele
- `sizeof(ukazatel)`
  - velikost ukazatele (typicky 4 nebo 8 bajtů)

Pozor, pole v C nehlídá meze!

- čtení/zápis mimo alokovanou paměť může způsobit pád nebo nežádoucí změnu jiných dat
-
```
int array[10]; array[100] = 1; // undefined behavior,
                               // likely runtime exception
                               // (like SIGSEGV)
```

Připomenutí: `sizeof(type)` == `sizeof`(proměnná toho `typu`)

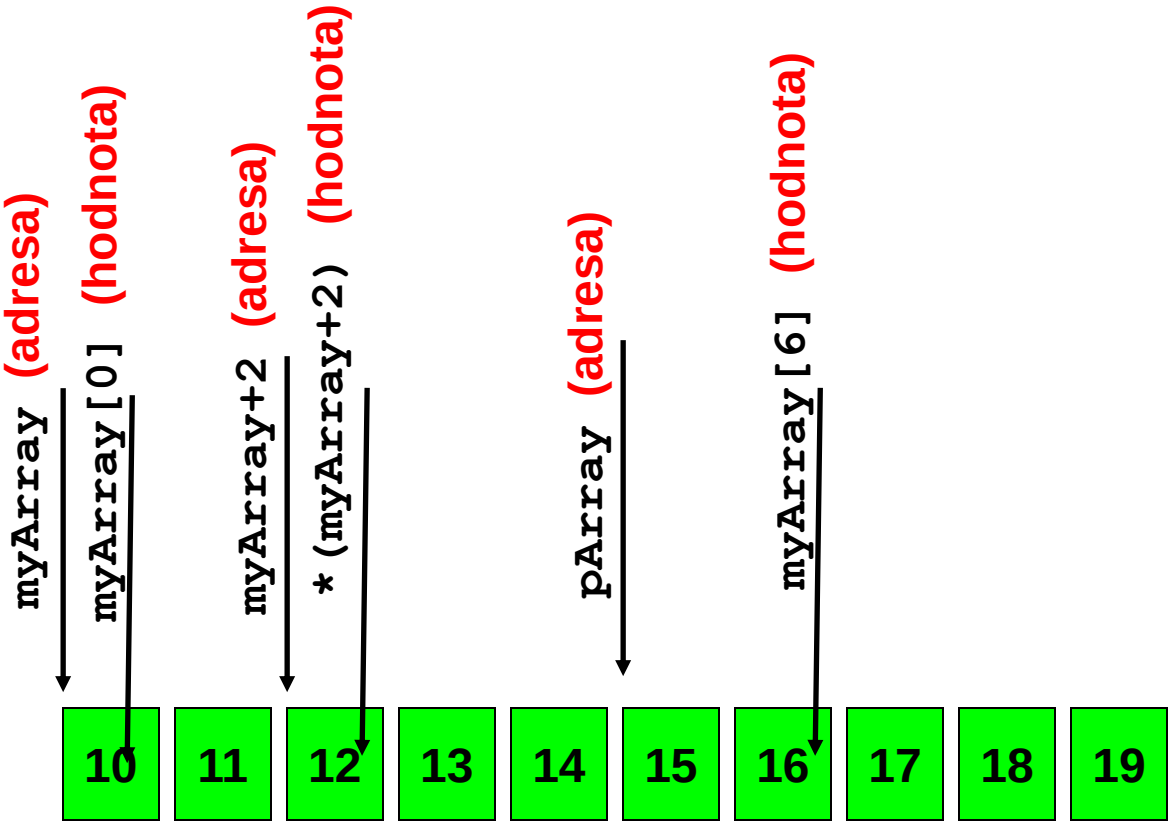
Ukazatelová aritmetika

- Aritmetické operátory prováděné nad ukazateli
- Využívá se faktu, že `array[X]` je definováno jako `*(array + X)`
- Operátor `+` přičítá k adrese na úrovni prvků pole

  - Nikoli na úrovni bajtů!
  - Např. pokud je `array` je typ `int*`
    - tak se přičte `X * sizeof(int)` bajtů
  - Obdobně pro -, ++ ...
- Adresu počátku pole lze přiřadit do ukazatele

Ukazatelová aritmetika - ilustrace

```
int myArray[10];
for (int i = 0; i < 10; i++) myArray[i] = i+10;
int* pArray = myArray + 5;
```



Demo ukazatelová aritmetika

```
void demoPointerArithmetic() {
 const int arrayLen = 10;
 int myArray[arrayLen];
 int* pArray = myArray; // value from variable myArray is assigned to variable pArra
 int* pArray2 = &myArray; // wrong, address of variable myArray,
                          //not value of variable myArray (warning)

 for (int i = 0; i < arrayLen; i++) myArray[i] = i;

 myArray[0] = 5; // OK, first item in myArray
 *(myArray + 0) = 6; // OK, first item in myArray
 //myArray = 10; // wrong, we are modifying address itself, not value on address

 pArray = myArray + 3; // pointer to 4th item in myArray
 //pArray = 5; // wrong, we are modifying address itself, not value on address
 *pArray = 5; // OK, 4th item
 pArray[0] = 5; // OK, 4th item
 *(myArray + 3) = 5; // OK, 4th item
 pArray[3] = 5; // OK, 7th item of myArray

 pArray++; // pointer to 5th item in myArray
 pArray++; // pointer to 6th item in myArray
 pArray--; // pointer to 5th item in myArray

 int numItems = pArray - myArray; // should be 4 (myArray + 4 == pArray)
}
```

Ukazatelová aritmetika - otázky

```
int myArray[10];
for (int i = 0; i < 10; i++) myArray[i] = i+10;
int* pArray = myArray + 5;
```

- Co vrátí myArray[10]?
- Co vrátí myArray[3]?
- Co vrátí myArray + 3?
- Co vrátí *(pArray – 2) ?
- Co vrátí pArray - myArray ?

Typické problémy při práci s poli

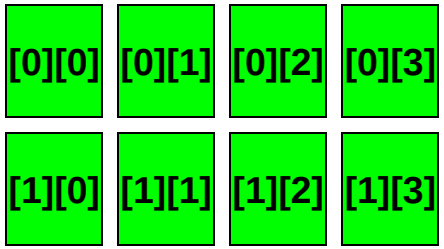
- Zápis do pole bez specifikace místa
  - `int array[10]; array = 1;`
  - proměnnou typu pole nelze naplnit (rozdíl oproti ukazateli)
- Zápis těsně za konec pole, častý “N+1” problém
  - `int array[N]; array[N] = 1;`
  - v C pole se indexuje od 0
- Zápis za konec pole
  - např. důsledek ukazatelové aritmetiky nebo chybného cyklu
  - `int array[10]; array[someVariable + 5] = 1;`
- Zápis před začátek pole
  - méně časté, ukazatelová aritmetika

- Čtení/zápis mimo alokovanou paměť může způsobit pád nebo nežádoucí změnu jiných dat (která se zde nachází)
  - `int array[10]; array[100] = 1; // runtime exception`

Multipole

Vícerozměrné pole

```
int array[2][4];
```

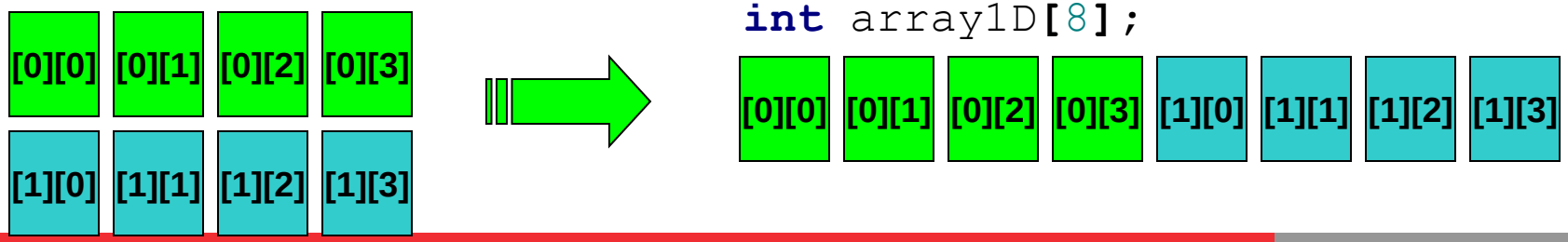


- Pravoúhlé pole N x M
  - stejný počet prvků v každém řádku
  - `int array[N][M];`
  - (pole hodnot o délce N, v každé hodnotě je pole **int***ů* o délce M)
- Přístup pomocí operátoru [] pro každou dimenzi
  - `array[7][5] = 1;`
- Lze i vícerozměrné pole
  - v konkrétním rozměru bude stejný počet prvků
  - `int array[10][20][30][7];`
  - `array[1][0][15][4] = 1;`

Reprezentace vícerozměrného pole jako 1D

- Vícerozměrné pole lze realizovat s využitím jednorozměrného pole
- Fyzická paměť není N-rozměrná
  - => překladač musí rozložit do 1D paměti
- Jak implementovat pole `array[2][4]` v 1D?
  - indexy v 0...3 jsou prvky array2D[0][0...3]
  - indexy v 4...7 jsou prvky array2D[1][0...3]
  - **`array2D[row][col]` → `array1D[row * NUM_COLS + col]`**
    - NUM_COLS je počet prvků v řádku (zde 4)

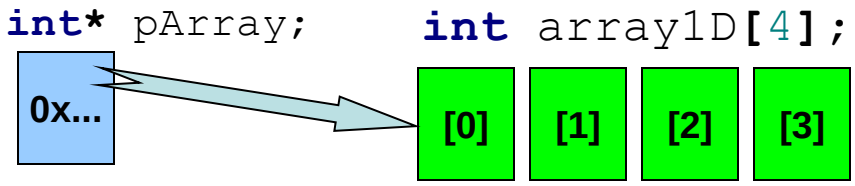
```
int array2D[2][4];
```



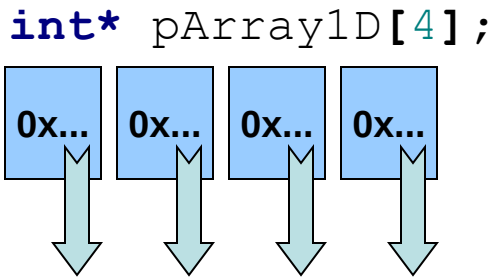
Pro zamyšlení

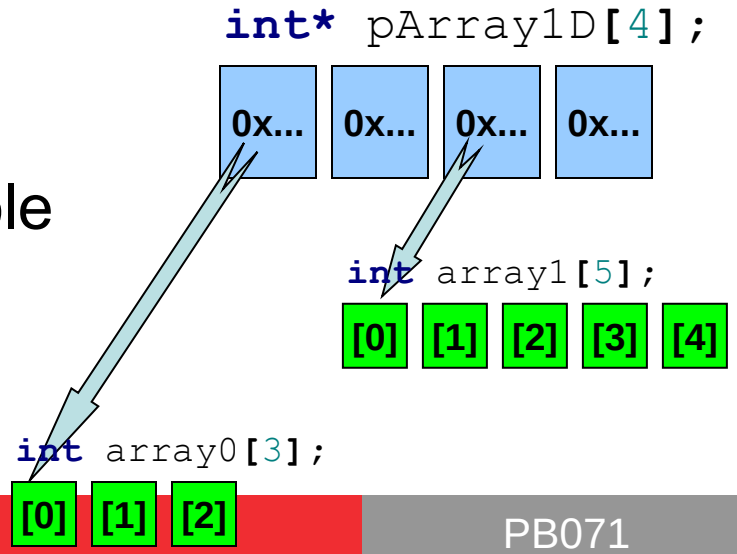
- Lze takto realizovat trojrozměrné pole?
  - `int array3D[NUM_X][NUM_Y][NUM_Z];`
- `array3D[x][y][z]` → ?
  - `int array1D[NUM_X*NUM_Y*NUM_Z];`
  - `array1D[x*(NUM_Y*NUM_Z) + y*NUM_Z + z]];`
- Proč?

Nepravoúhlé pole



- Pole, které nemá pro všechny “řádky” stejný počet prvků
  - dosahováno typicky pomocí dynamické alokace (později)
  - lze ale i pomocí statické alokace
- Ukazatel na pole
  - adresa prvního prvku
  - `int array[4]; int* pArray = array;`
- Pole ukazatelů
  - pole položek typu ukazatel
  - `int* pArray1D[4];`
- Do položek pArray1D přiřadíme pole
  - `pArray1D[0] = array0;`
  - `pArray1D[2] = array1;`





Typový systém

Typový systém - motivace

- Celé znaménkové číslo se reprezentuje nejčastěji v dvojkovém doplňkovém kódu

  iVal1 = 5 (dvojkový doplňkový)

  `00000000000000000000000000000101` `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

Problém: Autentizační modul v PHP

```
/var/www/htdocs/uag/functions/activeActiveCmd.php

function checkSharedKey($shared_key) {
    if (strlen($shared_key) != 32) {
        return false;
    }
    if (trim($shared_key) == "") {
        return flase;
    }
   // … check of expected shared_key value
}
```

- Překlep *flase* namísto false (logická FALSE)
- PHP nemusí mít silnou kontrolu typů
- return flase => řetězec “flase” => bool TRUE
- Ale až po kontrole délky (naneštěstí trim())
- https://medium.com/@DanielC7/remote-code-execution-gaining-domain-admin-privileges-due-to-a-typo-dbf8773df767

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 definovány 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 při kompilaci* – 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*` → `int` NE
4. Pokud nelze provést automatickou konverzi, ohlásí překladač chybu
   - *error: invalid conversion from 'int*' to 'int'*

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

5. Je zajištěno, že operace se provedou jen nad hodnotami povolených typů (POZOR: neznamená validní výsledek!)

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. Návratové hodnotě funkce

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

Zarovnání dat v paměti

- Proměnné daného typu můžou vyžadovat zarovnání v paměti (závislé na architektuře)
  - char na každé adrese
  - int např. na násobcích 4
  - float např. na násobcích 8
- Pokus o přístup na nezarovnanou paměť dříve způsoboval pád programu (těžké způsobit běžnými operacemi)
  - nyní jádro OS obslouží za cenu výrazného zpomalení programu
- Paměťový aliasing
  - na stejné místo v paměti více ukazatelů s různým typem
- Striktnější varianta: Strict aliasing
  - žádné dvě proměnné různého typu neukazují na stejné místo
  - zavedeno z důvodu možnosti optimalizace
  - standard vyžaduje, ale lze vypnout
- 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`
  - ...
  - Nemění hodnotu na bitové úrovni

```
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)

  6
  7             float fArray[10];
  8             char* cArray = (char*) fArray;
0x00401384   <+64>:                lea    0x1c(%esp),%eax
0x00401388   <+68>:                mov    %eax,0x44(%esp)
```

sémantická konverze

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 (`double` do `float`)
  - reprezentace čísel (`16777217` jako `float`)

Ukázka ztrátovosti konverze

```
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);
```

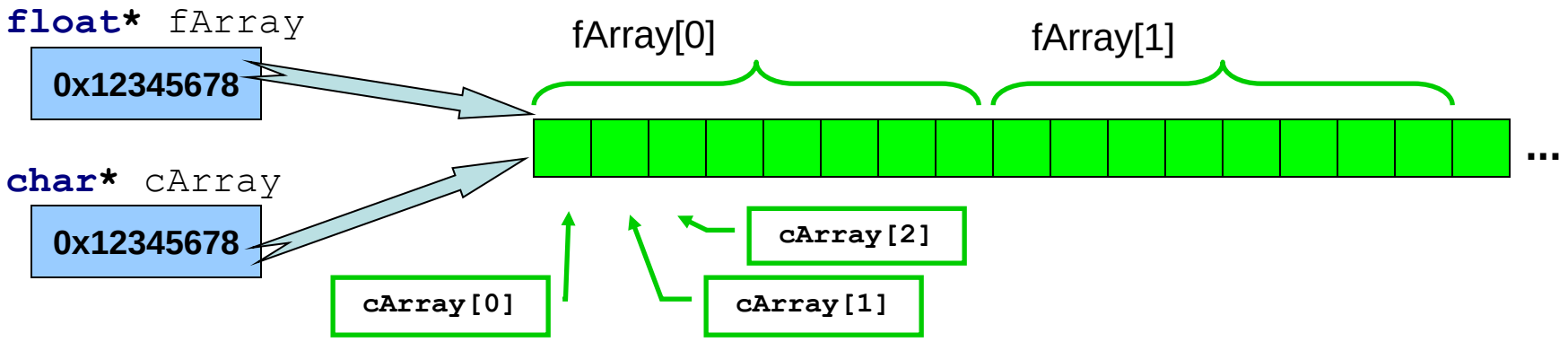
```
The integer is: 16777217
The double is: 16777217.000000
Their equality: 1
```

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

Přetypování ukazatelů

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

- 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



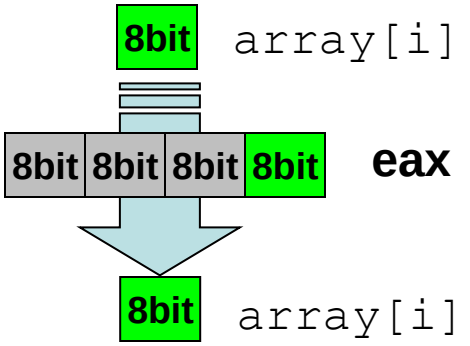
- Pozor na požadavek standardu na “strict aliasing“
  - Neměly by existovat dva ukazatele různého typu na stejnou paměť
    - s výjimkou (char*) a (void*)
  - omezuje možnost optimalizace překladačem

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)

```
const unsigned int array_len = 80;
unsigned char array[array_len];
// Will execute 80 iteration
for (int i = 0; i < array_len; i++) array[i] ^= 0x55;

// Will execute only 20 iteration (x86)
for (int i = 0; i < array_len / 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

Shrnutí

- Pole
  - Opakování datového prvku v paměti za sebou
  - Přístup pomocí ukazatele na první prvek
- Pole – nutné chápat souvislost s ukazatelem
- Ukazatelová aritmetika
- Typový systém

DÍKY ZA VÁŠ ČAS A V PONDĚLÍ ZASE NA VIDĚNOU