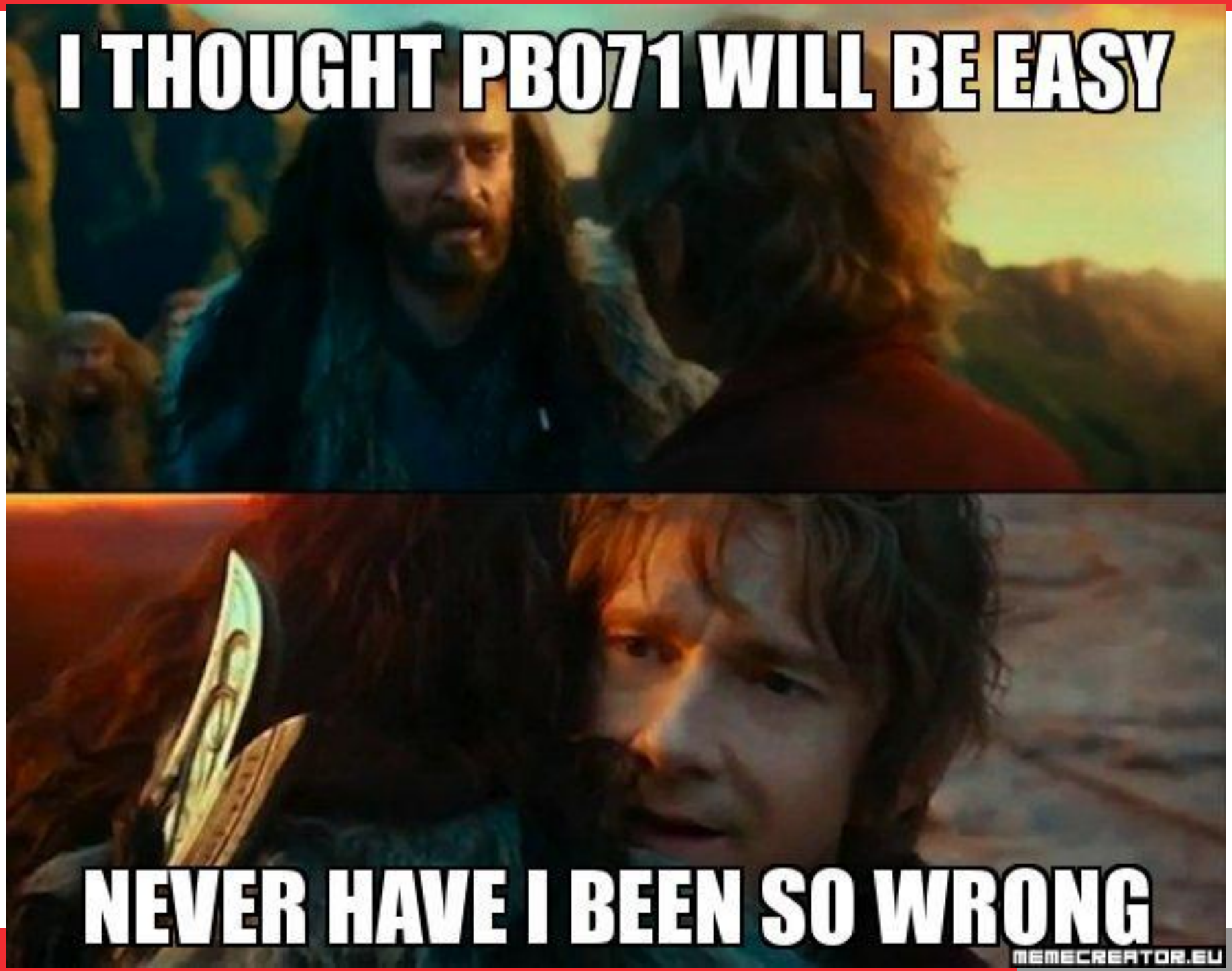
PB071
Principy nízkoúrovňového programování

Datové typy, operátory, řídící struktury

Slidy pro komentáře (děkuji!):

Slidy pro komentáře (děkuji!):
https://drive.google.com/file/d/1ndW8r7jRu-voB8e3KplED35bV-kP-_g6/view?usp=sharing

Organizační


I THOUGHT PB071 WILL BE EASY
NEVER HAVE I BEEN SO WRONG
MEMECREATOR.EU

Organizační – domácí úkoly

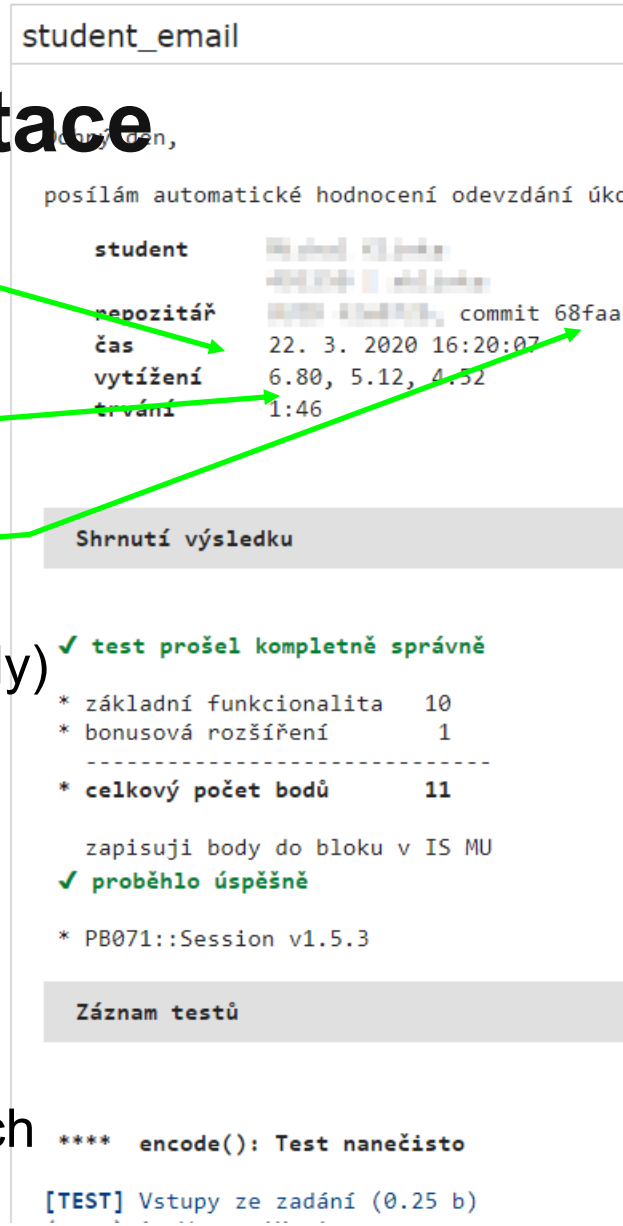
- Domácí úkol
  - zadání 1. úkolu
  - možnost odevzdání nanečisto (detaily na cvičení)
  - odevzdání do fakultního gitu, spuštění notifikačního skriptu
- Studentští poradci
  - hlavní počítačová hala, poblíž kopírky
  - dostupní pravidelně od tohoto týdne
  - https://fi.muni.cz/pb071
- Kudos: pokud vám poradce nebo cvičící dobře poradí, můžete mu udělit pochvalu:
  - https://is.muni.cz/auth/cd/1433/jaro2024/PB071/kudos

Kontrolní mail a jeho interpretace

Zátěž Aisy
http://en.wikipedia.org/wiki/Load_%28computing%29
Aisa má 64 jader, problémy se zátěží již nebývají

Doba běhu celé kontroly. Dlouhá doba signalizuje potenciální neoptimálnost vašeho řešení

Identifikace kontrolovaného kódu (commit)



- Tělo mailu
  - Výsledek testu, krátký popis při chybě, (body)
- Na ISu je založeno vlákno pro každý domácí úkol
  - Pište zde chybu nebo nepřesnost v zadaní
  - Není určeno pro diskuze nefunkčnosti nebo záseků vlastních kódů
    - využijte studentských poradců a cvičících

Top questions (1)

Anonymous 0

Pokud přeložím program překladačem gcc na Linuxu, bude spustitelný i na Windows?

Join at
slido.com
#pb071_2024

- **Pokud budete mít během poslechu přednášky dotaz, tak jej vložte na slido.com (#pb071_2024)**
- **Společně projedeme dotazy každé pondělí během přednášky Q&A**

Proměnné a jejich datové typy

Převod F2C

funkce `main`, bez argumentů, návratová hodnota `int`

datové typy, proměnné, přiřazení, konstanta

řídící struktury (cyklus), porovnávací operátor, zkrácený přiřazovací výraz, aritmetické operátory, pořadí vyhodnocování...

```c
#include <stdio.h>

int main(void) {
    int fahr = 0;
    float celsius = 0;
    int dolni = 0;
    int horni = 300;
    int krok = 20;

    for (fahr = dolni; fahr <= horni; fahr += krok) {
        celsius = 5.0 / 9.0 * (fahr - 32);
        // vypise prevod pro konkretni hodnotu fahrenheita
        printf("%3d \t %6.2f \n", fahr, celsius);
    }
    return 0;
}
```

Datové typy

- Datový typ objektu (proměnná, pole...) určuje:
  - (pozn.: objektem se zde nemyslí OOP objekt)
  1. hodnoty, kterých může objekt nabývat
     - např. `float` může obsahovat reálná čísla
  2. operace, které lze/nelze nad objektem provádět
     - např. k celému číslu lze přičíst 10
     - např. řetězec nelze podělit číslem 5
  3. množství paměti potřebné k jejímu uložení
- C má následující datové typy:
  - numerické (`int, float, double...`), znakový (`char`)
  - ukazatelový typ (později)
  - definované uživatelem (`struct, union, enum`) (později)

Jak silně typový jazyk C je?

- Co vrátí `2 + "2"` ?
- Síla typovosti dle míry omezení kladené na změnu typu
  - netypované jazyky – žádné typy nejsou, bez omezení
  - slabě typované jazyky – typy jsou, ale malé omezení
  - silně typované jazyky – výrazné omezení na změnu typu
  - http://en.wikipedia.org/wiki/Strongly_typed_programming_language
- Assembler << VB/PHP << C << C++/Java << Ada/SPARK
- C je spíše slabě typovaný jazyk, ale ne tolik jako VB/PHP
  - objekty mají přiřazen typ, ale mohou jej měnit
- Objekty mohou měnit svůj typ (tzv. konverze typů)
  - implicitní konverzi provádí automaticky překladač (Coercion)
  - explicitní konverzi provádí programátor
- Slabě typový proto, že typový systém lze obejít
  - a přetypovávat mezi nekompatibilními (potenciálně nebezpečné)

Primitivní datové typy

- Základní (primitivní) datové typy jsou:
  - `char` (znaménkový i neznaménkový – dle překladače, typicky použit na znaky)
  - `int` (znaménkový, celočíselný)
  - `float` (znaménkový, reálné číslo)
  - `double` (znaménkový, reálné číslo, typicky větší jak float)
  - `bool` (logická pravda 1 / nepravda 0, od C99)
    - #include <stdbool.h>
  - `wchar_t` (typ pro uchování UNICODE znaků)
- Viz http://en.wikipedia.org/wiki/C_data_types

Modifikátory datových typů

- **`unsigned`** – neznaménkový, uvolněný bit znaménka se použije na zvýšení kladného rozsahu hodnot (cca 2x)
  - unsigned int prom;
- **`signed`** – znaménkový (neuvádí se často, protože je to default)
  - Pozor, nemusí platit: `char == signed char`
- **`short`** – kratší verze typu
  - `short int prom`; nebo také `short prom`;
- **`long`** – delší verze typu (pokud překladač podporuje)
  - `long int prom;` nebo také `long prom;`
- **`long long`** – delší verze typu (pokud podporováno)

Primitivní datové typy – velikost

- Velikost typů se může lišit dle platformy a překladače
  - zjistíme operátorem `sizeof()`
- Standard předepisuje následující vztahy
  - `sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)`
  - `sizeof(char)<=sizeof(short)<=sizeof(float)<=sizeof(double)`
  - `sizeof(char) == 1` (nemusí být ale 8 bitů, některé DSP mají např. 16b)
- Zjištění pomocí `sizeof()` probíhá v době překladu
  - Kolik místa potřebujeme na uložení v paměti?
  - Velikost proměnné je ekvivalentní velikosti jejího typu
    - `int prom; sizeof(prom) == sizeof(int)`
  - Nezáleží na obsahu proměnné, ale jen na jejím typu
    - `int* prom = NULL; int* prom2 = 0x1234; sizeof(int*) == sizeof(void*) == sizeof(NULL) == sizeof(prom) == sizeof(prom2)`
  - Velikost pole není rovna velikosti ukazatele na pole
    - `sizeof(int[10]) != sizeof(int*)`

Velikost primitivních typů dle platformy

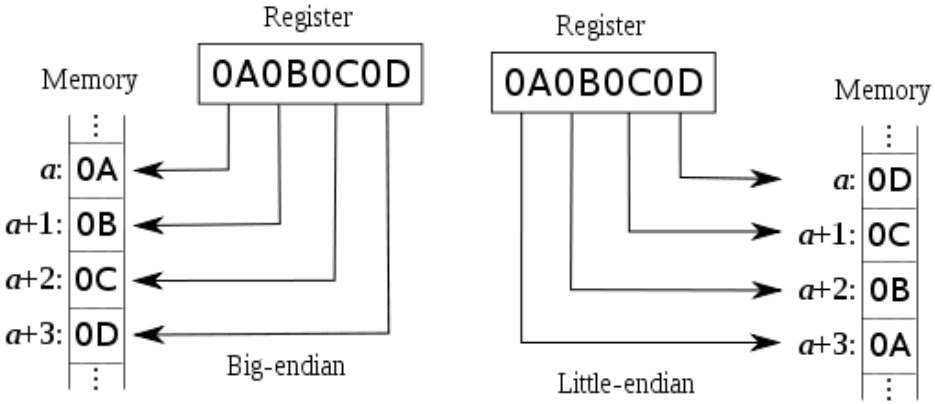
- Standard předepisuje vztahy velikostí, ne bitové délky
- Velikosti na architektuře x86 (32 bitů) jsou typicky:
  - char (8b), short (16b), int (32b), long (32b), float (32b), double (64b), long long (64b), long double (80b), pointer (32b)
- x64/ARM64 (64 bitů) typicky:
  - int (32b), long (64b), pointer (64b)
- Typické problémy při přechodu z x86 na x64
  - `sizeof(int)!=sizeof(long)!=sizeof(ukazatel)`
  - https://www.oracle.com/solaris/technologies/ilp32tolp64issues.html

Přesnost primitivních typů

- Typy s plovoucí desetinnou čárkou mají různou přesnost
- double (64 bitů) IEEE 754
  - double-precision binary floating-point format
  - https://en.wikipedia.org/wiki/Double-precision_floating-point_format
  - Přesnost na 15-17 desetinných míst
- Kde to vlastně potřebujeme?
- Představte si, že hledáme gravitační vlny
  - http://www.videacesky.cz/navody-dokumenty-pokusy/einstein-mel-pravdu-gravitacni-vlny-existuji
  - LIGO měřilo s přesností 10^{-21}
  - S doublem bychom je nenašli
- Algoritmus RSA – klíč délky 256 bajtů a více ($> 2^{2048}$)
  - Potřebujeme dodatečný typ “Big Integer”
  - Číslo je realizované jako pole, aritmetické operace nad polem
- Přesnost čísel pro různé jazyky: https://0.3000000000000004.com/

Big vs. little endian

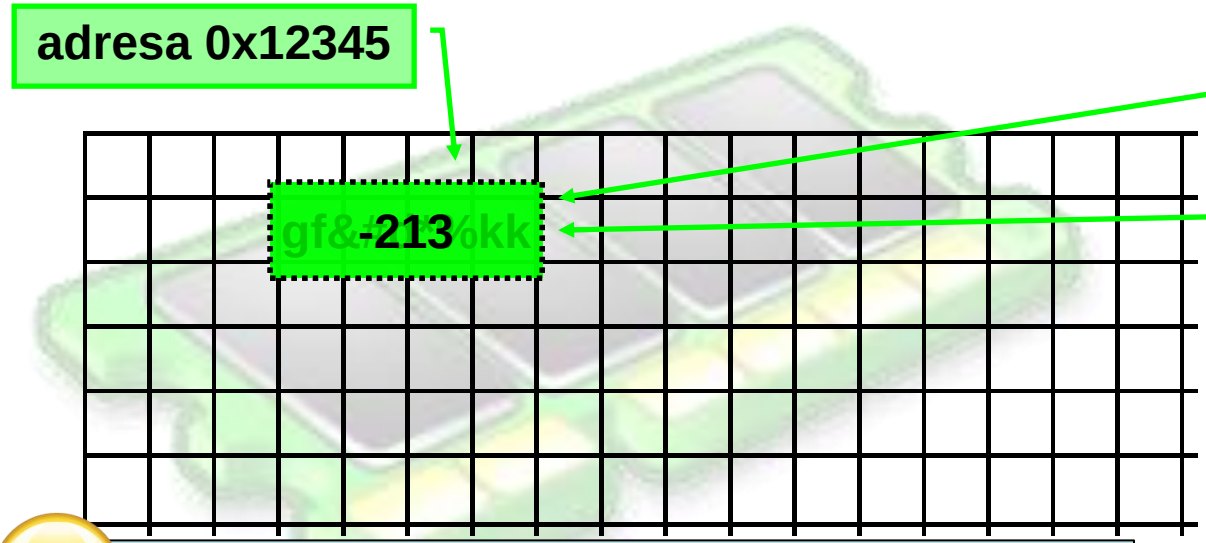
- Problém pořadí bajtů u vícebajtových typů (`int`)
- Big endian (významnější bajt na nižší adrese)
  - např. PowerPC, ARM (iniciálně)
  - využíváno pro přenos dat v sítích (tzv. *network order*)
- Little endian (méně významný bajt na nižší adrese)
  - např. Intel x86, x64
- Bi-endian
  - Lze přepínat
  - ARM, SPARC…



http://en.wikipedia.org/wiki/Little_endian

Co je proměnná?

- Pojmenované paměťové místo s připojenou typovou informací
  - **datový_typ** jméno_proměnné **[=**iniciální_hodnota**];**
  - např. **int** test **=** -213;
  - proměnná má přiřazen datový typ




```
int foo() {
    int test;

    test = -213;

    return 0;
}
```

Známé proměnné:
test → **int**, *0x12345*

Pro lokální proměnné je adresa relativní k rámci aktuální funkce: např. esp+0x12

Deklarace proměnné před prvním použitím

- Proměnná musí být definována/deklarována před prvním použitím
  - určujeme zároveň jméno (`test`) a typ (`int`)
  - můžeme zároveň inicializovat (přiřadit hodnotu)
    - silně doporučeno, jinak nespecifikovaná hodnota (předchozí obsah buňky paměti)
    - *warning: 'x' is used uninitialized in this function*
- Ukázky
  - `int x; // uninitialized variable`
  - `int x = 0; // variable initialized to 0`
  - `int x = 0, y = 1; // multiple variables, both initialized`

Proměnná - detailněji

- Každé místo v paměti má svou adresu
- Pomocí názvu proměnné můžeme číst nebo zapisovat do této paměti
  - např. `promenna = -213;`
- Překladač nahrazuje jméno proměnné její adresou
  - typicky relativní k zásobníku
  - `movl $0xfffff2b,0xc(%esp)`

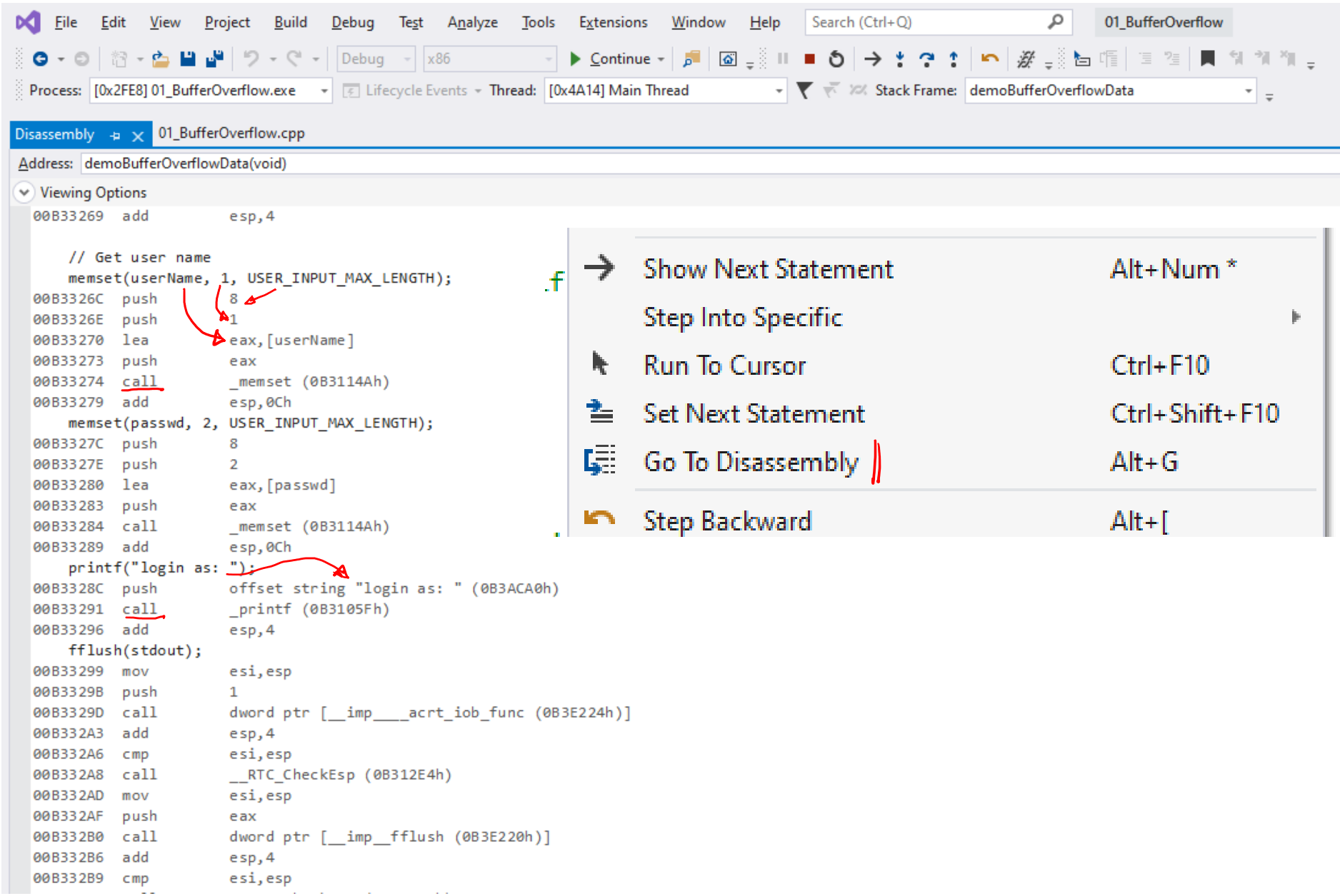
instrukce assembleru pro zápis do paměti

relativní adresa proměnné k adrese zásobníku

počáteční adresa zásobníku pro danou funkci

-213 hexadecimálně

Vyzkoušejte: QTCreator "instruction-wise mode"
Visual Studio: "Go To Disassembly"
Clion: Force step into (Alt+Shift+F7)


File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
01_BufferOverflow
Debug
x86
Continue
Process: [0x2FE8] 01_BufferOverflow.exe
Lifecycle Events
Thread: [0x4A14] Main Thread
Stack Frame: demoBufferOverflowData
Disassembly
01_BufferOverflow.cpp
Address: demoBufferOverflowData(void)
Viewing Options
00B33269 add esp,4
// Get user name
memset(userName, 1, USER_INPUT_MAX_LENGTH);
00B3326C push 8
00B3326E push 1
00B33270 lea eax,[userName]
00B33273 push eax
00B33274 call _memset (0B3114Ah)
00B33279 add esp,0Ch
memset(passwd, 2, USER_INPUT_MAX_LENGTH);
00B3327C push 8
00B3327E push 2
00B33280 lea eax,[passwd]
00B33283 push eax
00B33284 call _memset (0B3114Ah)
00B33289 add esp,0Ch
printf("login as: ");
00B3328C push offset string "login as: " (0B3ACA0h)
00B33291 call _printf (0B3105Fh)
00B33296 add esp,4
fflush(stdout);
00B33299 mov esi,esp
00B3329B push 1
00B3329D call dword ptr [__imp____acrt_iob_func (0B3E224h)]
00B332A3 add esp,4
00B332A6 cmp esi,esp
00B332A8 call __RTC_CheckEsp (0B312E4h)
00B332AD mov esi,esp
00B332AF push eax
00B332B0 call dword ptr [__imp__fflush (0B3E220h)]
00B332B6 add esp,4
00B332B9 cmp esi,esp
Show Next Statement Alt+Num *
Step Into Specific
Run To Cursor Ctrl+F10
Set Next Statement Ctrl+Shift+F10
Go To Disassembly Alt+G
Step Backward Alt+[

Jména proměnných

- Pravidla pojmenování (povinné, jinak syntaktická chyba)
  - musí začínat znakem anglické abecedy (a-zA-Z) nebo _
    - (NE např. `int 3prom; `)
  - může obsahovat pouze písmena, cifry a podtržítko
  - záleží na velikosti znaků (`int prom; int Prom;`)
  - nesmí být klíčové slovo jazyka (NE např. `int switch = 1;`)
- Konvence pojmenování (doporučené)
  - respektujte existující konvenci v projektu
  - volte výstižná jména
    - ANO `float divider = 1; int numberOfMembers = 0;`
    - NE a, aa, sajdksaj, PROMENNA
    - (jména `i` a `j` se používají jako řídící proměnná cyklu)
  - jména proměnných začínejte malým písmenem
  - nezačínejte proměnné __ nebo _X (X je libovolné velké písmeno)
    - rezervované, typicky různé platformě závislá makra apod.
  - oddělujte slova v názvu proměnné (`camelCase` nebo `raz_dva`)

Výrazy

- Výraz je kombinace proměnných, konstant, operátorů anebo funkcí
- `x = y + 4 * 5`
  - x a y jsou proměnné
  - 4, 5 jsou konstanty
  - + a * jsou aritmetické operátory, = je přiřazovací operátor
- Výsledkem vyhodnocení výrazu je hodnota s nějakým typem
  - `5 / 9` konstantní výraz s typem int
  - `5.0 / 9.0` konstantní výraz s typem double

Operátory

Operátory

- Operátory definují, jak mohou být objekty v paměti manipulovány
- Operátory mohou být dle počtu operandů:
  - unární (např. `prom++` nebo `--prom`)
  - binární (např. `prom1 + prom2`)
  - ternární (např. `(den > 15) ? 1 : 0`)
- Operátory mají různou prioritu
  - pořadí vyhodnocení, který vyhodnotit dříve
  - viz https://en.cppreference.com/w/c/language/operator_precedence

Aritmetické operátory

- Operátory pro aritmetické výrazy (+, -, *, /, %)
- Definovány pro všechny primitivní datové typy
  - typ vyhodnocení výrazu dle typu argumentů
  - defaultní typ je celočíselná hodnota
  - v plovoucí čárce, pokud je alespoň jeden operand float/double
- +, -, * (běžné)
  - `5 + prom; 365 * 24 * 60;`
  - (pozor, např. * má více významů)
- Operátor dělení /
  - v závislosti na argumentech celočíselné nebo s desetinnou čárkou
  - `5 / 9 * (fahr - 32)`
    - pozor na celočíselné dělení (`5 / 9` → `0`)
    - `5.0 / 9.0 * (fahr - 32)`
- Zbytek po celočíselném dělení % (modulo)
  - `5 % 9` → `5`
  - `21 % 2` → `1`

Porovnávací operátory

- Operátory pro porovnání dvou operandů
  - výsledkem je logická hodnota
  - v C je (libovolná) nenulová hodnota pravda (~TRUE)
    - (Relační operátory vrátí vždy 0 nebo 1)
  - 0 je nepravda (~FALSE)
    - pozor na 0 v reálném čísle (nemusí být přesně 0) 
- <, >, <=, >=, ==, !=
- Ukázky
  - `prom > 30`
  - `prom != 55`
  - `55 <= prom`
  - `55 != prom`

Porovnávací operátory

```c
#include <stdio.h>
int main(void) {
    int prom = 0;
    if (prom = 0) printf("Never printed");
    if (prom = 1) printf("Never say never!");
    if (1 = prom) // compilation error
    return 0;
}
```

Pozor na záměnu = a ==

- chceme testovat, zda je `prom` rovno 0
  - správně `prom == 0`
- zaměníme chybně == za =
  - `prom = 0` je validní výraz
- Dostaneme varování překladače, pokud použito např. s IF-ELSE
  - *`warning: suggest parentheses around assignment used as truth value`*

Pozor na == u reálných čísel

- omezená přesnost
- nemusí být shodné a operátor přesto vrátí `TRUE`

Logické operátory

- Operátory vyhodnocující logickou hodnotu výrazu
- `&&` (a zároveň, AND)
  - oba dva argumenty musí být pravda
  - `(prom == 1) && (prom2 % 10 == 5)`
- `||` (nebo, OR)
  - alespoň jeden argument musí být pravda
  - `(prom == 1) || (prom2 % 2 == 1)`
- `!` (logická negace, NOT)
  - logická inverze argumentu
  - `!(prom % 2 == 1)`

POZOR: Zkrácené vyhodnocování (líné, lazy)

- pokud je znám logický výsledek, zbytek výrazu se nevyhodnocuje
- podvýraz na FALSE pro `&&`, podvýraz na TRUE pro `||`
- pozor na vedlejší efekty (resp. jejich nepřítomnost)
- `if ((5 == 6) && (funkceFoo() == 1)) ...`

nebude vůbec zavoláno

Zvýšení a snížení o “1”

- Užitečná zkratka aritmetického operátoru + 1 resp. - 1
- Postfixová notace
  - A**++** je zkratka pro A **=** A **+** 1
  - A**--** je zkratka pro A = A – 1
  - B = A++; je zkratka pro B = A; A = A + 1;
    - ++ je vyhodnoceno a A změněno **PO** přiřazení
- Prefixová notace
  - **++**A je zkratka pro A = A + 1
  - **--**A je zkratka pro A = A – 1
  - B = ++A; je zkratka pro A = A + 1; B = A;
    - ++ je vyhodnoceno a A změněno **PŘED** přiřazením
- Pozor ale např. na a[i] = i++; (pozice `i` v poli `a`)
  - není definované, zda bude pozice i před nebo po ++
  - stejný problém nastává u funkce(x, ++x)
  - Více viz. Sekvenční body (logická místa provádění programu, kde jsou ukončeny dopady předchozích výrazů)
  - http://publications.gbdirect.co.uk/c_book/chapter8/sequence_points.html
- Ukázka chování při return
  - return A++; (nejprve vrátí hodnotu A, pak teprve zvýší o 1)
  - return ++A; (nejprve zvýší A, pak teprve vrátí obsah A)

Konstanty, konstantní výrazy

- Literál/výraz s hodnotou nezávislou na běhu programu
- Celočíselné, desetinná čárka, reálné
  - `int i = 192;` (dekadicky)
  - `int i = 0xC0;` (hexadecimálně)
  - `int i = 0300;` (osmičková)
  - `unsigned long i = 192UL;`
  - `float pi = 3.14;` (float)
  - `float pi = 3.14159F;` (float explicitně)
  - `double pi = 3.141592653589793L;` (double)
- Znakové konstanty: ‘A’, ‘\x1B’
- Řetězcové konstanty: “ahoj”
- Konstantní výrazy mohou být vyhodnoceny v době překladu
  - `5 * 4 + 13` → `33`
- Klíčové slovo **`const`** (později)

Bitové operátory

- `&`, `|`, `~`, `^`, `<<`, `>>`
- Pracují jako logické operátory, ale na úrovni jednotlivých bitů operandů
- AND: Z = X `&` Y;
- OR: Z = X `|` Y;
- XOR: Z = X `^` Y;
- INVERT: Z = ~X;
- LSHIFT: Z = X `<<` 2;
- RSHIFT: Z = X `>>` 2;

```
  0110
& 1100
= 0100
```

```
  0110
| 1100
= 1110
```

```
  0110
^ 1100
= 1010
```

```
~ 0110
= 1001
```

```
 00000110
<< 2
= 00011000
```

```
 00000110
>> 2
=
 00000001
```

bitový posun je ztrátový, pokud posunete jedničkový bit za hranici použitého datového typu

Bitové operátory - využití

- Sada příznaků TRUE/FALSE (pro úsporu prostoru)
  - např. do jednoho intu (32b) uschováme 32 hodnot
    - `unsigned int flags = 0x00000000;`
  - (pozn.: jednotlivé bity odpovídají násobkům dvojky)
- Vložení hodnoty na pozici X
  - vypočteme masku jako mask = $2^{(X-1)}$ (indexujeme od 0, tedy X - 1)
    - např. třetí bit → 2^2 → 4 → 0x04 hexadecimálně
  - aplikujeme operaci OR s vypočtenou maskou
    - `flags = flags | 0x04; // set 3rd bit to 1`
- Zjištění hodnoty z pozice X
  - vypočteme mask = $2^{(X-1)}$
  - `if (flags & 0x04) { /*...*/ }`
- Zjištění hodnoty dolního bajtu
  - maska pro celý bajt je 255, tedy 0xFF
  - `unsigned char lowByte = flags & 0x`

```
  0001
| 0100
= 0101
```

```
  0101
& 0100
= 0100
= TRUE
```

```
      .0101010000010101
     &        11111111
= .0000000000010101
```

! Nepoužívejte se signed hodnotami (není fixní bitová reprezentace)

Bitové operátory - využití

- Operace nutné na úrovni bitů
  - např. převod BASE64, šifrovací algoritmy...
  - maska pro dolních 6 bitů: $2^0+2^1+2^2+2^3+2^4+2^5 = 63_{10}$ = 0x3F
  - maska pro horní 2 bity: $2^6+2^7 = 192_{10}$ = 0xC0
- Rychlé násobení mocninou dvojky X * 2^{posun}
  - 0011_2 (3 dekadicky)
  - 0011_2 << 1 → 0110_2 (6 dekadicky)
  - 0011_2 << 2 → 1100_2 (12 dekadicky)
- Pozor na rozlišení & a &&, resp. | a ||
  - např. záměna & za &&
    - `1100 && 0011 == TRUE`
    - `1100 & 0011 == 0 (FALSE)`

Zkrácené přiřazovací operátory

- Často používané výrazy jsou ve tvaru
  - `prom = prom (op) výraz;`
  - např. `int prom = 0; prom = prom + 10;`
- C nabízí kompaktní operátory přiřazení
  - `prom (op)= výraz;`
  - `prom += 10;`
  - `prom /= 3;`
  - `prom %= 7;`
  - `prom &= 0xFF;`

Pořadí vyhodnocení operátorů

```
int prom1 = 1;
int prom2 = 10;
prom1 = 5 + prom1 * 18 % prom2 - prom1 == 2;
```

Jaká bude hodnota `prom1`?

- použijeme tabulku priority operátorů
  - http://en.cppreference.com/w/c/language/operator_precedence
- prom1 = 5 + (prom1 * 18) % prom2 - prom1 == 2;
- prom1 = 5 + ((prom1 * 18) % prom2) - prom1 == 2;
- prom1 =(5 + ((prom1 * 18) % prom2)) – prom1 == 2;
- prom1 =((5 + ((prom1 * 18) % prom2)) – prom1) == 2;
- prom1 =(((5 + ((prom1 * 18) % prom2)) – prom1) == 2);
- (prom1 = (((5 + ((prom1 * 18) % prom2)) – prom1) == 2));

https://en.cppreference.com/w/c/language/operator_precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	( )	Function call	
	[ ]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal (C99)	
2	++ --	Prefix increment and decrement [note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of [note 2]	
	_Alignof	Alignment requirement (C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional [note 3]	Right-to-left
14 [note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Pořadí vyhodnocení – co si pamatovat

- ++, --, (), [] mají nejvyšší prioritu
- *,/,% mají prioritu vyšší než + a -
- Porovnávací operátory (==, !=, <) mají vyšší prioritu než logické (&&, ||, !)
  - `if (a == 1 && !b)`
- Operátory přiřazení mají velmi malou prioritu
  - uložení vyhodnoceného výrazu do proměnné až nakonec
  - vyhodnocují se zprava doleva

Lze ovlivnit pomocí závorek ()

- využívejte co nejvíce, výrazně zpřehledňuje!
- nespoléhejte na „znalost“ priority operátorů

```
int prom1 = 1;
int prom2 = 10;
prom1 = (5 + ((prom1 * 18) % prom2) - prom1) == 12;
```

Řídící struktury

Řídící struktury

- `if` (*výraz*) {*příkaz1*} `else` {*příkaz2*}
- `for` (*init; podmínka; increment*) {*příkazy*}
- `while` (*podmínka_provedení*) {*příkazy*}
- `do` {*příkazy*} `while` (*podmínka_provedení*)
- `switch` **(***promenna***) { `case` ... }**

Podmíněné výrazy

```
if vyraz:
    prikaz1
else:
    prikaz2
```


- **`if`** (*výraz*) {*příkaz1*} **`else`** {*příkaz2*}
  - pokud je výraz == TRUE (podmínka), vykoná se *příkaz1*
  - jinak *příkaz2*
- Ternární operátor **`?`**
  - zkrácení if – else
  - `odd = (var1 % 2 == 1) ? 1 : 0;`

```
int var1 = 10;
_Bool odd = 0;_
if (var1 % 2 == 1) {
  // var1 is odd
  odd = 1;
}
else {
  // var1 is even
  odd = 0;
}
```

- Problematika závorek
  - pokud je ve větvi **`then/else`** jediný výraz, není nutné psát **`{ }`**
    - Ale raději vždy použijte (nezapomenete při pozdějším přidání)
  - pozor na problém při pozdějším rozšiřování kódu u **`else`**

```
int var1 = 10;
_Bool odd = 0;_
if (var1 % 2 == 1) odd = 1;
else odd = 0;
printf("Even");
```

Cyklus FOR

```
for (inicializační_výraz; podmínka_provedení; inkrementální_výraz) {
   // tělo cyklu, ... příkazy
}
```

- Cyklus FOR
  - *inicializační_výraz* – provede jen jednou, inicializace
    - typicky nastavení řídící proměnné
  - *podmínka_provedení* – pokud TRUE, tak proběhne tělo cyklu
    - typicky test řídící proměnné vůči koncové hodnotě
  - *inkrementální_výraz* – provede se po konci každé iterace
    - typicky změna řídící proměnné
- Používáno často pro cykly s daným počtem iterací
  - ne nutně fixním během překladu, ale ukončovací podmínka stejná
  - např. projití pole od začátku do konce

```
//  ...
for (fahr = dolni; fahr <= horni; fahr += krok) {
   celsius = 5.0 / 9.0 * (fahr - 32);
   // vypise prevod pro konkretni hodnotu fahrenheita
   printf("%3d \t %6.2f \n", fahr, celsius);
}
//  ...
```

```
for fahr in range(dolni, horni + 1, krok):
    # telo cyklu
```

Cyklus WHILE

```
while (podmínka_provedení) {
    // telo cyklu, ... prikazy
    // typicky zmena ridici promenne
}
```

- Cyklus WHILE
  - *podmínka_provedení* – pokud TRUE, tak proběhne tělo cyklu
    - typicky test řídící proměnné vůči koncové hodnotě
  - typicky v těle modifikujeme řídící proměnnou
- Používáno především pro cykly s předem neznámým počtem iterací
  - např. opakuj cyklus, dokud se nevyskytne chyba

```
// ...
fahr = dolni;
while (fahr <= horni) {
   celsius = 5.0 / 9.0 * (fahr - 32);
   // vypise prevod pro konkretni hodnotu fahrenheita
   printf("%d \t %d \n", fahr, celsius);
   // zmena ridici promenne
   fahr = fahr + krok;
}
//  ...
```

Cyklus DO - WHILE

```
do {
   // tělo cyklu, ... příkazy
   // typicky změna řídící proměnné
}
while (podmínka_provedení);
```

- Cyklus DO-WHILE
  - *podmínka_provedení* – pokud TRUE, tak proběhne další iterace cyklu
    - typicky test řídící proměnné vůči koncové hodnotě
    - testuje se PO těle cyklu
  - typicky v těle modifikujeme řídící proměnnou
  - tělo cyklu vždy proběhne alespoň jednou!

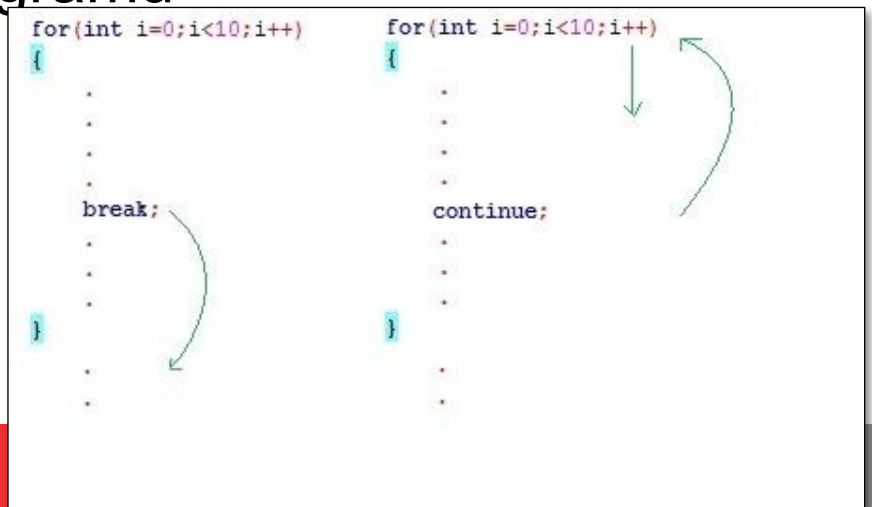
```
// ...
fahr = dolni;
if (fahr <= horni) {
 do {
    celsius = 5.0 / 9.0 * (fahr - 32);
    // vypise prevod pro konkretni hodnotu fahrenheita
    printf("%d \t %d \n", fahr, celsius);
    // zmena ridici promenne
    fahr = fahr + krok;
 } while (fahr <= horni);
}
// ...
```


Není přímý ekvivalent, lze snadno přes **while**

Předčasné ukončení cyklu

- **`break`** – ukončení cyklu a pokračování za cyklem
- **`continue`** – ukončení těla cyklu a pokračování další iterací
- (**`return`**) – ukončení celé funkce
  - preferujte pouze jeden return na konci funkce
  - s výjimkou iniciálního ošetření vstupních dat (tzv. “early return” návrhový vzor)
- (**`exit`**) – ukončení celého programu
- (goto)

- Lze použít pro všechny cykly

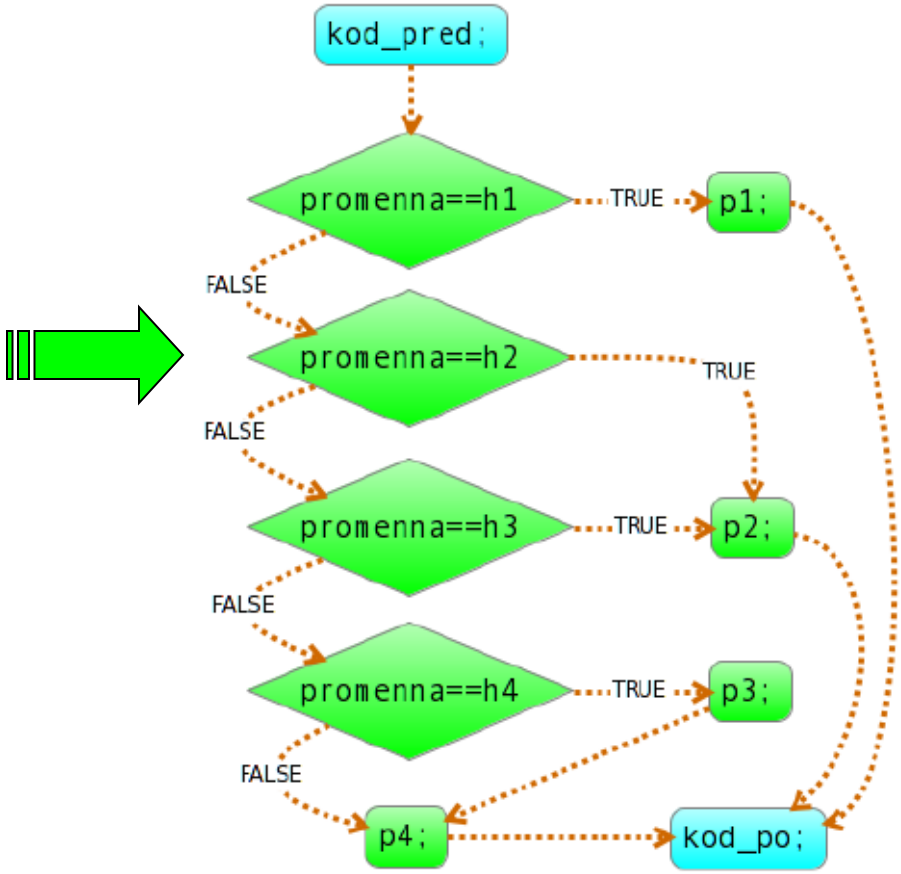


`switch`

- Podmíněný příkaz pro násobné větvení
- Využití pro typ int a typy, které na něj lze převést
  - další celočíselné typy (char, short, long)
- Umožňuje definovat samostatné větve pro různé hodnoty řídící proměnné
  - `case, break`
  - po nalezení shody se vykonává kód do nalezení klauzule break;
- Používejte `default` klauzuli
  - Aplikuje se, pokud žádný `case` neodpovídá
  - např. pro výpis chyby (zachytí nepředpokládanou hodnotu)

Switch – ukázka vyhodnocení

```
switch(promenna) {
   case h1: p1; break;
   case h2:
   case h3: p2; break;
   case h4: p3;
   default: p4; break;
}
```



Switch - ukázka

```
switcher = {
    1: 'Operation type A',
    2: 'Operation type B'
    #...
    }
switcher.get(value, 'Unknown value')
```

```
int value = 0;
// ...
switch (value) {
    case 1: {
      printf("Operation type A: %d\n", value);
      break;
    }
    case 2: {
      printf("Operation type A: %d\n",
      break;
    }
    case 3: {
      printf("Operation type B: %d\n",
      break;
    }
     default: {
      printf("Unknown value");
      break;
    }
}
```

Velmi vhodné explicitně okomentovat, aby bylo jasné, že nejde o opomenutí

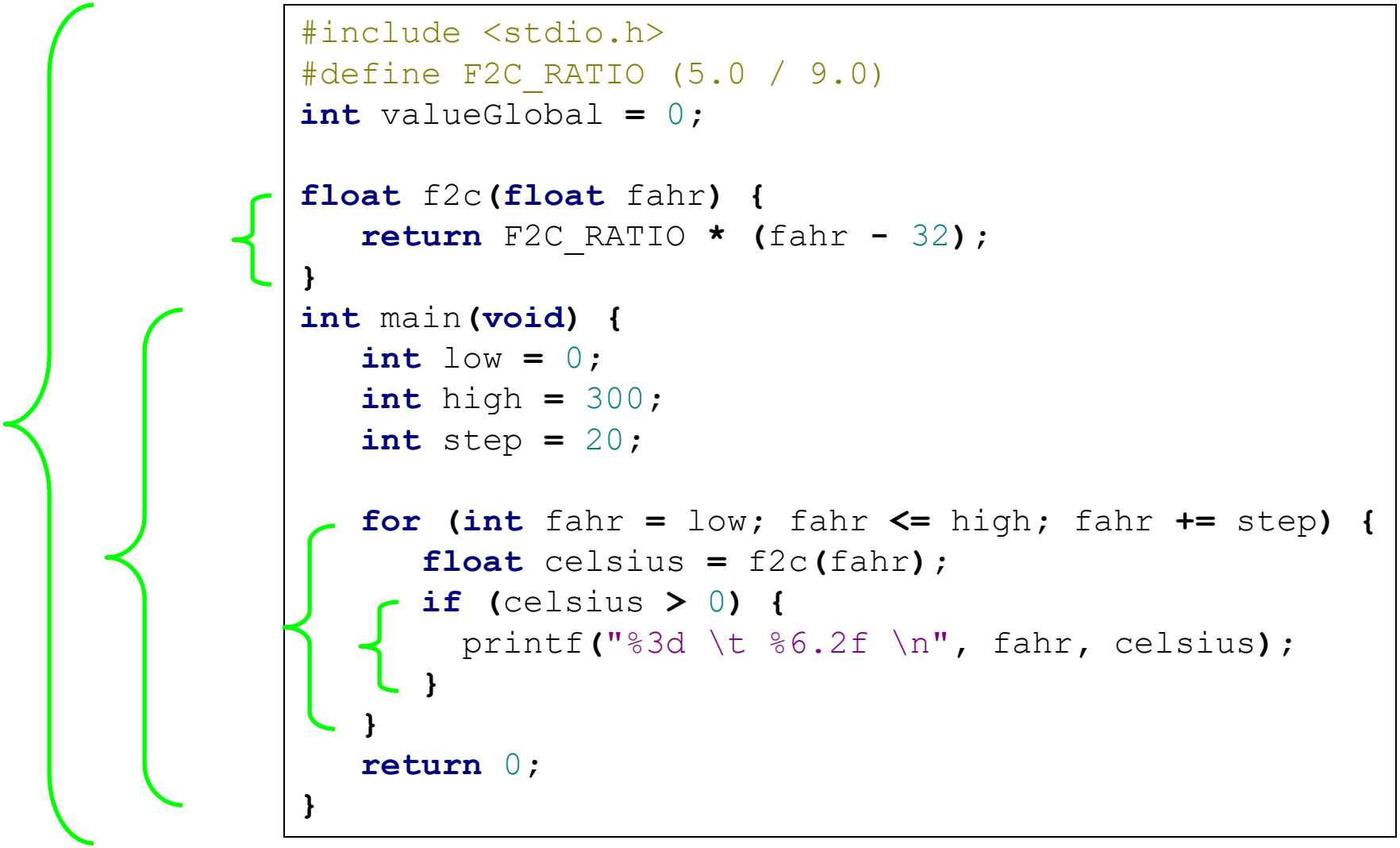
ukázka (ne)využití `break;`

```
int value = 0;
// ...
switch (value) {
    case 1: // no break
    case 2: {
      printf("Operation type A: %d\n", value);
      break;
    }
    case 3: {
      printf("Operation type B: %d\n", value);
      break;
    }
     default: {
      printf("Unknown value");
      break;
    }
}
```

Bloky

- Pomocí { } zkombinujeme několik příkazů do jednoho bloku
- Blok lze využít jako nahrazení pro příkaz
  - v místě, kde můžeme použít příkaz, lze použít i blok
  - využito např. u řídících konstrukcí jako IF-ELSE, cykly...
- Uvnitř bloku lze deklarovat proměnné, které automaticky zanikají na jeho konci
  - platnost proměnných je omezená na blok s deklarací
- Blok může být prázdný

Bloky - ukázka



```c
#include <stdio.h>
#define F2C_RATIO (5.0 / 9.0)
int valueGlobal = 0;

float f2c(float fahr) {
    return F2C_RATIO * (fahr - 32);
}
int main(void) {
    int low = 0;
    int high = 300;
    int step = 20;

    for (int fahr = low; fahr <= high; fahr += step) {
        float celsius = f2c(fahr);
        if (celsius > 0) {
            printf("%3d \t %6.2f \n", fahr, celsius);
        }
    }
    return 0;
}
```

Rozsah platnosti proměnných

- Část kódu, odkud je proměnná použitelná (scope)
- Často koresponduje s blokem, ve kterém je proměnná deklarována
- Lokální proměnná
  - proměnná s omezeným rozsahem platnosti
  - typicky proměnné v rámci funkce nebo bloku
- Globální proměnná
  - proměnné deklarované mimo funkce
  - nezaniká mezi voláními funkcí

Rozsah platnosti - ukázka

- Určete rozsah platnosti proměnných v kódu
  - globální, lokální celá funkce, lokální blok

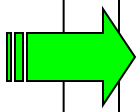
```
// ...
int valueGlobal = 0;

float f2c(float fahr) {
   return F2C_RATIO * (fahr - 32);
}
int main(void) {
   int low = 0;
   int high = 300;
   int step = 20;
   for (int fahr = low; fahr <= high; fahr += step) {
      float celsius = f2c(fahr);
      if (celsius > 0) {
        printf("%3d \t %6.2f \n", fahr, celsius);
      }
   }
   return 0;
}
```

Vnořené příkazy

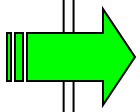
- Příkazy lze vnořit do sebe
  - další příkaz je součástí vnitřního bloku
  - např. vnořené if-else
- Pozor na nevhodné hluboké vnoření
  - řešením je přeformátování kódu
  - např. záměna za switch

Vnořené příkazy – ukázka přeformátování

```
if (day == 1) {
  printf("Pondeli");
}
else {
  if (day == 2) {
    printf("Utery");
  }
  else {
    if (day == 3) {
      printf("Streda");
    }
    else {
      // ....
    }
  }
}
```



```
if (day == 1) printf("Pondeli");
if (day == 2) printf("Utery");
if (day == 3) printf("Streda");
// ....
```



```
switch (day) {
   case 1: printf("Pondeli"); break;
   case 2: printf("Utery"); break;
   case 3: printf("Streda"); break;
   // ...
}
```

Shrnutí

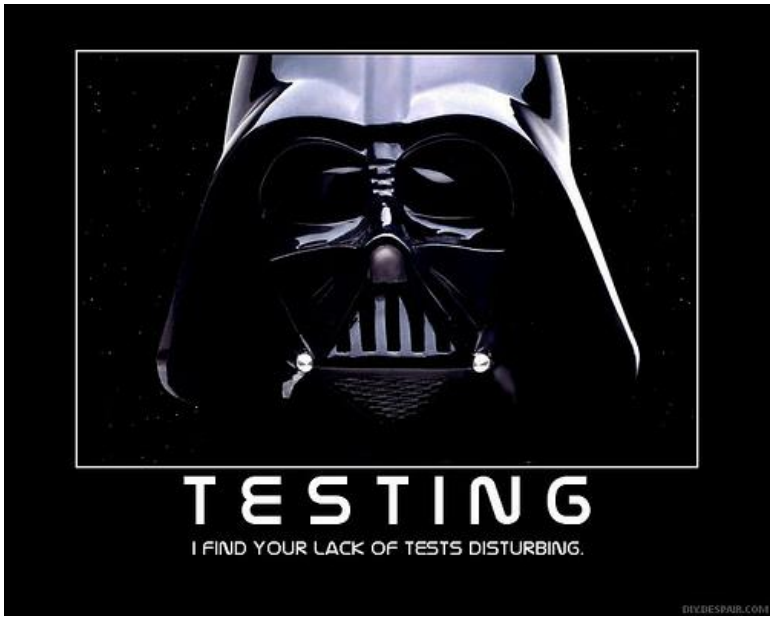
Funkce `main`, má své tělo v bloku ohraničeném {}

Vzniká 5 proměnných s různými datovými typy. Jsou ihned inicializované konstantou.

Řídící struktura for (cyklus) vykoná svoje tělo v bloku {}, obsahujícím zkrácený přiřazovací výraz obsahující aritmetické operátory. Cyklus se ukončí na základě výsledku porovnávacích operátorů.

```
#include <stdio.h>

int main(void) {
        int fahr = 0;
        float celsius = 0;
        int dolni = 0;
        int horni = 300;
        int krok = 20;

        for (fahr = dolni; fahr <= horni; fahr += krok) {
                celsius = 5.0 / 9.0 * (fahr - 32);
                // vypise prevod pro konkretni hodnotu fahrenheita
                printf("%3d \t %6.2f \n", fahr, celsius);
        }
        return 0;
}
```

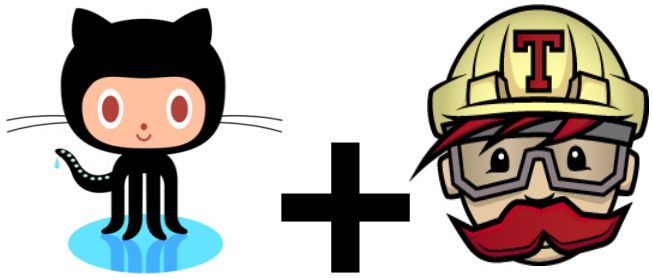
Testování, unit testing



Typy testování

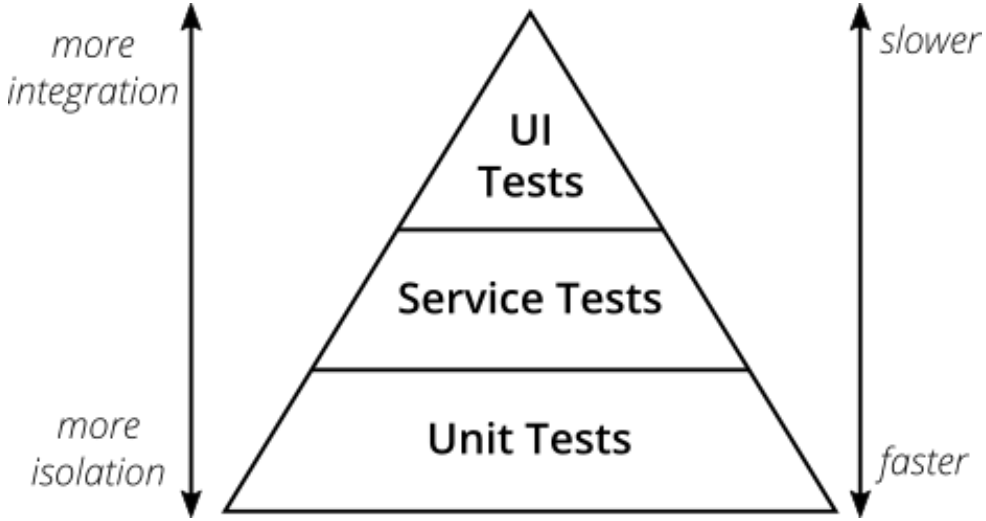
- Manuální vs. Automatické
- Dle rozsahu testovaného kódu
- Unit testing
  - testování elementárních komponent
  - typicky jednotlivé funkce
- Integrační testy
  - test spolupráce několika komponent mezi sebou
  - typicky dodržení definovaného rozhraní
- Systémové testy
  - test celého programu v reálném prostředí
  - ověření chování vůči specifikaci

Jaké testy, jak a kolik?



- Automatizujte testy
- Pište testy různých typů a s různým pokrytím
- Čím nízkoúrovňovější test je (směrem k unit-testům), tím více by jich mělo být

https://martinfowler.com/articles/practical-test-pyramid.html



Shrnutí

- Základní datové typy
  - volte vhodný, pozor na rozsah a přesnost
  - konstanty – různé možnosti zápisu (dekad., hexa.)
- Operátory
  - používejte závorky pro zajištění pořadí vyhodnocování
- Řídící struktury
  - není vždy jediná nejvhodnější
  - snažte se o čitelný kód
- Testování
  - automatizujte všechny nutné manuální činnosti

Top questions (1)

Anonymous 0

Pokud přeložím program překladačem gcc na Linuxu, bude spustitelný i na Windows?

Join at
slido.com
#pb071_2024

- **Pokud budete mít během poslechu přednášky dotaz, tak jej vložte na slido.com (#pb071_2024)**
- **Společně projedeme dotazy každé pondělí během přednášky Q&A**

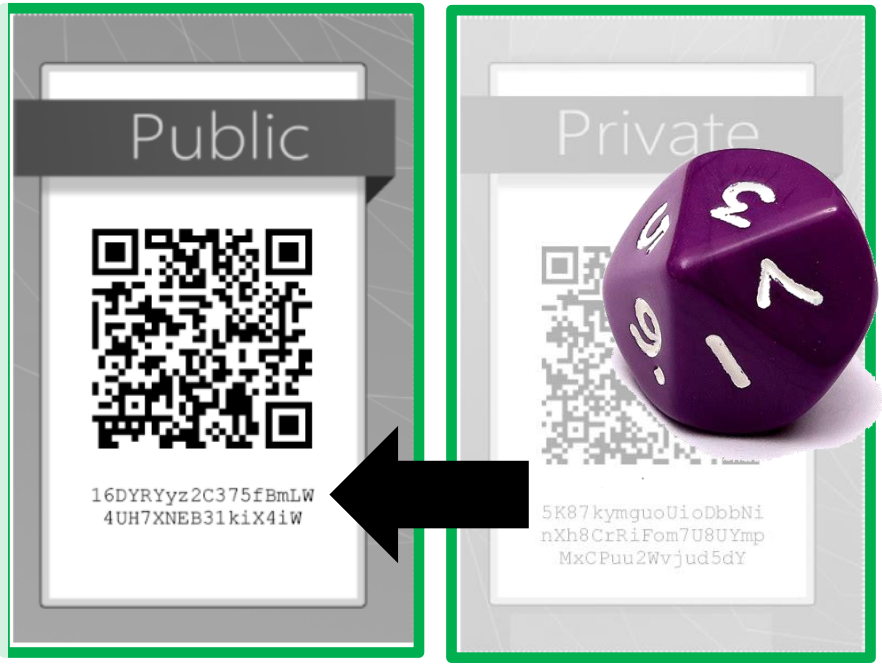
DÍKY ZA VÁŠ ČAS A V PONDĚLÍ ZASE NA VIDĚNOU

Bonus ☺

- Technické aspekty Bitcoinu
- Souvislost s programováním a jazykem C
- Každý týden jiná oblast

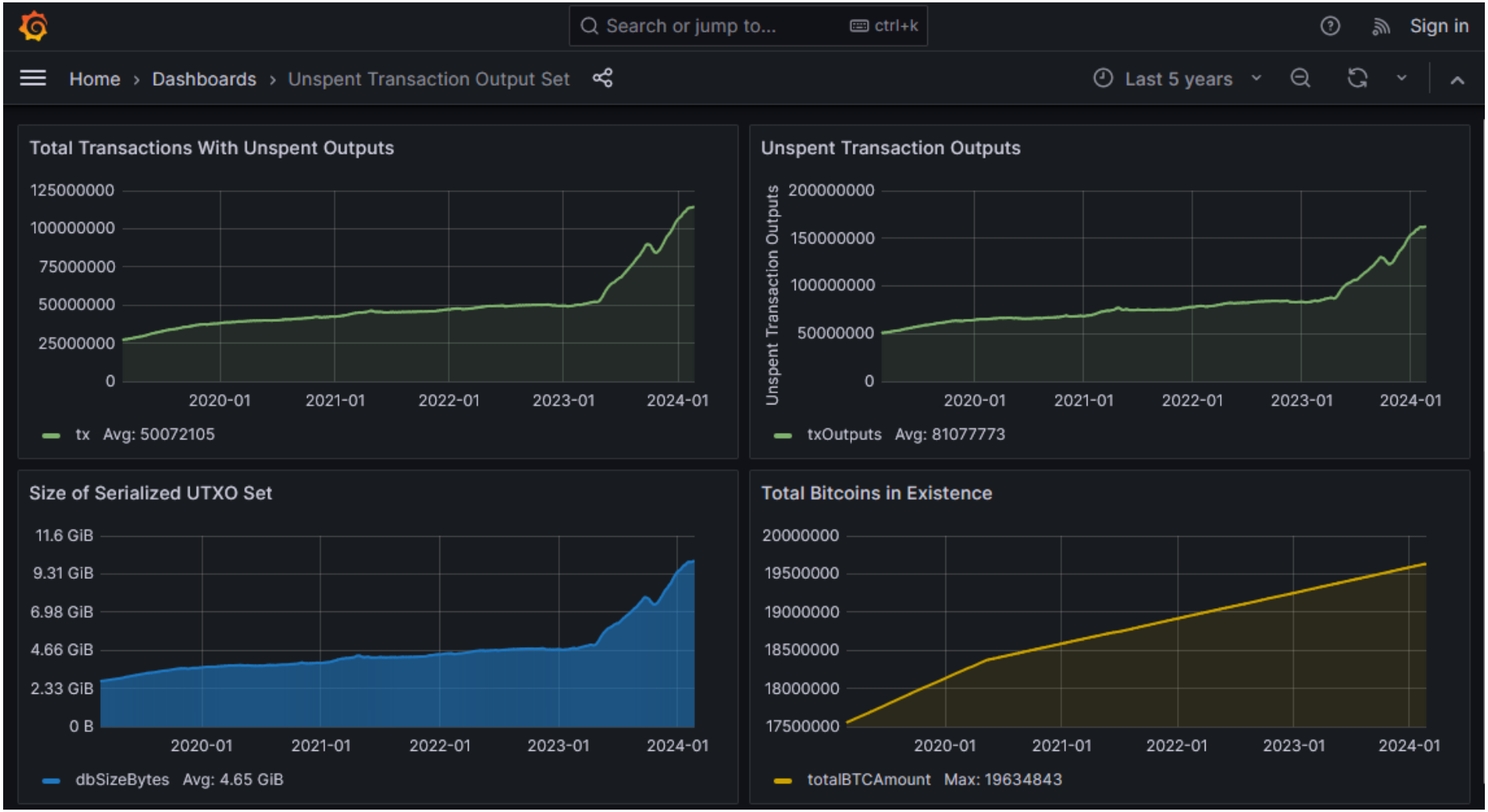
- Nebudeme probírat predikce ceny ani obchodování

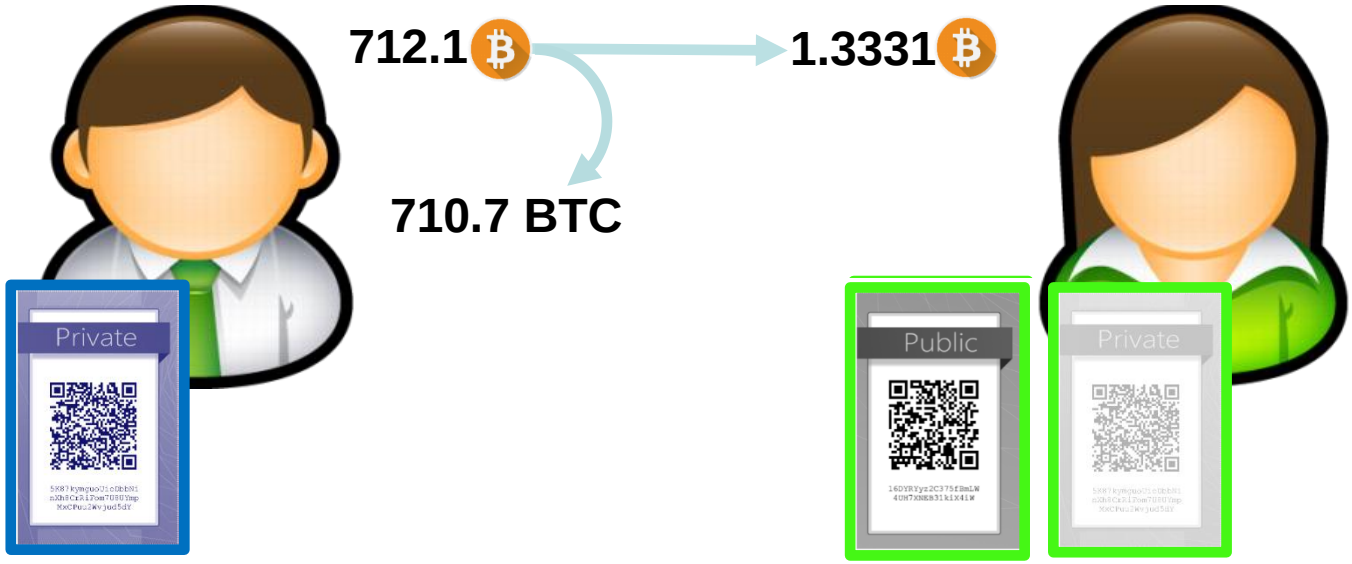
2014
0.5₿
HALF BITCOIN


Public
16DYRYyz2C375fBmLW
4UH7XNEB31kiX4iW
Private
5K87kymguoUioDbbNi
nXh8CrRiFom7U8UYmp
MxCPuu2Wvjud5dY

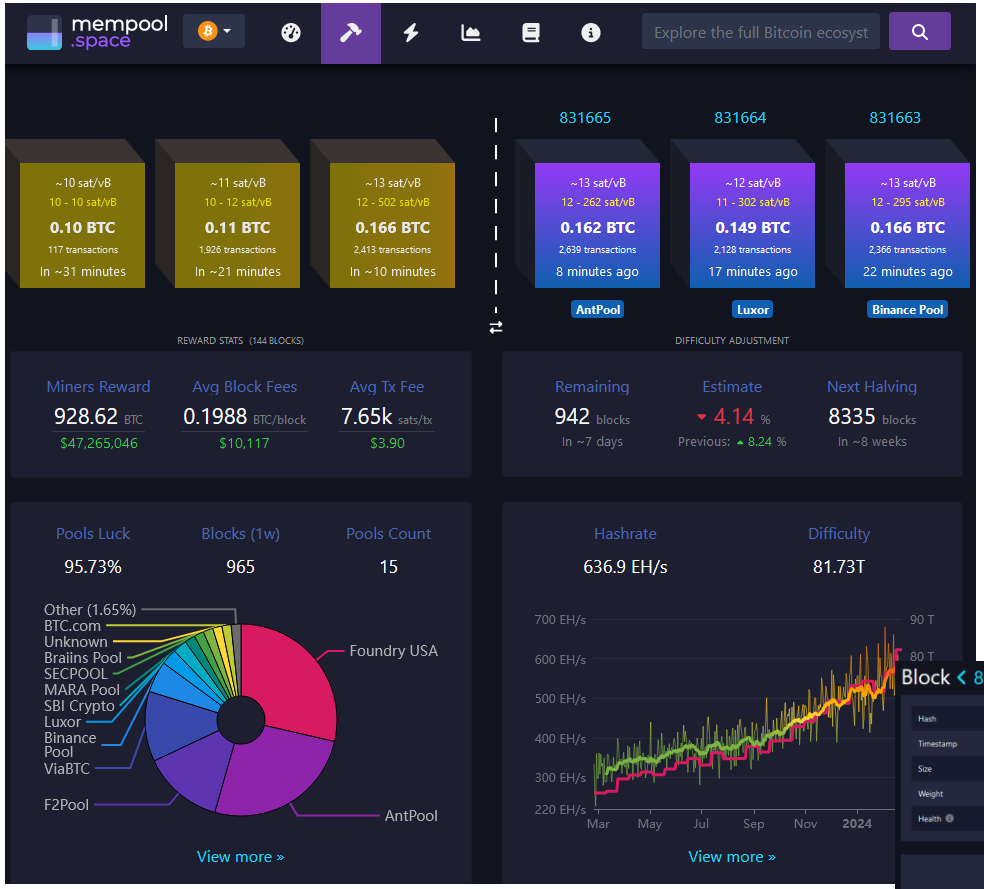
https://statoshi.info/d/000000009/unspent-transaction-output-set?orgId=1&refresh=10m&from=now-5y&to=now

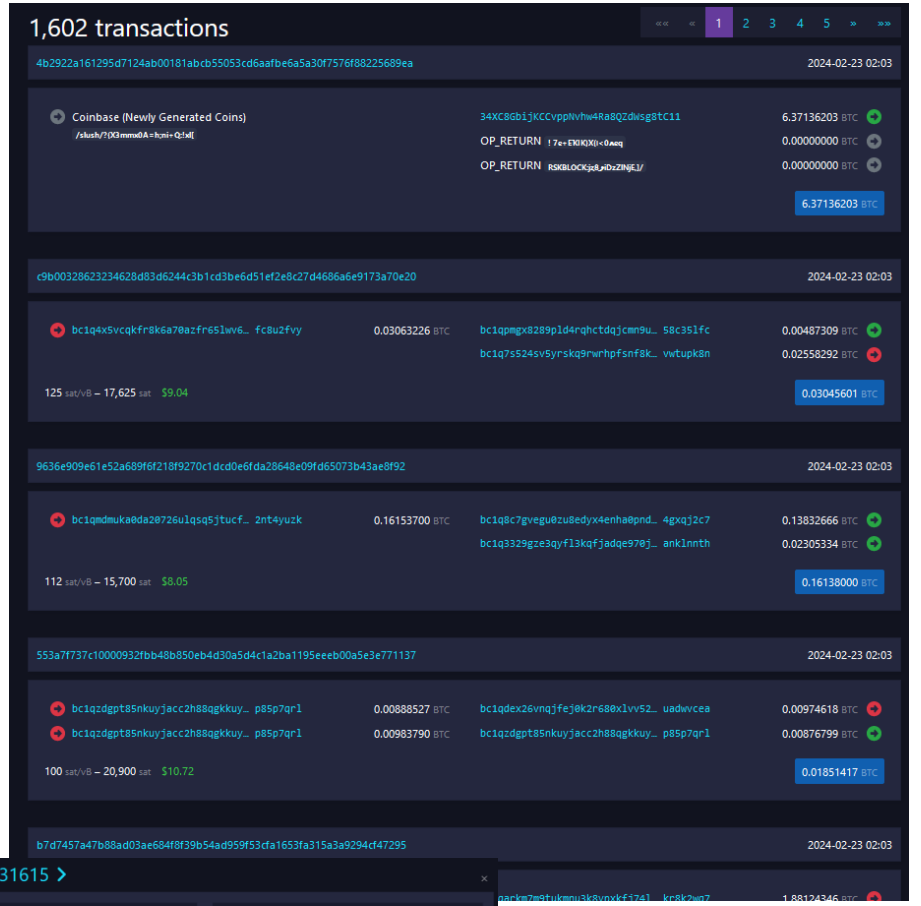
Kde jsou ty Bitcoiny? UTXO set




712.1
1.3331
710.7 BTC
Private
Public
Private

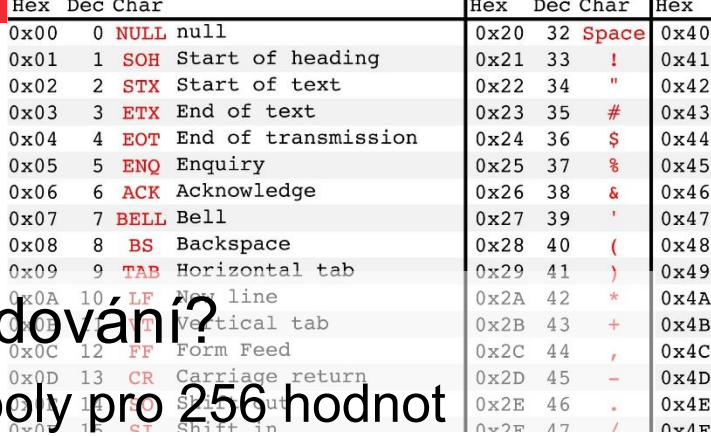
Blockchain explorer





- https://mempool.space/

Bonus – base-58 kódování

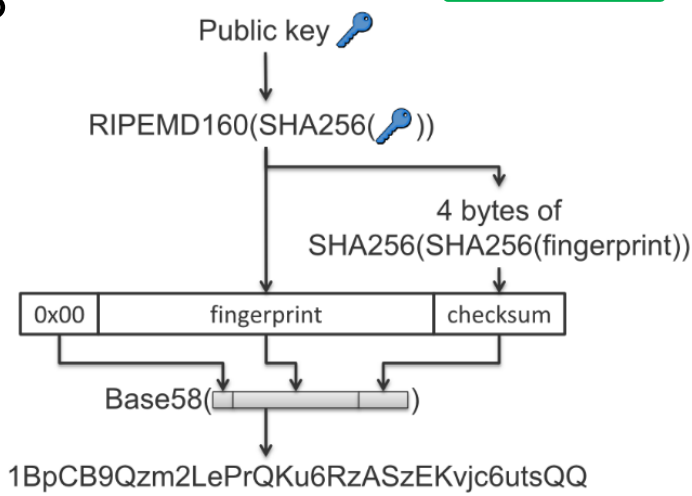


Hex	Dec	Char		Hex	Dec	Char		Hex
0x00	0	NULL	null	0x20	32	Space		0x40
0x01	1	SOH	Start of heading	0x21	33	!		0x41
0x02	2	STX	Start of text	0x22	34	"		0x42
0x03	3	ETX	End of text	0x23	35	#		0x43
0x04	4	EOT	End of transmission	0x24	36	$		0x44
0x05	5	ENQ	Enquiry	0x25	37	%		0x45
0x06	6	ACK	Acknowledge	0x26	38	&		0x46
0x07	7	BELL	Bell	0x27	39	'		0x47
0x08	8	BS	Backspace	0x28	40	(		0x48
0x09	9	TAB	Horizontal tab	0x29	41	)		0x49
0x0A	10	LF	New line	0x2A	42	*		0x4A
0x0B	11	VT	Vertical tab	0x2B	43	+		0x4B
0x0C	12	FF	Form Feed	0x2C	44	,		0x4C
0x0D	13	CR	Carriage return	0x2D	45	-		0x4D
0x0E	14	SO	Shift out	0x2E	46	.		0x4E
0x0F	15	SI	Shift in	0x2F	47	/		0x4F

- Kde není vhodné používat ASCII kódování?
  - ASCII tabulka – znaky a zástupné symboly pro 256 hodnot
  - Známý příklad – base-64
  - Typické užití je do URL nebo textu – lze kopírovat
- Proč base-58 místo base-64? (odstraněno 0, I, +, /, =)
  - Vhodné tam, kde se očekává že člověk ručně přepisuje
  - Nechceme `0OIl` znaky, které vypadají podobně
  - Některé systémy mohou mít obecné pole pro “číslo účtu”, kde mohou být jen alfanumerické znaky (tj. ne +, /, =)
  - Dvojklik v existujících editorech vybere celý text pokud je jenom z alfanumerických znaků
  - Typicky se nezalomí v emailu (není “na čem” z výjimkou tvrdého počtu znaků na řádek)
- Ale kde se base-58 vůbec používá?

Bitcoin adresy

- Bitcoin adresa může být kódována v base58
  - Base58Check: základ + verze + 4 bajty checksum
- Pay-to-pubkey-hash (p2pkh)
  - RIPEMD160(SHA256(ECDSA_publicKey))
  - Adresa začíná znakem 1, starší varianta
- Pay-to-script-hash (p2sh)
  - RIPEMD160(SHA256(redeemScript))
  - Adresa začíná s 3, novější varianta



- Novější formát bech32 (adresa začíná bc1), už nepoužívá base58
- Úplně nejnovější formát je bech32m pro Taproot (p2tr), začíná bc1p
- Přečtete si víc na
  - https://en.bitcoin.it/wiki/Base58Check_encoding
  - https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses#How_to_create_Bitcoin_Address
  - https://en.bitcoin.it/wiki/BIP_0173