

# PB071 – Programování v jazyce C

Datové typy, operátory, řídicí struktury

# 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 (details na cvičení)
- odevzdání do fakultního SVN, spuštění notifikačního skriptu

## ● Studentští poradci

- dočasně místnost B011 (hala je uzavřena)
- dostupní pravidelně od tohoto týdne
- <http://cecko.eu/public/pb071>
- Kudos: pokud vám poradce dobře poradí, můžete mu udělit pochvalu:
- <https://is.muni.cz/auth/cd/1433/jaro2014/PB071/kudos>

# Kontrolní mail a jeho interpretace

## ● Hlavička mailu

Zátěž Aisy

[http://en.wikipedia.org/wiki/Load\\_%28computing%29](http://en.wikipedia.org/wiki/Load_%28computing%29)

Pokud je nad 20, může být problém v zátěži

Vysledek pro studenta Tomas Brukner UCO 574580 (login xbrukner), SVN revize 83.  
Cvicici tohoto studenta je Tomas Brukner.  
Cas vyhodnoceni je 21. 2. 2013 17:45:23, load 0.77, 0.46, 0.43, doba provedeni 0:04.

## ● Tělo mailu

Doba běhu celé kontroly. Pokud je v řádu minut, může být problém v zatížení Aisy

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

# Organizační - materiály

- Slidy

- [http://cecko.eu/public/pb071\\_cviceni](http://cecko.eu/public/pb071_cviceni)
- odpoledne/večer po přednášce aktualizace na finální

- Video nahrávky

- <https://is.muni.cz/auth/el/1433/jaro2014/PB071/um/vi/>
- objevuje se typicky do druhého dne

- Tutoriály

- <http://cecko.eu/public/pb071>

- Twitter

- <https://twitter.com/rngsec>
- zveřejnění přípravy a slidů, občasné info
- hash tag **#pb071\_2014**
- (opravdu důležité věci budou rozesílány hromadně na IS mail)

# Správné použití SVN

- Vytvoříte repozitář (jednou)
  - Provedete Update (resp. SVN Checkout poprvé)
  - Děláte lokální změny v adresáři (editace souborů)
  - Provedete Commit
- NIKDY nemažte existující repozitář
  - všechny změny jsou uchovány v nejnovější revizi
  - není nutné repozitář vymazat nebo tak něco
  - pokud chcete odstranit soubor z repozitáře, smažete jej z lokálního adresáře a při Update zaškrtnete jeho odstranění
- SVN je case-sensitive (URL)
  - svn.fi.muni.cz interně neumožní PB071 a pb071
  - při pokusu o commit chyba ze strany SVN serveru
- <http://cecko.eu/public/svn>

# Proměnné a jejich datové typy



# Převod F2C ze cvičení

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

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

```
int main(void) {
```

```
    int fahr = 0;  
    float celsius = 0;  
    int dolni = 0;  
    int horni = 300;  
    int krok = 20;
```

datové typy, proměnné,  
přiřazení, konstanta

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

```
    for (fahr = dolni; fahr <= horni; fahr += krok) {  
        celsius = 5.0 / 9.0 * (fahr - 32);  
        // vypise prevod pro konkretni hodnotu fahrenheitu  
        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)
  - hodnoty, kterých může objekt nabývat
    - např. `float` může obsahovat reálná čísla
  - operace, které lze/nelze nad objektem provádět
    - např. k celému číslu lze přičíst 10
    - např. k řetězci nelze podělit číslem 5
- 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](http://en.wikipedia.org/wiki/Strongly_typed_programming_language)
- Assembler << VB/PHP << C << C++/Java << Ada/SPARK
- C je spíše **slabě** typový 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)
  - **wchar\_t** (typ pro uchování UNICODE znaků)
- Viz. [http://en.wikipedia.org/wiki/C\\_data\\_types](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
  - long int prom; nebo také long prom;
- **long long** – delší verze typu

# Primitivní datové typy - ukázka

## Základní typy:

Kategorie	Základní typ (kurzíva: jen v C99)	Modifikátory
Booleovský (Jen v C99)	<code>_Bool</code>	
Celočíselné	<code>int</code>	<code>short/long</code> (v C99 i <code>long long</code> ) <code>unsigned/signed</code>
	<code>char</code>	<code>unsigned/signed</code>
	<code>wchar_t</code> (Jen v C99)	<code>unsigned/signed</code>
Reálné	<code>float</code>	
	<code>double</code>	<code>long</code>
Komplexní (Jen v C99)	<code>_Complex</code> <code>_Imaginary</code>	<code>float/double/long double</code> (1 z těchto modifikátorů musí být uveden)
Prázdný	<code>void</code>	

Převzato z <http://www.fi.muni.cz/usr/jkucera/pb071/sl2.htm>

# Primitivní datové typy - velikost

- Velikost typů se může lišit dle platformy
  - 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)
- Velikosti na architektuře x86 jsou typicky
  - char (8b), short (16b), int (32b), long (32b), float (32b), double (64b), long long (64b)

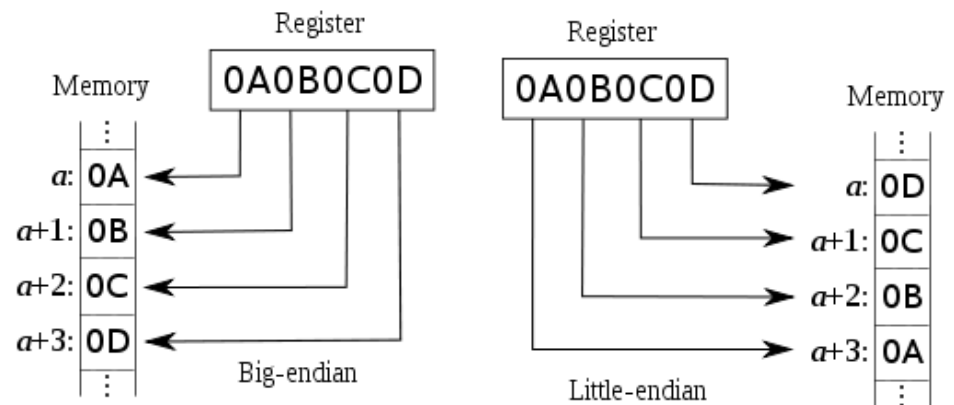
# Nejčastěji používané typy

- **char** – tisknutelný znak, součást řetězce
- **unsigned char** (někdy nazýván BYTE)
  - pro práci s binárními daty (např. obrázek)
- **int**
  - celé číslo se znaménkem, základní volba pro čítače u cyklů...
  - for (int i = 0...)
- **unsigned int** (někdy nazýván DWORD)
  - pro potenciálně velká neznaménková čísla
  - typicky offset (pozice) v rámci souboru, délky dat...
  - pozor na problém s přetečením DWORD! (fact)
- **size\_t** (od C99)
  - unsigned int typ alespoň 16 bitů velký
- **float/double**
  - číslo s desetinnými místy



# Big vs. little endian

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

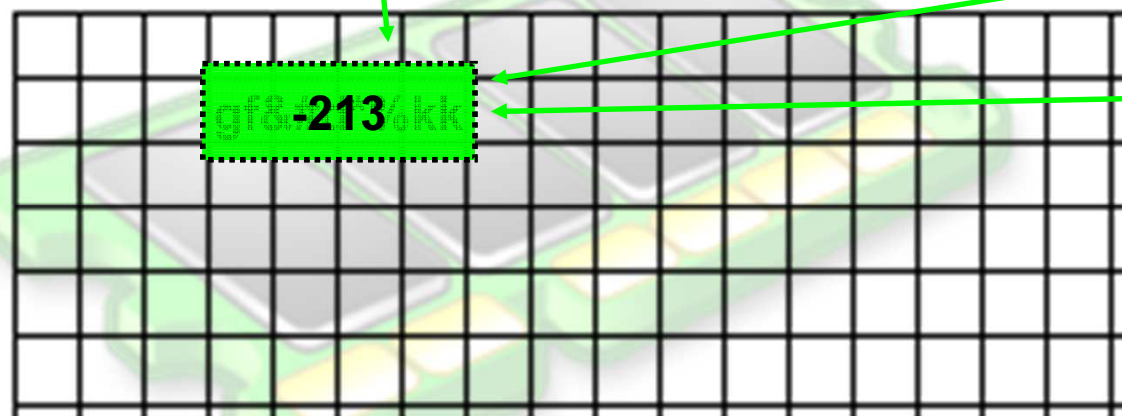


[http://en.wikipedia.org/wiki/Little\\_endian](http://en.wikipedia.org/wiki/Little_endian)

# 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

adresa 0x1234



```
int main() {  
    int test;  
  
    test = -213;  
  
    return 0;  
}
```

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

# Deklarace proměnné před prvním použitím

- Proměnná musí být definována/deklarována před prvním použitím
  - určíme zároveň jméno (**test**) a typ (**int**)
  - můžeme zároveň inicializovat (přiřadit hodnotu)
    - silně doporučeno, jinak náhodná 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

instrukce assembleru pro  
zápis do paměti

- např. `promenna = -213;`

- Překladač nahrazuje jméno proměnné její adresou

- typicky relativní k zásobníku

relativní adresa proměnné  
k adrese zásobníku

- `movl $0xffffffff2b,0xc(%esp)`

-213 hexadecimálně

počáteční adresa  
zásobníku pro  
danou funkci

- *Demo QTCreator instruction-wise mode*

# Jména proměnných

- Pravidla pojmenování (povinné, jinak syntaktická chyba)
  - musí začínat znakem nebo `_` (NE např. `int 3prom;` )
  - může obsahovat pouze znaky, cifry a podtržítko
  - záleží na velikosti znaků (`int prom; int Prom;`)
  - nesmí být klíčové slovo jazyka (NE např. `int switch = 1;`)
- Konvence pojmenování (doporučené)
  - 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 začínějte malým písmenem
  - nezačínějte proměnné `__` nebo `_X` (`X` je libovolné velké písmeno)
    - rezervované, typicky různé platformě závislá makra apod.
  - odděľujte 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
- 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. <http://www.fi.muni.cz/usr/jkucera/pb071/sl2.htm>
  - [http://www.difranco.net/compsci/C\\_Operator\\_Precedence\\_Table.htm](http://www.difranco.net/compsci/C_Operator_Precedence_Table.htm)



# 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)
  - 0 je nepravda (~FALSE)
    - pozor na 0 v reálném čísle (nemusí být přesně 0)
- <, >, <=, >=, ==, !=
- Ukázky
  - `prom > 30`
  - `prom != 55`
  - `55 <= prom`
  - `55 != prom`

# Porovnávací op

```
#include <stdio.h>
int main(void) {
    int prom = 0;
    if (prom = 0) printf("Never printed");
    if (prom = 1) printf("Never say never!");
    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](http://publications.gbdirect.co.uk/c_book/chapter8/sequence_points.html)

# 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

- $\&$ ,  $|$ ,  $\sim$ ,  $\wedge$ ,  $\ll$ ,  $\gg$
- Pracují jako logické operátory, ale na úrovni jednotlivých bitů operandů

- AND:  $Z = X \& Y$ ;

0110 & 1100 = 0100
--------------------------

- OR:  $Z = X | Y$ ;

0110   1100 = 1110
--------------------------

- XOR:  $Z = X \wedge Y$ ;

0110 $\wedge$ 1100 = 1010
---------------------------------

- INVERT:  $Z = \sim X$ ;

$\sim$ 0110 = 1001
-----------------------

- LSHIFT:  $Z = X \ll 2$ ;

00000110 $\ll 2$ = 00011000
-----------------------------------

- RSHIFT:  $Z = X \gg 2$ ;

00000110 $\gg 2$ = 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)

```
0001
| 0100
= 0101
```

- Vložení hodnoty na pozici X

- vypočteme masku jako  $mask = 2^{(X-1)}$  (indexujeme od 0, tedy X - 1)

- např. třetí bit  $\rightarrow 2^2 \rightarrow 4 \rightarrow 0x04$  hexadecimálně

- aplikujeme operaci OR s vypočtenou maskou

- `flags = flags | 0x04; // set 3th bit to 1`

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

- Zjištění hodnoty z pozice X

- vypočteme masku  $= 2^{(X-1)}$

- `if (flags & 0x04) { /* ... */ }`

- Zjištění hodnoty dolního bajtu

- maska pro celý bajt je 255, tedy 0xFF

- `unsigned char lowByte = flags & 0xFF;`

```
...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  $X * 2^{\text{posun}}$ 
  - $0011_2$  (3 dekadicky)
  - $0011_2 \ll 1 \rightarrow 0110_2$  (6 dekadicky)
  - $0011_2 \ll 2 \rightarrow 1100_2$  (12 dekadicky)
- Pozor na rozlišení & a &&, resp. | a ||
  - např. záměna & za &&
    - $1100 \ \&\& \ 0011 == \text{TRUE}$
    - $1100 \ \& \ 0011 == 0 \ (\text{FALSE})$

# 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://www.difranco.net/compsci/C\\_Operator\\_Precedence\\_Table.htm](http://www.difranco.net/compsci/C_Operator_Precedence_Table.htm)
- $\text{prom1} = 5 + (\text{prom1} * 18) \% \text{prom2} - \text{prom1} == 2;$
- $\text{prom1} = 5 + ((\text{prom1} * 18) \% \text{prom2}) - \text{prom1} == 2;$
- $\text{prom1} = (5 + ((\text{prom1} * 18) \% \text{prom2})) - \text{prom1} == 2;$
- $\text{prom1} = ((5 + ((\text{prom1} * 18) \% \text{prom2})) - \text{prom1}) == 2;$
- $\text{prom1} = (((5 + ((\text{prom1} * 18) \% \text{prom2})) - \text{prom1}) == 2);$
- $(\text{prom1} = (((5 + ((\text{prom1} * 18) \% \text{prom2})) - \text{prom1}) == 2));$

# Pořadí vyhodnocení operátorů

Nejvyšší priorita - zleva doprava:	
()	Volání funkce, např. <code>sqrt(2*alfa)</code> (něco jiného jsou uzávorkované výrazy)
[]	Prvek pole
.	Výběr prvku struktury
->	Výběr prvku struktury zadané ukazatelem
Nižší priorita - ZPRAVA DOLEVA:	
!	Logická negace
~	Jednotkový doplněk (z nulových bitů udělá jednotkové a naopak)
+	Potvrzení znaménka (unární plus)
-	Změna znaménka (unární minus)
++	Inkrementace (přičtení 1). Prefixový a postfixový tvar! Jak <code>i++</code> tak <code>++i</code> přičte k <code>i</code> jedničku, ale jako výsledek nevrací totéž!
--	Dekrementace (odečtení 1). Prefixový a postfixový tvar!
&	Vytvoření ukazatele: <code>&amp;x</code> vytvoří ukazatel (adresu) na <code>x</code>
*	Dereference ukazatele: je-li <code>p</code> ukazatel, <code>*p</code> je hodnota, na niž ukazuje
(typ)	Přetypování (konverze typu): <code>(double)pocet</code> převede hodnotu proměnné <code>pocet</code> na hodnotu typu <code>double</code>
sizeof	Velikost objektu (násobek délky typu <code>char</code> , v C99 přímo počet bajtů). Operandem je (typ) nebo výraz: <code>sizeof(long double)</code> , <code>sizeof pocet</code>
Nižší priorita - zleva doprava:	
*	Násobení (binární *)
/	Dělení
%	Modulo (zbytek po celočíselném dělení)
Nižší priorita - zleva doprava:	
+	Sčítání
-	Odčítání

Nižší priorita - zleva doprava:	
<<	Bitový posun vlevo: $x \ll 3$ posune o 3 bity doleva, čili násobí 8
>>	Bitový posun vpravo: $x \gg n$ posune o $n$ bitů doprava, čili dělí $2^n$
Nižší priorita - zleva doprava:	
<	Menší než - výsledkem je 1 (true) nebo 0 (false)
<=	Menší nebo rovno (dtto)
>	Větší než (dtto)
>=	Větší nebo rovno (dtto)
Nižší priorita - zleva doprava:	
==	Test na rovnost (dtto)
!=	Test na nerovnost (dtto)
Nižší priorita - zleva doprava:	
&	Logický součin bit po bitu (and) - provádí se nad stejnoelektrými bity obou operandů
Nižší priorita - zleva doprava:	
^	Logické vylučovací nebo bit po bitu (xor) (dtto)
Nižší priorita - zleva doprava:	
	Logický součet bit po bitu (or) (dtto)
Nižší priorita - zleva doprava:	
&&	Logický součin (and) - celý operand se považuje za true, je-li nenulový nebo za false je-li roven nule; výsledkem je 1 (true) nebo 0 (false)
Nižší priorita - zleva doprava:	
	Logický součet (or) (dtto)
Nižší priorita - ZPRAVA DOLEVA:	
?:	Podmíněný výraz: <i>podmínka</i> ? <i>výraz1</i> : <i>výraz2</i> znamená, že je-li <i>podmínka</i> splněna, výsledkem je <i>výraz1</i> , v opačném případě <i>výraz2</i>
Nižší priorita - ZPRAVA DOLEVA:	
= += -= *= /= %= <<= >>= &= ^=  =	Přřazení, resp. kombinace jiná operace spolu s přřazením. Např. $x*=y$ znamená $x=x*y$
Nejnižší priorita - zleva doprava:	
,	Postupné provedení (Umožňuje zapsat více výrazů tam, kde syntakticky smí být jen jeden; výsledkem je hodnota posledního výrazu. Hodnota předchozích výrazů se zapomene, mohou mít ovšem vedlejší účinky.)

# 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!
- nespolehejte 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** (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;`

- Problematika závorek

- pokud je ve větvi **then/else** jediný výraz, není nutné psát { }
- pozor na problém při pozdějším rozšiřování kódu u **else**

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

```
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 řídicí proměnné
- *podmínka\_provedení* – pokud TRUE, tak proběhne tělo cyklu
  - typicky test řídicí proměnné vůči koncové hodnotě
- *inkrementální\_výraz* – provede se po konci každé iterace
  - typicky změna řídicí proměnné

- Používáno často pro cykly s daným počtem iterací

- ne nutně fixním během překladu, ale ukončovací podmínka stejná
- např. projítí pole od začátku do konce

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

# 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 fahrenheitu  
    printf("%d \t %d \n", fahr, celsius);  
    // zmena ridici promenne  
    fahr = fahr + krok;  
}  
// ...
```

# Cyklus DO - WHILE

- Cyklus DO-WHILE

- *podmínka\_provedení* – pokud TRUE, tak proběhne další iterace cyklu
  - typicky test řídicí proměnné vůči koncové hodnotě
  - testuje se PO těle cyklu
- typicky v těle modifikujeme řídicí proměnnou
- tělo cyklu vždy proběhne alespoň jednou!

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

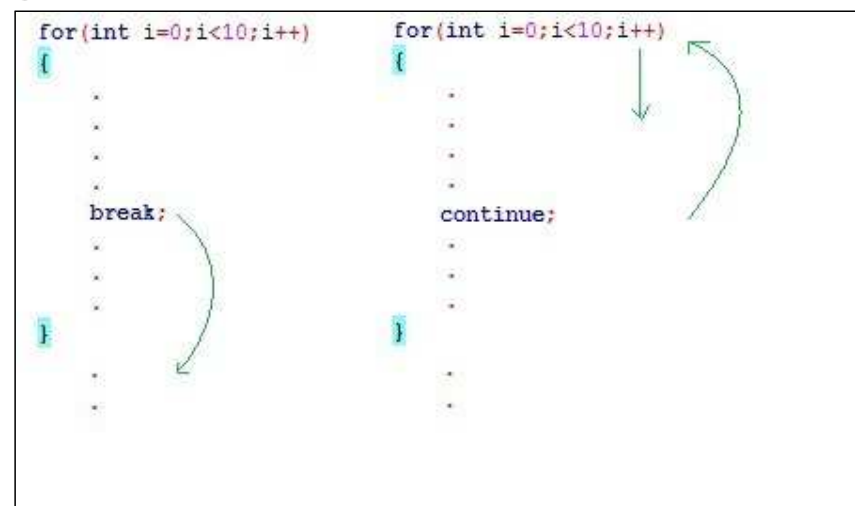
```
// ...  
fahr = dolni;  
if (fahr <= horni) {  
    do {  
        celsius = 5.0 / 9.0 * (fahr - 32);  
        // vypise prevod pro konkretni hodnotu fahrenheitu  
        printf("%d \t %d \n", fahr, celsius);  
        // zmena ridici promenne  
        fahr = fahr + krok;  
    } while (fahr 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

# 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
- (**exit**) – ukončení celého programu
- (goto)



Lze použít pro všechny cykly

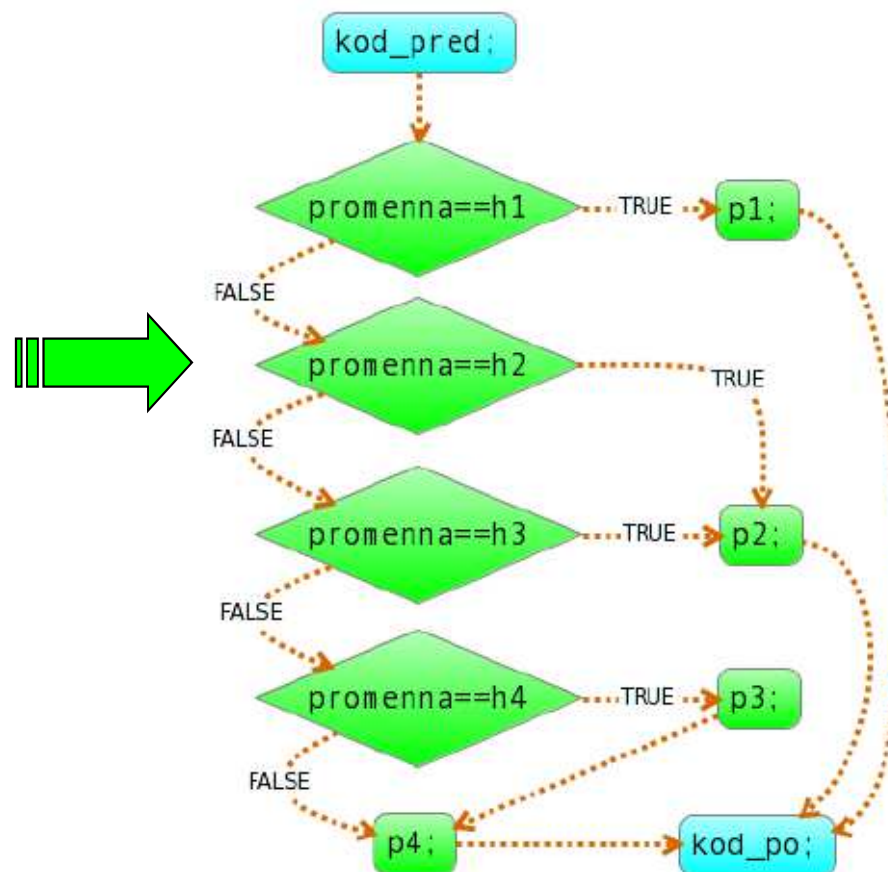


# switch

- 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
  - 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

```
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;
    }
}
```

*ukázka (ne)využití **break**;*

```
int value = 0;
// ...
switch (value) {
    case 1:
    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;
    }
}
```



# Operátor sekvenčního provedení ,

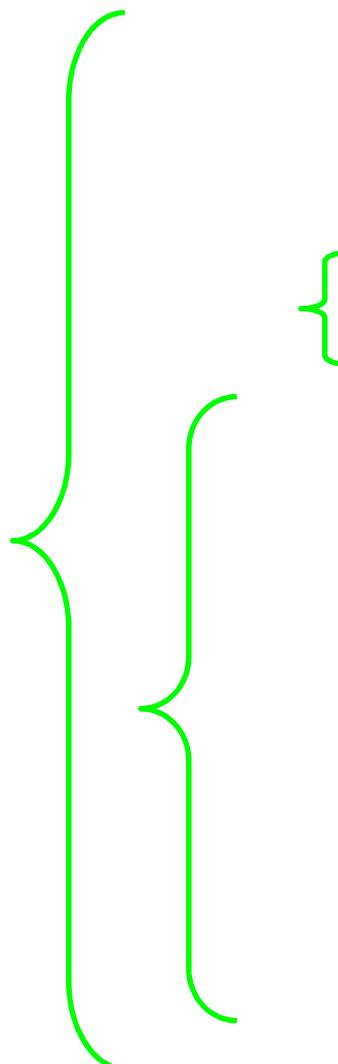
- Zřetězí vykonávání několika výrazů
  - vyhodnotí se všechny zleva doprava
- Lze umístit tam, kde lze umístit jeden výraz
  - jako výstup zřetězení je výstup posledního výrazu
- Nepoužívejte
  - spoléhá na vedlejší efekty předchozích výrazů
    - jejich vyhodnocení je zahozeno, tedy vliv jen vedlejšími
  - ztěžuje porozumění kódu

```
int x = 2;  
if (x == 3) printf("x == 3");  
if (x++, x == 3) printf("x == %d", x);  
if (foo(&x), x == 3) printf("x == %d", x);
```

# 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 řídicích 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



```
#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);
}

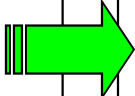
int main(void) {
    int low = 0;
    int high = 300;
    int step = 20;
    for (int fahr = low; fahr <= high; fahr += step) {
        float celsius = f2c(fahr);
        if (celsius > 0) {
            printf("%3d \t %6.2f \n", fahr, celsius);
        }
    }
    return 0;
}
```

# Vnořené příkazy

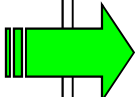
- Příkazy lze vnořit do sebe
  - další příkaz je součástí vnitřního bloku
  - např. vnořené if-else
- Pozor na nevhodné hluboké vnoření
  - řešením je přeformátování kódu
  - např. záměna za switch

# 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;  
    // ...  
}
```

# Vnořené příkazy

- Pozor na příslušnost **else** k odpovídajícímu **if**
  - **if if else** -> **if příkaz**;
  - vhodné zdůraznit pomocí { }

```
if (day > 5) if (day == 7) printf("Nedele");  
else printf("Pracovni den nebo Sobota?");
```

```
if (day > 5) {  
    if (day == 7) printf("Nedele");  
    else printf("Sobota?");  
}
```



# Shrnutí

```
#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 fahrenheitu  
        printf("%3d \t %6.2f \n", fahr, celsius);  
    }  
    return 0;
```

```
}
```

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() {
    int i = 0;
    printf("while (i++) {\n");
    while (i++) {
        printf("%d\n", i);
        if (i == 2) break;
    }
    i = 0;
    printf("while (++i) {\n");
    while (++i) {
        printf("%d\n", i);
        if (i == 2) break;
    }
    printf("for (i = 0; i <=2; i++) {\n");
    for (i = 0; i <=2; i++) {
        printf("%d\n", i);
    }
    printf("for (i = 0; i <=2; ++i) {\n");
    for (i = 0; i <=2; ++i) {
        printf("%d\n", i);
    }
    return 0;
}

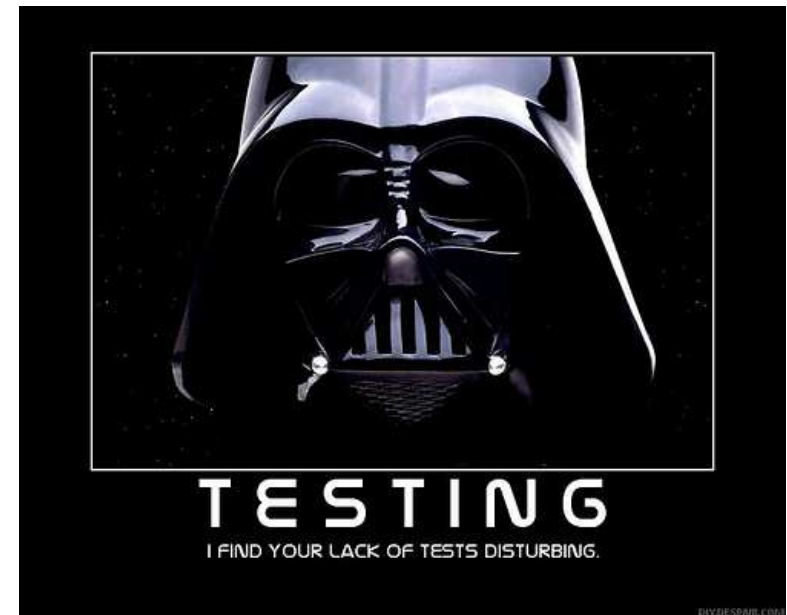
```

```

VÝSTUP:
while (i++) {
while (++i) {
1
2
for (i = 0; i <=2; i++) {
0
1
2
for (i = 0; i <=2; ++i) {
0
1
2

```

# Testování



# 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

# Co testovat na vlastním kódu?

- V zadání domácího úkolu jsou ukázkové vstupy a odpovídající výstupy
- Ukázková implementace pro domácí úkol umožňuje vytvořit seznam vstupů a očekávaných výstupů
- Vrací celý program 0 při korektním průběhu?
- Vrací program hodnotu  $\neq 0$  při chybě?

```
int main() {  
    printf("Hello World")  
    return 0;  
}
```



```
int main() {  
    int status = 0;  
    printf("Hello World")  
    return status;  
}
```

# Jak testovat?

- Test funkce (různé vstupy a očekávané výstupy)
  - Typické možnosti
  - Krajní možnosti
  - Negativní testy (opravdu nefunguje tam kde nemá fungovat)?
- Test celého programu
  - (soubor se standardním vstupem + diff)
- Automatizace všech ručně zadávaných hodnot
  - Ukázka na scanf (cvičení)
- Automatizace kontroly výstupu
  - Diff + očekávaný výstup
  - Další vlastní program na kontrolu

# 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