

# B0B99PRPA – Procedurální programování

## Řídící struktury, výrazy

Stanislav Vítek

Katedra radioelektroniky  
Fakulta elektrotechnická  
České vysoké učení v Praze

# Přehled témat

- Část 1 – Řídicí struktury

  - Kódovací styl

  - Řízení běhu programu

  - Konečnost cyklu

- Část 2 – Výrazy

  - Výrazy a operátory

  - Přiřazení

- Část 3 – Zadání 3. domácího úkolu

# Část I

## Řídicí struktury

# I. Řídicí struktury

Kódovací styl

Řízení běhu programu

Konečnost cyklu

## Jak psát programy?

```

1  /* International Obfuscated C Code Contest 1990 */
2  v,i,j,k,l,s,a[99]
3  main()
4  {
5      for(scanf("%d", &s);*a-s;v=a[*j=v]-a[i],k=i<s,j+=(v=j<s
        &&(!k&&!printf(2+"\n\n%c"-!l<<!j)," #Q"[1^v?(1^j)
        &1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-j))&&(1%=s),v||(i==
        j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);
6  }
```

<https://www.ioccc.org/1990/baruch.hint>

# I. Řídicí struktury

Kódovací styl

Řízení běhu programu

Konečnost cyklu

## Příkazy řízení běhu programu

- Podmíněné řízení běhu programu
  - Podmíněný příkaz: `if ()` nebo `if () ... else`
  - Programový přepínač: `switch () case ...`
- Cykly
  - `for ()`
  - `while ()`
  - `do ... while ()`
- Nepodmíněné větvení programu
  - `continue`
  - `break`
  - `return`
  - `goto`

## Podmíněné větvení `if`

```
if (podminka) prikaz1 else prikaz2
```

- Je-li `podminka` `!= 0`, provede se příkaz `prikaz1` jinak `prikaz2`  
Příkazem může být i složený příkaz nebo blok
- Část `else` je nepovinná
- Podmíněné příkazy mohou být vnořené a můžeme je řetězit

### Příklad

```
int max;  
if (a > b) {  
    if (a > c) {  
        max = a;  
    }  
}
```

```
if (a > b) {  
    // ...  
} else if (a < c) {  
    // ...  
} else if (a == b) {  
    // ...  
} else {  
    // ...  
}
```



## Podmíněné větvení **if** – příklad

- Jestliže v případě splnění či nesplnění podmínky má být provedeno více příkazů, je třeba z nich vytvořit složený příkaz nebo blok.

**Příklad** jestliže  $x < y$ , vyměňte hodnoty těchto proměnných

```
if (x < y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

Co se stane, když za příkazem větvení nebude blok?

```
if (x < y)  
    int tmp = x;  
    x = y;  
    y = tmp;
```

## Podmíněné větvení **if** – příklad

Do proměnné **min** uložte menší z čísel **x** a **y** a do proměnné **max** uložte větší z čísel.

```
if (x < y) {  
    min = x;  
    max = y;  
} else {  
    min = y;  
    max = x;  
}
```

Špatné řešení:

```
if (x < y)  
    min = x;  
    max = y;  
else min = y; //unexpected token else  
    max = x;
```

## Programový přepínač `switch`

- Příkaz `switch` (přepínač) umožňuje větvení programu do více větví na základě různých hodnot výrazu výčtového (celočíslného) typu, jako jsou např. `int`, `char`, `short`, `enum`.
- Tvar příkazu

```
switch (vyraz) {  
    case konstanta1: prikazy1; break;  
    case konstanta2: prikazy2; break;  
    : : :  
    case konstantaN: prikazyN; break;  
    default: prikazydef; break;  
}
```

- konstanty jsou téhož typu jako výraz a příkazy jsou posloupnosti příkazů

U větvení `switch` se nepoužívají složené příkazy.

## Programový přepínač `switch`

- Přepínač `switch(vyraz)` větří program do `N` větví
- Hodnota `vyraz` je porovnávána s `N` konstantními výrazy typu `int`  
příkazy `case konstantai: ...`
- Hodnota `vyraz` musí být celocíselná a hodnoty `konstantai` musí být navzájem různé
- Pokud je nalezena shoda, program pokračuje od tohoto místa dokud nenajde příkaz `break` nebo konec příkazu `switch`
- Pokud shoda není nalezena, program pokračuje nepovinnou sekcí `default`  
Sekce `default` se zpravidla uvádí jako poslední
- Příkazy `switch` mohou být vnořené

## Programový přepínač **switch** – příklad

```
switch (n) {  
  case 1:  
    printf("*");  
    break;  
  case 2:  
    printf("**");  
    break;  
  case 3:  
    printf("***");  
    break;  
  case 4:  
    printf("****");  
    break;  
  default:  
    printf("----");  
}
```

```
if (n == 1) {  
  printf("*");  
} else if (n == 2) {  
  printf("**");  
} else if (n == 3) {  
  printf("***");  
} else if (n == 4) {  
  printf("****");  
} else printf("----");
```

- Co se vypíše, pokud ve větvích nebudou příkazy **break** a  $n=3$ ?

# Cykly

- Cyklus **for** a **while** testuje podmínku opakování před vstupem do těla cyklu
  - **for** – inicializace, podmínka a změna řídicí proměnné jsou součástí syntaxe

```
for (int i = 0; i < 5; ++i) {  
    // telo cyklu  
}
```
  - **while** – řídicí proměnná v režii programátora

```
int i = 0;  
while (i < 5) {  
    // telo cyklu  
    i += 1;  
}
```
- Cyklus **do** testuje podmínku opakování po prvním provedení cyklu

```
int i = -1;  
do {  
    // telo cyklu  
    i += 1;  
} while (i < 5);
```

## Cykly **while** a **do-while**

- Tvar příkazu **while**

```
while (podminka) prikaz
```

- Tvar příkazu **do-while**

```
do prikaz while (podminka)
```

### Příklad

```
q = x;  
while (q >= y) {  
    q = q - y;  
}
```

```
q = x;  
do {  
    q = q - y;  
} while (q >= y);
```

- Jaká je hodnota proměnné **q** po skončení cyklu?

## Cyklus **while** – příklad

Program na výpočet faktoriálu přirozeného čísla

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      int i = 1, f = 1, n;
6      printf("zadejte prirozene cislo: ");
7      scanf("%d", &n);
8      while (i<n) {
9          i = i+1;
10         f = f*i;
11     }
12     printf("%d! = %d\n", n, f);
13     return 0;
14 }
```

$$n! = \prod_{k=1}^n k$$



## Cyklus for

- Tvar příkazu

```
for (inicializace; podminka; zmena) prikaz
```

- Odpovídá cyklu while ve tvaru:

```
inicializace;  
while (podminka) {  
    prikaz;  
    zmena;  
}
```

### Příklad

```
for (int i = 0; i < 10; ++i) {  
    printf("i = %i\n", i);  
}
```

Změnu řídicí proměnné lze zapsat operátorem inkrementace `++` nebo dekrementace `--`, lze též použít zkrácený zápis přiřazení, např. `+=`.

- Výrazy `inicializace` a `zmena` mohou být libovolného typu
- Libovolný z výrazů lze vynechat
- `break` – cyklus lze nuceně opustit příkazem `break`
- `continue` – část těla cyklu lze vynechat příkazem `continue`
- Při vynechání řídicího výrazu `podminka` se cyklus bude provádět nepodmíněně

## Cyklus **for** – příklady

- Různé varianty zápisu:

```
for (i = 0; i < 10; i++)
```

```
for (; a < 4.0; a += 0.2)
```

```
for (; i < 10; )
```

```
for (;;) /* Nekonecny cyklus */
```

```
for (;;) /* Nekonecny cyklus, ekv. while(1) */
```

- Nesprávné použití cyklu **for**:

```
for ( ) /* Chybi stredniky */
```

```
for (i = 1, i == x, i++) /* Carky místo středníku */
```

```
for ( x < 4 ) /* Chybi středníky */
```

## Příkaz `continue`

- Příkaz návratu na vyhodnocení řídicího výrazu
- Příkaz `continue` lze použít pouze v těle cyklu
  - `for ()`
  - `while ()`
  - `do-while ()`
- Příkaz `continue` způsobí prerušení vykonávání těla cyklu a nové vyhodnocení řídicího výrazu

### Příklad

```
int i;
for (i = 0; i < 20; ++i) {
    if (i % 2 == 0) {
        continue;
    }
    printf("%d\n", i);
}
```

## Příkaz **break**

- Příkaz nuceného ukončení cyklu
- Příkaz **break** lze použít pouze v těle řídicích struktur
  - `for()`
  - `while()`
  - `do...while()`
  - `switch()`
- Po opuštění těla cyklu nebo těla `switch()` program pokračuje následujícím příkazem

### Příklad

```
int i = 10;
while (i > 0) {
    if (i == 5) {
        printf("i reaches 5, leave the loop\n");
        break;
    }
    i--;
    printf("End of the while loop i: %d\n", i);
}
```

## Příkaz `goto`

- Příkaz nepodmíněného lokálního skoku
- Syntax `goto navesti`;
- Příkaz `goto` lze použít pouze v těle funkce
- Příkaz `goto` předá řízení na místo určené návěštím `navesti`
- Skok `goto` nesmí směřovat dovnitř bloku, který je vnořený do bloku, kde je příslušné `goto` umístěno

### Příklad

```
int test = 3;
for (int i = 0; i < 10; i++) {
    if (i == test) {
        goto OUT;
    }
    printf ("i = %i\n", i);
}
return 0;
OUT: return -1;
```

# I. Řídicí struktury

Kódovací styl

Řízení běhu programu

Konečnost cyklu

## Konečnost cyklu

- Konečnost algoritmu – pro přípustná data skončí v konečné době
- Aby byl algoritmus konečný, musí každý cyklus v něm uvedený skončit po konečném počtu kroků
- Jedním z důvodů neukončení programu je zacyklení
  - Program opakovaně vykonává cyklus, jehož podmínka ukončení není nikdy splněna.

```
while (i != 0) {  
    j = i - 1;  
}
```

- Cyklus se neprovede ani jednou, nebo neskončí.
- Záleží na hodnotě řídicí proměnné `i` před voláním cyklu

## Konečnost cyklu

- Základní pravidlo pro konečnost cyklu
  - Provedením těla cyklu se musí změnit hodnota proměnné použité v podmínce ukončení cyklu

```
for (int i = 0; i < 5; ++i) {  
    ...  
}
```

- Uvedené pravidlo konečnost cyklu nezaručuje

```
int i = -1;  
while ( i < 0 ) {  
    i = i - 1;  
}
```

Konečnost cyklu závisí na hodnotě proměnné před vstupem do cyklu.



## Konečnost cyklu

```
while (i != n) {  
    ... //prikazy nemenici hodnotu promenne i  
    i++;  
}
```

- Vstupní podmínka konečnosti uvedeného cyklu
  - $i \leq n$  pro celá čísla

Jak by vypadala podmínka pro proměnné typu double?

---

- Splnění vstupní podmínky konečnosti cyklu musí zajistit příkazy předcházející příkazu cyklu
- Zabezpečený program testuje přípustnost vstupních dat

# Část II

## Výrazy

## II. Výrazy

Výrazy a operátory

Přiřazení

## Výrazy

- Výraz předepisuje výpočet hodnoty určitého vstupu
- Výraz může obsahovat
  - **operandy** – proměnné, konstanty, volání funkcí nebo jiné výrazy
  - **operátory**
  - **závorky**
- Pořadí operací předepsaných výrazem je dáno **prioritou** a **asociativitou** operátorů.

### Příklad

`10 + x * y // poradi vyhodnoceni 10 + (x * y)`

`10 + x + y // poradi vyhodnoceni (10 + x) + y`

\* má vyšší prioritu než +, + je asociativní zleva

## Výrazy a operátory

- **Výraz** se skládá z operátorů a operandů
  - Výraz sám může být operandem
  - Výraz má typ a hodnotu (Pouze výraz typu `void` hodnotu nemá.)
  - Výraz zakončený středníkem `;` je příkaz
- **Operátory** jsou vyhrazené znaky (ev. sekvence) pro zápis výrazu
  - Postup výpočtu výrazu s více operátory je dán prioritou operátoru
  - Postup výpočtu lze předepsat použitím kulatých závorek `( a )`
  - Obecně (mimo konkrétní případy) není pořadí vyhodnocení operandů definováno (nezaměňovat s asociativitou!)
    - Např. pro součet `f1() + f2()` není definováno, který operand se vyhodnotí jako první (tj. jaká funkce se zavolá jako první).
    - Pořadí vyhodnocení je definováno pro operandy v logickém součinu `AND` a součtu `OR`
- **Nedefinované chování** – vyhodnocení některých specifických výrazů není definováno a záleží na překladači: `i = ++i + i++;`

[https://en.cppreference.com/w/c/language/eval\\_order](https://en.cppreference.com/w/c/language/eval_order)

# Operátory

- Binární operátory
  - Aritmetické – sčřtání, odčřtání, násobení, dělení
  - Relační — porovnání hodnot (menřř, větřř, ... )
  - Logické — logický součet a součin
  - Operátor přřazenř – na levé straně operátoru = je proměnná
- Unární operátory
  - indikující kladnou/zápornou hodnotu: + a -  
operátor - modifikuje znaménko výrazu za nřm
  - modifikující proměnou: ++ a --
  - logický operátor doplněk: !
  - bitová negace (negace bit po bitu): ~
  - operátor pretypování: (jměno typu)
- Ternární operátor
  - podmíněné přřazenř hodnoty: ? :

[http://www.tutorialspoint.com/cprogramming/c\\_operators.htm](http://www.tutorialspoint.com/cprogramming/c_operators.htm)

## Aritmetické operátory

- Operandů aritmetických operátorů mohou být libovolného číselného typu

Výjimkou je operátor zbytek po dělení % definovaný pro int

*	Násobení	$x*y$	
/	Dělení	$x/y$	
%	Dělení modulo	$x\%y$	Zbytek po dělení $x$ a $y$
+	Sčítání	$x+y$	
-	Odčítání	$x-y$	
+	Kladné zn.	$+x$	
-	Záporné zn.	$-x$	
++	Inkrementace	$++x, x++$	Inkrementace před/po vyhodnocení výrazu
--	Dekrementace	$--x, x--$	Dekrementace před/po vyhodnocení výrazu

## Unární aritmetické operátory

- Unární operátory `++` a `--` mění hodnotu svého operandu  
Operand musí být **l-hodnota**, tj. výraz, který má adresu, kde je uložena hodnota výrazu (např. promenná)
  - lze zapsat prefixově, např. `++x` nebo `--x`  
Operace je provedena **před** vyhodnocením výrazu.
  - nebo postfixově např. `x++` nebo `x--`  
Operace je provedena **po** vyhodnocení výrazu.
  - v obou případech se však liší výsledná hodnota výrazu!

### Příklad

```
int i = 1, a;  
a = i++;    // i=2 a=1  
a = ++i;    // i=3 a=3  
a = ++(i++); // nelze, hodnota i++ není l-hodnota
```



## Relační operátory

- Operandy relačních operátorů mohou být číselného typu, ukazatele shodného typu nebo jeden z nich `NULL` nebo typ `void`

<code>&lt;</code>	Menší než	<code>x&lt;y</code>	1 pro x je menší než y, jinak 0
<code>&lt;=</code>	Menší nebo rovno	<code>x&lt;=y</code>	1 pro x menší nebo rovno y, jinak 0
<code>&gt;</code>	Větší	<code>x&gt;y</code>	1 pro x je větší než y, jinak 0
<code>&gt;=</code>	Větší nebo rovno	<code>x&gt;=y</code>	1 pro x větší nebo rovno y, jinak 0
<code>==</code>	Rovná se	<code>x==y</code>	1 pro x rovno y, jinak 0
<code>!=</code>	Nerovná se	<code>x!=y</code>	1 pro x nerovno y, jinak 0

## Logické operátory

- Operandy mohou být číselné typy nebo ukazatele
- Výsledek 1 má význam true, 0 má význam false
- Ve výrazech `&&` a `||` se vyhodnotí nejdříve levý operand
- pokud je výsledek dán levým operandem, pravý se nevyhodnocuje

`&&` Logické AND `x&&y` 1 pokud x ani y není rovno 0, jinak 0

`||` Logické OR `x||y` 1 pokud alespon jeden z x, y není rovno 0,  
jinak 0

`!` Logické NOT `!x` 1 pro x rovno 0, jinak 0

- Operace `&&` a `||` se vyhodnocují zkráceným způsobem, tj. druhý operand se nevyhodnocuje, pokud lze výsledek určit již z hodnoty prvního operandu.

## Bitové operátory

- Bitové operátory vyhodnocují operandy bit po bitu

&	Bitové AND	$x \& y$	1 když x i y je rovno 1
	Bitové OR	$x   y$	1 když x nebo y je rovno 1
^	Bitové XOR	$x \wedge y$	1 pokud pouze x nebo pouze y je 1
~	Bitové NOT	$\sim x$	1 pokud x je rovno 0
<<	Posun vlevo	$x \ll y$	Posun x o y bitů vlevo
>>	Posun vpravo	$x \gg y$	Posun x o y bitů vpravo

### Příklad

21 & 56	=	00010101 & 00111000	=	00010000
21   56	=	00010101   00111000	=	00101101
21 ^ 56	=	00010101 ^ 00111000	=	00111101
~21	=	~00010101	=	11101010
21 << 2	=	00010101 << 2	=	01010100
21 >> 1	=	00010101 >> 1	=	00001010

## Operace bitového posunu

- Operátory bitového posunu posouvají celý bitový obraz proměnné nebo konstanty o zvolený počet bitů vlevo nebo vpravo
  - Při posunu vlevo jsou uvolněné bity zleva doplňovány 0
  - Při posunu vpravo jsou uvolněné bity zprava
    - u čísel kladných nebo čísel typu `unsigned` plněny 0
    - u záporných čísel buď plněny 0 (logický posun) nebo 1 (aritmetický posun vpravo), dle implementace překladače.
- Operátory bitového posunu mají nižší prioritu než aritmetické operátory!
  - `i << 2 + 1` znamená `i << (2 + 1)`

## Bitové operátory – příklad 1/2

- Nastavení N-tého bitu celého čísla

```
unsigned char cislo;  
cislo |= (1<<N);
```

- Nulování N-tého bitu celého čísla

```
unsigned char cislo;  
cislo &= ~(1<<N);
```

- Inverze N-tého bitu celého čísla

```
unsigned char cislo;  
cislo ^= (1<<N);
```

- Získání hodnoty N-tého bitu celého čísla

```
unsigned char cislo;  
char bit = (cislo & (1<<N)) >> N;
```

## Bitové operátory – příklad 2/2

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  int main()
5  {
6      uint8_t a = 10;
7
8      for (int i = 7; i >= 0; --i)
9      {
10         printf ("%i", (a >> i) & 1);
11     }
12
13     printf("\n");
14
15     return 0;
16 }
```

## Operátory přístupu do paměti

- V C lze přímo přistupovat k adrese paměti proměnné, kde je hodnota proměnné uložena
- Přístup do paměti je prostřednictvím ukazatele (pointeru)
  - nesmírně silná vlastnost programovacího jazyka
  - vyžaduje pochopení principu práce s pamětí
  - podrobněji v 5. přednášce

<code>&amp;</code>	adresa proměnné	<code>&amp;x</code>	ukazatel na <code>x</code>
<code>*</code>	neprímá adresa	<code>*p</code>	proměnná adresovaná <code>p</code>
<code>[]</code>	prvek pole	<code>x[i]</code>	prvek pole <code>x</code> s indexem <code>i</code>
<code>.</code>	prvek struct/union	<code>s.x</code>	prvek <code>x</code> struktury <code>s</code>
<code>-&gt;</code>	prvek struct/union	<code>p-&gt;x</code>	prvek struktury adresovaný <code>p</code>

## Další operátory

<code>()</code>	volání funkce	<code>f(x)</code>	volání funkce <code>f</code> s argumentem <code>x</code>
<code>(type)</code>	přetypování	<code>(int)x</code>	změna typu <code>x</code> na <code>int</code>
<code>sizeof</code>	velikost prvku	<code>sizeof(x)</code>	velikost <code>x</code> v bajtech
<code>?:</code>	podmíněný příkaz	<code>x?y:z</code>	proved <code>y</code> pokud <code>x != 0</code> jinak <code>z</code>
<code>,</code>	postupné vyhodnocení	<code>x,y</code>	vyhodnotí <code>x</code> pak <code>y</code> , výsledek operátoru je výsledek posledního výrazu

- Operandem operátoru `sizeof()` může být jméno typu nebo výraz

```
int a = 10;
printf("%lu %lu\n", sizeof(a), sizeof(a + 1.0));
```

- Příklad použití operátoru čárka

```
for (c = 1, i = 0; i < 3; ++i, c += 2) {
    printf("i: %d c: %d\n", i, c);
}
```



## Operátor pretypování

- Zmena typu za behu programu se nazývá pretypování
  - Explicitní pretypování (cast) zapisuje programátor

```
int i;
float f = (float)i;
```

- Implicitní pretypování provádí prekladac automaticky pri prekladu
- Možné konverze při přiřazení

<b>typ hodnoty</b>	<b>typ proměnné</b>	<b>poznámka ke konverzi</b>
racionální	kratší racionální	zaokrouhlení mantisy
racionální	delší racionální	doplnění mantisy nulami
racionální	celočíselný	odseknutí necelé části
celočíselný	racionální	možná ztráta přesnosti
celočíselný	kratší celočíselný	odseknutí vyšších bitů
celočíselný unsgn.	delší celočíselný	doplnění nulových bitů
celočíselný sgn.	delší celočíselný	rozšíření znaménka

## Priority operátorů

priorita	operátory	asociativita
1	. -> () []	zleva
2	+ - ++ -- ! ~ (typ) & * sizeof	zprava
3	* / %	zleva
4	+ -	zleva
5	<< >>	zleva
6	< > <= >=	zleva
7	== !=	zleva
8	&	zleva
9	~	zleva
10		zleva
11	&&	zleva
12		zleva
13	?:	zprava
14	= += -= *= /= %= <<= >>= &=  = ≐	zprava
15	,	zleva

## II. Výrazy

Výrazy a operátory

Přiřazení

## Přířazení

- Nastavení hodnoty proměnné – inicializace místa v paměti
- Tvar přiřazovacího operátoru

**proměnná** = **výraz**                      Výraz je literál, proměnná, volání funkce, ...

- Proměnné lze přiřadit hodnotu výrazu pouze identického typu
- Příklad implicitní konverze při přiřazení

```
int i = 320.4; // implicitni konverze 320.4 -> 320
char c = i; // implicitni oriznuti 320 -> 64
```

- Zkrácený zápis přiřazení

**proměnná** **operátor** = **výraz**

```
int i = 10;
double j = 12.6;
i = i + 1;
j = j / 0.2;
```

```
int i = 10;
double j = 12.6;
i += 1;
j /= 0.2;
```

# Část III

## Zadání 3. domácího úkolu

## Zadání 3. domácího úkolu (HW03)

### Téma: Kreslení (ASCII art)

- **Motivace:** Zábavným a tvůrčím způsobem získat praktickou zkušenost s cykly a jejich parametrizací na základě uživatelského vstupu.
- **Cíl:** Použití cyklů a vnořených cyklů.
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/b0b99prpa/hw/hw03>
  - Načtení parametrizace pro vykreslení obrázku domečku s využitím vybraných ASCII znaků
  - Ošetření vstupních hodnot
  - **Volitelné zadání** rozšiřuje obrázek domečku o plot
- **Termín odevzdání:** 3.11.2018, 23:59:59



# Shrnutí přednášky



## Diskutovaná témata

- Řídící struktury
  - Větvení
  - Cyklus
  - Přepínač
  - Příkazy `break` a `continue`
  - Konečnost cyklů
- Operátory
  - Přehled operátorů a jejich priorit
  - Přiřazení a zkrácený způsob zápisu
- Příště: pole, ukazatel, textový řetězec





## Diskutovaná témata

- Řídící struktury
  - Větvení
  - Cyklus
  - Přepínač
  - Příkazy `break` a `continue`
  - Konečnost cyklů
- Operátory
  - Přehled operátorů a jejich priorit
  - Přiřazení a zkrácený způsob zápisu
- Příště: pole, ukazatel, textový řetězec