

Řídicí struktury, výrazy a funkce

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 03

B0B36PRP – Procedurální programování

Část I

Řídicí struktury

Přehled témat

■ Část 1 – Řídicí struktury

Příkaz a složený příkaz

Příkazy řízení běhu programu

Konečnost cyklu

S. G. Kochan: kapitoly 5 a 6

P. Herout: kapitola 5

■ Část 2 – Výrazy

Výrazy a operátory

Přiřazení

S. G. Kochan: kapitola 4, 12

P. Herout: kapitola 3, 15

■ Část 3 – Zadání 2. domácího úkolu (HW02)

Příkaz a složený příkaz (blok)

■ Příkaz je výraz zakončený středníkem

Příkaz tvořený pouze středníkem je prázdný příkaz

■ Blok je tvořen seznamem deklarací a seznamem příkazů

■ Uvnitř bloku musí deklarace předcházet příkazům

Záleží na standardu jazyka, platí pro ANSI C (C89, C90)

■ Začátek a konec bloku je vymezen složenými závorkami { a }

■ Bloky mohou být vnořené do jiného bloku

```
void function(void)
{ /* function block start */
  /* inner block */
  for(i = 0; i < 10; ++i)
  {
    //inner for-loop block
  }
}

void function(void) { /* function
block start */
  /* inner block */
  for(int i = 0; i < 10; ++i) {
    //inner for-loop block
  }
}
```

Různé kódovací konvence

Kódovací konvence a štabní kultura

- Důležitá je štabní kultura, které podporuje přehlednost a čitelnost
https://www.gnu.org/prep/standards/html_node/Writing-C.html
- Formátování patří k úplným základům
Nastavte si automatické formátování textovém editoru
- Volba výstižného jména identifikátorů podporuje čitelnost
Co může být jasné nyní, za pár dní či měsíců může být jinak
- **Cvičte se ve štabní kultuře i za cenu zdánlivě pomalejšího startu. Přehlednost je důležitá, zvláště pokud hledáte chybu**
Nezřídkou je užitečné nebát se začít úplně znovu a lépe.
- Doporučená konvence v rámci PRP


```
1 void function(void)
2 { /* function block start */
3   for(int i = 0; i < 10; ++i) {
4     //inner for-loop block
5     if (i == 5) {
6       break;
7     }
8   }
9 }
```

 - Pište zdrojové kódy pokud možno anglicky (identifikátory)
 - Pro proměnné volte podstatná jména
 - Pro funkce volte slovesa

Osobní preference přednášejícího: odsazení 3 znaky, mezery místo tabulátor.

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`

Kódovací konvence

- Existuje mnoho různých kódovacích konvencí
- Inspirujte se existujícími doporučeními
- Inspirujte se čtením cizích kódu (reprezentativních)

<http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html>
<https://www.doc.ic.ac.uk/lab/cplus/cstyle.html>
http://en.wikipedia.org/wiki/Indent_style
<https://google.github.io/styleguide/cppguide.html>
<https://www.kernel.org/doc/Documentation/CodingStyle>
<https://google.github.io/styleguide/cppguide.html>

Podmíněné větvení – if

- `if (vyraz) prikaz1; else prikaz2`
- Je-li hodnota výrazu `vyraz != 0` provede se příkaz `prikaz1` jinak `prikaz2`
Příkaz může být blok příkazů
- Část `else` je nepovinná
- Podmíněné příkazy mohou být vnořené a můžeme je řetězit

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

Příklad zápisu

```
1 if (x < y) {
2     int tmp = x;
3     x = y;
4     y = tmp;
5 }
```

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

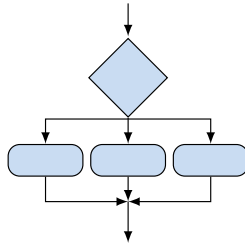
```
1 if (x < y) {
2     min = x;
3     max = y;
4 } else {
5     min = y;
6     max = x;
7 }
```

Jaký je smysl těchto programů?

Příkaz větvení `switch`

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

```
switch (výraz) {
    case konstanta1: příkazy1; break;
    case konstanta2: příkazy2; break;
    ...
    case konstantan: příkazyn; break;
    default: příkazydef; break;
}
```



kde *konstanty* jsou téhož typu jako *výraz* a *příkazy_i* jsou posloupnosti příkazů

Sémantika: vypočte se hodnota výrazu a provedou se ty příkazy, které jsou označeny konstantou s identickou hodnotou. Nemá-li vybrána žádná větev, provedou se příkazy_{def} (jso-li uvedeny).

Programový přepínač `switch` – Příklad

```
switch (v) {
    case 'A':
        printf("Upper A\n");
        break;
    case 'a':
        printf("Lower a\n");
        break;
    default:
        printf(
            "It is not A nor a\n");
        break;
}
```

```
if (v == 'A') {
    printf("Upper A\n");
} else if (v == 'a') {
    printf("Lower a\n");
} else {
    printf(
        "It is not A nor a\n");
}
```

lec03/switch.c

Programový přepínač – `switch`

- Přepínač `switch(vyraz)` větví program do n směrů
- Hodnota *vyraz* je porovnávána s n konstantními výrazy typu `int` příkazy `case konst_x: ...`
- Hodnota *vyraz* musí být celočíselná a hodnoty *konst_x* musejí 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.

Větvení `switch` – pokračování ve vykonávání dalších větví

- Příkaz `break` dynamicky ukončuje větev, pokud jej neuvedeme, pokračuje se v provádění další větve

Příklad volání více větví

```
1 int part = ?
2 switch(part) {
3     case 1:
4         printf("Branch 1\n");
5         break;
6     case 2:
7         printf("Branch 2\n");
8     case 3:
9         printf("Branch 3\n");
10        break;
11    case 4:
12        printf("Branch 4\n");
13        break;
14    default:
15        printf("Default branch\n");
16        break;
17 }
```

■ part ← 1
Branch 1

■ part ← 2
Branch 2
Branch 3

■ part ← 3
Branch 3

■ part ← 4
Branch 4

■ part ← 5
Default branch

lec03/demo-switch_break.c

Příklad větvení switch vs if-then-else

- Napište konverzní program, který podle čísla dnu v týdnu vytiskne na obrazovku jméno dne. Ošetřete případ, kdy bude zadané číslo mimo platný rozsah (1 až 7).

Příklad implementace

```
int day_of_week = 3;
if (day_of_week == 1) {
    printf("Monday");
} else if (day_of_week == 2)
{
    printf("Tuesday");
} else ... {
} else if (day_of_week == 7)
{
    printf("Sunday");
} else {
    fprintf(stderr, "Invalid
week");
}

int day_of_week = 3;
switch (day_of_week) {
case 1:
    printf("Monday");
    break;
case 2:
    printf("Tuesday");
    break;
...
case 7:
    printf("Sunday");
    break;
default:
    fprintf(stderr, "Invalid week");
    break;
}
lec03/demo-switch_day_of_week.c
```

Oba způsoby jsou sice funkční, nicméně elegantněji lze vyřešit úlohu použitím datové struktury pole nebo ještě lépe asociativním polem / (hash mapou).

Cyklus while a do-while

- Základní příkaz cyklu **while** má tvar **while (podmínka) příkaz**
- Základní příkaz cyklu **do-while** má tvar **do příkaz while (podmínka)**

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 pro hodnoty

- $x \leftarrow 10$ a $y \leftarrow 3$

while: 1, do-while: 1

- $x \leftarrow 2$ a $y \leftarrow 3$

while: 2, do-while: -1

lec03/demo-while.c

Cykly

- Cyklus **for** a **while** testuje podmínku opakování před vstupem do těla cyklu

- for** – inicializace, podmínka a změna řídicí proměnné součást syntaxe

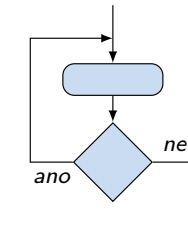
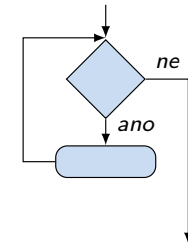
```
for (int i = 0; i < 5; ++i) {
    ...
}
```

- while** – řídicí proměnná v režii programátora

```
int i = 0;
while (i < 5) {
    ...
    i += 1;
}
```

- Cyklus **do** testuje podmínku opakování cyklu po prvním provedení cyklu

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



Ekvivalentní provedení 5ti cyklů.

Cyklus for

- Základní příkaz cyklu **for** má tvar **for (inicializace; podmínka; změna) příkaz**

- Odpovídá cyklu while ve tvaru:

```
inicializace;
while (podmínka) {
    příkaz;
    změna;
}
```

- Změnu řídicí proměnné lze zkráceně zapsat operátorem inkrementace nebo dekrementace **++** a **--**

- Alternativně lze též použít zkrácený zápis přiřazení, např. **+=**

Příklad

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

Cyklus for(; ;)

- Příkaz **for** cyklu má tvar `for ([vyraz1]; [vyraz2]; [vyraz3]) prikaz;`
- Cyklus **for** používá řídicí proměnnou a probíhá následovně:
 1. **vyraz1** – Inicializace (zpravidla řídicí proměnné)
 2. **vyraz2** – Test řídicího výrazu
 3. Pokud **vyraz2 !=0** provede se **prikaz**, jinak cyklus končí
 4. **vyraz3** – Aktualizace proměnných na konci běhu cyklu
 5. Opakování cyklu testem řídicího výrazu
- Výrazy **vyraz1** a **vyraz3** mohou být libovolného typu
- Libovolný z výrazů lze vynechat
- **break** – cyklus lze nuceně opustit příkazem **break**
- **continue** – část těla cyklu lze vynechat příkazem **continue**

Příkaz přeruší vykonávání těla (blokového příkazu) pokračuje vyhodnocením vyraz3.
- Při vynechání řídicího výrazu **vyraz2** se cyklus bude provádět nepodmíněně

```
for (;;) {...}
```

Nekonečný cyklus

Předčasné ukončení průchodu cyklu – příkaz **continue**

- Někdy může být užitečné ukončit cyklus v nějakém místě uvnitř těla cyklu
 - Například ve vnořených **if** příkazech
- Příkaz **continue** předepisuje **ukončení průchodu** těla cyklu

Platnost pouze v těle cyklu!

```
for (int i = 0; i < 10; ++i) {
    printf("i: %i ", i);
    if (i % 3 != 0) {
        continue;
    }
    printf("\n");
}
```

`clang demo-continue.c`
`./a.out`
`i:0`
`i:1 i:2 i:3`
`i:4 i:5 i:6`
`i:7 i:8 i:9`

`lec03/demo-continue.txt`

Příkaz **continue**

- Příkaz návratu na vyhodnocení řídicího výrazu – **continue**
- Příkaz **continue** lze použít pouze v těle cyklů
 - `for ()`
 - `while ()`
 - `do...while ()`
- Příkaz **continue** způsobí přerušení 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);
}
```

`lec03/continue.c`

Příkaz **break**

- Příkaz nuceného ukončení cyklu **break**;
- Příkaz **break** lze použít pouze v těle cyklů
 - `for()`
 - `while()`
 - `do...while()`
- a v těle programového přepínače **switch()**
- Příkaz **break** způsobí opuštění těla cyklu nebo těla **switch()**,
- program pokračuje následujícím příkazem, např.

```
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);
}
```

`lec03/break.c`

Předčasné ukončení vykonávání cyklu – příkaz `break`

- Příkaz `break` předepisuje ukončení cyklu

Program pokračuje následujícím příkazem po cyklu

```

for (int i = 0; i < 10; ++i) {
    printf("i: %i ", i);
    if (i % 3 != 0) {
        continue;
    }
    printf("\n");
    if (i > 5) {
        break;
    }
}

clang demo-break.c
./a.out
i:0
i:1 i:2 i:3
i:4 i:5 i:6

lec03/demo-break.java

```

Vnořené cykly

- `break` ukončuje vnitřní cyklus

```

for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        printf("i-j: %i-%i\n", i, j);
        if (j == 1) {
            break;
        }
    }
}

i-j: 0-0
i-j: 0-1
i-j: 1-0
i-j: 1-1
i-j: 2-0
i-j: 2-1

```

- Vnější cyklus můžeme ukončit příkazem `break` se jménem

```

for (int i = 0; i < 5; ++i) {
    for (int j = 0; j < 3; ++j) {
        printf("i-j: %i-%i\n", i, j);
        if (j == 2) {
            goto outer;
        }
    }
}
outer:

i-j: 0-0
i-j: 0-1
i-j: 0-2

lec03/demo-goto.c

```

Příkaz `goto`

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

```

1 int test = 3;
2 for(int i = 0; i < 3; ++i) {
3     for (int j = 0; j < 5; ++j) {
4         if (j == test) {
5             goto loop_out;
6         }
7         fprintf(stdout, "Loop i: %d j: %d\n", i, j);
8     }
9 }
10 return 0;
11 loop_out:
12 fprintf(stdout, "After loop\n");
13 return -1;

```

lec03/goto.c

Konečnost cyklů 1/3

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

```

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

```

- Cyklus se provede jednou,
- nebo neskončí.
- Záleží na hodnotě `i` před voláním cyklu

Konečnost cyklů 2/3

- 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.

Příklad – test, je-li zadané číslo prvočíslem

```
#include <stdbool.h>
#include <math.h>

_Bool isPrimeNumber(int n) {
    _Bool ret = true;
    for (int i = 2; i <= (int)sqrt((double)n); ++i) {
        if (n % i == 0) {
            ret = false;
            break;
        }
    }
    return ret;
}
```

lec03/demo-prime.c

- **break** – po nalezení 1. dělitele nemusíme dále testovat
- Hodnota výrazu `(int)sqrt((double)n)` se v cyklu nemění a je zbytečné výraz opakovaně vyhodnocovat

```
_Bool ret = true;
const int maxBound = (int)sqrt((double)n);
for (int i = 2; i <= maxBound ; ++i) {
    ...
}
```

Příklad kompilace spuštění demo-prime.c: clang demo-prime.c -lm; ./a.out 13

Konečnost cyklů 3/3

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

lec03/demo-loop_byte.c

- Vstupní podmínka konečnosti uvedeného cyklu

- $i \leq n$ pro celá čísla

Jak by vypadala podmínka pro proměnné typu double?

lec03/demo-loop.c

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

Kódovací konvence

- Příkazy **break** a **continue** v podstatě odpovídají příkazům skoku.
- Obecně můžeme říci, že příkazy **break** a **continue** nepřidávají příliš na přehlednosti

Nemyslíme tím break v příkazu switch
- Přerušení cyklu **break** nebo **continue** můžeme využít v těle dlouhých funkcí a vnořených cyklech

Ale funkce bychom měli psát krátké a přehledné
- Je-li funkce (tělo cyklu) krátké je význam **break/continue** čitelný
- Podobně použití na začátku bloku cyklu např. jako součást testování splnění předpokladů, je zpravidla přehledné
- Použití uprostřed bloku je však už méně přehledné a může snížit čitelnost a porozumění kódu

<https://www.scribd.com/doc/38873257/>

[Knuth-1974-Structured-Programming-With-Go-to-Statements](#)

Část II

Výrazy

Výrazy a operátory

- Výraz se skládá z operátorů a operandů
 - Nejjednodušší výraz tvoří konstanta, proměnná nebo volání funkce
 - 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 pro zápis výrazů

Nebo posloupnost znaků
- Postup výpočtu výrazu s více operátory je dán prioritou operátorů

Postup výpočtu lze předepsat použitím kulatých závorek (a)
- Operátory: aritmetické, relační, logické, bitové
 - Arita operátoru (počet operandů) – unární, binární, ternární
 - 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 (jaká funkce se zavolá jako první).

Chování i = ++i + i++; není definované, závisí na překladači.
 - Pořadí vyhodnocení je definováno pro operandy v logické součinu **AND** a součtu **OR**

http://en.cppreference.com/w/c/language/eval_order

Výrazy

- **Výraz** předepisuje výpočet hodnoty určitého vstupu
- Struktura výrazu obsahuje *operandy, operátory a závorky*
- Výraz může obsahovat
 - literály
 - unární a binární operátory
 - proměnné
 - volání funkcí
 - konstanty
 - 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*

Základní rozdělení operátorů

- Můžeme rozlišit čtyři základní typy binárních operátorů
 - Aritmetické operátory – sčítání, odčítání, násobení, dělení
 - Relační operátory – porovnání hodnot (menší, větší, ...)
 - Logické operátory – logický součet a součin
 - **Operátor přiřazení** - na levé straně operátoru **=** je proměnná
- Unární operátory
 - indikující kladnou/zápornou hodnotu: **+** a **-**

operátor – modifikuje znaménko výrazu za ním
 - modifikující proměnou: **++** a **--**
 - logický operátor doplněk: **!**
 - bitová negace : **~** (negace bit po bitu)
- Ternární operátor – podmíněné přiřazení hodnoty

*Jediný ternární operátor je podmíněný příkaz **?:***

http://www.tutorialspoint.com/cprogramming/c_operators.htm

Aritmetické operátory

- Operandy aritmetických operátorů mohou být libovolného aritmetického typu

Výjimkou je operátor zbytek po dělení % definovaný pro int

*	Násobení	$x * y$	Součin x a y
/	Dělení	x / y	Podíl x a y
%	Dělení modulo	$x \% y$	Zbytek po dělení x a y
+	Sčítání	$x + y$	Součet x a y
-	Odčítání	$x - y$	Rozdíl a y
+	Kladné znam.	$+x$	Hodnota x
-	Záporné znam.	$-x$	Hodnota -x
++	Inkrementace	$++x/x++$	Inkrementace před/po vyhodnocení výrazu x
--	Dekrementace	$--x/x--$	Dekrementace před/po vyhodnocení výrazu x

Relační operátory

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

<	Menší než	$x < y$	1 pro x je menší než y, jinak 0
<=	Menší nebo rovno	$x <= y$	1 pro x menší nebo rovno y, jinak 0
>	Větší než	$x > y$	1 pro x je větší než y, jinak 0
>=	Větší nebo rovno	$x >= y$	1 pro x větší nebo rovno y, jinak 0
==	Rovná se	$x == y$	1 pro x rovno y, jinak 0
!=	Nerovná se	$x != y$	1 pro x nerovno y, jinak 0

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. proměnná)

- Lze zapsat prefixově např. ++x nebo --x
- nebo postfixově např. x++ nebo x--
- v obou případech se však liší výsledná hodnota výrazu!

int i; int a;	hodnota i	hodnota a
i = 1; a = 9;	1	9
a = i++;	2	1
a = ++i;	3	3
a = ++(i++);	nelze, hodnota i++ není l-hodnota	

V případě unárního operátoru i++ je nutné v paměti uchovat původní hodnotu i a následně inkrementovat hodnotu proměnné i. Případě použití ++i pouze inkrementujeme hodnotu i. Proto může být použití ++i efektivnější.

Logické operátory

- Operandy mohou být aritmetické 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

Zkrácené vyhodnocování – složité výrazy

&&	Logické AND	$x \&\& y$	1 pokud x ani y není rovno 0, jinak 0
	Logické OR	$x \ \ y$	1 pokud alespoň 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, Ize-li výsledek určit již z hodnoty prvního operandu

Bitové operátory

- Bitové operátory vyhodnocují operandy bit po bitu
- Operátory bitového posunu posouvají celý bitový obraz o zvolený počet bitů vlevo nebo vpravo
 - Při posunu vlevo jsou uvolněné bity zleva plněny 0
 - Při posunu vpravo jsou uvolněné bity zprava
 - u čísel kladných nebo typu unsigned plněny 0
 - u záporných čísel buď plněny 0 (logical shift) nebo 1 (arithmetic shift right), dle implementace překladače.

&	Bitové AND	$x \& y$	1 když x i y je rovno 1 (bit po bitu)
	Bitové OR	$x y$	1 když x nebo y je rovno 1 (bit po bitu)
^	Bitové XOR	$x \wedge y$	1 pokud oba x a y jsou 0 nebo 1 (bit po bitu)
~	Bitové NOT	$\sim x$	1 pokud x je rovno 0 (bit po bitu)
<<	Posun vlevo	$x \ll y$	Posun x o y bitů vlevo
>>	Posun vpravo	$x \gg y$	Posun x o y bitů vpravo

Operátory přístupu do paměti

Zde pro úplnost, více v následujících přednáškách

- V C lze přímo přistupovat k adrese paměti proměnné, kde je hodnota proměnné uložena
- Přístup do paměti je prostřednictvím ukazatele (*pointeru*)

Dává velké možnosti, ale také vyžaduje zodpovědnost.

Operátor	Význam	Příklad	Výsledek
&	Adresa proměnné	$\&x$	Ukazatel (pointer) na x
*	Nepřímá adresa	$*p$	Proměnná (nebo funkce) adresovaná pointerem p
[]	Prvek pole	$x[i]$	$*(x+i)$ – prvek pole x s indexem i
.	Prvek struct/union	$s.x$	Prvek x struktury s
->	Prvek struct/union	$p->x$	Prvek struktury adresovaný ukazatelem p

Operandem operátoru & nesmí být bitové pole a proměnná typu register.

*Operátor nepřímé adresy * umožňuje přístup na proměnné přes ukazatel.*

Příklad – bitových operací

```
uint8_t a = 4;
uint8_t b = 5;
```

```
a  dec: 4 bin: 0100
b  dec: 5 bin: 0101
a&b dec: 4 bin: 0100
a|b dec: 5 bin: 0101
a^b dec: 1 bin: 0001
```

```
a>>1 dec: 2 bin: 0010
a<<1 dec: 8 bin: 1000
```

`lec02/bits.c`

Ostatní operátory

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

()	Volání funkce	$f(x)$	Volání funkce f s argumentem x
(type)	Přetypování (cast)	$(int)x$	Změna typu x na <code>int</code>
sizeof	Velikost prvku	$sizeof(x)$	Velikost x v bajtech
? :	Podmíněný příkaz	$x ? y : z$	Proved' y pokud $x \neq 0$ jinak z
,	Postupné vyhodnocení	x, y	Vyhodnotí x pak y

- Příklad použití operátoru čárka

Operátor přetypování

- Změna typu za běhu programu se nazývá přetypování
- Explicitní přetypování (cast) zapisuje programátor uvedením typu

v kulatých závorkách, např.

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

- Implicitní přetypování provádí překladač automaticky při překladu
- Pokud nový typ může reprezentovat původní hodnotu, přetypování ji vždy zachová
- Operandů typů `char`, `unsigned char`, `short`, `unsigned short`, případně bitová pole, mohou být použity tam kde je povolen typ `int` nebo `unsigned int`. C očekává hodnoty alespoň typu `int`
 - Operandů jsou automaticky přetypovány na `int` nebo `unsigned int`.

Přehled operátorů a jejich priorit 1/3

Priorita	Operátor	Asociativita	Operace
1	<code>++</code>	P/L	pre/post inkrementace
	<code>--</code>		pre/post dekrementace
	<code>()</code>	L→P	volání metody
	<code>[]</code>		indexace do pole
	<code>.</code>		přístup na položky struktury/unionu
	<code>-></code>		přístup na položky přes ukazatel
2	<code>! ~</code>	P→L	logická a bitová negace
	<code>- +</code>		unární plus (minus)
	<code>()</code>		přetypování
	<code>*</code>		nepřímé adresování (dereference)
	<code>&</code>		adresa (reference)
	<code>sizeof</code>		velikost

Asociativita priorit operátorů

- Binární operace `op` na množině **S** je **asociativní**, jestliže platí $(x \text{ op } y) \text{ op } z = x \text{ op } (y \text{ op } z)$, pro každé $x, y, z \in \mathbf{S}$
- U **neasociativních operací** je nutné řešit v jakém pořadí jsou operace implicitně provedeny
 - asociativní zleva – operace jsou seskupeny zleva
Např. výraz $10 - 5 - 3$ je vyhodnocen jako $(10 - 5) - 3$
 - asociativní zprava – operace jsou seskupeny zprava
Např. $3 + 5^2$ je 28 nebo $3 \cdot 5^2$ je 75 vs. $(3 \cdot 5)^2$ je 225
- Přirazení je asociativní zprava
Např. $y=y+8$
Vyhodnotí se nejdříve celá pravá strana operátoru `=`, která se následně přiřadí do proměnné na straně levé.
- Priorita binárních operací vyjadřuje v algebře pořadí, v jakém jsou binární operace prováděny
- Pořadí operací lze definovat důsledným **závorkováním**

Přehled operátorů a jejich priorit 2/3

Priorita	Operátor	Asociativita	Operace
3	<code>*,/,%</code>	L→R	násobení, dělení, zbytek
4	<code>+ -</code>		sčítání, odečítání
5	<code>>> <<</code>		bitový posun vlevo/vpravo
6	<code><, >, <=, >=</code>		porovnání
7	<code>==, !=</code>		rovno, nerovno
8	<code>&</code>		bitový AND
9	<code>^</code>		bitový XOR
10	<code>^</code>		bitový OR
11	<code>&&</code>		logický AND
12	<code> </code>		logický OR

Přehled operátorů a jejich priorit 3/3

Priorita	Operátor	Asociativita	Operace
13	? :	P→L	ternární operátor
14	=		přirazení
	+ = - =		přirazení součtu / rozdílu
	* = / = % =	P→L	přirazení součinu/podílu/zbytku
	<<= >>=		přirazení bitového posunu vlevo/vpravo
	& = ^ = =		přirazení bitového AND/XOR/OR
15	,	L→P	operátor čárka

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

Zkrácený zápis přirazení

■ Zápis

$\langle \text{proměnná} \rangle = \langle \text{proměnná} \rangle \langle \text{operátor} \rangle \langle \text{výraz} \rangle$

■ Lze zapsat zkráceně

$\langle \text{proměnná} \rangle \langle \text{operátor} \rangle = \langle \text{výraz} \rangle$

Příklad

```
int i = 10;           int i = 10;
double j = 12.6;     double j = 12.6;

i = i + 1;           i += 1;
j = j / 0.2;         j /= 0.2;
```

■ Přirazení je výraz

```
int x, y;

x = 6;
y = x = x + 6;
```

„syntactic sugar“

Přirazení

■ Nastavení hodnoty proměnné

Uložení definované hodnoty na místo v paměti

■ Tvar přiřazovacího operátoru

$\langle \text{proměnná} \rangle = \langle \text{výraz} \rangle$

Výraz je literál, proměnná, volání funkce, ...

■ C je staticky typovaný jazyk

■ Proměnné lze přiřadit hodnotu výrazu pouze identického typu

Jinak je nutné provést typovou konverzi

■ Příklad nedovoleného příkazů přirazení

```
int i = 1.4;
```

■ C je typově bezpečné v omezeném kontextu kompilace, např. na `printf("%d\n", 10.1);` kompilátor upozorní na chybu

■ Obecně není C typově bezpečné

Za běhu programu může dojít například k zápisu mimo vyhrazenou paměť a tím může dojít k nedefinovanému chování.

Výraz a příkaz

■ Příkaz provádí akci a je zakončen středníkem

```
robot_heading = -10.23;
robot_heading = fabs(robot_heading);
printf("Robot heading: %f\n" + robot_heading);
```

■ Výraz má určený typ a hodnotu

```
23      typ int, hodnota 23
14+16/2 typ int, hodnota 22
y=8     typ int, hodnota 8
```

■ Přirazení je výraz a jeho hodnotou je hodnota přiřazená levé straně

■ Z výrazu se stává příkaz, pokud je ukončen středníkem

Část III

Zadání 3. domácího úkolu (HW03)

Zadání 3. domácího úkolu HW03

-
- Termín odevzdání: 29.10.2016, 23:59:59 AoE

AoE – Anywhere on Earth

Shrnutí přednášky