

Řídicí struktury, výrazy a funkce

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 04

B0B36PRP – Procedurální programování

Přehled témat

■ Část 1 – Řídicí struktury

Příkaz a složený příkaz

Příkazy řízení běhu programu

Konečnost cyklu

S. G. Kochan: kapitoly 5 a 6

■ Část 2 – Výrazy

Výrazy a operátory

Přirazení

Nedefinované chování

S. G. Kochan: kapitola 4, 12

■ Část 3 – Zadání 3. domácího úkolu (HW03)

Část I

Řídicí struktury

Příkaz a složený příkaz (blok)

■ Příkaz je výraz zakončený středníkem.

Příkaz tvořený pouze středníkem je prázdný příkaz.

■ Blok je tvořen seznamem definic proměnných a příkazů.

■ Uvnitř bloku zpravidla definice proměnných předchází příkazům.

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

■ Začátek a konec bloku je vymezen složenými závkami { 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

Srozumitelnost, čitelnost kódu - kódovací konvence a styl (čistota kódu)

- Konvence a styl je důležitý, protože podporuje přehlednost a čitelnost.
https://www.gnu.org/prep/standards/html_node/Writing-C.html
- Formátování patří k úplným základům. *Nastavte si automatické formátování v textovém editoru.*
- Volba výstižného jména identifikátorů podporuje čitelnost.

Co může být jasné nyní, za pár dní či měsíců může být jinak.

- Cvičte se v kódovací konvenci a zvoleném stylu i za cenu zdánlivě pomalejšího zápisu kódu. Přehlednost je důležitá, zvláště pokud hledáte chybu.

- Doporučená konvence v rámci PRP

Nezřídka je užitečné nebát se začít úplně znovu a lépe.

- Pište zdrojové kódy pokud možno anglicky (identifikátory).

- Pro proměnné volte podstatná jména.

- Pro funkce volte slovesa.

```
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 }
```

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

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

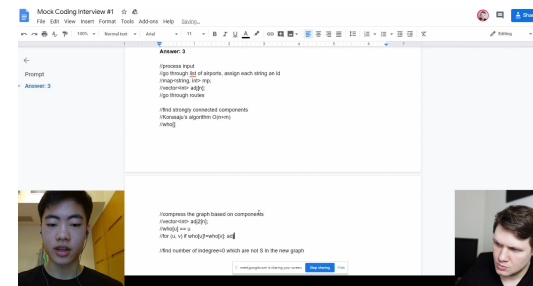
- `continue`
- `break`
- `return`
- `goto`

Srozumitelnost a čitelnost kódu - kódovací konvence

- Existují různé kódovací konvence; inspirujte se existujícími doporučeními a čtením reprezentativních kódů.



Clean Code - Uncle Bob / Lesson 1
<https://youtu.be/7EmboKQH81M>



Google Coding Interview with a High School Student
<https://youtu.be/qz9tK1F431k>

<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/process/coding-style.rst>

Podmíněné větvení – if

- `if (vyraz) prikaz1; else prikaz2`
- Je-li hodnota výrazu `vyraz != 0`, provede se příkaz `prikaz1` jinak `prikaz2`.
- Část `else` je nepovinná. *Příkaz může být blok příkazů.*
- 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 {
    ...
}
```

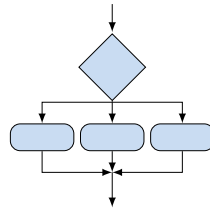
```
1 if (x < y) {
2     min = x;
3     max = y;
4 } else {
5     min = y;
6     max = x;
7 }
```

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*; jsou posloupnosti příkazů.

Sémantika: vypočte se hodnota výrazu a provedou se ty příkazy, které jsou označeny konstantou s identickou hodnotou. Není-li vybrána žádná větev, provedou se příkazy_{def} (pokud jsou 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");
}
```

lec04/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 konstanta _{i} : ...
- Hodnota `vyraz` musí být celočíselná a hodnoty konstanta _{i} 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

lec04/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 number");
}

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 number");
        break;
}
```

lec04/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.

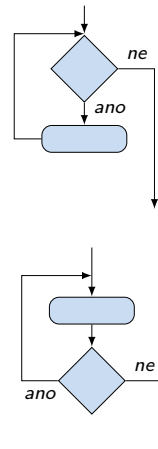
Cykly

- 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) {
    ...
}
```
 - `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 pěti cyklů.

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

while: 1, do-while: 1

while: 2, do-while: -1

lec04/demo-while.c

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
```

`lec04/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 přerušování 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);
}
```

`lec04/continue.c`

Příkaz break

- Příkaz nuceného ukončení cyklu `break`; lze použít pouze v těle cyklů.
 - `for()`
 - `while()`
 - `do...while()`
- a v těle programového přepínače `switch()`.
- `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);
}
```
- Z hlediska přehlednosti a čitelnosti je vhodné změnu řídicí proměnné realizovat na konci cyklu.

`lec04/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
```

lec04/demo-break.c

Příkaz `goto`

- Příkaz nepodmíněného lokálního skoku `goto` předá řízení na místo určené návěstím `navesti` – syntax `goto navesti;`.
- Návěstí má tvar `navesti příkaz`. *Definice proměnné není příkaz.*
- Příkaz `goto` lze použít pouze v těle funkce a skok je možný pouze rámci jediné funkce.

```
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;
```

lec04/goto.c

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 `goto`.

```
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
```

lec04/demo-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 neprovede ani 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; // leave the loop once if it sure
            break; // n is not a prime number
        }
    }
    return ret;
}
```

lec04/demo-prime.c

- **break** – po nalezení prvního dělitele nemusíme dále testovat.
- Hodnota výrazu `(int)sqrt((double)n)` se v cyklu nemění.

```
_Bool ret = true; // zbytecne vypocet opakovat
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++;
}
```

lec04/demo-loop_byte.c

- Vstupní podmínka konečnosti uvedeného cyklu

- $i \leq n$ pro celá čísla.

Jak by vypadala podmínka pro proměnné typu double?

lec04/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.
- Přerušování 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>

Výrazy a operátory Přířazení Nedefinované chování

Část II

Výrazy

Jan Faigl, 2020 B0B36PRP – Přednáška 04: Řídící struktury, výrazy a funkce 32 / 61

Výrazy a operátory Přířazení Nedefinované chování

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

$10 + x * y$	poradí vyhodnocení $10 + (x * y)$
$10 + x + y$	poradí vyhodnocení $(10 + x) + y$

* má vyšší prioritu než +
+ je asociativní zleva

Jan Faigl, 2020 B0B36PRP – Přednáška 04: Řídící struktury, výrazy a funkce 34 / 61

Výrazy a operátory Přířazení Nedefinované chování

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.
- Postup výpočtu výrazu s více operátory je dán prioritou operátorů. *Případně posloupnost znaků.*
- Operátory: aritmetické, relační, logické, bitové. *Postup výpočtu lze předepsat použitím kulatých závorek ().*
 - 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 definováno, závisí na překladači.
 - Pořadí vyhodnocení je **definováno pro operandy v logickém součinu AND** a **součtu OR**.
http://en.cppreference.com/w/c/language/eval_order

Jan Faigl, 2020 B0B36PRP – Přednáška 04: Řídící struktury, výrazy a funkce 35 / 61

Výrazy a operátory Přířazení Nedefinované chování

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 (l-hodnota reprezentující místo v paměti).
- Unární operátory:
 - indikující kladnou/zápornou hodnotu: + a -;
 - modifikující proměnou: ++ a --;
 - logický operátor doplněk: !;
 - bitová negace: ~ (negace bit po bitu).

operátor – modifikuje znaménko výrazu za ním.
- Ternární operátor – podmíněný příkaz. *Jediný ternární operátor v C je podmíněný příkaz ? :*

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

Jan Faigl, 2020 B0B36PRP – Přednáška 04: Řídící struktury, výrazy a funkce 36 / 61

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 **liši výsledná hodnota výrazu!**

int i; int a;	hodnota i	hodnota a
<code>i = 1; a = 9;</code>	1	9
<code>a = i++;</code>	2	1
<code>a = ++i;</code>	3	3
<code>a = ++(i++);</code>	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`. V 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.

<code>&&</code>	Logické AND	$x \&\& y$	1 pokud x ani y není rovno 0, jinak 0.
<code> </code>	Logické OR	$x \ \ y$	1 pokud alespoň jeden z x , y není rovno 0, jinak 0.
<code>!</code>	Logické NOT	<code>!x</code>	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.

<code>&</code>	Bitové AND	<code>x & y</code>	1 když x i y je rovno 1 (bit po bitu).
<code> </code>	Bitové OR	<code>x y</code>	1 když x nebo y je rovno 1 (bit po bitu).
<code>^</code>	Bitové XOR	<code>x ^ y</code>	1 pokud pouze x nebo pouze y je 1 (exkluzivně právě jedna z variant) (bit po bitu).
<code>~</code>	Bitové NOT	<code>~x</code>	1 pokud x je rovno 0 (bit po bitu).
<code><<</code>	Posun vlevo	<code>x << y</code>	Posun x o y bitů vlevo.
<code>>></code>	Posun vpravo	<code>x >> y</code>	Posun x o y bitů vpravo.

Operace bitového posunu

- 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 (logický posun) nebo 1 (aritmetický posun vpravo), dle implementace překladače.
- Operátory bitového posunu **mají nižší prioritu než aritmetického operátory!**
 - `i << 2 + 1` znamená `i << (2 + 1)`.

Nebuďte zaskočení nečekanou interpretací – závorkujte!

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
```

`lec04/bits.c`

See recursive version in `lec04/bits-recursive.c`

Operátory přístupu do paměti

Zde pro úplnost, více v následujících přednáškách.

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

Dává velké možnosti, ale také vyžaduje zodpovědnost.

Operátor	Význam	Příklad	Výsledek
<code>&</code>	Adresa proměnné	<code>&x</code>	Ukazatel (pointer) na <code>x</code>
<code>*</code>	Nepřímá adresa	<code>*p</code>	Proměnná (nebo funkce) adresovaná pointerem <code>p</code>
<code>[]</code>	Prvek pole	<code>x[i]</code>	<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>-></code>	Prvek struct/union	<code>p->x</code>	Prvek struktury adresovaný ukazatelem <code>p</code>

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

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

Ostatní operátory

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

<code>()</code>	Volání funkce	<code>f(x)</code>	Volání funkce <code>f</code> s argumentem <code>x</code>
<code>(type)</code>	Přetypování (cast)	<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>	Proveď <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.


```
lec04/sizeof.c
for (c = 1, i = 0; i < 3; ++i, c += 2) {
    printf("i: %d c: %d\n", i, c);
}
```

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.
- Operandy typů `char`, `unsigned char`, `short`, `unsigned short`, případně bitová pole, mohou být použity tam kde je povolen typ `int` nebo `unsigned int`.
 - Operandy jsou automaticky přetypovány na `int` nebo `unsigned int`.

Asociativita a prioritita 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 $x, y, z \in S$.
- 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
- Přířazení 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 provedení operací lze definovat důsledným **závorkováním**.

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
2	<code>-></code>		přístup na položky přes ukazatel
	<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

Přehled operátorů a jejich priorit 2/3

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

Přřazení

- Nastavení hodnoty proměnné.
- Tvar přiřazovacího operátoru.

Uložení definované hodnoty na místo v paměti.

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

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 implicitní konverze při přiřazení.

```
int i = 320.4; // implicit conversion from 'double' to 'int' changes value from
              320.4 to 320 [-Wliteral-conversion]
```

```
char c = i; // implicit truncation 320 -> 64
```

- 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.

Přehled operátorů a jejich priorit 3/3

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

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

Zkrácený zápis přiřazení

- 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;
double j = 12.6;
```

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

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

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

- Přiřazení je výraz

```
int x, y;
```

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

„syntactic sugar“

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
```

- Přřazení je výraz a jeho hodnotou je hodnota přřazená levé straně.
- Z výrazu se stává příkaz, pokud je ukončen středníkem.

Přřklad nedefinovaného chování

- Standard C nepředpisuje chování při přetečení celého čísla (**signed**)
 - V přřpadě doplňkového kódu může být např. hodnota výrazu `127 + 1` typu `char` rovna `-128`, viz `lec04/demo-loop_byte.c`.
 - Reprezentace celých čísel však může být realizována jinak dle architektury např. přřmým kódem nebo inverzním kódem.
- Zajištění předepsaného chování tak může být výpočetně komplikované, proto standard nedefinuje chování při přetečení.
- **Chování programu není definované a závisí na kompilátoru**, např. přřkladače `clang` a `gcc` bez/s optimalizacemi `-O2`.

```
for (int i = 2147483640; i >= 0; ++i) {
    printf("%i %x\n", i, i);
}
```

`lec04/int_overflow-1.c`

Bez optimalizací program vypíše 8 řádků, pro `-O2` program zkompilovaný `clang` vypíše 9 řádků, `gcc` program skončí v nekonečné smyčce.

```
for (int i = 2147483640; i >= 0; i += 4) {
    printf("%i %x\n", i, i);
}
```

`lec04/int_overflow-2.c`

Program zkompilovaný `gcc` s `-O2` po spuštění padá.

Analyzujte kód `asm` generovaný přřpínačem `-S`.

Nedefinované chování

- Dle standardu C mohou některé příkazy (výrazy) způsobit **nedefinované chování**.
 - `c = (b = a + 2) - (a - 1);`
 - `j = i * i++;`
- Program se může chovat rozdílně podle použitého kompilátoru, přřpadně nemusí jít zkompilovat, spustit, nebo dokonce padat a chovat se neobvykle či produkovat nesmyslné výsledky.
- To se může například také stát v přřpadě, že nejsou proměnné inicializovány.
- **Vyhýbejte se příkazům (výrazům), které mohou vést na nedefinované chování!**

Část III

Zadání 3. domácího úkolu (HW03)

Zadání 3. domácího úkolu HW03

Téma: Kreslení (ASCII art)

Povinné zadání: **2b**; Volitelné zadání: **2b**; Bonusové zadání: *není*

- **Motivace:** Zábavným a tvůrčím způsobem získat praktickou zkušenost s cykly a jejich parametrizací na základě uživatelského vstupu.
- **Cíl:** Osvojit si použití cyklů a vnořených cyklů.
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw03>
 - Načtení parametrizace pro vykreslení obrázku domečku s využitím vybraných ASCII znaků. https://en.wikipedia.org/wiki/ASCII_art
 - Ošetření vstupních hodnot.
 - **Volitelné zadání** rozšiřuje obrázek domečku o plot.
- **Termín odevzdání:** **23.10.2021, 23:59:59 PDT.**

PDT – Pacific Daylight Time

Shrnutí přednášky

Diskutovaná témata

- Řídící struktury - přepínač, cykly, vnořené cykly, **break** a **continue**
- Konečnost cyklů
- Kódovací konvence
- Výrazy - unární, binární a ternární
- Přehled operátorů a jejich priorit
- Přiřazení a zkrácený způsob zápisu
 - Příkazy a nedefinované chování
- **Příště: Pole, ukazatel, textový řetězec, vstup a výstup programu**