

Řídicí struktury, výrazy a funkce

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 03

B0B36PRP – Procedurální programování

Přehled témat

- Část 1 – Řídicí struktury
 - Příkaz a složený příkaz
 - Příkazy řízení běhu programu
 - Podmíněné řízení běhu programu
 - Konečnost cyklu

S. G. Kochan: kapitoly 5 a 6

- Část 2 – Výrazy
 - Výrazy a operátory
 - Přiřazení

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, definice proměnných zpravidla předchází 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.

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

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

```
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.

■ Nezdědka je užitečné nebat se začít úplně znovu a lépe.

■ Píšte zdrojové kódy pokud možno anglicky (identifikátory).

■ Pro proměnné volte podstatná jména.

■ Pro funkce volte slovesa.

Složený příkaz a zanořování 1/2

Čtyři úrovně zanoření.

```
1 int get_sum_of_even_numbers(int from, int to)
2 {
3   if (from < to) {
4     int sum = 0;
5     for (int number = from; number <= to; ++number) {
6       if (number % 2 == 0) {
7         sum += number;
8       }
9     } // end for loop
10    return sum;
11  } else {
12    return 0;
13  }
14 }
```

Míříme na čitelnější podobu.

```
1 int get_sum_of_even_numbers(int from, int to)
2 {
3   if (from > to) return 0;
4   int sum = 0;
5   for (int number = from; number <= to; ++number) {
6     sum += filter_odd(number);
7   } // end for loop
8   return sum;
9 }
```

Vyjmutí (definice nové funkce).

```
1 int filter_odd(int number);
2
3 int get_sum_of_even_numbers(int from, int to)
4 {
5   if (from < to) {
6     int sum = 0;
7     for (int number = from; number <= to; ++number) {
8       sum += filter_odd(number);
9     } // end for loop
10    return sum;
11  } else {
12    return 0;
13  }
14 }
15
16 int filter_odd(int number)
17 {
18   if (number % 2 == 0) {
19     return number;
20   }
21   return 0;
22 }
```

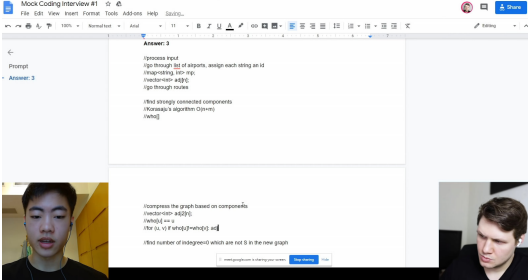
■ Použitím technik vyjmutí a inverze redukuje počet zanoření.
<https://youtu.be/CFRhGnuXG-4>

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/qz9tKlF431k>

<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>

Složený příkaz a zanořování 2/2

Inverze (záměna podmínky hodnoty vstupu).

```
1 int filter_odd(int number);
2
3 int get_sum_of_even_numbers(int from, int to)
4 {
5   if (from > to) {
6     return 0;
7   }
8   int sum = 0;
9   for (int number = from; number <= to; ++number) {
10    sum += filter_odd(number);
11  } // end for loop
12  return sum;
13 }
14
15 int filter_odd(int number)
16 {
17   if (number % 2 == 0) {
18     return number;
19   }
20   return 0;
21 }
```

Finální „zkompaktnění“.

```
1 int filter_odd(int number);
2
3 int get_sum_of_even_numbers(int from, int to)
4 {
5   if (from > to) return 0;
6   int sum = 0;
7   for (int number = from; number <= to; ++number) {
8     sum += filter_odd(number);
9   } // end for loop
10  return sum;
11 }
12
13 int filter_odd(int number)
14 {
15   return (number % 2 == 0) ? number : 0;
16 }
17
```

- Použitím technik vyjmutí a inverze redukuje počet zanoření.
<https://youtu.be/CFRhGnuXG-4>

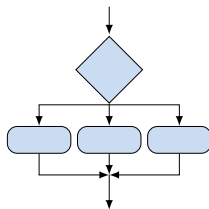
Příkazy řízení běhu programu

- 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`

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.

- Příkaz větvení `switch` mohou být vnořené.

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).

Podmíněné větvení – `if`

- `if (výraz) příkaz1; else příkaz2`
- Je-li hodnota výrazu `výraz != 0 (TRUE)`, provede se příkaz `příkaz1` jinak `příkaz2`.
- Část `else` je nepovinná. *Příkaz může být blok příkazů.*
- Podmíněné příkazy mohou být vnořené a můžeme je řetězit.

```
1 int max;
2 if (a > b) {
3     if (a > c) {
4         max = a;
5     }
6 }
```

- Příklad zápisu

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

```
1 int max;
2 if (a > b) {
3     ...
4 } else if (a < c) {
5     ...
6 } else if (a == b) {
7     ...
8 } else {
9     ...
10 }
```

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

Jaký je smysl těchto programů?

Programový přepínač – `switch`

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

Programový přepínač **switch** – Příklad

```

1 switch (v) {
2     case 'A':
3         printf("Upper 'A'\n");
4         break;
5     case 'a':
6         printf("Lower 'a'\n");
7         break;
8     default:
9         printf("It is not 'A' nor 'a'\n");
10        break;
11 }

1 if (v == 'A') {
2     printf("Upper 'A'\n");
3 } else if (v == 'a') {
4     printf("Lower 'a'\n");
5 } else {
6     printf("It is not 'A' nor 'a'\n");
7 }
```

lec03/switch.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).

```

1 int day_of_week = 3;
3 if (day_of_week == 1) {
4     printf("Monday");
5 } else if (day_of_week == 2) {
6     printf("Tuesday");
7 } else ... {
8 } else if (day_of_week == 7) {
9     printf("Sunday");
10 } else {
11     fprintf(stderr, "Invalid
12     number");
12 }

1 int day_of_week = 3;
2 switch (day_of_week) {
3     case 1:
4         printf("Monday");
5         break;
6     case 2:
7         printf("Tuesday");
8         break;
9     ...
10    case 7:
11        printf("Sunday");
12        break;
13    default:
14        fprintf(stderr, "Invalid number");
15        break;
16 }
```

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.

Větvení **switch** – pokračování ve vykonávání dalších větví

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

```

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 }
```

lec03/demo-switch_break.c

Podmíněný výraz / ternární operátor

- V příkladu hledání největšího společného dělitele můžeme podmíněné nastavení proměnné **d** větvením **if** a **else** nahradit podmíněným výrazem **?** :
- Téhož lze zkráceně dosáhnout podmíněným výrazem **expr₁ ? expr₂ : expr₃**.

```

1 int getGreatestCommonDivisor(int x, int y)
2 {
3     int d;
4     if (x < y) {
5         d = x;
6     } else {
7         d = y;
8     }
9     while ( (x % d != 0) || (y % d != 0) ) {
10        d = d - 1;
11    }
12    return d;
13 }
```

```

1 int getGreatestCommonDivisor(int x, int y)
2 {
3     int d = x < y ? x : y;
4     while ( (x % d != 0) || (y % d != 0) ) {
5         d = d - 1;
6     }
7     return d;
8 }
```

lec03/demo-gcd.c

Cykly

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

- for** – inicializace, podmínka a změna řídicí proměnné.

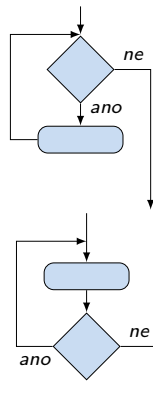
```
1 for (int i = 0; i < 5; ++i) {
2     ...
3 }
```

- while** – řídicí proměnná v těle cyklu.

```
1 int i = 0;
2 while (i < 5) {
3     ...
4     i += 1;
5 }
```

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

```
1 int i = -1;
2 do {
3     ...
4     i += 1;
5 } while (i < 5);
```



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

```
1 q = x;
2 while (q >= y) {
3     q = q - y;
4 }
```

```
1 q = x;
2 do {
3     q = q - y;
4 } 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

lec03/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

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

Cyklus for(; ;)

- Příkaz **for** cyklu má tvar **for ([vyraz₁]; [vyraz₂]; [vyraz₃]) příkaz;**
- Cyklus **for** používá řídicí proměnnou a probíhá následovně:
 - vyraz₁** – Inicializace (zpravidla řídicí proměnné);
 - vyraz₂** – Test řídicího výrazu;
 - Pokud **vyraz₂ != 0** provede se **příkaz**, jinak cyklus končí;
 - vyraz₃** – Aktualizace proměnných na konci běhu cyklu;
 - 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

Příkaz continue

- Příkaz návratu na vyhodnocení řídicího výrazu – continue.
- Příkaz continue lze použít pouze v těle cyklů.
 - for ()
 - while ()
 - do...while ()

```
1 int i;
2 for (i = 0; i < 20; ++i) {
3     if (i % 2 == 0) {
4         continue;
5     }
6     printf("%3d", i);
7 }
8 putchar('\n');
```

lec03/continue.c

```
$ clang continue.c && ./a.out
1 3 5 7 9 11 13 15 17 19
```

```
1 for (int i = 0; i < 10; ++i) {
2     printf("i: %i ", i);
3     if (i % 3 != 0) {
4         continue;
5     }
6     printf("\n");
7 }
```

lec03/demo-continue.c

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

Jan Faigl, 2025

B0B36PRP – Přednáška 03: Řídicí struktury, výrazy a funkce

24 / 60

Příkaz break – Ukončení cyklu

- Příkaz break způsobí opuštění těla cyklu. Program pokračuje dalším příkazem za tělem cyklu.

■ Příklad break ve while cyklu.

```
1 int i = 10;
2 while (i > 0) {
3     if (i == 5) {
4         printf("i reaches 5, leave the loop\n");
5         break;
6     }
7     i--;
8     printf("End of the while loop i: %d\n", i);
9 }
```

lec03/break.c

```
$ clang break.c
$ ./a.out
End of the while loop i: 9
End of the while loop i: 8
End of the while loop i: 7
End of the while loop i: 6
End of the while loop i: 5
i reaches 5, leave the loop
```

■ Příkazy continue a break ve for cyklu.

```
1 for (int i = 0; i < 10; ++i) {
2     printf("i: %i ", i);
3     if (i % 3 != 0) {
4         continue;
5     }
6     printf("\n");
7     if (i > 5) {
8         break;
9     }
10 }
```

lec03/demo-break.c

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

Jan Faigl, 2025

B0B36PRP – Přednáška 03: Řídicí struktury, výrazy a funkce

25 / 60

Příkaz goto

- Příkaz nepodmíněného lokálního skoku goto předá řízení na místo určené návěstím navesti – syntax goto navesti;.
- Návěští má tvar navesti 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"); // goto can jump to a label that
13                                     represents statement (there must be an address to be jumped at).
14 return -1;
```

lec03/goto.c

Definice proměnné není příkaz.

Jan Faigl, 2025

B0B36PRP – Přednáška 03: Řídicí struktury, výrazy a funkce

26 / 60

Vnořené cykly

- Příkaz break ukončuje vnitřní cyklus.

```
1 for (int i = 0; i < 3; ++i) {
2     for (int j = 0; j < 3; ++j) {
3         printf("i-j: %i-%i\n", i, j);
4         if (j == 1) {
5             break;
6         }
7     }
8 }
```

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.

```
1 for (int i = 0; i < 5; ++i) {
2     for (int j = 0; j < 3; ++j) {
3         printf("i-j: %i-%i\n", i, j);
4         if (j == 2) {
5             goto outer;
6         }
7     }
8 }
9 outer:
```

lec03/demo-goto.c

Jan Faigl, 2025

B0B36PRP – Přednáška 03: Řídicí struktury, výrazy a funkce

27 / 60

Příkaz a složený příkaz

Příkazy řízení běhu programu

Podmíněné řízení běhu programu

Konečnost cyklu

Příklad goto

```

1  #include <stdio.h>
3  int main(void)
4  {
5      for (int i = 0; i < 5; ++i) {
6          for (int j = 0; j < 3; ++j) {
7              printf("i-j: %i-%i\n", i, j);
8              if (j == 2) {
9                  goto outer;
10             }
11         }
12     }
13 outer:
14     int a;
15     return 0;
16 }

```

```

1  #include <stdio.h>
3  int main(void)
4  {
5      for (int i = 0; i < 5; ++i) {
6          for (int j = 0; j < 3; ++j) {
7              printf("i-j: %i-%i\n", i, j);
8              if (j == 2) {
9                  goto outer;
10             }
11         }
12     }
13 outer;; // za návěštím musí být příkaz
14     int a; // definice není příkaz
15     return 0;
16 }

```

lec03/goto_outer.c

Jan Faigl, 2025

B0B36PRP – Přednáška 03: Řídící struktury, výrazy a funkce

28 / 60

Příkaz a složený příkaz

Příkazy řízení běhu programu

Podmíněné řízení běhu programu

Konečnost cyklu

Konečnost cyklů 2/3

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

```

1  for (int i = 0; i < 5; ++i) {
2      ...
3  }

```
- Uvedené pravidlo konečnost cyklu nezaručuje.

```

1  int i = -1;
3  while ( i < 0 ) {
4      i = i - 1;
5  }

```

Konečnost cyklu závisí na hodnotě proměnné před vstupem do cyklu.

Jan Faigl, 2025

B0B36PRP – Přednáška 03: Řídící struktury, výrazy a funkce

31 / 60

Příkaz a složený příkaz

Příkazy řízení běhu programu

Podmíněné řízení běhu programu

Konečnost cyklu

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.

```

1  while (i != 0) {
2      j = i - 1;
3  }

```

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

Jan Faigl, 2025

B0B36PRP – Přednáška 03: Řídící struktury, výrazy a funkce

30 / 60

Příkaz a složený příkaz

Příkazy řízení běhu programu

Podmíněné řízení běhu programu

Konečnost cyklu

Konečnost cyklů 3/3

```

1  while (i != n) {
2      ... //příkazy nemenící hodnotu promenne i
3      i++;
4  }

```

lec03/demo-loop_byte.c

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

*Jak by vypadala podmínka pro proměnné typu double?
Co se stane pokud by proměnná i byla typu unsigned char?*

lec03/demo-loop.c

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

Jan Faigl, 2025

B0B36PRP – Přednáška 03: Řídící struktury, výrazy a funkce

32 / 60

Příklad – test zda-li je číslo prvočíslem – isPrimeNumber() 1/3

```
1 #include <stdbool.h>
2 #include <math.h>
4 _Bool isPrimeNumber(int n)
5 {
6     _Bool ret = true;
7     for (int i = 2; i <= (int)sqrt((double)n); ++i) {
8         if (n % i == 0) {
9             ret = false; // leave the loop once it is sure
10            break; // n is not a prime number
11        }
12    }
13    return ret;
14 }
```

lec03/demo-prime.c

- break – po nalezení prvního dělitele nemusíme dále testovat.

Příklad – test zda-li je číslo prvočíslem – isPrimeNumber() 3/3

- Místo příkazu break můžeme testovat hodnotu ret v podmínce opakování cyklu.

```
1 _Bool isPrimeNumber(int n)
2 {
3     _Bool ret = true;
4     const int maxBound = (int)sqrt((double)n);
5     for (int i = 2; ret && i <= maxBound; ++i) {
6         if (n % i == 0) {
7             ret = false; // leave the loop if it is sure n is not a prime number
8         }
9     }
10    return ret;
11 }
```

- Nebo ještě přidat nastavení ret při aktualizaci hodnoty řídící proměnné.

```
1 _Bool isPrimeNumber(int n)
2 {
3     _Bool ret = true;
4     const int maxBound = (int)sqrt((double)n);
5     for (int i = 2; ret && i <= maxBound; ret = !(n % i++ == 0));
6     return ret;
7 }
```

Čitelnost je v tomto případě diskutabilní.

Příklad – test zda-li je číslo prvočíslem – isPrimeNumber() 2/3

- Hodnota výrazu (int)sqrt((double)n) se v cyklu nemění.

```
1 for (int i = 2; i <= (int)sqrt((double)n); ++i) {
2     ...
3 }
```

Striktně vzato ji počítáme znovou a znovu při každém testu podmínky opakování cyklu.

- Můžeme použít operátor čárky a inicializovat proměnou maxBound.

```
1 for (int i = 2, maxBound = (int)sqrt((double)n);
2     i <= maxBound; ++i) {
3     ...
4 }
```

- Nebo můžeme definovat proměnnou maxBound jako konstantní.

```
1 _Bool ret = true;
2 const int maxBound = (int)sqrt((double)n);
3 for (int i = 2; i <= maxBound ; ++i) {
4     ...
5 }
```

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

Kódovací konvence

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

Ne nutně break v příkazu switch.

- Prerušení 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. *Případně posloupnost znaků.*

- Postup výpočtu výrazu s více operátory je dán prioritou operátorů.

*Postup výpočtu lze předsat 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 definováno, závisí na překladači.
- Pořadí vyhodnocení je **definováno pro operandů v logickém 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    // pořadí vyhodnocení 10 + (x * y)
10 + x + y    // pořadí vyhodnocení (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 (l-hodnota reprezentující místo v paměti).
- 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 – podmíněný příkaz.

*Jediný ternární operátor v C 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 celá čísla (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. 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.

&&	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, 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 (bit po bitu).
 - |** Bitové OR $x | y$ 1 když x nebo y je rovno 1 (bit po bitu).
 - ^** Bitové XOR $x ^ y$ 1 pokud pouze x nebo pouze y je 1 (exkluzivně právě jedna z variant) (bit po bitu).
 - ~** Bitové NOT $\sim x$ 1 pokud x je rovno 0 (bit po bitu).
 - <<** Posun vlevo $x << y$ Posun x o y bitů vlevo.
 - >>** Posun vpravo $x >> y$ 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

```
1 #include <inttypes.h>
3 uint8_t a = 4;
4 uint8_t b = 5;

6 a      dec: 4 bin: 0100
7 b      dec: 5 bin: 0101
8 a & b dec: 4 bin: 0100
9 a | b dec: 5 bin: 0101
10 a ^ b dec: 1 bin: 0001
12 a >> 1 dec: 2 bin: 0010
13 a << 1 dec: 8 bin: 1000
```

[lec03/bits.c](#)
Vizte rekurzivní implementaci [lec03/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.
Potřebujeme v scanf()!
 - 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.*

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>	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.

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

lec03/sizeof.c
- Příklad použití operátoru čárka.

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

Asociativita a priorita 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
- 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 provedení operací lze definovat důsledným **závorkováním**.

Operátor přetypování

- Změna typu za běhu programu se nazývá přetypování.
- Explicitní přetypování (cast) zapisuje programátor uvedením typu v kulatých závorkách, např.

```
1 int i;
2 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`.

C očekává hodnoty alespoň typu `int`.
- Operandy jsou automaticky přetypovány na `int` nebo `unsigned int`.

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, ...
 - Přirazení je výraz, který můžeme použít v jiném výrazu, např. `a = b = c = 10;`

Je to výraz v příkazu přiřazení.
 - C je staticky typovaný jazyk.
 - Proměnné lze přiřadit hodnotu výrazu pouze identického typu.
 - Příklad implicitní konverze při přiřazení.

Jinak je nutné provést typovou konverzi.
- ```
1 int i = 320.4; // implicit conversion from 'double' to 'int' changes value from 320.4 to 320 [-Wliteral-conversion]
3 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 typově bezpečné.**

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

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

■ Přiřazení je výraz

```
int x, y;
x = 6;
y = x = x + 6;
```

„syntactic sugar“

Část III

Zadání 3. domácího úkolu (HW03)

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 <b>int</b> , hodnota 23 |
| 14+16/2 | typ <b>int</b> , hodnota 22 |
| y=8     | typ <b>int</b> , hodnota 8  |

■ Přiřazení je výraz a jeho hodnotou je hodnota přiřazená levé straně.

■ Z výrazu se stává příkaz, pokud je ukončen středníkem.

Zadání 3. domácího úkolu HW03

Téma: Kreslení (ASCII art)

Povinné zadání: 3b; 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](https://en.wikipedia.org/wiki/ASCII_art)
  - Ošetření vstupních hodnot.
  - Volitelné zadání rozšiřuje obrázek domečku o plot.
- **Termín odevzdání:** 25.10.2025 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.**

# Část V

## Appendix

# Kódovací příklad – Tisk hodnot v šestnáctkové soustavě

- Reprezentace `float` hodnot.
  - Hodnota 85.125 je `0x42aa4000`.
  - Hodnota 0.1 je sice `0x3dcccccd`, ale je kódována `0x3dcccccd`. *Protože chyba je absolutně menší.*
- Implementujeme funkci pro tisk paměťové reprezentace hodnoty typu `float` jako posloupnosti hodnot bajtů v šestnáctkové soustavě.
- Přístup k `float` jako posloupnosti bajtů a tisk hex hodnot `"%02x"` funkcí `printf()`.
  - Adresním operátorem `&` získáme adresu proměnné.
  - Přetypujeme adresu jako ukazatel na hodnotu `char`.
  - Použijeme nepřímý adresní operátor `*` k přístupu k hodnotě na adrese uložené v ukazateli.

```
1 #include <stdio.h>
2
3 void print_float_hex(float v);
4
5 int main(void)
6 {
7 print_float_hex(85.125);
8 print_float_hex(0.1);
9 return 0;
10 }
11
12 void print_float_hex(float v)
13 {
14 ...
15 }
```

## Kódovací příklad – Tisk hodnot v šestnáctkové soustavě 1/3

- Získáme adresu proměnné `float v` operátorem `&v`.
- K hodnotám na adrese `&v` budeme přistupovat jako k bajtům, proto přetypujeme adresu na ukazatel (adresu) na hodnoty typu `char`.

```
unsigned char *p = (unsigned char*)&v;
```

- Hodnotu uloženou na adrese `p` získáme operátorem nepřímého adresování `*p`.

- Adresu následujícího bajtu za adresou uloženou v `p` získáme `p = p + 1`.

*Protože se jedná o ukazatel na `char`, probíhá inkrementace o `sizeof(char)`, tj. o 1 (ukazatelová aritmetika).*

- Vytisknuté hodnoty jsou v opačném než očekávaném pořadí **0x42aa4000** a **0x3dcccccd**.

```
1 int main(void)
2 {
3 print_float_hex(85.125);
4 print_float_hex(0.1);
5 ...
6 void print_float_hex(float v)
7 {
8 unsigned char *p = (unsigned char*)&v;
9 printf("Value %13.10f is 0x", v);
10 for (int i = 0; i < sizeof(float); ++i,
11 p = p + 1) {
12 printf("%02x", *p); // or use p[i]
13 }
14 putchar('\n');
```

```
1 $ clang floats.c -o floats && ./floats
2 Value 85.1250000000 is 0x0040aa42
3 Value 0.1000000015 is 0xcdcccc3d
```

## Kódovací příklad – Tisk hodnot v šestnáctkové soustavě 3/3

- Detekce uložení můžete být založena na různých principech.

- Intuitivně můžeme uložit definovanou hodnotu, která má pouze jeden bajt nenulový a ostatní nulové.

- Využijeme složeného typu `union`, ve kterém položky sdílejí paměť a umožňuje nám tak různý pohled na konkrétní block paměti.

1. Definujeme celočíselnou proměnnou o čtyřech bajtech, např., `uint32_t` z knihovny `stdint.h`.
2. Nastavíme hodnotu na `0x01 00 00 00`.
3. Otestujeme první bajt paměťové reprezentace.

```
1 #include <stdint.h>
2
3 _Bool is_big_endian(void)
4 {
5 union {
6 uint32_t i;
7 char c[4]; // 4 is fine here as we use
8 uint32_t
9 } e = { 0x01000000 };
10 return e.c[0];
11 }
```

## Kódovací příklad – Tisk hodnot v šestnáctkové soustavě 2/3

- Očekávaná reprezentace v šestnáctkové soustavě je pro `85.125` výstup **0x42aa4000** a pro `0.1` výstup **0x3dcccccd**. Namísto toho dostáváme `0x0040aa42` a `0xcdcccc3d`.

- Výstup je závislý na reprezentaci více bajtových hodnot v paměti. Pro architekturu (amd64) je to tzv. little endian.

<https://en.wikipedia.org/wiki/Endianness>

- Proto potřebujeme detekovat, jak jsou hodnoty uloženy, například funkcí

```
_Bool is_big_endian(void);
```

- a případně vytiskneme hodnoty v opačném pořadí.

```
1 void print_float_hex(float v)
2 {
3 const _Bool big_endian = is_big_endian();
4 // cast pointer to float to pointer to char
5 unsigned char *p = (unsigned char*)&v
6 + (big_endian ? 0 : (sizeof(float) - 1));
7 printf("Value %13.10f is 0x", v);
8 for (int i = 0; i < sizeof(float); ++i) {
9 printf("%02x",
10 *(big_endian ? p++ : p--));
11 }
12 printf("\n");
13 }
14 }
```

```
$ clang floats.c -o floats && ./floats
Value 85.1250000000 is 0x42aa4000
Value 0.1000000015 is 0x3dcccccd
```

## 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, případně nemusí jít zkompileovat, spustit, nebo dokonce padat a chovat se neobvykle či produkovat nesmyslné výsledky.

- To se může například také stát v případě, že nejsou proměnné inicializovány.

- **Vyhýbejte se příkazům (výrazům), které mohou vést na nedefinované chování!**

Příklad nedefinovaného chování

- Standard C nepředpisuje chování při přetečení celého čísla (*signed*)
  - V případě doplňkového kódu může být např. hodnota výrazu `127 + 1` typu `char` rovna `-128`, viz `lec03/demo-loop_byte.c`.
  - Reprezentace celých čísel však může být realizována jinak dle architektury např. přímým 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. překladače `clang` a `gcc` bez/s optimalizacemi `-O2`.

```
1 for (int i = 2147483640; i >= 0; ++i) {
2 printf("%i %x\n", i, i);
3 }
```

lec03/int\_overflow-1.c

Bez optimalizací program vypíše 8 řádků, pro `-O2` program zkompilovaný `clang` vypíše 9 řádků, `gcc` program skončí v nekonečné smyčce.1 for (int i = 2147483640; i >= 0; i += 4) {
2 printf("%i %x\n", i, i);
3 }

lec03/int\_overflow-2.c

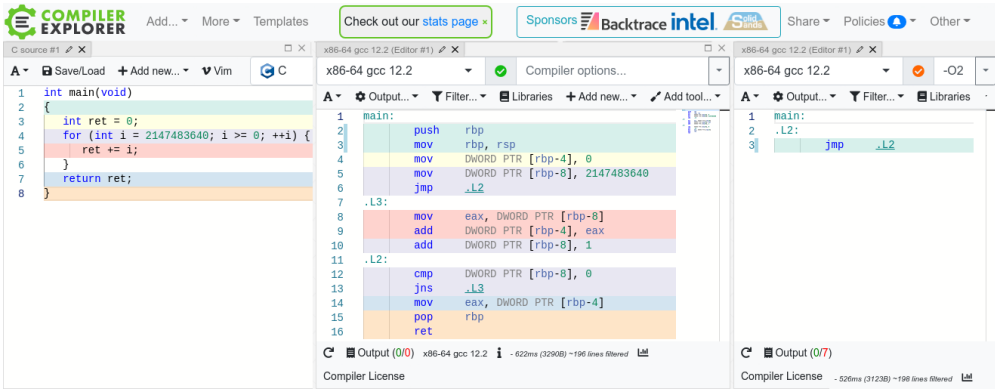
Program zkompilovaný `gcc` s `-O2` po spuštění (může) padá(at).  
Analyzujte kód `asm` generovaný přepínačem `-S`.

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>-&gt;</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>&amp;</code>  |              | adresa (reference)                  |
|          | <code>sizeof</code> |              | velikost                            |

Compiler Explorer – Analýza optimalizovaného kódu

- Vliv optimalizace `-O2` na výsledný kód, který obsahuje nedefinované chování, přetečení celého čísla.



<https://godbolt.org/z/G3GEz4vbv>

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>&gt;&gt;, &lt;&lt;</code>       |              | bitový posun vlevo, vpravo |
| 6        | <code>&lt;, &gt;, &lt;=, &gt;=</code> |              | porovnání                  |
| 7        | <code>==, !=</code>                   |              | rovno, nerovno             |
| 8        | <code>&amp;</code>                    |              | bitový AND                 |
| 9        | <code>^</code>                        |              | bitový XOR                 |
| 10       | <code>~</code>                        |              | bitový OR                  |
| 1        | <code>&amp;&amp;</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       | =             |              | přřazení                               |
|          | + =, - =      |              | přřazení součtu, rozdílu               |
|          | * =, / =, % = | P→L          | přřazení součinu, podílu a zbytku      |
|          | << =, >> =    |              | přřazení bitového posunu vlevo, vpravo |
|          | & =, ^ =,   = |              | přřazení bitového AND, XOR, OR         |
| 15       | ,             | L→P          | operátor čárka                         |

[http://en.cppreference.com/w/c/language/operator\\_precedence](http://en.cppreference.com/w/c/language/operator_precedence)