Y36PJC Programování v jazyce C/C++

# Datové typy, deklarace, operátory a výrazy.

Ladislav Vagner

# Dnešní přednáška

- Datové typy v C a C++.
- Zápis konstant.
- Deklarace proměnných.
- Operátory v C a C++.
- Vyhodnocení výrazů v C a C++.
- Obvyklé chyby.

# Minulá přednáška

- Organizace Y36PJC:
  - http://service.felk.cvut.cz/courses/Y36PJC
- Historie C a C++.
- Jednoduché ukázkové programy:
  - standardní vstup a výstup,
  - výrazy,
  - cykly,
  - jednoduché funkce, rekurze.

# Datové typy obecně

Určují:

- operace, které s prom. daného typu lze provádět,
- rozsah hodnot, které do něj lze uložit.

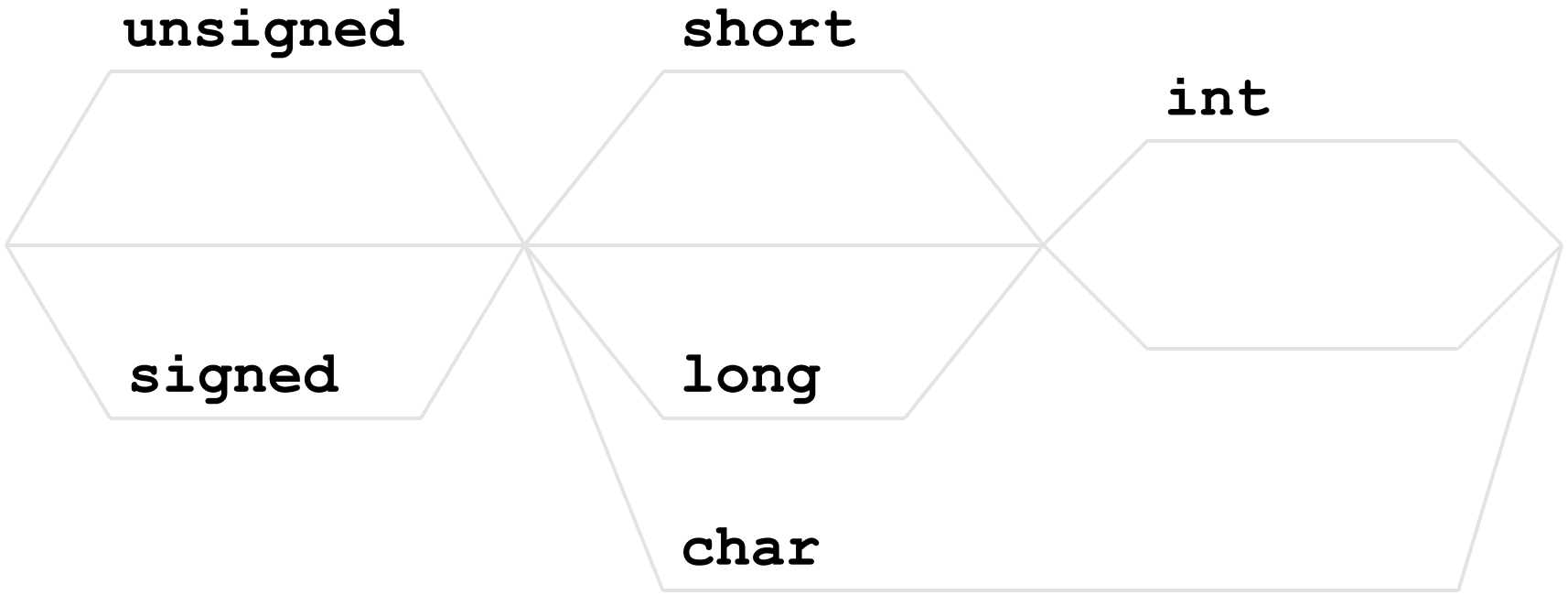
Původ:

- dané specifikací jazyka (built-in),
- uživatelem definované.

# Datové typy C/C++

- Jednoduché (skalární, primitivní):
  - celočíselné,
  - desetinné,
  - znakové a (řetězcové),
  - ukazatele,
  - výčty (**`enum`**)
- Strukturované:
  - struktury (**`struct`**),
  - třídy (**`class, struct`**),
  - unie (**`union`**),
- Datový typ **`void`**.

# Celočíselné typy




- Celkem 20 celočíselných datových typů (8 různých).
- Nepřenositelné datové typy:
  - `long long int` – gcc,
  - `__int64` – MSVC

# Celočíselné typy

- Mnoho typů je ekvivalentních:

```
short int = short = signed short =
                              = signed short int
long int = long = signed long = signed long int
int = signed int
unsigned = unsigned int
unsigned short = unsigned short int
unsigned long = unsigned long int
```

Buď:

```
char = signed char
```

nebo:

```
char = unsigned char (méně obvyklé)
```

- Proč existuje takové množství typů?

# Celočíselné typy

- C/C++ norma neurčuje rozsah datových typů
- Garantuje pouze:

```
          short int <= int <= long int
unsigned short <= unsigned int <= unsigned long
```

- Vnitřní reprezentace – většinou doplňkový kód.
- Rozsahy – dané implementací, většinou:

| | `char` | `short` | `int` | `long` |
|---|---|---|---|---|
| 16b | 1B | 2B | 2B | 4B |
| 32b | 1B | 2B | 4B | 4B |
| 64b | 1B | 2B | 4B | 8B |

# Celočíselné typy

- Zápis celočíselných konstant:
  - desítkově: `123, 56789`
  - šestnáctkově: `0x12, 0x5E`
  - osmičkově: `012, 0377`

- S udáním typu:
  - long: `567l`
  - unsigned: `890u`
  - long long: `999LL`
  - __int64: `987i64`

- Kombinace: `0123u, 0x567lu`

# Desetinné typy

- Pouze 3 typy:
  - **`float`**
  - **`double`**
  - **`long double`**
- Rozsah – implementačně závislý. Platí:
  **`float <= double <= long double`**
- Vnitřní reprezentace – většinou dle IEEE 754:

| | vel | rozsah | cifer |
|---|---|---|---|
| **`float`** | 4B | ~3.4E38 | 7-8 |
| **`double`** | 8B | ~1.8E308 | 15-16 |
| **`long double`** | 10B | ~3.4E4932 | 19-20 |

# Desetinné typy

- Zápis desetinných konstant:
  - desítkově: `1.23, 567.89`
  - s exponentem: `1e15, 2.36e-9`

- S udáním typu:
  - float: `567.31f`
  - long double: `890.25l`

# Znaky

Datový typ

- **`char`** pro ASCII znaky:
  - velikost většinou 8-bit,
  - v aktuální kódové stránce.

- **`wchar_t`** pro UNICODE znaky:
  - Windows: vel 16-bit, kódování UTF-16 (~UCS-2),
  - Linux: vel 32-bit, kódování UTF-32 (~UCS-4).

# Znaky

Znakové konstanty:

```
'a', 'b', '*'
```

Escape-sekvence:

```
'\'', '\"', '\\'
'\n', '\t', '\r',
```

Zápis pomocí ASCII hodnoty:

```
'\012', '\x61'
```

UNICODE znaky:

```
L'a', L'\\', L'\n'
```

# Řetězce

Neexistuje vlastní vestavěný datový typ:

- ukazatel na první znak (C, C++),
- knihovní třída **`string`** (C++).

Řetězcové konstanty – ASCIIZ konvence:

- znaky uložené v paměti za sebou,
- ukončené znakem s hodnotou bin. `0`,
- znak `'0'` není ukončující (bin. **`0x30`**).

# Řetězce

Řetězcové konstanty:

- přímý zápis v uvozovkách: `"abc"`
- escape sekvence: `\",\\,\n,\r,\t`
- zápis ASCII hodnotou: `\012, \x41`
- UNICODE řetězce: `L"text"`

```
"I said \"Hello\""
"\"\x47\111\x20\x21\42"
"\\\"\"\\"
"\\""\"\\"
L"This is \"UNICODE\42 string"
"C:\autoexec.bat"
```

# Výčtový typ

- Symbolické pojmenování možných hodnot:
  - reprezentován datovým typem **`int`**,
  - deklarace – klíčové slovo **`enum`**.

```
enum EDays { SUNDAY, MONDAY, TUESDAY,
  WEDNESDAY, THURSDAY, FRIDAY, SATURDAY };

EDays a;

a = MONDAY;
cout << a << endl; // zobrazi 1
cout << (a + 1) << endl; // zobrazi 2
```

# Deklarace

Obecný tvar deklarace:

```
<pam. třída> <kvalifikátor> <dat. typ> <deklarátor>
```

Paměťová třída:

| | |
|---|---|
| **`auto`** | alokovat lokálně na zásobníku, implicitní pro proměnné, |
| **`register`** | umístit do registru CPU (doporučení), |
| **`static`** | statické přidělení (dat. segment), |
| **`extern`** | nealokovat paměť, pouze deklarace, implicitní pro funkce. |

# Deklarace

```
int         a;          // globalni (staticky alok.)
static int b;           // globalni, není videt 'ven'
extern int c;           // globalni, 'odjinud'
void foo ( void )       // funkce, je videt 'ven'
 {
   int         d;       // lokalni = auto int d;
   static int e = 5;    // staticky alokovana
   register int f;
   …
 }

static void bar ( void ) // funkce, neni videt 'ven'
 {
   static int e = 10; // ok, nekoliduje s e ve foo
   …
 }
```

# Deklarace

Kvalifikátor (nepovinný):

| | |
|---|---|
| **`const`** | konstanta (nelze měnit), |
| **`volatile`** | neoptimalizovzat přístupy k proměnné, (např. pro paměťově mapované I/O). |

Příklad:

```
int                          a;
register int                 x;
const int                    y = 10;
extern const double          pi;
volatile int               * timerPtr;

y = 20;   // !!!
```

# Deklarace

Deklarátor:

| | |
|---|---|
| `<identifikátor>` | |
| `'&' <identifikátor>` | reference |
| `'*' [ 'const' ] <deklarátor>` | ukazatel |
| `<deklarátor> '[' <konst. výraz> ']'` | pole |
| `<deklarátor> '(' <seznam param.> ')'` | funkce |
| `'(' <deklarátor> ')'` | |

Příklad:

```
int      a;
int    * b;
int      foo ( void );
int      pole [30];
```

# Složitější deklarace

Příklady:

```
int * a;                  // ukazatel na int
const int * b;            // ukazatel na konstantu
int * const c;            // konstantní ukazatel
const int * const d;      // konstantní + na konstantu
int * e (void);           // fce vrací ukazatel na int
int (*f)(void);           // ukazatel na funkci
int * g [20];             // pole 20 ukazatelů na int
int (*h)[40];             // ukazatel na pole 40 int
int (*(i(void)))(int)     // fce vrací ukazatel na fci
int (*j[30])(int)         // pole 30 ukazatelu na fci
int (*(*k)[50])(int)      // ukazatel na 50 prvkove
                          // pole ukazatelu na fci
int & l = *a;             // reference na int
int * &m = a;             // reference na ukazatel
int & *n = a;             // !!! nelze uk na referenci
```

# Inicializace

Staticky alokované proměnné:
- vyplněné hodnotou `0`,
- inicializované na počátku programu.

Lokální proměnné:
- jednoduché dat. typy - nejsou inicializované,
- objektové dat. typy - konstruktor volán na počátku bloku (funkce).

Explicitní inicializace – může být součástí deklarace:
- lokální proměnné – libovolný výraz,
- staticky alokované – konstantní výraz.

# Inicializace

```
int a = 10;       // globalni prom inic. na 10
int b;            // globalni prom inic. na 0
int main ( int argc, char * argv [] )
 {
   int c;         // lok prom, není inic.
   static int d; // staticky alokovana, inic. na 0

   for ( int i = 0; i < 10; i ++ )
    {
      int e = 20;// lok. prom s inicializaci
      d = d + e;
      b ++; e++;
    }
   cout << d << " " << c << " " << b << endl;
   return ( 0 );
 }
```

# Operátory

C/C++ operátory:

- Aritmetické: unární `-`, `+`, `-`, `*`, `/`, `%` (modulo),
- Bitové: `~`, `&`, `|`, `^`, `<<`, `>>`,
- Logické: `!`, `&&`, `||`,
- Relační: `<`, `<=`, `>`, `>=`, `==`, `!=`,
- Přiřazovací: `+=`, `-=`, `%=`, `>>=`, `&=`, …,
- Ternární: `? :`
- Inkrement/dekrement: `++`, `--`
- Volání funkce, indexace: `()`, `[]`
- Přístup ke složkám struktury: `.`, `->`
- Reference a dereference: `&`, `*`

# Operátory

Aritmetické operátory:

- zápis podobný jako v jiných pgm. jazycích,
- zbytek po dělení - `%`,
- typ výsledku je určen typem operandů,
- automatické konverze datových typů před provedením operace.

Příklad:

```
double x;
x      = 2   / 4;     // x = 0.0, proc ?
x      = 2.0 / 4;     // x = 0.5
x      = 2   / 4.0f; // x = 0.5
```

# Operátory

Bitové operátory:

- `|` or,
- `&` and,
- `^` xor,
- `~` bitová negace,
- `>>`, `<<` aritmetický posuv vpravo / vlevo.
- Jak zařídit bitový posuv (Java operátory `<<<` a `>>>`)? Unsigned operandem.

Příklad:

```
int x;          // 37 = 0010 0101
x = 37 | 94;    // 94 = 0101 1110
                // x  = 0111 1111 => 127
```

# Operátory

Příklady použití bitových operátorů:

- Vytvoření masky, kde je nastaven pouze i-tý bit:

```
mask = 1 << i
```

- Vytvoření masky, kde je nulován pouze i-tý bit:

```
mask = ~(1 << i)
```

- Nastavení i-tého bitu na 1:

```
val = val | (1 << i)
```

- Nastavení i-tého bitu na 0:

```
val = val & ~(1 << i)
```

- Překlopení i-tého bitu:

```
val = val ^ (1 << i)
```

- Test, zda je i-tý bit nastaven:

```
(val & ( 1 << i )) != 0
```

# Operátory

Logické operátory:

- `||` or,
- `&&` and,
- `!` logická negace,
- výsledkem je hodnota `0` (**`false`**) nebo `1` (**`true`**),
- zkrácené vyhodnocení (ukončí se v okamžiku, kdy je jasný výsledek).

Příklad:

```
x = 37 || 94;  // x = 1 (true)
x = 37 && 94;  // x = 1 (true)
x = ! 37;      // x = 0 (false)
x = x && delAll ( "C:\\" ); // nesmaze
```

# Operátory

Relační operátory:

- `<, <=, …,`
- výsledkem je hodnota `0` (**`false`**) nebo `1` (**`true`**),
- pozor na asociativitu.

Příklad:

```
x = 5;
if ( 10 < x < 30 ) doJob (); // does
 // ( ( 10 < x ) < 30 )

if ( a == b == c == d ) …
 // ((( a == b ) == c ) == d )
```

# Operátory

Přiřazovací operátory:

- `=, +=, -=, …,`
- pravě asociativní, lze seskupovat,
- vedlejší efekt (zápis do paměti) není serializovaný.

Příklad:

```
x = y = z = 0; // x = ( y = ( z = 0 ));

x += 10;       // x = x + 10

x -= 20 + 30;  // x = x - ( 20 + 30 );

x *= x *= 20;  // nedefinovano
```

# Operátory

Ternární operátor:

- **`podmínka ? hod_pravda : hod_nepravda`**
- vyhodnotí právě jeden z výrazů pravda / nepravda,
- výrazy větví pravda a nepravda musejí mít stejný (konvertovatelný) typ.

Příklad:

```
max2 = ( x > y ) ? x : y;
absx = x >= 0 ? x : -x;
cout << "X je " << (X > 0 ? "kladne" :
     "nekladne") << endl;
cout << "X je " << (X > 0 ? "kladne" :
     X ) << endl;
```

# Operátory

Další operátory:

- `++, --` pre/post inkrement/dekrement,
- `,` operátor "zapomenutí",
- `*,&` (jako unární) dereference / reference,
- `.,->` přístup k složkám třídy / struktury
- `(),[]` (jako postfixové) volání funkce, indexace.

Příklad:

```
int a = 4, b;
b = a ++;          // b = 4, a = 5
b = ++ a;          // b = 6, a = 6
b = a ++ ++;       // !!
```

# Operátory

| Pr. | Operátory | Asociativita |
|---|---|---|
| 1 | `() [] -> .` | zleva doprava |
| 2 | `! ~ ++ -- + - (typ) * & sizeof` | zprava doleva |
| 3 | `* / %` | zleva doprava |
| 4 | `+ -` | zleva doprava |
| 5 | `<< >>` | zleva doprava |
| 6 | `< > >= <=` | zleva doprava |
| 7 | `== !=` | zleva doprava |
| 8 | `&` | zleva doprava |
| 9 | `^` | zleva doprava |
| 10 | `\|` | zleva doprava |
| 11 | `&&` | zleva doprava |
| 12 | `\|\|` | zleva doprava |
| 13 | `?:` | zleva doprava |
| 14 | `= += -= *= /= %= >>= <<= &= \|= ^=` | zprava doleva |
| 15 | `,` | zleva doprava |

# Výrazy

- Výraz je vyhodnocován podle priorit operátorů a jejich asociativity.
- Změna priority či asociativity pomocí závorek.
- Pořadí vyhodnocení není garantováno, k dispozici pro optimalizaci.
- Pořadí je definované pro ternární operátor a čárku.
- Zkrácené vyhodnocení logických operátorů.
- Pořadí vyhodnocení může být důležité, pokud má podvýraz vedlejší efekt (volání funkce, přiřazení, ++,...).
- V C/C++ lze zapsat nedefinované výrazy.

# Výrazy

- l-value:
  - výraz, který může stát na levé straně operátoru =,
  - má paměťovou reprezentaci (je kam uložit výsledek),
  - lze na něj vytvořit ukazatel či referenci.
- r-value:
  - výraz, který může stát na pravé straně operátoru =,
  - má hodnotu, ale nemá paměťovou reprezentaci,
  - např. může existovat pouze v registru CPU během výpočtu, pak je zapomenut.

# Výrazy

Příklady l-value a r-value:

```
int a, *b = &a, *c[5], & d = a;
```

| l-value | r-value |
|---|---|
| `a` | `a + 5` |
| `b` | `2 * a` |
| `c[3]` | `&a` |
| `d` | `&b` |
| `*b` | `c` |
| `*c` | `&c[3]` |
| `*(&a)` | `a ++` |
| `*(b+1)` | `a > 0 ? *b : d` |

# Výrazy

Pozor na prioritu a asociativitu:

```
int a, b, c, *d = &a;

c = a + b >> 1;     // c = (a + b) >> 1

c = a & 2 == 2;     // c = a & (2 == 2)

c = 3 * a / 4 * b; // c = (3 * a / 4) * b

c = *d ++;          // c = *(d ++)
```

# Výrazy

Nedefinované výrazy:

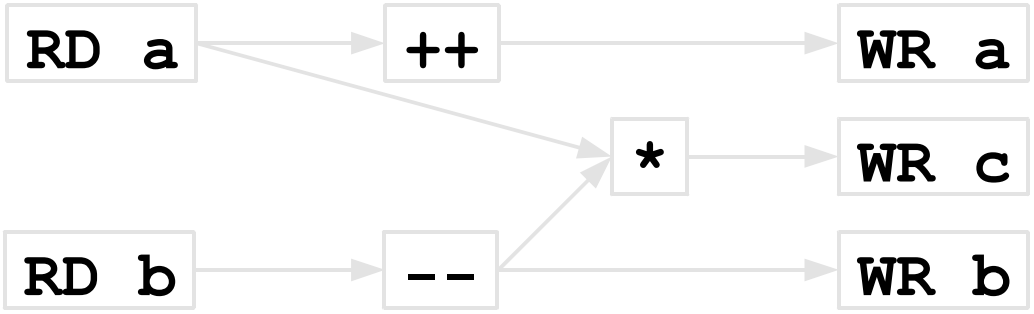
- Není definováno v jakém pořadí jsou vyhodnoceny podvýrazy.
- Není definováno, kdy se do paměti zapíše výsledek, pokud má operátor vedlejší efekt (`++`, `=`, `+=`, ...).
- Všechny vedlejší efekty se uplatní nejpozději po skončení příkazu, operátorech `||`, `&&`, `?:` a `,`.

Příklad:

```
int a = 1, c;
c = a++ + a++; // a = 2, 3 ?
               // c = 2, 3 ?
```

# Výrazy - příklad

```
c = a ++ * -- b;
```




Možné překlady (uspořádání):

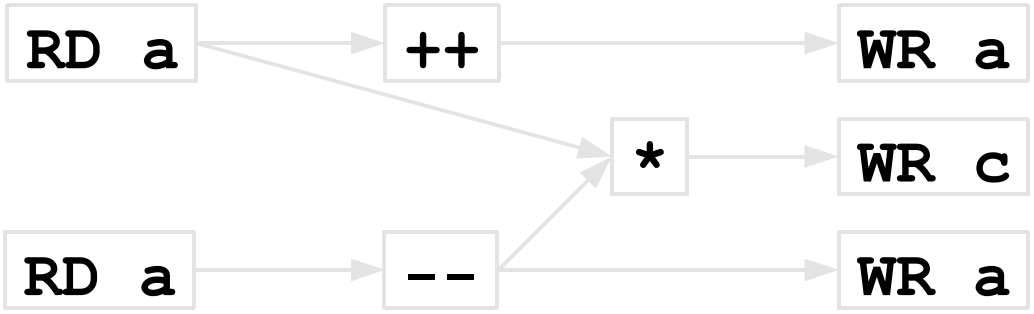
```
RD a  RD b  --  *  WR c  WR b  ++  WR a

RD a  ++  WR a  RD b  --  WR b  *  WR c

RD b  --  WR b  RD a  ++  WR a  *  WR c
```

# Výrazy - příklad

```
c = a ++ * -- a; // !!
```




Možné překlady (uspořádání):

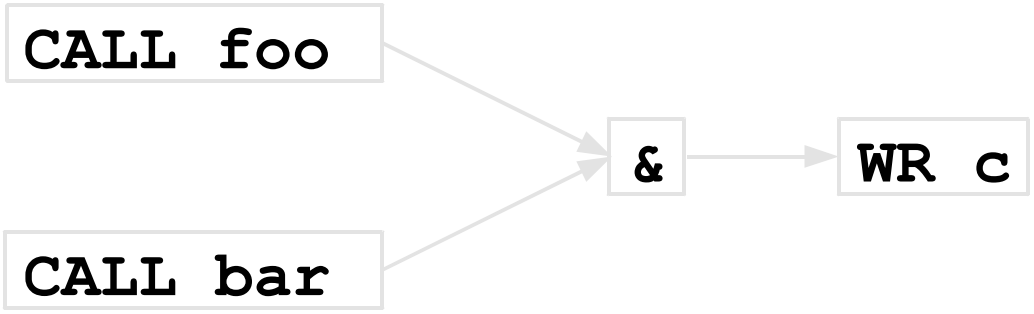
```
RD a  RD a  --  *  WR c  WR a  ++  WR a

RD a  ++  WR a  RD a  --  WR a  *  WR c

RD a  --  WR a  RD a  ++  WR a  *  WR c
```

# Výrazy - příklad

```
c = foo () & bar (); //  !!!
```



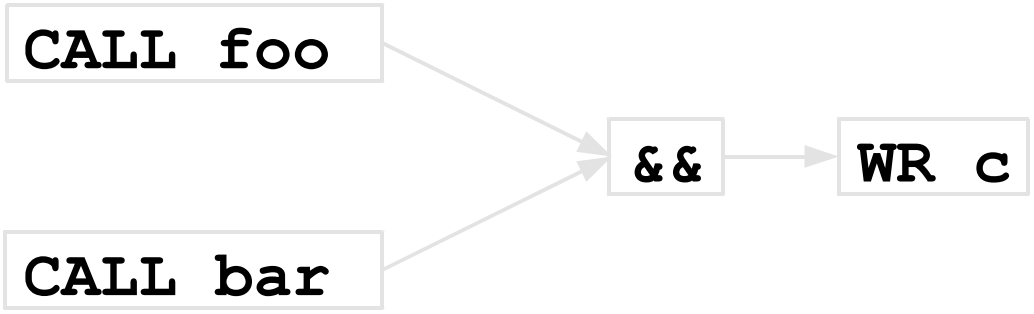

Možné překlady (uspořádání):

```
CALL foo   CALL bar   &   WR c

CALL bar   CALL foo   &   WR c
```

# Výrazy - příklad

```
c = foo () && bar ();
```




Možné překlady (uspořádání):

```
CALL foo   CALL bar   &&   WR c
```

~~`CALL bar   CALL foo   &&   WR c`~~

# Dotazy …

Děkuji za pozornost.