Medvědův průvodce po Céčku

Martin Mareš
mj@ucw.cz

Katedra Aplikované Matematiky
MFF UK Praha

2019

Typový systém: Základní typy

Základní typy: (norma, GCC/amd64)

- Celočíselné typy:

`_Bool (bool)`	1 bit	1 b
`char`, `(un)signed char`	8 bitů, $\forall$ znak	8 b
`(unsigned) short int`	$\geq$ 16 bitů	16 b
`(unsigned) int`	$\geq$ 16 bitů	32 b
`(unsigned) long int`	$\geq$ 32 bitů	64 b
`(unsigned) long long int`	$\geq$ 64 bitů	64 b
bit-fields		

- Floating-point typy:

`float`	$\geq$ 6 číslic, $\geq 10^{37}$	32 b
`double`	$\geq$ 10 číslic, $\geq 10^{37}$	64 b
`long double`	$\geq$ 10 číslic, $\geq 10^{37}$	80 b
jejich komplexní varianty (`_Complex`)		

- Výčty (`enum`), chovají se celočíselně
- Typ `void`

Typový systém: Odvozené typy

Odvozené typy:

- Pole
- Struktury
- Uniony
- Funkce
- Ukazatele (speciální: `void *`)

Částečné typy:

- `struct who_knows_what *`
- `int f()`
- `int x[]`

Typový systém: Reprezentace hodnot

Reprezentace:

- Hodnoty jsou složeny z bytů (`char`ů), výjimka: bitová pole
- K reprezentaci lze přistupovat pomocí `unsigned char`ů
- Může obsahovat padding a vyžadovat zarovnání
- Některé reprezentace mohou být nekorektní (trap)

Celá čísla:

- `unsigned` čistě binární, může mít padding (třeba paritu)
- `unsigned char` nemá padding
- `signed` má znaménkový bit, bity hodnoty a padding
- Hodnota z `signed` ∩ `unsigned` vypadá v obou stejně

Složené typy:

- Struktury: padding mezi prvky, prefixová vlastnost
- Pole: bez paddingu

Literály

- Integery: `12345`, `0xdeadbeef`, `0177`
- Typované integery: `12345U`, `12345L`, `12345ULL`
- Floaty: `1.5`, `1.`, `.1`, `1e20`, `1e-20`
- Typované floaty: `1.5D`, `1.5L`
- Hexadecimální floaty: `0x1.fffep10`
- Znaky (int): `'x'`, `'\n'`, `'\033'`, `'\x1b'`
- Řetězce (const char []): `"brekekex\n"`, `"str" "ing"`
- “Široké” znaky (wchar_t): `L'ž'`, `L"žluťoučký kůň"`
- Unicode: `L'\u2302'`, `L'\U00002302'` (též v identif.)
- Složené literály: `(char []){ 1, 2, 3 }`
  (pozor na to, kde jsou uložené)
- Pojmenovaně: `(struct point){ .x=1, .y=2 }`

Operátory a jejich priority

1. `(`*podvýraz*`)` *ident* *literál* `_Generic`
2. `[`*index*`]` `(`*volání*`)` `.`*prvek* `->`*prvek* `++` `--` (postfixové)
3. `++` `--` `sizeof` `_Alignof` `&` `*` `+` `-` `~` `!` `(`*typ*`)` (prefixové)
4. `*` `/` `%`
5. `+` `-`
6. `<<` `>>`
7. `<` `>` `<=` `>=`
8. `==` `!=`
9. `&`
10. `^`
11. `|`
12. `&&`
13. `||`
14. `?:`
15. `=` `+=` `*=` `...` (asoc. zprava)
16. `,`

Priority: drobné zákeřnůstky

Chytáky:

```
x&1 == y&1
```

```
1<<4 + 1<<5
```

Priority: drobné zákeřnůstky

Chytáky:

```
x&1 == y&1 ... x&(1==y)&1
```

```
1<<4 + 1<<5 ... 1<<(4+1)<<5
```

Konverze typů

Automatické konverze:

- Integery menší než `int` → `int` / `unsigned int`
- Pole → ukazatel na první prvek
- Funkce → ukazatel na funkci
- 0 ↔ null pointer (vyžaduje-li to kontext)
- Ke konverzím nedochází u `sizeof`, `_Alignof` a `&`

U binárních operátorů (mimo přiřazení a posuvů):

- `_Complex`
- `long double`
- `double`
- `float`
- "větší integer"
  - stejně velký unsigned a signed → unsigned

Konverze typů II

Na vyžádání (nebo při přiřazení):

- Integer na `bool`: 0 → 0, nenula → 1
- Integer na menší unsigned: modulo
- Integer na menší signed: závislé na implementaci (i trap)
- Float na integer: zaokrouhluje k nule (ořezává)
- Integer na float: může ztratit přesnost
- Complex na real: zahodí imaginární část

V parametrech funkce:

- Typ je určen prototypem funkce, existuje-li. Jinak:
- Malé celočíselné typy se předají jako `int`
- `float` se předá jako `double`

Pořadí vyhodnocování

Liší se pořadí **vyhodnocování** (podle asociativity operátorů) a **provádění side-efektů** (vesměs nedefinováno).

Synchronizační body (sequence points):

- operátory `&&`, `||` a `?`
- čárka jako operátor
- konec výrazu (příkaz, podmínka v cyklu ...)
- volání funkce a návrat z ní

Mezi synchronizačními body je zakázáno měnit jeden objekt vícekrát nebo současně číst a zapisovat (výjimka: přiřazení).

Chyták: `printf("%d %d %d", a++, a++, a++);`

Deklarace

```
static volatile int *(* const f)[10] = NULL;
```

???

Deklarace

```
static volatile int *(* const f)[10] = NULL;
```

Třída uložení

- `static`, `extern`, `auto`, `register`, `_Thread_local`
- Ovlivňuje alokaci, dobu života i viditelnost
- Také sem patří `typedef`

Kvalifikátory

- `const`, `volatile`, `restrict`
- `_Alignas(...)`

Typy

- Čteme jako výraz

Inicializátor

- Pro staticky alokované implicitně 0

Deklarace funkcí

```
int main(int argc, char **argv);
```
(klasika)

```
int f();
```
(o argumentech nic neříkám)

```
int f(void);
```
(argumenty nemá)

```
int f(char x, float f);
```
(typované argumenty)

```
int f(int x, ...);
```
(jeden nebo více argumentů)

```
typedef struct { int x,y; } vec;
vec add(vec x, vec y);
```
(struktury předám hodnotou)

```
void sum(int pole[]);
```
(ale pole odkazem)

```
void sum(int n, int matice[n][n]);
int cmp(int x[restrict], int y[restrict]);
```

```
static inline int cmp(int x, int y);
```

Příkazy

- prázdný příkaz
- *výraz*
- `{` *příkaz*`;` *deklarace*`;` `...` `}`
- `if (`*výraz*`)` *příkaz* `else` *příkaz*
- `while (`*výraz*`)` *příkaz*
- `do` *příkaz*`;` `while (`*výraz*`)`
- `for (`*a*`;`*b*`;`*c*`)` *příkaz*
- `for (`*deklarace*`;`*b*`;`*c*`)` *příkaz*
- `break`, `continue`, `goto` *návěští*
- `return` *výraz*
- `switch`, `case`, `default`
- `_Static_assert`(*podmínka*, *zpráva*)

Funkce s proměnlivým počtem argumentů

```
#include <stdarg.h>

int f(char *s, ...) {
  va_list args;
  va_start(args, s);
  while (*s)
    switch (*s++) {
      case 'i': got_int(va_arg(args, int));
                break;
      case 's': got_str(va_arg(args, char *));
                break;
    }
  va_end(args);
}
```

(pozor na implicitní typové konverze)

Zpracování argumentů vícekrát

```
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>

char *memprintf(char *s, ...) {
  va_list args;
  va_start(args, s);

  int len = vsnprintf(NULL, 0, s, args) + 1;
  char *p = malloc(len);
  vsnprintf(p, len, s, args);

  va_end(args);
  return p;
}
```

Zpracování argumentů vícekrát

```
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>

char *memprintf(char *s, ...) {
  va_list args, args2;
  va_start(args, s);
  va_copy(args2, args);
  int len = vsnprintf(NULL, 0, s, args2) + 1;
  char *p = malloc(len);
  vsnprintf(p, len, s, args);
  va_end(args2);
  va_end(args);
  return p;
}
```

Fáze překladu

1. Konverze do kompilační znakové sady a nahrazení trigrafů (??= → # apod.)
2. Slepení řádků končících na '\'
3. Rozdělení na pp-tokeny a mezery, komentáře → mezery
4. Preprocesor (direktivy, makra, includy)
5. Konverze do cílové znakové sady
6. Slepení navazujících stringových literálů
7. Konverze pp-tokenů na tokeny, zapomenutí mezer
8. Syntaktická a sémantická analýza
9. Linker

Direktivy preprocesoru

- `#if` výraz, `#else`, `#endif`, `#elif` výraz
  - integerové konstantní výrazy bez přetypování
  - počítají se v (u)`intmax_t`
  - navíc `defined(` ident `)`, `defined` ident
  - všude jinde se nahrazují makra
  - nedefinované identifikátory (i klíčová slova) $\rightarrow$ 0
- `#ifdef` ident, `#ifndef` ident
- `#define` ident [`(`arg`)`] expanze, `#undef`
  - pozor na mezery před `(`arg`)`
- `#include` <jméno> / `"`jméno`"` / makra
  - pozor, po expanzi maker se neslepují řetězce
  - ochrana proti vícenásobné inkluzi
- `#warning`, `#error`
- `#line` řádek [`"`soubor`"`]
- `#pragma`, `_Pragma`

Nahrazování maker

1. Máme opravdu nahrazovat?
   - “funkcovité” makro nelze volat bez závorek
   - uvnitř direktiv se někdy nenahrazuje
2. Posbírají se argumenty (včetně vnořených závorek)
3. Argumenty se rekurzivně expandují
4. Volání makra se nahradí jeho definicí s argumenty
   - Argumenty operátorů `#` a `##` se nahrazují neexpandovaně
5. Provedou se preprocesorové operátory:
   - `#` stringifikuje
   - `##` slepuje tokeny (i prázdné)
6. Ve výsledné posloupnosti pp-tokenů se hledají další makra
7. Případně blokujeme rekurzi

Použití preprocesoru

Jednoduchá makra:

```
#define PLUS(a,b) ((a)+(b))
#define MIN(a,b) ((a)<(b) ? (a) : (b))
#define ARRAY_SIZE(a) (sizeof(a) / sizeof(*(a)))
```

Makro jako příkaz:

```
#define DBG(msg) do { \
  if (debug) puts(msg); } while(0)
```

Blokování rekurze je užitečné:

```
int f(int x);
#define f f
```

Makra s proměnlivým počtem argumentů:

```
#define F(...) do { \
  if (debug) printf(__VA_ARGS__) } while(0)
```

Temné kouty preprocesoru

Na tokenech občas záleží:

```
#define Ex +2
```

Pak `12Ex` není totéž jako `12␣Ex`

Pořadí vyhodnocování:

```
#define LEP(x,y) x ## y
#define LEP2(x,y) LEP(x,y)
#define A aa
LEP(a,1) →
```

Temné kouty preprocesoru

Na tokenech občas záleží:

```
#define Ex +2
```

Pak `12Ex` není totéž jako `12␣Ex`

Pořadí vyhodnocování:

```
#define LEP(x,y) x ## y
#define LEP2(x,y) LEP(x,y)
#define A aa
```

`LEP(a,1)` → `a1`

`LEP(A,1)` →

Temné kouty preprocesoru

Na tokenech občas záleží:

```
#define Ex +2
```

Pak `12Ex` není totéž jako `12␣Ex`

Pořadí vyhodnocování:

```
#define LEP(x,y) x ## y
#define LEP2(x,y) LEP(x,y)
#define A aa
```

`LEP(a,1)` → `a1`

`LEP(A,1)` → `A1`

`LEP2(A,1)` →

Temné kouty preprocesoru

Na tokenech občas záleží:

```
#define Ex +2
```

Pak `12Ex` není totéž jako `12␣Ex`

Pořadí vyhodnocování:

```
#define LEP(x,y) x ## y
#define LEP2(x,y) LEP(x,y)
#define A aa
```

`LEP(a,1)` → `a1`

`LEP(A,1)` → `A1`

`LEP2(A,1)` → `aa1`

Temné kouty preprocesoru

Na tokenech občas záleží:

```
#define Ex +2
```

Pak `12Ex` není totéž jako `12␣Ex`

Pořadí vyhodnocování:

```
#define LEP(x,y) x ## y
#define LEP2(x,y) LEP(x,y)
#define A aa
LEP(a,1) → a1
LEP(A,1) → A1
LEP2(A,1) → aa1
```

Parametry s čárkami:

```
#define LEFT -1,0
#define WALK0(dx,dy) x+=dx, y+=dy
#define WALK(a) WALK0(a)
WALK(LEFT); → WALK0(-1,0); → x+=-1, y+=0;
```

Kreslíme obrázky preprocesorem

```
#define X )*2+1
#define _ )*2
#define s ((((((((((((((((0
static const uint16_t stopwatch[] = {

s _ _ _ _ _ X X X X X _ _ _ X X _ ,
s _ _ _ X X X X X X X X _ X X X ,
s _ _ X X X _ _ _ _ _ X X X _ X X ,
s _ X X _ _ _ _ _ _ _ _ _ X X _ _ ,
s _ X X _ _ _ _ _ _ _ _ _ X X _ _ ,
s X X _ _ _ _ _ _ _ _ _ _ X X _ ,
s X X _ _ _ _ _ _ _ _ _ _ X X _ ,
s X X _ X X X X X _ _ _ _ X X _ ,
s X X _ _ _ _ _ X _ _ _ _ X X _ ,
s X X _ _ _ _ _ X _ _ _ _ X X _ ,
s _ X X _ _ _ _ X _ _ _ _ X X _ _ ,
s _ X X _ _ _ _ X _ _ _ _ X X _ _ ,
s _ _ X X X _ _ _ _ _ X X X _ _ _ ,
s _ _ _ X X X X X X X X _ _ _ _ ,
s _ _ _ _ _ X X X X X _ _ _ _ _ _ ,
s _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

};
```

Bludiště knihovních typů

- limits.h:
  - `CHAR_BIT`
  - `CHAR_MAX`, `SHRT_MAX`, `INT_MAX`, `UINT_MAX`, `LONG_MAX`, `...`
- float.h: podobně pro floatové typy
- stddef.h:
  - `ptrdiff_t`, `size_t`, `wchar_t`, `NULL`, `offsetof()`
- stdbool.h: `bool` (namísto `_Bool`)
- stdint.h:
  - `(u)intN_t`, `(u)int_leastN_t`, `(u)int_fastN_t`
  - `(u)intptr_t`, `(u)intmax_t`
- inttypes.h:
  - `PRIfN`, `PRIfMAX`, `SCNfN`, `...` (nebo `%jd`, `%zd`)
  - `strtoimax()`, `strtoumax()`
- signal.h: `sig_atomic_t`
- uchar.h: `char16_t`, `char32_t`

setjmp()

“Goto do dřívějšího bodu běhu programu”, např. k ošetření výjimek:

```
#include <setjmp.h>
jmp_buf exception;

int main(void) {
    /* Try */ if (!setjmp(exception)) {
        program();
    } else {  /* Catch */
        printf("exception\n");
    }
}

void program(void) {
    /* Throw */ longjmp(exception, 1);
}
```

GNU rozšíření: Syntaxe

- Vnořené funkce
- a `?:` b
- `0b10001010`
- `case 1...5:`
- Anonymní prvky struktur (v C11 jsou)
- `__auto_type` *proměnná* = *hodnota* (GCC 4.9)

GNU rozšíření: Příkazové výrazy

```
#define MIN(a,b) ({ \
  typeof(a) _a=(a), _b=(b); _a < _b ? _a : _b; })

#define stk_strcat(a,b) ({ \
  char *_a=(a), *_b=(b); \
  int la=strlen(_a), lb=strlen(_b); \
  char *z = alloca(la + lb + 1); \
  memcpy(z, _a, la); \
  memcpy(z+la, _b, lb+1); \
  z; })
```

GNU rozšíření: Typy

- `__int128` – 128-bitový celočíselný typ (nemá literály)
- `__float128` – 128-bitový floatový typ (nemá literály)
- Fixed-point typy podle draftu N1169:
  - `_Fract` (rozsah $[-1, 1]$)
  - `_Accum` (alespoň 4 bity celé části)
  - `unsigned _Fract`, `short _Fract`, ...
  - modifikátor `_Sat` (saturace)
  - literály `12.3r`
  - printf: `%r` (`_Fract`), `%R` (`unsigned _Fract`), `%k` (`_Accum`)
  - `<stdfix.h>`
- Decimální floaty podle draftu N1312:
  - `_Decimal32`, `_Decimal64`, `_Decimal128`
  - literály `12.3d`, `12.3dd`, `12.3dl`
  - printf: `%Hf`, `%Df`, `%DDf`
  - `<float.h>`

GNU rozšíření: Inline assembler

```
static inline void cpuid(int op, int *a, int *b,
                                 int *c, int *d)
{
  asm("cpuid"
    : "=a" (*a), "=b" (*b), "=c" (*c), "=d" (*d)
    : "0" (op));
}

static inline uns bit_ffs(uns w)
{
  asm("bsfl %1,%0" : "=r" (w) : "rm" (w));
  return w;
}

#define wmb() asm volatile ("": : :"memory")
```

GNU rozšíření: Atributy funkcí, typů a proměnných

Příklad použití:

```
double sin __attribute__((const)) (double f);
```

Zajímavé atributy:

- Konstruktory: `constructor`
- Pro optimalizaci: `const`, `pure`, `malloc`, `hot`, `cold`, `optimize`, `noreturn` [`_Noreturn`]
- Inlinování: `noinline`, `always_inline`, `flatten`
- Proměnné a typy: `aligned`, `packed`
- Varování: `unused`, `format`, `warn_unused_result`, `warning`, `error`

GNU rozšíření: Zabudované funkce

- `__builtin_constant_p(value)`
- `__builtin_types_compatible_p(type1, type2)`
- `__builtin_choose_expr(condition, if-true, if-false)`
- `__builtin_expect(value, expectation)`
- `__builtin_unreachable()`
- `__builtin_prefetch(address, ...)`
- `__builtin_add_overflow(x, y, &result)`
- `__builtin_ffs(integer)`
- `__builtin_bswap(integer)`
- `__builtin_popcount(integer)` a další strojové instrukce

Viz též `_Generic(expr, type:value, ..., default:value)`.

Konec