Programování v jazyce C

3 Datové typy, deklarace a definice

- Přehled základních datových typů
- Deklarace proměnných
- Definice konstant
- Definice uživatelských datových typů
- Definice (a deklarace) funkcí
- Ukazatele a pole

Deklarace proměnných
Spojení identifikátoru s objektem

Deklarace = přiřazení jména (identifikátoru) a typu nějakému objektu jazyka, nejčastěji proměnné, funkci, konstantě, apod.

- V jazycích se **statickým typovým systémem** je třeba objekt nejprve deklarovat – předtím, než je v programu použit.
- Deklarací je překladač informován o vzniku příslušného objektu v programu – je-li to zapotřebí, **alokuje** pro takový objekt **paměť**, např. pro proměnnou.

- V ANSI C musí být deklarace **vždy pouze na začátku bloku!** (deklarace kdekoliv uvnitř bloku dovoluje až norma ISO/IEC 9899:1999, tj. tzv. C99)
- Deklarace proměnných **nesmí být v hlavičkových souborech** (kromě tzv. `extern` deklarací, kvůli duplicitě – viz dále).

Deklarace proměnných
Komentovaná ukázka

```
int a, b = 5;

int main(int argc, char *argv[]) {
    int a = 3, b = 5;
    float c, d;

    c = d = multiply(a, b) / 2.0;

    return 0;
}

int multiply(int x, int y) {
    return x * y;
}
```

globální proměnné

lokální proměnné (zakrývají globální proměnné, platnost pouze uvnitř bloku)

Jednoduché datové typy
Celočíselné datové typy

`[signed] short [int]`	-32768 .. 32767
`[signed] int` `signed`	v moderních překladačích ANSI C obvykle -2^{31} .. $2^{31}-1$
`[signed] long [int]`	-2^{31} .. $2^{31}-1$ nebo -2^{63} .. $2^{63}-1$
`unsigned short [int]`	0 .. 65535
`unsigned [int]`	obvykle 0 .. $2^{32}-1$
`unsigned long [int]`	0 .. $2^{32}-1$ nebo 0 .. $2^{64}-1$
`[signed | unsigned] char`	-128 .. 127 | 0 .. 255

Jednoduché datové typy
Celočíselné datové typy

Tabulka 5-1: **Hodnoty definované v limits.h (ANSI C)**

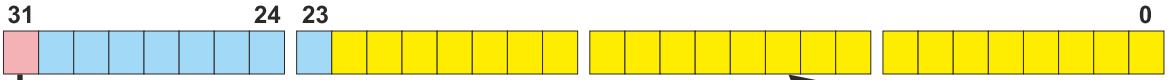
Jméno	Minimální hodnota	Význam
CHAR_BIT	8	šířka typu char, v bitech
SCHAR_MIN	-127	minimální hodnota pro signed char
SCHAR_MAX	127	maximální hodnota pro signed char
UCHAR_MAX	255	maximální hodnota pro unsigned char
SHRT_MIN	-32,767	minimální hodnota pro short int
SHRT_MAX	32,767	maximální hodnota pro short int
USHRT_MAX	65,535	maximální hodnota pro unsigned short
INT_MIN	-32,767	minimální hodnota pro int
INT_MAX	32,767	maximální hodnota pro int
UINT_MAX	65,535	maximální hodnota pro unsigned int
LONG_MIN	-2,147483647	minimální hodnota pro long int
LONG_MAX	2,147483647	maximální hodnota pro long int
ULONG_MAX	4,294967295	maximální hodnota pro unsigned long
CHAR_MIN	SCHAR_MIN nebo 0❶	minimální hodnota pro char
CHAR_MAX	SCHAR_MAX nebo UCHAR_MAX❷	maximální hodnota pro char

implementace se od textu normy obvykle liší (např. v GCC 4.8 jsou v limits.h **`int`**-y 32-bitové a **`long`**-y 64-bitové) ⇒ **používat `sizeof`()**

Jednoduché datové typy
Reálné číselné datové typy

`float`	cca. $\pm 3{,}403 \times 10^{38}$
`double`	cca. $\pm 1{,}798 \times 10^{308}$
`long double`	obvykle jako **double** (někdy 80 bitů)

IEEE Standard for Binary-Point Arithmetic No. 754-1985



***exp*onent** (kód s posunutou 0; 0 ~ 127)

znaménko *s*
(0 = +, 1 = -)

normalizo-vaná *man*tisa

$$h_{float} = (-1)^{s} \times 2^{(exp\ -\ 127)} \times (1 + man)$$

Jednoduché datové typy
Ukazatel (*Pointer*)

- 32-bitové (nebo 48- či 64-bitové) celé číslo s významem adresy v paměti ⇒ lze s ním provádět aritmetické operace (sčítání, odčítání), tzv. ***ukazatelovou aritmetiku***

- obvykle „ukazuje” na nějaký objekt v paměti, typicky např. na proměnnou, tj. adresa této proměnné v paměti je uložena v ukazateli – je-li to proměnná typu **`int`**, hovoříme o **ukazateli na int**

- ukazatel na něco se vyrobí pomocí znaku **`*`**, např. **`int *p;`**

- tzv. ***generický ukazatel*** může ukazovat na jakýkoliv objekt, tzn. datový typ objektu, jehož adresu ukazatel obsahuje, není explicitně určen

Jednoduché datové typy
Ukazatel (*Pointer*)

```
int *i;
float *hodnota;
char *zn;

float (*vypocet) ();

void *uk;
```

ukazatel na celé číslo

ukazatel na reálné číslo

ukazatel na znak

ukazatel na funkci, která vrací reálné číslo

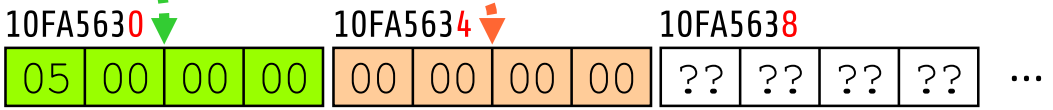
generický ukazatel (může ukazovat na cokoli)

klíčové slovo **`void`** označuje prázdný typ, má význam „nic”

Poznámka: Proměnnou typu **`void`** pochopitelně nelze deklarovat (neexistovala by, nelze vytvořit v paměti „nic”), ovšem lze tento typ použít např. jako návratovou hodnotu funkce nebo ve spojení s ukazatelem.

Jednoduché datové typy
Ukazatel (*Pointer*) – situace v paměti

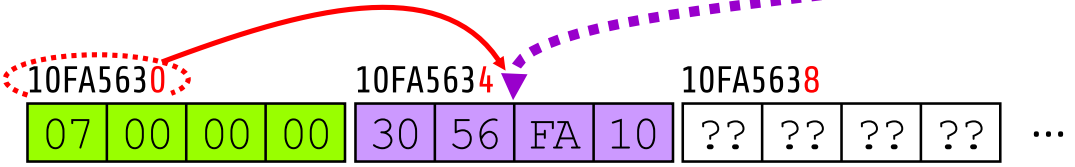
```
int num = 5;
int *num_ptr = 0;
```



dereferenční operátor (užívá adresu pro přístup k objektu)

referenční operátor (získává adresu objektu)

```
num_ptr = &num;
*num_ptr = 7;
```



- ukazatel `num_ptr` nyní ukazuje na proměnnou `num`

Pole
Jedno- a vícerozměrná pole

- pole **nelze** vytvořit z typu **void** a typu funkce (ale z typu ukazatel na funkci samozřejmě jde)
- pole je **vždy indexované od nuly**

```
int i[3];
char zn[20];
int *uk[3];
float matice[5][5];
```

pole 3 intů
pole 20 znaků
pole 3 ukazatelů na **int**-y
pole 5x5 floatů (matice)

```
int i[20], j;
int *ip[20];

for (j = 0; j < 20; j++)
   ip[j] = &i[j];
```

Pole
Vztah datového typu pole a ukazatel

- vztah je těsný – často lze pole a ukazatel zaměnit
- proměnná typu pole je vlastně ukazatel na první prvek oblasti, která je v paměti pro toto pole alokovaná

```
int i[20], *ip;

ip = i;
ip = &i[0];
```

} ekvivalentní operace

- pole a ukazatel jsou vázané ukazatelovou aritmetikou:
  `a[i]` ⇔ `*((a) + (i))`

- **indexování pole** = přičtení výrazu (index × velikost bázového typu pole) k bázi, tj. k ukazateli uloženému v proměnné typu pole

Výčtový typ
Množina identifikátory daných hodnot

- kvůli zjednodušení zápisu, pro pohodlnost programátora
- množina celočíselných hodnot reprezentovaných identifikátory (tzv. **výčtové konstanty**)

```
enum ryby {kapr, uhor, stika, pstruh};
enum zvirata {prase, krava, koza,
   kocka, kur} v_chlivku, v_kurniku;

void main(void) {
   enum ryby k_veceri;
   k_veceri = stika;
}
```

při definici výčtového typu lze rovnou deklarovat proměnné daného typu

- překladač přiřadí každé výčtové konstantě hodnotu (první zleva dostane hodnotu 0)
- můžeme to také udělat po svém

Výčtový typ
Vlastní určení hodnot výčtových konstant

```
enum ryby {kapr = 1, uhor = 2, stika = 3,
  pstruh = 4} chytil_jsem;
```

- Neurčí-li hodnoty všech výčtových konstant programátor, dodělá to překladač – to ale může vést k problémům...

0 1 4 (!!!)

```
enum ryby {kapr, uhor, stika = 3, lin}
  chytil_jsem;
```

- Je možné přiřadit dvěma různým výčtovým konstantám stejnou hodnotu, pak ale **pozor**!

```
enum ryby {kapr = 1, uhor = 1, stika = 2,
  pstruh = 3} chytil_jsem;
```

tato zjevně nesmyslná podmínka je splněna

```
if (kapr == uhor) { ... }
```

Datový typ struktura
Mechanismus definice nových datových typů

```
struct complex {
    double real;
    double imag;
};

struct complex a, b;
struct complex c = {1, -0.5};

a.real = 5.0;
a.imag = -2.0;
```

zde nemá smysl inicializovat, jedná se o abstraktní definici
(tj. v paměti ještě nic neexistuje)

- V paměti může struktura zabírat **více místa**, než je součet velikostí typů jednotlivých komponent (z důvodů tzv. **zarovnávání**, viz později příkaz preprocesoru **`#pragma align`**).

Funkce pracující s typem struktura
Příklad předání struktury podprogramu

- Struktura může být parametrem funkce a může být funkcí vrácena.

argumenty se předávají v zásobníku ⇒ hodně přesunů dat mezi pamětí a zásobníkem

```
struct complex {
    double re;
    double im;
};

struct complex mul(struct complex a,
                   struct complex b) {
    struct complex prod;

    prod.re = (a.re * b.re - a.im * b.im);
    prod.im = (a.re * b.im + a.im * b.re);

    return prod;
}
```

Funkce pracující s typem struktura
Příklad předání struktury podprogramu

- Z důvodu omezení množství dat na zásobníku je praktičtější (zejm. u větších struktur) předávat ukazatel.

```
struct complex {
    double re;
    double im;
};

struct complex mul(struct complex *a,
                   struct complex *b) {
    struct complex prod;

    prod.re = (a->re * b->re - a->im * b->im);
    prod.im = (a->re * b->im + a->im * b->re);

    return prod;
}
```

Ukazatel na typ struktura
Speciální případy definice struktury

- Někdy je třeba provést tzv. **neúplnou definici** – odkaz na ještě nedodefinovanou strukturu je uvnitř její definice.

```
struct tree_node {
    struct tree_node *left_son;
    struct tree_node *right_son;
    int data;
};
```

definice struktury je úplná až zde

tzn. zde se odkazujeme na zatím nedodefinovanou strukturu

- Překladač samozřejmě nemůže dovolit, aby definovaná struktura byla komponentou sebe sama (nevěděl by kolik místa pro takovou komponentu alokovat), ovšem velikost ukazatele je známá vždy (nezáleží na velikosti struktury).

Ukazatel na typ struktura
Speciální případy definice struktury

- Lze zajít ještě dál a provést definici dvou vzájemně se na sebe odkazujících struktur.

```
struct cell;

struct header {
    struct cell *first;
    ...
};

struct cell {
    struct header *head;
    ...
};
```

nutno uvést tuto neúplnou definici před použitím zde

Struktura jako bitové pole
Teoretická možnost úspory místa

- Teoreticky lze „ušetřit” místo tím, že pro celočíselné složky, které nezabírají rozsah celého datového typu, vyhradíme jen tolik bitů, kolik opravdu potřebují.

```
struct planner {
    unsigned day_of_week:3;
    unsigned day:5, month:4;
    ...
};
```

- Výslednou velikost ovšem ovlivní zarovnávání, tzv. nemusí se ušetřit tolik místa, kolik očekáváme.
- **Užívaná technika pro vazbu na hardware či vnitřní struktury operačního systému.**

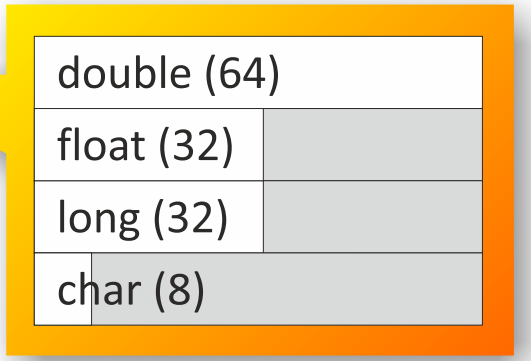
Union
Elegantní řešení přístupu do paměti

- definuje se podobně jako **`struct`**

```
union value {
    double d;
    float f;
    unsigned long int i;
    char c;
};

union value v;

v.f = 3.25;
printf("%X\n", v.i);
```



- union může v daném čase obsahovat pouze jednu hodnotu
- velikost unionu v paměti je dána jeho největší složkou

Datový typ funkce
Deklarace proměnné typu funkce

- Jazyk C dovoluje deklarovat **proměnnou typu funkce**, je to vlastně **ukazatel**, v němž je adresa vstupního bodu funkce.

```
float add(float a, float b) {
  return a + b;
}

float mul(float a, float b) {
  return a * b;
}

void main() {
  float (*func) (float, float);
  float a = 5, b = 5;

  func = add;
  printf("%f + %f = %f\n", a, b, func(a, b));

  func = mul;
  printf("%f * %f = %f\n", a, b, func(a, b));
}
```

deklarace proměnné **func**, což je funkce 2 **float** parametrů, vracející **float**

do proměnné typu funkce přiřadíme adresu existující funkce

Datový typ funkce
Skutečně nepříjemné podoby deklarace

- inicializované pole ukazatelů na funkci:

```
void * (*funcs[]) (int, int *[]) = { NULL, … };
  ⋮
int *nums[] = { &var1, &var2, &var3, … };
void *myfunc(int n, int *a[]) { … }
  ⋮
funcs[0] = myfunc;
funcs[0](0, nums);
```

volání funkce (oindexováním pole ukazatelů na funkce)

uložení adresy existující (v ZK definované) funkce do pole ukazatelů na pozici 0

Datový typ funkce
Skutečně nepříjemné podoby deklarace

- neinicializované (dynamické) pole ukazatelů na funkci:

```
void * (**funcs) (int, int *[]);
  ⋮
int *nums[] = { &var1, &var2, &var3, … };
void *myfunc(int n, int *a[]) { … }
  ⋮
funcs = malloc(10 * sizeof(void *));
funcs[0] = myfunc;
funcs[0](0, nums);
  ⋮
free(funcs);
```

- pole (latentní) je deklarované jako ukazatel na počátek bloku ukazatelů na funkci
- skutečně je vytvořeno až alokací paměti (funkcí `malloc(·)`), jinak funguje stejně jako v předch. případě

Příkaz typedef
Pojmenování definovaného datového typu

```
typedef int cele_cislo;

typedef struct { double re;
  double im; } complex;

typedef struct thenode {
    int key;
    struct thenode *left_son;
    struct thenode *right_son;
} node;

cele_cislo i = 5;
node uzel;
complex z;
```

pojmenování nového typu

definice nového datového typu

Definice konstant
Pomocí preprocesoru nebo kvalifikátoru typu

```
#define PI 3.14159

const float EUL = 2.71;

float compute(float i) {
  float a = EUL * i;
  return 2 * PI * a;
}
```

zde nesmí být nikdy středník

kvalifikátor typu

- konstanty definované pomocí příkazu preprocesoru
  (i) **šetří místo** (není to proměnná)
  (ii) **způsobují problémy při ladění** programu (překladač nezná jejich názvy, detaily o nich nejsou uloženy v ladicích informacích ⇒ debugger je nevidí)

Definice konstant
Technika *Casting away the const-ness*

```
#include <stdio.h>

const float PI = 3.14;

void main() {
  *((float *) &PI) = 3.14159;
  printf("%f\n", PI);
}
```

casting away the const-ness

- U některých překladačů tento trik nefunguje, protože umisťují konstanty do samostatného segmentu, který má nastavená práva pouze pro čtení.

Explicitní určení datového typu konstanty

Celočíselné konstanty

desítková

osmičková

šestnáctková

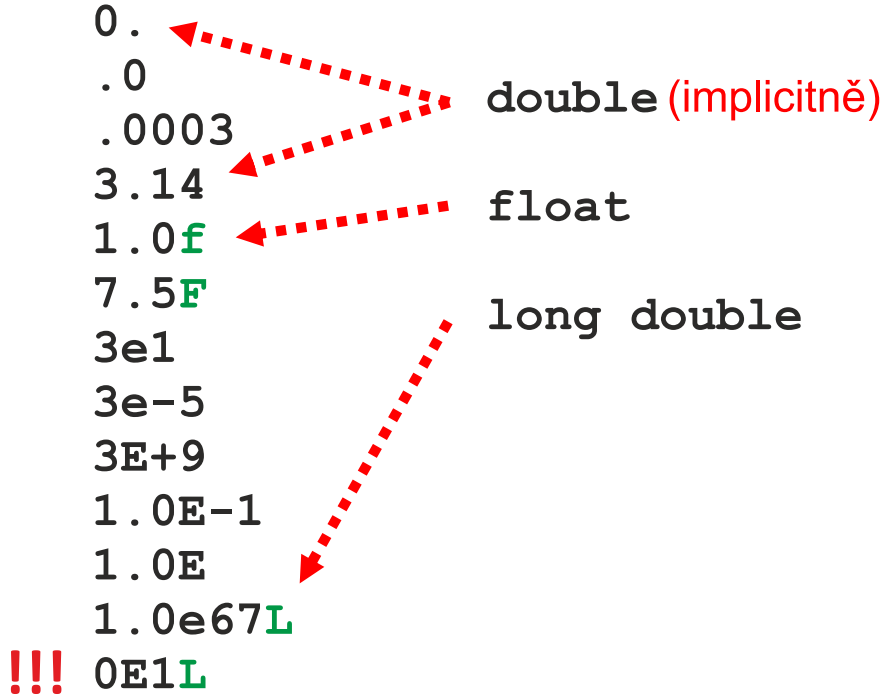
```
12345678     int
01777777     unsigned int
0X8FDD2A     long int
0x8FDD2A     unsigned long int

12345678U    unsigned int
01777777U    unsigned long int
0x8FDD2AU

12345678L    long int
01777777L    unsigned long int
0x8FDD2AL

12345678UL   unsigned long int
01777777UL
0x8FDD2AUL
```

Explicitní určení datového typu konstanty
Reálné konstanty



- Lze použít i l (malé L), ale snadno se splete s 1, takže raději **nepoužívat**.

Explicitní určení datového typu konstanty
Příklady použití

- Explicitní určení typu konstanty se používá tam, kde není ze zápisu zjevné, jakého typu má konstanta být nebo je-li třeba vynutit typovou konverzi.

```
x = y * 10L;
...
zoom(2.0F);
...
if (x > 1U) ...
```

- Obvykle tehdy, když je konstanta zapsaná přímo v kódu ⇒ důsledný boj proti „magickým číslům" činí tuto techniku zbytečnou.

vynucení konverze na neznaménkový typ

- Vynucení typové konverze může mít někdy zásadní význam (např. tehdy, když vinou implicitní typové konverze dojde k přetečení, apod.).

Definice funkcí
Základní podoba definice funkce

návratový typ **lokální proměnné** **argumenty** (parametry)

```
int addition(int a, int b) {
    int c;

    c = a + b;

    return c;
}
```

```
013D1000: push ebp
013D1001: mov  ebp, esp
013D1003: push ecx
013D1004: mov  eax, dword ptr 8[ebp]
013D1007: add  eax, dword ptr 12[ebp]
013D100A: mov  dword ptr -4[ebp], eax
013D100D: mov  eax, dword ptr -4[ebp]
013D1010: mov  esp, ebp
013D1012: pop  ebp
013D1013: ret  0
```

- Vnořené funkce nejsou dovoleny.
- Rekurze je dovolena (a syntakticky se nespecifikuje).
- **Funkce nesmí vracet typ pole a typ funkce!** (ale `(*f)` ano)

Definice funkcí
Proč nemůže funkce vracet typ pole?

- Vnitřní reprezentací pole je pouze bázový ukazatel, tj. adresa bloku paměti, ve kterém je pole uloženo → **neznáme jeho délku!**

- Vše, co funkce vrací, se předává buď ve všeobecném stradači CPU (x86: **EAX**, x86-64: **RAX**) nebo v zásobníku. Když není známá délka pole, neví proces volající funkci, kolik položek ze zásobníku vybrat.
- Ani vlastně není jak to syntakticky korektně zapsat:

```
func03.c:4:5: error: expected identifier or '(' before '[' token
    4 | int [] return_array(int len, int val) {
      |     ^
```

```
int [] return_array(int len, int val) { … }

int * return_array(int len, int val) { … }
```

tohle znamená něco jiného...

Definice funkcí
Jak na to? Může funkce „vyrobit” pole?

- Vždy to jde **pomocí ukazatele** (tzv. *dynamické pole*, viz dále)
- Nebo pomocí klíčového slova **`static`**:

```
#include <stdio.h>
#define ARRAY_LEN 20

int * make_array(int val) {
  static int arr[ARRAY_LEN], i;

  for (i = 0; i < ARRAY_LEN; i++) arr[i] = val;
  return arr;
}

void main() {
  int i, *arr;

  arr = make_array(5);
  for (i = 0; i < ARRAY_LEN; i++)
    printf("%d ", arr[i]);
}
```

překladač vytvoří pole jako globální, trvale platnou proměnnou **v heapu**

vnitřní reprezentací pole je ukazatel, tzn. funkce vrací ukazatel na int, a ten také musí uložit volající proces

Definice funkcí
Předávání argumentů v zásobníku

- Překladač **uloží** před voláním funkce její **argumenty do zásobníku**. Pokud to programátor vynutí, může jít o velký objem dat (→ vyčerpání zásobníku):

```
#include <stdio.h>
#define ARR_LEN 100

typedef struct {
  int items[ARR_LEN];
} array;

array sum(array a, array b) {
  int i; array c;

  for (i = 0; i < ARR_LEN; i++)
    c.items[i] =
      a.items[i] +
      b.items[i];

  return c;
}
```

```
int main() {
  int i; array a1, a2, t;

  t = sum(a1, a2);

  for (i = 0; i < 20; i++)
    printf("%d ", t.items[i]);
}
```

kopíruje se celá struktura (zde to nejde jinak — vnitřní reprezentací struktury je celý blok dat této struktury o známé velikosti)

Definice funkcí
Předávání argumentů v zásobníku

- → **Při praktickém programování v C** předávají funkcím kromě primitivních dat. typů **výhradně bázové ukazatele**.
- Ukazatel má vždy pevně danou velikost (32/64 bitů), lze předat snadno i v registru (urychlení volací konvencí **fastcall**, viz dále) a nedochází k tak rychlému vyčerpávání zásobníku...
- Uvedený příklad ilustruje, že „velké" objekty funkci předat lze v zásobníku, ale **nedělá se to tak!**

```
array sum(array a, array b) {
  int i; array c;

  for (i = 0; i < ARR_LEN; i++)
    c.items[i] =
      a.items[i] +
      b.items[i];

  return c;
}
```

```
$LN3@sum:
; Line 30
mov ecx, 100
lea esi, DWORD PTR _c$[ebp]
mov edi, DWORD PTR $T1[ebp]
rep movsd
mov eax, DWORD PTR $T1[ebp]
```

zbytečné kopírování dat také zpomaluje běh prg

Používání funkcí
Volání funkce z programu (tj. z jiné funkce)

```
int addition(int a, int b) {
    int c;

    c = a + b;

    return c;
}

void main() {
   int d = addition(1, 2);
   printf("%d\n", d);
}
```

parametry se funkcím předávají prostřednictvím zásobníku, konkrétní způsob se řídí tzv. ***volací konvencí***

cdecl

```
013D1021: …
013D1024: push 2
013D1026: push 1
013D1028: call 013D1000 ;_addition
013D102D: add  esp, 8
013D1030: mov  dword ptr -4[ebp], eax
013D1033: …
```

Používání funkcí
Volací konvence (*Calling Convention*) jazyka C

`__cdecl` – **implicitní volací konvence v jazyce C.** Parametry se předávají v zásobníku, ukládají se zprava doleva, tj. první parametr je na vrcholu zásobníku. Uložené parametry ze zásobníku odstraňuje volající proces. Jménu funkce je předřazeno podtržítko. Návratová hodnota se předává v registru EAX.

```
int func(int a, int b, int c) { … }
…
int r, x, y, z;
…
r = func(x, y, z);
```

```
push z
push y
push x
call _func
add  esp, 12
mov  r, eax
```

```
pop  ?
pop  ?
pop  ?
```

Používání funkcí
Volací konvence (*Calling Convention*) jazyka C

`__stdcall` – Win32 API, parametry v zásobníku, ukládají se zprava doleva, zásobník čistí volaný proces. Jména funkcí jsou doplněna podtržítkem na začátku a @ na konci, následuje velikost parametrů v bytech.

`__fastcall` – první dva 32-bitové parametry v registrech ECX a EDX, další v zásobníku, uložené zprava doleva. Zásobník čistí volaný proces. Jména funkcí jsou doplněna @ na začátku i na konci, následuje velikost parametrů v bytech.

`__thiscall` – pouze v C++, jako **cdecl**, ale v registru ECX se předává ukazatel na volající instanci třídy (`this`).

pascal – opak **cdecl**, parametry zleva doprava, zásobník čistí volaný proces.

Používání funkcí
Volací konvence – použití

- Volací konvenci lze změnit, ovšem každý překladač používá jiný syntaktický aparát (toto nedefinuje norma)...

GCC (UNIX/Linux)

```
int __attribute__((__cdecl__)) fnc(...) {
    int c;
    ...
    return c;
}
```

MSVC, GCC (Win32)

```
int __cdecl fnc(...) {
    int c;
    ...
    return c;
}
```

Používání funkcí
Volací konvence – smysl a účel

- Vynucení použití konkrétní volací konvence je zejména důležité při volání funkcí z knihoven, které nejsou součástí ekosystému překladače.
- Poskytuje-li např. nějaká knihovna, kterou využívá náš projekt, funkci **`get_max`**`(...)` s mechanismem přenosu parametrů **fastcall**, je nutné to „vysvětlit” překladači:

```
#include <stdio.h>

extern int __fastcall get_max(int, int);

int main() {
  printf("%d", get_max(-5, 5));
  return 0;
}
```