PB111 Nízkoúrovňové programování (shrnutí jazyka C)

P. Ročkai

Část 1: Výpočetní stroj

Tato kapitola jazyk C nepoužívá.

Část 2: Lokální proměnné, řízení toku

Počínaje touto kapitolou budeme většinu programů psát ve zjednodušené verzi jazyka C. V tomto kurzu budeme psát programy do jednoho souboru, který bude sestávat z definic typů (uvidíme později) a podprogramů. Na diskusi o sémantice podprogramů zatím nejsme připraveni, proto je budeme chápat jako syntaktickou obálku pro kód, který budeme psát.

Program bude typicky vypadat takto:

```
int podprogram( int parametr1, int parametr2 )
{
    …
}

int main()
{
    assert( podprogram( 1, 2 ) == 3 );
    …
}
```

Podprogram s názvem `main` bude v tomto kurzu vždy obsahovat testy, které ověřují základní funkcionalitu ostatních podprogramů. Můžete si do něj vždy přidat svoje vlastní testy. Zápis `podprogram( 1, 2 )` je volání (použití) podprogramu – prozatím jej nebudeme mimo testy potřebovat, protože jediné podprogramy, které budeme moct v tomto předmětu použít, jsou ty, které si sami napíšeme.

1 Hodnoty, objekty a proměnné Proměnné znáte již z kurzu IB111 – proměnné v jazyce C mají s těmi v Pythonu mnoho společného, ale mají také důležité odlišnosti. Prvním, v zásadě syntaktickým, rozdílem je, že v jazyce C musíme každou proměnnou **deklarovat** – to provedeme zápisem `typ jméno;` případně `typ jméno = výraz;`. První forma proměnnou pouze deklaruje, ale její počáteční hodnotu ponechá neurčenu – tuto hodnotu **není dovoleno použít**.

Typ proměnné určuje, jakých hodnot může nabývat – k dispozici máme prozatím tyto zabudované typy:

- `unsigned` – celé číslo v rozsahu 0 až 65535,[1]
- `int` – celé číslo v rozsahu -32768 do 32767,[2]
- `bool` – celé číslo, 0 nebo 1, které typicky reprezentuje pravdivostní hodnotu – 0 pro `false`, 1 pro `true`,
- `signed char` – celé číslo v rozsahu -128 až 127,
- `unsigned char` – celé číslo v rozsahu 0 až 255,
- `char` – typ se stejným rozsahem jako jeden z předchozích dvou (který z nich je určeno implementací), ale přesto z pohledu kontroly typů od obou odlišný.

Proměnná je v jazyce C pevně svázaná[3] s **objektem**. Objekt je **abstrakce paměti** – reprezentuje entitu, která je schopna pamatovat si **hodnotu**, již můžeme z objektu **přečíst** nebo do objektu **uložit** novou (a tím tu předchozí přepsat). Objekt tak můžeme chápat jako dvojí zobecnění paměťové buňky:

- místo jednoho bajtu si pamatuje **hodnotu** (která může mít potenciálně složitou vnitřní strukturu, i když takové zatím neumíme v jazyce C sestrojit),
- místo adresy má **identitu** – objekt můžeme „uchopit“ a pracovat s ním – obvykle tak, že tento objekt svážeme s proměnnou.

Realizace objektů je důležitým prvkem implementace programovacího jazyka a může se případ od případu lišit. Zejména není pravda, že by byl objekt pevně svázán s nějakou adresou nebo registrem – překladač může objekt transparentně přesouvat dle potřeby výpočtu.[4]

2 Život a rozsah platnosti Objekt, který je s proměnnou svázaný, vznikne právě deklarací, a zanikne opuštěním rozsahu platnosti této proměnné. Čtení objektu je implicitní – provede se kdykoliv proměnnou použijeme jako hodnotu ve výrazu, zápis do objektu pak provedeme operátorem přiřazení (viz také další sekce).

Podobně jméno proměnné je platné počínaje deklarací, a konče pravou složenou závorkou, která ukončuje nejbližší uzavírající blok (složený příkaz nebo tělo funkce – podrobněji rozebereme dále). Například:

```
{
    // zde x ještě není deklarováno
    int x;
    {
        int y;
        … // zde můžeme použít jak x tak y
    } // zde končí rozsah platnosti y
    … // zde již y není lze použít
} // zde končí rozsah platnosti x
```

U proměnných je tak syntakticky zaručeno, že jsou svázány s živým objektem – kdykoliv můžeme jméno proměnné použít, objekt, který tato proměnná pojmenovává, existuje.

3 Výrazy Na úrovni jazyka C je základní jednotkou výpočtu **výraz** – podobně jako v jazyce Python můžeme výrazy tvořit induktivně. Jsou-li:

- e_1, e_2 … e_n výrazy,
- `var` jméno proměnné,
- `lit` číselný literál (konstanta),

existují také výrazy tvaru:[5]

1. `lit` (konstanta) je výraz,
2. `var` (jméno proměnné) je výraz,
3. použití aritmetického operátoru (binární v infixovém zápisu, unární v prefixovém):
   - e_1 `+` e_2, e_1 `-` e_2,
   - e_1 `*` e_2, e_1 `/` e_2, e_1 `%` e_2 (modulo)
   - unární mínus `-`e_1,
4. relační operátory:
   - e_1 `==` e_2 (rovnost), e_1 `!=` e_2 (nerovnost)
   - e_1 `<=` e_2, e_1 `>=` e_2, e_1 `<` e_2, e_1 `>` e_2
5. bitové logické operace a posuvy:
   - binární e_1 `&` e_2 (and), e_1 `|` e_2 (or), e_1 `^` e_2 (xor),
   - unární `~`e_1 – bitová negace,
   - bitové posuvy zapisujeme e_1 `>>` e_2, e_1 `<<` e_2,
6. **operátory** přiřazení (pozor na změnu oproti jazyku Python – v jazyce C je přiřazení výraz, nikoliv příkaz):
   - jednoduché `var =` e_1,
   - složené `var +=` e_1, `var -=` e_1,
   - dále `var *=` e_1, `var /=` e_1, `var %=` e_1,
   - s bitovým posuvem `var <<=` e_1, `var >>=` e_2,
   - s bitovou operací `var &=` e_1, `var ^=` e_1, `var |=` e_1,
7. operátory zvýšení a snížení proměnné o jedničku:
   - prefixové `++var`, `--var`,
   - postfixové `var++`, `var--`,
8. operátor čárka, e_1`,` e_2,
9. booleovské logické operace:
   - binární e_1 `&&` e_2 (and), e_1 `||` e_2 (or),
   - unární `!`e_1,
   - ternární e_1 `?` e_2 `:` e_3,

[1] Pro typy `int` a `unsigned` je konkrétní rozsah přípustných hodnot daný implementací – na mnoha systémech jsou tyto typy 32bitové.

[2] Starší standardy jazyka C neurčují, jaké kódování se použije pro znaménkové typy, novější již požadují dvojkový doplňkový kód (viz také předchozí kapitola).

[3] Na rozdíl od jazyka Python, kde je možné vazbu proměnné na objekt změnit přiřazením. To v jazyce C možné není.

[4] Překladače jazyka C například běžně přesouvají objekty mezi registry a zásobníkem podle aktuální situace. Tentýž objekt může být tedy v různých fázích výpočtu fyzicky uložen na různých místech.

[5] S dalšími operátory se setkáme v pozdějších kapitolách.

4 Vyhodnocení výrazu

Nyní víme, jak výrazy vypadají (jakou mají **syntaxi**), můžeme tedy přistoupit k otázce, co takové výrazy **znamenají** (jakou mají **sémantiku**). Všechny zde uvedené výrazy[6] popisují nějakou **hodnotu** a výraz samotný je návodem, jak tuto hodnotu získat.

Vyhodnocení výrazu (provedení výpočtu tímto výrazem popsaného) budeme samozřejmě realizovat pomocí již zavedeného výpočetního stroje tiny. Abychom mohli výpočet skutečně provést, musíme určit registr, do kterého má být výsledek zapsán – budeme mluvit o **vyhodnocení výrazu E do registru R**.[7]

1. Výraz lit se vyhodnotí přímo na číselnou hodnotu zapsanou ve zdrojovém kódu. Například vyhodnocení výrazu 7 do registru rv se realizuje instrukcí put 7 → rv.
2. Výraz var se vyhodnotí na hodnotu, která je v momentě vyhodnocení tohoto výrazu uložena v objektu svázaném s proměnnou var. Prozatím uvažujeme pouze situace, kdy je objekt svázaný s var uložen přímo v registru. Je-li např. var uloženo v l1, vyhodnocení výrazu var do registru rv realizujeme instrukcí copy l1 → rv.
3. Uvažme nyní výraz tvaru e_1 + e_2. Víme, že e_1 a e_2 popisují nějaké hodnoty. Abychom mohli vyčíslit hodnotu e_1 + e_2, budeme nejprve potřebovat tyto hodnoty. Na to použijeme **dočasné registry** – vyhodnocení e_1 + e_2 do rv bude vypadat takto:
   a. vyhodnoť e_1 do registru t_1,
   b. vyhodnoť e_2 do registru t_2,
   c. proveď add t_1, t_2 → rv.

   Musíme samozřejmě zabezpečit, že výpočet e_2 nepřepíše registr t_1 – jak přesně se toho dosáhne budeme zkoumat později.[8]

   Analogicky se vypočtou ostatní aritmetické, bitové, atd. operátory (k hodnotám s/bez znaménka a operacím dělení se ještě vrátíme).
4. Výrazy tvaru var = e_1 mají krom hodnoty také **vedlejší efekt** – zápis do objektu svázaného s proměnnou var. Jejich realizace vypadá takto – vyhodnocujeme do registru rv, objekt svázaný s var nechť žije v l1:
   a. vyhodnoť e_1 do registru rv
   b. proveď copy rv → l1.

   Všimněte si, že hodnota e_1 je zároveň hodnotou celého výrazu, a zůstává uložená v registru rv, jak bylo požadováno.
5. Složené přiřazení var += e_1 je analogické, pouze je operaci copy předřazena příslušná aritmetická nebo logická operace:[9]
   a. vyhodnoť e_1 do registru t_1
   b. proveď add t1, l1 → rv,
   c. proveď copy rv → l1.

   Výrazy zvýšení a snížení o jedničku jsou analogické, liší se pouze ve výsledné hodnotě. Prefixové verze, ++var, --var jsou pouze syntaktické zkratky pro var += 1 resp. var -= 1, ale postfixové se liší – vyhodnocení var++ do rv proběhne takto (var je svázáno s l1):
   a. proveď copy l1 → rv,
   b. proveď add l1, 1 → l1.

   Hodnota výrazu var++ je tedy **původní** hodnota var, předtím, než bylo provedeno zvýšení proměnné o jedničku.
6. Výraz e_1, e_2 představuje „zapomenutí hodnoty“ výrazu e_1 – výraz e_1 je proveden pouze pro svoje vedlejší efekty (např. výše uvedené přiřazení). Vyhodnocení e_1, e_2 do registru rv lze realizovat např. takto:
   a. vyhodnoť e_1 do rv,
   b. vyhodnoť e_2 do rv.
7. Zbývá zatím nejsložitější typ výrazů, a to jsou booleovské logické operace. XXX

Uvažme nyní několik konkrétních příkladů:

1. var + 1 se vypočte XXX

5 Příkazy

- výraz + středník
- složený příkaz
- if, else
- for
- while
- break
- continue

[6] Toto tvrzení v jazyce C neplatí obecně pro všechny výrazy – existují i takové, které hodnotu nemají.

[7] Tato konstrukce skutečně tvoří základ překladu výrazů v překladači jazyka C. Rozdílem je, že překladač pracuje s dočasnými registry mnohem hospodárněji, než naivní překlad zde popsaný – tím šetří nejen volné registry, ale i instrukce, které by hodnoty mezi registry zbytečně přesouvaly. Toto platí i pro velmi jednoduché překladače (např. také tinycc).

[8] Prozatím si vystačíme s představou, že při překladu udržujeme množinu volných dočasných registrů (takových, které jsme zatím nepoužili, nebo kterých hodnotu už jsme upotřebili, a nebudeme ji v dalším výpočtu potřebovat). Je asi jasné, že ať začneme s jakkoliv velkou konečnou množinou dočasných registrů, při výpočtu dostatečně složitého výrazu nám musí dojít – jak se s tímto problémem vypořádat si ukážeme v příští kapitole.

[9] Pro výrazy tvarů, které jsme zatím zavedli, je var += e_1 ekvivalentní výrazu var = var + e_1. V obecném případě, kdy je na levé straně složeného přiřazení složitější výraz (a nikoliv pouze název proměnné), to už ale neplatí!

Část 3: Podprogramy

Tato kapitola zavádí důležitý nový koncept, totiž **podprogram** a s ním související nový typ výrazu – volání (použití) podprogramu.

1 **Definice** Syntaxi **definice podprogramu** již zběžně známe:

```
typ₀ jméno₀( typ₁ jméno₁, … ,typₙ jménoₙ )
{
    příkaz₁
    příkaz₂
    …
    příkazₘ
}
```

Jednotlivé prvky zápisu mají tento význam:

- typ_0 je tzv. **návratový typ** – může být prozatím pouze jeden z již známých typů (int, unsigned, …),
- $jméno_0$ je název zaváděného podprogramu,
- typ_1 … typ_n jsou typy jednotlivých **parametrů**,
- $jméno_1$ … $jméno_n$ jsou **jména** parametrů,
- $příkaz_1$ … $příkaz_m$ tvoří **tělo** podprogramu.

2 **Výrazy** Hlavní nový typ výrazů je v této kapitole **použití podprogramu** nebo také **volání podprogramu** (možná znáte také jako „volání funkce“). Tento typ výrazu má následovný tvar:

```
funcall ≡ jméno( expr₁, expr₂, …, exprₙ )
```

Předpokládejme, že jméno odpovídá definici z předchozí sekce. Výraz je pak typově správný v případě, že:

- typ výrazu $expr_1$ je možné implicitně převést na typ_1,
- typ $expr_2$ na typ typ_2, atd., až $expr_n$ na typ_n.

Typ výrazu funcall jako celku je pak typ_0 z definice výše.

Vyhodnocení tohoto výrazu probíhá následovně:

1. Pro každý formální parametr je vytvořen objekt odpovídajícího typu; uvnitř těla podprogramu pak jména formálních parametrů pojmenovávají právě tyto objekty.
2. Výrazy $expr_1$ až $expr_n$ jsou vyhodnoceny na hodnoty (v blíže neurčeném pořadí!) a každá takto získaná hodnota je zapsána do příslušného objektu z předchozího bodu.
3. Řízení je předáno tělu podprogramu (vzniká při tom mimo jiné nový rozsah platnosti jmen).
4. Hodnota celého výrazu funcall je pak určena prvním spuštěným příkazem return uvnitř těla. Tento příkaz zároveň předá řízení zpátky volajícímu podprogramu.

3 **Příkazy** Nový příkaz return expr; má dva efekty:

1. vyhodnotí výraz expr a jeho hodnotu **vrátí** volajícímu (viz předchozí podsekce),
2. ukončí vykonávání podprogramu a předá řízení volajícímu.

Část 4: Adresy a ukazatele

Tato kapitola přidává ukazatele a (TODO!) operátor přetypování.

1 Definice V definici podprogramu umožníme, aby se na místě návratového typu objevilo klíčové slovo `void`, které značí, že výsledek vyhodnocení podprogramu **není hodnota**.

2 Hodnoty a typy V této kapitole přidáváme velmi důležitou novou třídu hodnot, a společně s ní příslušné typy. Je-li `T` libovolný typ, pak `T *` je typ „ukazatele na objekt typu `T`“. Hodnota typu `T *` je reprezentovaná celým číslem bez znaménka, které odpovídá **adrese**, na které je uložen příslušný objekt.[10]

Klíčové slovo `void` označuje (pseudo)typ `void`. Nejedná se o typ v běžném smyslu, protože neexistuje žádná hodnota typu `void`. Abychom odlišili „běžné“ typy (s existujícími hodnotami), zavedeme pojem **hodnotový typ**. Typ `void` je tak prvním typem, který hodnotový není.

Přesto, že nemůže existovat hodnota typu `void`, je možné napsat **výraz** tohoto typu. Takový výraz ale není možné ve většině případů použít jako podvýraz. Jediná místa, kde se **podvýraz** typu `void` objevit smí, jsou:

- libovolný operand operátoru čárka (nezávisle),
- e_1 a e_2 ve výrazu `ternary ? e1 : e2` – pak je typ výrazu `ternary` jako celku také `void`,[11]
- e_1 ve výrazu přetypování `( void ) e1`.

Je-li `expr` výraz typu `void`, může se také objevit v příkazu tvaru `expr;` (příkaz tvořený výrazem – v tomto případě se ovšem nejedná o podvýraz).

3 Výrazy Tato kapitola zavádí tři nové tvary výrazů. Abychom ale mohli tyto výrazy správně popsat, musíme nejprve upravit způsob, jakým uvažujeme o vyhodnocení výrazů a zavést několik nových pojmů.

Některé výrazy je možné **vyhodnotit na objekt** – prozatím se jedná pouze o výrazy tvaru `jméno` kde `jméno` pojmenovává nějakou proměnnou.[12] Vyhodnotí-li se nějaký výraz na objekt, další postup závisí na **kontextu** v jakém se objevuje:

1. Většina podvýrazů tvoří **r-kontexty**, tzn. takové, které očekávají hodnotu – např. operandy aritmetických nebo relačních operátorů, parametry předávané podprogramu, pravý operand operátoru přiřazení, atd. Je-li nějaký podvýraz vyhodnocen na objekt v **r-kontextu**, je z tohoto objektu navíc **přečtena hodnota** a výsledkem výrazu je až tato hodnota.
2. Některé podvýrazy jsou ale **l-kontextem** a v takovém případě se vyhodnocení zastaví určením **objektu** a čtení hodnoty ze získaného objektu se **neprovede**. Jedná se zejména o levý operand přiřazovacího operátoru (odtud také název **l-kontext**).

Objektu získanému vyhodnocením výrazu se říká také **l-hodnota** (chceme-li pak zdůraznit, že mluvíme o „normálních“ **hodnotách** a nikoliv **objektech**/**l-hodnotách**, můžeme použít také pojem **r-hodnota**).

Nyní můžeme konečně přistoupit k popisu nových tvarů výrazů:

1. Výraz $*expr_1$, tzv. **dereferenci**, lze použít pouze vyhodnotí-li se $expr_1$ na **platný ukazatel** (jedná-li se o ukazatel neplatný, program je chybný a jeho chování není určeno), a vyhodnotí se na **objekt** (l-hodnotu), který je tímto ukazatelem popsán. Je-li výraz $expr_1$ typu `T *`, je výraz $*expr_1$ typu `T`.
2. Výraz $\&expr_1$ lze použít jen tehdy, vyhodnotí-li se $expr_1$ na **objekt** (je l-hodnotou) a výsledkem je **ukazatel** na tento objekt. Unární operátor `&` se anglicky nazývá „address of“, nicméně jeho výsledkem není striktně vzato adresa.[13] Je-li výraz $expr_1$ typu `T`, je výraz $\&expr_1$ typu `T *`.
3. $expr_1$ `=` $expr_2$ (jedná se o zobecnění již zavedeného `var = expr2`) lze použít, vyhodnotí-li se $expr_1$ na objekt (tzn. $expr_1$ popisuje l-hodnotu). Efektem tohoto výrazu je, že hodnota uložená v takto popsaném objektu se přepíše na hodnotu, kterou získáme vyhodnocením výrazu $expr_2$.

4 Příkazy Příkaz `return` v definici podprogramu bez návratové hodnoty (návratový typ je `void`) píšeme `return;` – nesmí se zde zejména objevit výraz.

[10] To neznamená, že objekt je s touto adresou pevně svázán – pouze to, že kdykoliv může být objekt **sekvenčně pozorován**, bude k nalezení na této adrese.

[11] Pravidla pro typy e_1 a e_2 zároveň vynucují, že musí být `void` oba nebo ani jeden.

[12] Na rozdíl od jazyka C++ výrazy tvaru `var = e1` ani tvaru `e0 ? e1 : e2` nikdy objekt nepopisují (nejsou l-hodnotami).

[13] Tuto zkratku si můžeme dovolit pouze proto, že použití operátoru `&` donutí překladač „zafixovat“ pozorovatelnou adresu objektu – každé použití `&` na objekt musí vrátit stejnou hodnotu a každá dereference takto získaného ukazatele musí, po dobu jeho platnosti, popisovat tento objekt.

Část 5: Pole

Tato kapitola přináší **pole** – první složený typ, se kterým budeme v tomto kurzu pracovat. Speciální vlastností pole v jazyce C je, že neexistují **hodnoty** typu pole, pouze **objekty**. Protože se jedná o složené objekty, budou sestávat z **podobjektů** – jednotlivých položek. Všechny podobjekty (položky) daného pole jsou stejného typu.

1 Deklarace Pole vytvoříme podobně jako jiné objekty **deklarací proměnné**. Krom objektu tak vznikne jméno, kterým můžeme takto vytvořený objekt odkázat. Deklarace pole má tento tvar:

```
typ jméno[výraz];
```

Přitom typ může být libovolný dosud zavedený hodnotový typ (nemůže to tedy být pole, protože pole není hodnotový typ).

Část 6: Struktury, zřetězený seznam

Tato kapitola přináší novou třídu složených typů – struktury, neboli záznamové typy.

1 Definice Definice struktury má tento tvar:

```
struct jméno0
{
    typ1 jméno1;
    typ2 jméno2;
    …
    typn jménon;
};
```

Každé jméno uvnitř definice ($jméno_1$ až $jméno_n$) definuje **složku** struktury odpovídajícího typu.[14]

Konečně $jméno_0$ je jménem struktury (pozor, není totéž jako jméno typu!). Definujeme-li strukturu `foo`, na místech, kde je očekáván typ, můžeme psát `struct foo`.

2 Výrazy Se strukturami souvisí také dva nové tvary výrazů:

- $expr_1$`.jméno` – přístup ke složce; typ výrazu $expr_1$ musí být struktura, která má složku pojmenovanou `jméno`. Vyhodnotí-li se $expr_1$ na objekt, výraz jako celek pojmenovává příslušný podobjekt (a může tedy stát na levé straně přiřazení).
- $expr_1$`->jméno` – nepřímý přístup ke složce skrze ukazatel – podobně jako dereference, vstupní podmínkou je, že $expr_1$ je **platný** ukazatel. Výsledkem je vždy objekt (l-hodnota).

[14] Složky můžeme sdružovat podle typu, nicméně pozor na ukazatele – stejně jako u lokálních proměnných, deklarace položek `int *x, y;` zavádí položku `x` typu `int *` a položku typu `y` typu `int`. Dvě položky typu `int *` zapisujeme jako `int *x, *y;`.