Jazykové konštrukcie: Výrazy Rekurzívne algoritmy

1. Výrazy v programovacom jazyku C
   - údajové objekty a l-hodnota
   - operátory, priorita a asociatívnosť
   - aritmetické výrazy
   - logické výrazy
   - relačné výrazy
   - bitové operácie
2. Rekurzívne algoritmy
   - vlastnosti, priama a nepriama rekurzia
   - výhody použitia resp. nepoužitia rekurzie
   - príklad rýchleho triedenia (quicksort)

Výrazy v programovacom jazyku C

- Prostriedok pre vyjadrenie **spôsobu výpočtu údajových objektov**
- Stupne abstrakcie
  - vyjadrenie výpočtu na strojovej úrovni číslicového počítača
  - vyjadrenie výpočtu na vyššej úrovni (aritmetické, logické, relačné výrazy)
- Syntax výrazu
  - presne definovaná postupnosť **operandov** a **operátorov**
    (`a = b + 2`)
- Sémantika výrazu
  - presne definovaný spôsob výpočtu údajových objektov na základe použitých operátorov, typov operandov a štruktúry výrazu
    (k hodnote premennej `b` pripočítaj číslo 2 a výsledok ulož do premennej `a`)

Údajové objekty ako operandy

- Operandom výrazu je **údajový objekt**
- Každý údajový objekt má
  - **adresu** – miesto jeho uloženia v pamäti
  - **hodnotu** – údaj uložený na príslušnej adrese podľa typu údajového objektu
- Pri výpočte výrazu môžeme ako operand použiť
  - priamo **hodnotu** údajového objektu
    `a = b + 1`
  - nepriamo **adresu** na hodnotu údajového objektu (použitie **smerníka** na údajový objekt)
    `scanf("%d", &a)`

  (Pri použití adresy údajového objektu môžeme navyše pracovať s touto adresou ako s hodnotou)

L-hodnota

- Prístup k **hodnote** údajového objektu je pomocou jeho **adresy**
- **L-hodnota** je výraz vyjadrujúci údajový objekt, s ktorého **hodnotou môžeme manipulovať**
  - pre každý výraz, ktorý **je** l-hodnota **vieme určiť adresu** a pomocou nej manipulovať s jeho hodnotou
  - pre výraz, ktorý **nie je** l-hodnota **nevieme určiť adresu**, poznáme iba jeho hodnotu, ktorú nemôžeme meniť
- Použitie l-hodnoty súvisí s výrazom priradenia (napr. `a = 1`)
  - **na ľavej strane priradenia musí byť vždy l-hodnota**

L-hodnota (príklady)

- `a = 10`

  `a` je l-hodnota (vieme určiť adresu premennej) a 10 nie je l-hodnota (nevieme určiť adresu konštanty)
- `a + b`

  výraz nie je l-hodnota (nevieme určiť adresu hodnoty vypočítanej súčtom hodnôt premenných `a` a `b`)
- `a + 1 = 5`

  výraz je nesprávny, pretože výraz `a + 1` nie je l-hodnota (nevieme určiť adresu hodnotu vypočítanej súčtom hodnoty premennej a a konštanty 1) a nemôže byť na ľavej strane priradenia
- `&(a + 1)`

  výraz je nesprávny, pretože výraz `a + 1` nie je l-hodnota (viď vyššie) a nemôžeme použiť operátor & pre získanie adresy tohoto výrazu
- `3 = a`

  výraz je nesprávny, pretože konštanta `3` nie je l-hodnota

Implicitná konverzia typov operandov

- Vo výraze sa môžu vyskytovať hodnoty operandov rôznych typov (`a = 1 + 0.5`)
- Pri výpočte výrazu sa tieto hodnoty rôznych typov automaticky (implicitne) konvertujú na jeden z použitých typov
- Konverzia pre výraz priradenia
  - typ pravej strany sa konvertuje na typ ľavej strany

int = **char**	s rozšírením znamienka alebo bez (vec implementácie)
char = **int**	orezanie vyšších rádov
char, **int** = **long**	orezanie vyšších rádov
int = **float**	zanedbanie desatinnej časti
float = double	zaokrúhlenie

- Konverzia pre ostatné výrazy
  - operand s „nižším“ typom sa prevedie na „vyšší“ typ iného opernadu vo výraze

int ⇨ **unsigned** ⇨ **long** ⇨ **double**
⇧ ⇧
char, **short** **float**

Operátory a ich vlastnosti

- Operátory určujú druh operácie nad operandmi
- Stupeň operátora
  - **unárny** – viaže sa s jedným operandom (napr. negácia)
  - **binárny** – viaže sa s dvoma operandmi (napr. sčítanie)
  - **ternárny** – viaže sa s troma operandmi (napr. podmienka)
- **Priorita**
  - poradie vykonávania operátora vo výraze vzhľadom k ostatným operátorom vo výraze
  - priorita môže byť **explicitne vyjadrená pomocou zátvoriek ( )**
- **Komutatívnosť**
  - možnosť zmeny poradia operandov
- **Asociatívnosť**
  - poradie vykonávania operátorov vo výraze s operátormi rovnakej priority

`a + b + c * d / e + f`	sa vypočíta takto	`(((a + b) + ((c * d) / e)) + f)`
`a = b = c = 1`	sa vypočíta takto	`(a = (b = (c = 1)))`
`a + b >> 3`	sa vypočíta takto	`((a + b) >> 3)`
`f = fopen() == NULL`	sa vypočíta takto	`(f = (fopen() == NULL))`

Rozdelenie operátorov

aritmetické (súčet, rozdiel, súčin, podiel, zvyšok po delení, unárne mínus)	`+ - * / % -`
relačné (menší, menší alebo rovný, rovný, rôzny, väčší alebo rovný, väčší)	`< <= == != >= >`
logické (logický súčin, logický súčet, negácia)	`&& || !`
bitové (súčin, súčet, neekvivalencia, jednotkový doplnok, posuv doľava, doprava)	`& | ^ ~ << >>`
inkrementačný operátor	`++`
dekrementačný operátor	`--`
priraďovacie (aktualizácia hodnoty pripočítaním, odpočítaním, atď.)	`= += -= *= /= %= &= |= ^= <<= >>=`
operátor výpočtu adresy	`&`
operátor indexácie prvku poľa	`[ ]`
operátor podmieneného výrazu	`? :`
operátor zmeny typu	`(typ)`
operátor zistenia veľkosti údajového typu alebo premennej	`sizeof()`
operátor čiarka (operátor zabudnutia, zániku hodnoty)	`,`
operátor nepriameho prístupu (indirekcie, nepriamej adresácie, dereferencie)	`*`
operátory sprístupnenia zložky štruktúry (kvalifikácia a nepriama kvalifikácia)	`. ->`

Prehľad priorít a asociatívnosti operátorov

Priorita	Operátor	Asociatívnosť	
15	`( ) [ ] -> .`	→	zátvorky, indexy, kvalifikácia
14	`! ~ ++ -- -`	←	unárne operátory
	`(typ) * & sizeof()`	←	
13	`* / %`	→	multiplikatívne operátory
12	`+ -`	→	aditívne operátory
11	`<< >>`	→	posuvy
10	`< <= > >=`	→	relačné operátory
9	`== !=`	→	relačné operátory: rovný, rôzny
8	`&`	→	logický súčin po bitoch
7	`^`	→	neekvivalencia po bitoch
6	`|`	→	logický súčet po bitoch
5	`&&`	→	logický súčin
4	`||`	→	logický súčet
3	`? :`	←	podmienkový operátor
2	`= += -= *= /= %= &=`	←	priraďovacie operátory
	`|= ^= <<= >>=`	←	
1	`,`	→	operátor čiarka

Aritmetické výrazy

- Súčet `+` , rozdiel `-` , súčin `*` , podiel `/` , zvyšok po delení `%` , negácia `-`
- Problém komutatívnosti
  - výsledok výrazu `a + b + c` je z matematického hľadiska rovnaký ako výsledok výrazu `b + c + a`

    ```
    int a = 32000;
    int b = 10000;
    int c = -20000;
    ```

    výpočet výrazu `a + b + c` je problém (nastane pretečenie), ale výpočet výrazu `b + c + a` je v poriadku
- Problém poradia vyhodnotenia operandov
  - vo výraze `a + b` nie je zaručené poradie vyhodnotenie výrazov `a` a `b`
  - ak vo výraze `x = a + f()` volanie funkcie `f()` ovplyvňuje premennú `a`, potom `a + f()` nie je to isté ako `f() + a` (je lepšie použiť dva výrazy `b = f()` a `x = a + b`)
- Podiel celočíselných výrazov
  - ak `a`, `b` sú celé čísla, potom výraz `a / b` bude tiež celé číslo (aj ak tento výraz priradíme reálnej premennej)

Logické výrazy

- Logický súčet `||` , logický súčin `&&` , negácia `!`
- V jazyku C neexistuje typ pre logické hodnoty (pravda, nepravda) a používa sa typ celé čísla (int)
  - hodnota **0 je nepravda** a ostatné **nenulové hodnoty sú pravda**
  - výsledkom logického výrazu je vždy **hodnota 0 (nepravda) resp. 1 (pravda)**

a	b	a && b	a || b	!a	!b
0	0	0	0	1	1
0	N	0	1	1	0
N	0	0	1	0	1
N	N	1	1	0	0

je zameniteľné s `b == 0`

`N` – nenulová hodnota

- Logiský výraz sa vyhodnocuje zľava doprava a vyhodnotenie **sa zastaví**, ak je známa logická hodnota výsledku, t.j. nemusia sa vyhodnotiť všetky operandy logického výrazu (**vyhodnocovanie je závislé od priebehu výpočtu**)

Relačné výrazy

- Menší `<` , menší alebo rovný `<=` , väčší `>` , väčší alebo rovný `>=` , rovný `==` , nerovný `!=`
- Výsledkom relačného výrazu je vždy **logická hodnota** (pravda resp. nepravda), pričom tá je v jazyku C stále reprezentovaná celým číslo (0 resp. 1)
- Je potrebné dávať pozor pri používaní relačných výrazov s **reálnymi číslami** (v číslicovom počítači sú vždy reprezentované iba s **určitou presnosťou**)

```
double a = 0.0;
do {
  ...
  a = a + 0.1;
} while (a != 1.0);
```

môže nastať problém

```
double a = 0.0;
do {
  ...
  a = a + 0.1;
} while (a <= 1.0);
```

správne riešenie

Operátory inkrementácie a dekrementácie

- Inkrementácia `++` , dekrementácia `--`
- Tieto operátory sa používajú na efektívnejší a bezpečnejší zápis výrazov
  `a = a + 1` resp. `a = a - 1`
  spôsobom
  `a++` (alebo `++a`) resp. `a--` (alebo `--a`)
- Spôsob použitia
  - **prefixný** `++a` resp. `--a`
    - úprava hodnoty premennej `a` sa vykoná ešte **pred** začatím vyhodnocovania výrazu (vo výraze sa pracuje už so zmenenou hodnotou)
  - **postfixný** `a++` resp. `a--`
    - úprava hodnoty premennej `a` sa vykoná až **po** vyhodnotení celého výrazu (vo výraze sa pracuje ešte s pôvodnou hodnotou)

Priraďovacie operátory

- Priradenie `=`
  - zabezpečí priradenie hodnoty výrazu na pravej strane priradenia do výrazu na ľavej strane priradenia
  - ľavá strana výrazu priradenia musí byť l-hodnota
- Priradenie `OP=`
  - `OP` môže byť jeden z binárnych operátorov `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`
  - výraz **`x OP= y`** je ekvivalentný výrazu **`x = (x) OP (y)`**
  - efektívnejší a bezpečnejší zápis výrazu priradenia, kde výraz na ľavej strane sa vyskytuje aj vo výraze na pravej strane priradenia
  - ľavá strana výrazu priradenia musí byť l-hodnota

Operátor podmieneného výrazu

- Podmienený výraz e_1 `?` e_2 `:` e_3
  - ak výraz e_1 je nenulový, potom výsledkom výrazu je výraz e_2, v opačnom prípade výsledkom výrazu je výraz e_3
- Používa sa na zjednodušenie zápisu príkazu vetvenia napr.

```
if (a > b)
  max = a;
else
  max = b;
```

môžeme nahradiť výrazovým príkazom

```
max = a > b ? a : b;
```


Operátor spojenia výrazov

- Spojenie výrazov e_1`,` e_2
  - spája rôzne výrazy e_1 a e_2 do jedného výrazu
  - výsledkom spojenia výrazov je výraz e_2 na pravej strane spojeného výrazu
- Používa sa pre zjednodušenie zápisu postupnosti viacerých výrazov, napr.

```
a = (b = 10, c = 20, b + c);
```

predstavuje vykonanie nasledovných výrazových príkazov

```
b = 10;
c = 20;
a = b + c;
```

- Pozor, symbol `,` sa používa aj ako oddeľovač iných jazykových konštrukcií (definície premenných, zoznam parametrov funkcie, zoznam prvkov poľa atď.)

Operátor pretypovania

- Pretypovanie `(typ)`
- Operátor pretypovania slúžni na explicitnú konverziu typu výrazu
  - výraz **`(typ) e`** zmení používanie hodnoty výrazu **`e`** na základe uvedeného typu **`typ`**
- Používa sa tam, kde je implicitná konverzia typu nevyhovujúca, napr.

```
int a;
float b;
a = 5;
b = (float) a / 2;
```

umožní správny výpočet hodnoty reálnej premennej `b` z celočíselnej hodnoty premennej `a` a celočíselnej konštanty `2` pretypovaním výrazu

Rekurzívne algoritmy

- Prvok je rekurzívny, ak sa čiastočne skladá, alebo je definovaný pomocou samého seba
- Rekurzia je silným nástrojom najmä pri matematických definíciách
  - napr. prirodzené čísla, stromové štruktúry, funkcie (faktoriál, Fibonacciho čísla, atď.)
- Rekurzia umožňuje definovať nekonečnú množinu prvkov konečným príkazom a podobne možno **nekonečný počet výpočtov opísať pomocou konečného rekurzívneho programu**
- Rekurzívne algoritmy sú najvhodnejšie pri riešení problémov a pri výpočtoch funkcií alebo spracovaní takých štruktúr údajov, ktoré už samy osebe sú definované rekurzívnym spôsobom

Rekurzívna funkcia

- Schému rekurzívneho algoritmu môžeme vyjadriť vzťahom

$$P \equiv \mathcal{P}(S, P)$$

  ktorý predstavuje program P ako kompozíciu $\mathcal{P}$ príkazov S (neobsahujúcich P) a samotného P
- **Prostriedkom pre vyjadrenie rekurzie v programoch je podprogram (funkcia), ktorej identifikátor slúži na rekurzívnu aktiváciu jeho tela**

Priama a nepriama rekurzia

- Ak podprogram P obsahuje priamy odkaz na seba, tak o ňom hovoríme, že je **priamo rekurzívny**
- Ak podprogram P obsahuje odkaz na iný podprogram Q, ktorý obsahuje (priamy alebo nepriamy) odkaz na P, tak P je **nepriamo rekurzívny**
- Použitie rekurzie preto nemusí byť vždy priamo zrejmé z textu samotného podprogramu

Problém ukončenia rekurzie

- Rekurzívne funkcie dávajú možnosť nekonečných výpočtov a je potrebné riešiť problém ukončenia výpočtu
- Rekurzívne volanie funkcie *P* musí byť riadené podmienkou *B*, ktorá môže byť za určitých okolností nesplnená

$$P \equiv \mathcal{P}(S, \text{ak } B \text{ potom } P)$$

- Dôkazom konečnosti rekurzívneho programu je preukázanie konečnej hĺbky rekurzie, tzn. každé volanie *P* znižuje počet ďalších volaní *P*
- Často je dôležité poukázať nielen na konečnosť rekurzie, ale aj na **malú hĺbku tejto rekurzie**, pretože každé volanie rekurzívnej funkcie vyžaduje pridelenie určitého množstva pamäti

Kedy používať a kedy nepoužívať rekurziu

- Rekurzívny program je vhodný, ak riešený problém alebo používané údaje sú definované rekurzívne
- Použitie rekurzie **nezaručuje**, že algoritmus predstavuje najlepšie riešenie problému
- Vo všeobecnosti sa schéma

$$P \equiv \mathcal{P}(S, \text{ak } B \text{ potom } P)$$

dá transformovať na nerekurzívny (iteratívny) tvar

$$P \equiv \mathcal{P}(\text{nech } x = x_0, \text{ pokiaľ } B \text{ opakuj } S)$$

- **Každý rekurzívny program možno transformovať na iteratívny tvar (pomocou cyklov)**
- Pri zložitejších rekurzívnych schémach je však transformácia na iteratívny tvar veľmi nepraktická
- **Programy, ktoré sú svojou podstatou rekurzívne (skôr než iteratívne) majú byť implementované pomocou rekurzívnych funkcií**

Príklady rekurzívnych algoritmov

- Výpočet faktoriálu

  Faktoriál $n!$ pre celé číslo $n \geq 0$:
  - $0! = 1$
  - ak $n > 0$, tak $n! = n(n-1)!$
- Výpočet Fibonacciho čísel

  Fibonacciho číslo $f(n)$ pre celé číslo $n > 0$
  - $f(1) = 1$ a $f(2) = 1$
  - ak $n > 2$, tak $f(n) = f(n-1) + f(n-2)$
- Rýchle triedenie (Quicksort)
  - vybrať ľubovoľný prvok x z triedeného poľa a; prehľadať prvky zľava, pokiaľ sa nenájde prvok $a_i > x$; prehľadať prvky sprava, pokiaľ sa nenájde $a_j < x$; vymeniť prvky a_i a a_j; pokračovať v prehľadávaní a výmene, kým sa ľavé a pravé prehľadávania nestretnú v strede poľa

    $$M_1 = \frac{1}{n}\sum_{x=1}^{n}\frac{x-1}{n}(n-x+1) = \frac{n}{6} - \frac{1}{6n}$$

    $$C = n\log n \qquad M = \frac{n}{6}\log n$$
  - v najhoršom prípade je rádu n^2 a rozhodujúcim krokom je výber prvku x