Bitové operace a bitová pole

Jazyk C je nízkoúrovňový programovací jazyk, který nabízí programátorům na rozdíl od vysokoúrovňových programovacích jazyků i operátory, pomocí kterých mohou nepřímo pracovat i s jednotlivými bity.

Bitové operace

Práce s jednotlivými bity většinou vyžaduje hlubší znalost operačního systému, například jakým stylem se ukládají čísla. Operace s jednotlivými bity však dokáže výrazně zrychlit program a některé operace ani pomocí dříve zmíněných konstrukcí není možné provést, jelikož pro ně byl nejmenší jednotkou informace byte.
V jazyce C máme k dispozici operátory *bitový součin*, odpovídající logické operaci AND, *bitový součet*, který je ekvivalentem operace OR, *bitový exkluzivní součet* (XOR), *bitový posun doleva*, *bitový posun doprava* a *jedničkový doplněk* (NOT). Pro všechny platí, že jejich argumenty mohou být pouze celá čísla (jak `signed` tak `unsigned`). Priorita bitových operátorů je nízká, ve složitějších výrazech je potřeba používat závorky.

Bitový součin

Bitový má následující syntaxi.

```
x & y
```

`i`-tý bit výsledku bitového součinu `x & y` bude roven `1` pokud `i`-tý bit `x` i `y` budou rovny `1`, jinak bude výsledný bit roven `0`.
Výsledkem operace `100 & 50` bude 32, jelikož platí následující vztah.

```
    100   0 1 1 0 0 1 0 0
     50   0 0 1 1 0 0 1 0
100 & 50  0 0 1 0 0 0 0 0
```

Bitový součin se často používá k výběru jen určitých bitů (nastavení ostatních bitů na 0). Například, pokud chceme zjistit, zda je číslo liché, stačí nám znát hodnotu nejméně významného bitu (0. bitu).[126]

[126] V textu předpokládáme, že nejméně významný bit je vpravo.

```
#define liche(x) (1 & (x))
```

Je třeba si uvědomit rozdíl mezi bitovým a logickým součinem.

```
int i = 1, j = 2, k, l;

k = i && j;
l = i & j;
```

Výsledky se budou lišit. Zatímco `k` je rovno `1`, protože obě `i` i `j` jsou nenulová, tak `l` bude rovno `0`. A to proto, že čísla mají následující binární vyjádření.

```
1   =   00000001
2   =   00000010
```

Bitový součet

Bitový součet má následující syntaxi.

```
x | y
```

`i`-tý bit výsledku bitového součtu `x | y` bude roven `1` pokud `i`-tý bit `x` nebo `y` bude roven `1`. Budou-li oba nulové výsledný bit bude roven `0`. Často se používá k nastavení určitých bitů na `1`, s tím, že ostatní bity si ponechají původní hodnotu.
Následující příklad ukazuje použití bitového součtu pro převod velkého písmena na malé.

```
#define na_mala(c) (c | 0x20)
```

`0x20` odpovídá binárně `0 0 1 0 0 0 0 0`, což odpovídá hodnotě 32. Ze znalosti ASCII tabulky víme, že právě malá a velká písmena se v této hodnotě liší.

Bitový exkluzivní součet

Bitový exkluzivní součet se zapisuje následujícím způsobem.

```
x ^ y
```

`i`-tý bit výsledku bitového exkluzivního součtu `x ^ y` bude roven `1`, pokud se `i`-tý bit `x` nerovná `i`-tému bitu `y`. Budou-li oba shodné (oba rovny `0`, nebo `1`) výsledný bit bude roven `0`.
Následující kód slouží k porovnání dvou čísel.

```
if(x ^ y)
    /* cisla jsou rozdilna */
```

Pokud jsou `x` a `y` rozdílná, pak alespoň jeden bit bude roven `1` a tedy výsledek bude nenulový (pravdivý).

Bitový posun doleva

Bitový posun doleva má následující zápis.

```
x << n
```

Posune bity v `x` o `n` pozic doleva. Při tomto posunu se bity zleva ztrácí a vpravo jsou doplňovány `0`. Typickým použitím je použití k násobení mocninou 2. `x = y << 1` vynásobí číslo 2.

```
1       =  00000001
1 << 1  =  00000010
```

`x = y << 3` vynásobí číslo `8` (2^3).
Takovéto násobení je rychlejší než běžné násobení. Než násobit číslo číslem `80`, je lepší ho vynásobit `64` (posun o 6 bitů) a `16` (posun o 4 bity) a výsledky sečíst. Viz následující příklad.

```
i = j * 80; /* pomalejsi */

i = (j << 6) + (j << 4); /* rychlejsi */
```

Bitový posun doprava

Bitový posun doprava má následující syntaxi.

```
x >> n
```

Posune bity v `x` o `n` pozic doprava. Při tomto posunu se bity zprava ztrácí a vlevo jsou doplňovány `0`.[127] Má opačný význam než bitový posun doleva. Často se používá k celočíselnému dělení mocninou 2. `x = y >> 1` vydělí číslo číslem 2, `x = y >> 3` vydělí číslo 8. Stejně jako u bitového posunu doleva, takovéto dělení mocninou 2 je rychlejší, než klasické dělení.

[127] Toto tvrzení platí pro `unsigned` typy. Pro `signed` je výsledek implementačně závislý.

Jedničkový doplněk (negace bit po bitu)

Jedničkový doplněk se zapisuje následujícím způsobem.

```
~x
```

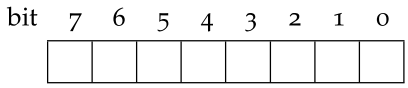
Nulové bity nahradí `1` a jedničkové `0`. Nulovány jsou i počáteční nuly, takže je výsledek závislý na typu čísla (kolik bitů obsahuje). Jiný výsledek bude pro `char` (8 bitů) a `int` (32 bitů).

```
int main()
{
    int i = 10, i2;
    unsigned char j = 10, j2;

    i2 = ~i;
    j2 = ~j;

    printf("%d %d \n", sizeof(char), sizeof(int));
    printf("%u %d", i2, j2);
}
```

Práce se skupinou bitů

Bitové operace se často používají pro práci se skupinou bitů, která přestavuje například stavové slovo definované následovně.

```
unsigned int status;
```



Definujeme si konstanty, které určí pozice příznakových bitů ve stavovém slově. Například použijeme `3.`, `4.` a `5.` bit pro příznaky číst, psát, vymazat.

```
#define READ 0x8
#define WRITE 0x10
#define DELETE 0x20
```

Pro nastavení všech příznaků na `1` použijeme následující příkaz.

```
status |= READ | WRITE | DELETE;
```

[128] Operátor | představuje bitový součet. Operátor |= je operátor přiřazení s bitovým součtem. Obdoba už známých aritmetických přiřazovacích operátorů +=, -= a pod.

Pokud chceme na 1 nastavit jen příznaky pro čtení a zápis, použijeme obdobný výraz, konkrétně následující výraz.

```
status |= READ | WRITE;
```

Pro nastavení všech příznaků na 0 použijeme bitový součin následujícím způsobem.

```
status &= ~(READ | WRITE | DELETE);
```

Chceme-li nastavit na 0 jen příznak pro čtení, použijeme výraz

```
status &= ~READ;
```

Pro testování, zda je nějaký příznak nastaven na 0 použijeme operátor bitového součinu. Konkrétně, pokud chceme zjistit, zda je příznak čtení roven 0, použijeme následující podmínku.

```
if (! (status & READ ))
```

Bitové pole

Speciálním případem struktur je *bitové pole*. Tato struktura je omezená velikostí typu `int`. Nejmenší délka jedné položky v této struktuře je jeden bit. Definuje se obdobně, jako se definují nám již známé struktury, pouze položky struktury nejsou určeny libovolným typem a jménem, ale jménem a délkou v bitech. Typy jednotlivých položek mohou být jen `unsigned` (neznaménkové) nebo `signed` (znaménkové) a za každým členem uvádíme za dvojtečkou počet bitů, které budou pro tento člen vyhrazeny.
Předchozí příklad řešený pomocí bitového pole je následující.[129]

[129] Položka `zacatek` je zde jen kvůli přeskočení prvních 3 bitů.

```
typedef struct{
    unsigned zacatek : 3; // bity od 0 do 2
    unsigned read : 1; // bit 3
    unsigned write : 1; // bit 4
    unsigned delete : 1; // bit 5
} STAV;


STAV status;
```

S jednotlivými bity se pracuje pomocí tečkové notace, jako je tomu u struktur. Konkrétně pro nastavení příznaku pro čtení na 1 použijeme příkaz `status.read = 1;`, pro nastavení příznaku pro zápis

na 0 příkaz `status.write = 0;` a podobně. Pro test, zda je příznak pro čtení nastaven na 1 použijeme podmínku ve tvaru `if(status.read)`.

Cvičení

Úkol 84
Z příkladu v sekci o bitovém součtu víme, že se malé písmeno a odpovídající velké liší v 5. bitu. Napište makro, které převede malé písmeno na velké.

Úkol 85
Napište funkci, která pro zadané číslo `cislo` a číslo `n` zjistí hodnotu `n`-tého bitu čísla `cislo`. Například hodnota 1. bitu čísla 3 je 1, 2. bit má hodnotu 0. Bude se vám hodit operace bitového posunu.

Úkol 86
Vyzkoušejte si práci se stavovým slovem. Vytvořte funkci, která bude brát jako vstup celé číslo `n` a `stav` a vypíše všechna čísla od 0 do `n` a bude vynechávat násobky podle `stav`. `stav` bude uchovávat informaci, zda se mají vypisovat násobky 2, 3 a 5. Například ve `stav` je nastaven příznak, že se nemají vypisovat násobky 2, pak funkce vypíše jen lichá čísla do `n`.

Úkol 87
Upravte předchozí kód tak, aby stav nastavoval uživatel po dotazu v konzoli.

Úkol 88
Upravte předchozí příklad tak, aby `stav` byl definován jako bitové pole.

Úkol 89
Napište funkci, která způsobí rotaci (ne posun) čísla `x` o `n` bitů doprava. Pro číslo 3 (00000011) a `n = 3` dostaneme 96 (01100000) (pracujeme-li nad `char`).

Úkol 90
Napište program, který bude využívat bitového pole pro úschovu času (hodiny, minuty, vteřiny). Vytvořte i funkce na převod času do bitového pole a funkci, která vypíše bitové pole jako čas.