Baze de numerație, scrierea constantelor și operatori pe biți

În prima parte a acestui material ne vom asigura că stăpânim câteva noțiuni matematice, legate de lucrul cu numere în alte baze de numerație. Nu vom intra în detalii, scopul fiind acela de a înțelege semnificația scrierii constantelor în C/C++ în alte baze decât 10 precum și modul intern de operare cu datele întregi stocate ca secvențe de 0 și 1. Pe noi ne interesează lucrul cu numere în baza 2, în baza 8 și în baza 16. Vom enunța însă câteva lucruri valabile la orice bază.

Lucrul cu baze de numerație

Într-o anumită bază B, cifrele au valoarea cuprinsă între 0 și B-1. Estre strânsă legătură între această afirmație și faptul că prin împărțirea unui număr la B resturile obținute sunt cuprinse între 0 și B-1. De exemplu cifrele în baza 10 sunt 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, cifrele în baza 4 sunt 0, 1, 2, 3 iar cifrele în baza 2 sunt 0 și 1. Putem lucra și cu baze mai mari de 10 dar atunci trebuie făcute convenții de notare a cifrelor mai mari decât 9. Putem spune că este o regulă în informatică de a nota cifrele mai mari ca 9 în baza 16 primele 6 litere ale alfabetului, fie mari fie mici (A, B, C, D, E, F).

Când scriem un număr într-o bază oarecare vom considera cifrele sale numerotate de la dreapta (de la cifra unităților) aceasta considerându-se pe poziția 0.

Regula de trecere a unui număr dintr-o bază B în baza 10 este următoarea:

fie numărul: $C_kC_{k-1}C_{k-2}$ ... C_1C_0, în baza B. Valoarea sa în baza 10 este următoarea:

$C_k*B^k + C_{k-1}*B^{k-1} + C_{k-2}*B^{k-2} + ... + C_1*B^1 + C_0*B^0$. Am notat B^i ca fiind B ridicat la puterea i.

Exemple

Numărul dat	Baza	Formula desfășurată	Valoarea în baza 10
2101	4	$2*4^3 + 1*4^2 + 0*4^1 + 1*4^0$	145
110101	2	$1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$	53
2FA	16	$2*16^2 + 15*16^1 + 10*16^0$	762
FF	16	$15*16^1 + 15*16^0$	255

Operația inversă, aceea de a trece un număr din baza 10 într-o anumită bază B, se realizează conform următorului algoritm:

- se păstrează restul împărțirii numărului inițial la B ca fiind cifra unităților
- numărul dat se înlocuiește cu câtul împărțirii sale la B
- se păstrează restul împărțirii noului număr la B ca fiind cifra zecilor
- numărul curent se înlocuiește iarăși cu câtul împărțirii sale la B

Acest procedeu se repetă până când numărul devine 0. Mai pe scurt spus, se păstrează resturile împărțirii repetate la B, în ordine inversă.

Iată un exemplu: Dorim scrierea numărului 145 în baza 4.

- 145 % 4 dă rezultatul 1, deci ultima cifră va fi 1 și vom continua cu 145/4 = 36
- 36 % 4 dă rezultatul 0, deci penultima cifră va fi 0 și vom continua cu 36/4 = 9
- 9 % 4 dă rezultatul 1, deci următoarea cifră va fi 1 și se continuă cu 9/4 = 2
- 2 % 5 dă rezultatul 2, așadar umătoarea cifră va fi 2 și nu se mai continuă întrucât 2/4 = 0.

Scriind în ordine inversă cifrele date obținem 2101, adică avem cumva proba pentru exemplul din tabelul de mai sus.

Este foarte util la informatică următorul rezultat: trecerea din baza 2 într-o bază putere de 2, se spunem 2^k se poate face și astfel: se grupează cifrele numărului în baza 2 începând din dreapta, câte k, și se scrie în baza 10 valoarea obținută cu fiecare astfel de grup. Acestea sunt cifrele numărului în baza 2^k.

Iată câteva exemple:

Numărul în baza 2	Baza în care trecem		
1011100110	4	Grupăm cifrele de la dreapta, câte două căci $4 = 2^2$ 10 11 10 01 10 2 3 2 1 2	23212
1011100110	8	1 011 100 110 1 3 4 6	1346
1011100110	16	10 1110 0110 2 14 6	2E6

Observăm că grupul cel mai din stânga poate fi cu mai puțin de k cifre (considerăm că vom completa în față cu cifre 0 până când avem k cifre).

Ca exercițiu, vă invităm să transformați (cu regula prezentată la începutul materialului) numărul 1011100110 din baza 2 în baza 10, numărul 23212 din baza 4 în baza 10, numărul 1346 din baza 8 în baza 10 și numărul 2E6 din baza 16 în baza 10. Va trebui să obțineți același rezultat.

Operațiile cu numere în altă bază decât 10 se fac după aceiași algoritmi învățați în clasele primare la matematică.

Exemplificăm efecrtând o operație de adunare direct în baza 8.

Considerăm numerele 243 și 361 (ambele scrise în baza 8). Se adună 3 cu 1, și se obține 4 (avem deci ultima cifră a rezultatului ca fiind 4 și fără transport). Se adună 6 cu 4. În baza 10 acest rezultat este 10 dar noi lucrăm în baza 8. Cifra care se trece la rezultat este restul împărțitii lui 10 la 8 (adică 2) iat câtul 10/8 = 1 va fi transport. Așadar, până acum, ultimele cifre ale rezultatului sunt 24. Ultima adunare este 2 + 3 + transportul(1) și rezultatul este 6, care de nu depășește ordinul. Așadar, operația de adunare între 243 și 361 considerate în baza 8 dă rezultatul 624 (tot în baza 8). Puteți face proba transformând în baza 10 cele trei numere și apoi să verificați egalitatea. La finalul acestei secțiuni veți găsi explicat foarte detaliat modul în care adunăm două numere în baza 2.

Este interesant de reținut că rezultatul adunării cu 1 în baza 2 este echivalent cu cel obținut prin aplicarea următorului algoritm: toate valorile de 1 de la final se transformă în 0, primul 0 întâlnit devine 1 iar restul valorilor rămân la fel:

De exemplu:

```
10100111 + 1
Ne imaginăm operația astfel:
10100111+
00000001
Rezultat:
10101000
```

Aplicând regula de la adunare, prima operație este 1+1 care dă 2, adică cifra 0 și transport 1. În continuare, cât timp avem sus 1 (căci jos este de acum 0) se adună cu acest transport și va da iarăși 0 cu transport 1.

Prima dată când sus avem 0 se va obține rezultatul 1 (de la transport), iar pentru restul cifrelor nu va mai apărea nicio influență (jos este 0 și nici transport nu mai avem).

De fapt, rezultatul mai general, la adunarea cu 1 în orice bază, este următorul: cât timp la final avem cifra maximă a bazei, se pune 0, iar prima cifră întâlnită diferită de cea maximă crește cu 1. Celelalte cifre rămân nemodificate.

Exemplul 1: 92399 + 1 (în baza 10) dă 92400.

Exemplul 2: 46277 + 1 (în baza 8) dă 46300

Este esențial să ne obișnuim cu semnificația scrierii numerelor în baza 2, pentru că acesta este modul intern memorare a numerelor și direct în această bază se fac și calculele. În tabelul următor se află reprezentarea în baza 2 a tuturor numerelor naturale cuprise între 0 și 15.

Număr scris în baza 10	Scrierea numărului în baza 2 pe număr minim necesar de biți	Scrierea numărului în baza 2 extins, pe 4 biți
0	0	0000
1	1	0001
2	10	0010
3	11	0011
4	100	0100
5	101	0101
6	110	0110
7	111	0111
8	1000	1000
9	1001	1001
10	1010	1010
11	1011	1011
12	1100	1100
13	1101	1101
14	1110	1110
15	1111	1111

Iată câteva observații pe care pe putem face analizând tabelul de mai jos

- pentru numerele cuprinse între 2^{k-1} și 2^k-1 avem nevoie de k biți.

- dacă neglijăm cifrele 0 nesemnificative, numerele de forma "*putere de 2 − 1*" conțin doar cifre de 1 (observați reprezentările lui 1, 3, 7, 15)

- numerele de forma putere de 2 au exact o cifră de 1 în scrierea în baza 2.

Iată încă un exemplu de adunare a două numere în baza 2, explicat detaliat:

```
9876543210
1011110110 +
0000110100
```

Pe prima linie este poziția cifrei (numerotate din dreapta începând cu valoarea 0. Am completat în față cu 0 numărul mai mic pentru claritate.
Prima dată de face adunarea cifrelor de la poziția 0, adică 0+0. Rezultatul este 0 și nu avem nici transport pentru pasul următor.

```
9876543210
1011110110 +
```

```
0000110100
          0     transport 0
```

Urmează adunarea cifrelor de la poziția `1`: `1+0+0`(transportul). Rezultatul este `1`, fără transport.

```
9876543210
1011110110 +
0000110100
         10     transport 0
```

Urmează adunarea cifrelor de la poziția `2`: `1+1+0`(transportul). Rezultatul este `2`. Pentru baza `2` acest lucru înseamnă cifra `0` și transport `1`. *Cifra este restul împărțirii la baza în care se lucrează* iar *transportul este câtul împărțirii la baza în care se lucrează*. `0` (cifra) este `2%2` iar `1` (transportul) este `2/2`.

```
9876543210
1011110110 +
0000110100
        010     transport 1
```

Urmează adunarea cifrelor de la poziția `3`: `0+0+1`(transportul). Rezultatul este `1`, fără transport.

```
9876543210
1011110110 +
0000110100
       1010     transport 0
```

La pasul următor se adună cifrele de la poziția `4`: `1+1+0`(transportul). Rezultatul este `0`, iar transportul este 1.

```
9876543210
1011110110 +
0000110100
      01010     transport 1
```

La pasul următor se adună cifrele de la poziția `5`: `1+1+1`(transportul). Rezultatul este `3%2 = 1`, iar transportul este `3/2 = 1`.

```
9876543210
1011110110 +
0000110100
     101010     transport 1
```

Urmează să se adune cifrele de la poziția `6`: `1+0+1`(transportul). Rezultatul este `0`, iar transportul este `1`.

```
9876543210
1011110110 +
0000110100
    0101010     transport 1
```

Urmează să se adune cifrele de la poziția `7`: `1+0+1`(transportul). Rezultatul este `0`, iar transportul este `1`.

```
9876543210
1011110110 +
0000110100
   00101010     transport 1
```

Acum se vor aduna cifrele de la poziția `8`: `0+0+1`(transportul). Rezultatul este `1`, iar transportul este `0`.

```
9876543210
```

```
1011110110 +
0000110100
 100101010     transport 0
```

În fine, se adună cifrele de la poziția `9`: `1+0+0`(transportul). Rezultatul este `1`, iar transportul este `0`.

```
9876543210
1011110110 +
0000110100
1100101010     transport 0
```

Nu avem transport la ultima operație, așa că rezultatul va avea același număr de cifre ca și operandul mai lung. În caz contrar ar mai fi apărul în față o cifră egală cu `1` (transportul).

Putem face ușor proba transformând toate cele trei numere de la final în baza `10` și obținem: `758 + 52 = 810`.

Mai multe moduri de a specifica în limbaj constantele

Am prezentat un mod de a scrie constantele în C++, și anume pe acela mai apropiat de matematică. Pe scurt, regula este: constantele întregi se scriu ca la matematică (succesiune de cifre precedate eventual de `+` sau `−`), iar la constantele reale folosim caracterul *punct* pe post de separator între partea întreagă și cea zecimală.

Limbajul pune la dispoziție și alte moduri de a scrie constantele.

Scrierea constantelor întregi

- *scrierea zecimală*: succesiune de cifre în baza `10`, precedată, eventual de plus sau minus și în care să nu fie `0` prima cifră.

Exemple: `23`, `−45`, `+109`

- *scrierea octală* (în baza `8`): succesiune de cifre de la `0` la `7`, care să înceapă cu cifra `0` și care să fie eventual precedată de plus sau minus.

Exemple: `034`, `−077`, `0234`

- scrierea *hexazecimală* (în baza `16`): succesiune de cifre în baza `10`, litere mari și/sau mici de la `a` la `f` (`A - F`) care să aibă prefixul `0x` (sau `0X`) și precedată eventual de plus sau minus.

Exemple: `0X23`, `−0xFFF`, `0x12A`

Observăm că apariția în față a lui `0` este semn că avem o constantă specificată altfel decât în modul clasic. Apariția lui `x` sau `X` după acest `0` este semnul că vorbim despre o constantă specificată în baza `16`.

Exemple

Constantă	Valoare	Explicație
`10`	`10`	
`14`	`14`	
`014`	`12`	Constanta a fost specificată în baza `8`.
`0x14`	`20`	Constanta a fost specificată în baza `16`
`FF`		Scriere greșită.
`068`		Scriere greșită, este semn că am indica o constantă în baza `8` dar nu putem folosi cifra `8`.
`0XFF`	`255`	
`-023`	`-19`	

`0xffg`		Scriere greșită, este semn că am indica o constantă în baza `16` dar nu putem folosi caracterul `g`.
`029`		Scriere greșită, prefixul anunță scrierea unei constante în baza `8` dar cifrele care urmează sunt și în afara intervalului de cifre octale (`0` .. `7`).

Observație: `16`, `0x10` și `020` sunt trei moduri corecte de a indica valoarea `16` de tip întreg.

Așadar, codul:

`int a;` `a = 16;`	este echivalent cu:	`int a;` `a = 020;`	și cu:	`int a;` `a = 0x10;`

În limbaj, calculele se realizează implicit în tipul int. Ce înseamnă acest lucru? Dacă în expresie tipul cel mai cuprinzător al vreunui operand este `int` sau unul mai mic, operanzii sunt convertiți la `int` și apoi se realizează operația. Dacă avem un operand de tip mai mare decât `int`, calculele se fac în tipul acelui operand.

Constantele sunt implicit de tip int. Putem însă folosi sufixe și atunci anunțăm că acea constantă să fie considerată de alt tip decât cel implicit. Sunt des utilizate sufixele:

`u` - pentru a anunța constanta de tip unsigned `int`

`LL` - pentru a anunța constanta de tip `long long`

Vă rog să urmăriți cu mare atenție exemplele următoare din care sunt foarte multe de învățat.

Secvență de cod	Ce se afișează	Explicație

`int a;`
`a = 2000000000 * 3;`
`cout<<a;`
	`1705032704`	Nici nu are importanță dacă se afișează exact această valoare. Probabil că ea este ceea ce se obține cu ultimii `32` de biți ai valorii `6` miliarde. Valoarea expresiei din dreapta depășește maximul reprezentabil pe tipul int și atunci ea nu mai este calculată corect, pentru că ambii operanzi sunt de tip int, deci calculele se fac în int.

`long long a;`
`a = 2000000000 * 3;`
`cout<<a;`
	`1705032704`	Evident că efectul este același, trunchierea se realiează încă din timpul efectuării calculelor din dreapta lui = pentru că acolo ambii operanzi sunt de tip int.

`int a;`
`a = 2000000000 * 3LL;`
`cout<<a;`
	`1705032704`	Motivul pentru care se obține același lucru este următorul: calculul expresiei din dreapta se face acum corect, dar valoarea obținută se copiază într-o zonă de memorie neîncăpătoare, așadar iar se trunciază, salvându-se valoarea cu biții care încap pe int.

`long long a;`
`a = 2000000000 * 3LL;`
`cout<<a;`
	`6000000000`	Corect

`int a = 10u;` `cout<<a;`	`10`	`10` are aceeași valoare fie că este considerat de tip int și fie că este considerat de tip unsigned int.

Scrierea constantelor reale

Vom extinde regula cunoscută astfel: poate apărea ca sufix caracterul `e` (sau `E`) iar după acesta o secvență de cifre zecimale (imediat după `e` poate fi plus sau minus). Numărul format cu aceste cifre reprezintă exponentul puterii lui `10` cu care se înmulțește valoarea din fața lui `e`, pentru a obține valoarea întregii constante. Iată, mai bine pe exemple:

Constantă reală	Semnificație	Valoare
`2.3`		`2,3`
`-3.05`		`-3,05`
`2.3E2`	$2,3 * 10^2$	`230,0`
`3e4`	$3*10^4$	`30000,0`
`-2.3e-2`	$-2,3 * 10^{-2}$	`-0,023`
`1e-7`	$1 * 10^{-7}$	`0,0000001`

Observăm că poate lipsi caracterul *punct*, dar prezența lui `e` face constanta să fie oricum de tip real.

Operatorii pe biți

Sunt operatori care se aplică datelor de **tipuri întregi**. Pentru a înțelege efectul aplicării lor, trebuie să stăpânim modul de reprezentare internă a numerelor întregi în zona de memorie rezervată.

Pentru a fi mai simplu de scris exemplele vom considera variabile întregi pe `2` octeți (`16` biți), ceea ce vom prezenta fiind valabil și pentru celelalte tipuri de date întregi.

Prima regulă este aceea că valorile pozitive se memorează pe biții rezervați lor conform scrierii în baza `2` a valorii.

Exemplu:

```
short a;
a = 43;
```

Variabila a va ocupa 16 biți și în urma atribuirii, configurația lor va fi următoarea:

Poziție bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Valoare bit	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1

Conform regulii de scriere în baza `10` am avea $2^5 + 2^3 + 2^1 + 2^0 = 43$

O variabilă de tip `short` am arătat că poate lua valori între -2^{15} și $2^{15}-1$ (sunt, în total 2^{16} valori, adică pentru fiecare configurație de `0` și `1` de lungime `16`, câte o valoare din acest interval). Așadar valoarea pozitivă maximă este $2^{15}-1$.

Dar acest număr este egal cu valoarea expresiei: $2^{14} + 2^{13} + \ldots + 2^1 + 2^0$. Acesta este un lucru cunoscut de la matematică dar vom vedea că sunt mai multe interpretări de la info care justifică acest rezultat. Deducem că pentru valoarea maximă, toți biții, mai puțin primul, au valoarea `1`.

Deci, reprezentarea internă a lui `32767` este:

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Astfel, observăm că restul valorilor pozitive au ca reprezentări interne configurații care au valori `0` în locul valorilor `1` existente. Concluzia este următoarea: toate configurațiile care au primul bit `0` corespund valorilor pozitive.

Dacă variabila noastră ar fi fost de tip `unsigned short`, valoarea `1` pentru primul bit (bitul din față, bitul `15` sau, mai des spus, *bitul cel mai semnificativ*) ar fi reprezentat valorile pozitive cuprinse între 2^{15} și $2^{16}-1$. Asta pentru că la valorile pozitive de la short se adună 2^{15} (pentru că este `1` bitul cel mai semnificativ).

Pentru tipul `short` (și celelalte tipuri întregi ce nu sunt scrise `unsigned`), configurațiile care au bitul cel mai semnificativ egal cu 1 sunt reprezentări interne ale numerelor negative.

Vom prezenta mai departe regula după care se deduce modulul când configurația memorează o valoare negativă. Dacă ne-am grăbi, am putea spune că prezența lui `1` în față înseamnă număr negativ iar trecerea în baza `10` a configurației cu ceilalți biți este valoarea absolută. Nu este însă chiar așa.

Iată o justificare simplă, printr-un exemplu:

numărul `1` memorat într-o variabilă de tip short are reprezentarea:

`0000000000000001`

Numărul `-1`, pe raționamentul (greșit) de mai sus ar fi:

`1000000000000001`

Ar trebui ca adunarea celor două valori să dea 0. Dar este evident că nu se întîmplă așa

`0000000000000001 +`

`1000000000000001`

`1000000000000010`

Iată și regula corectă (numită *regula complementului față de 2*): se complementează toți biții (valorile `1` se înlocuiesc cu `0` și cele `0` se înlocuiesc cu `1`), apoi se adună cu valoarea `1` (în baza `2`).
Iată un exemplu:

Reprezentarea internă	`1111111100010100`
După complementare	`0000000011101011`
După ce se adună `1`	`0000000011101100` (biții de `1` de la final devin `0` și primul `0` devine `1`, restul de valori ale biților se păstrează).

Acum, scriind în baza `10` numărul obținut (`0000000011101100`) avem $2^7+2^6+2^5+2^3+2^2 = 236$
Așadar secvența inițială, adică `1111111100010100` reprezintă reprezentarea internă a numărului `-236`.

Iată și o probă:

`-236 +`	`1111111100010100 +`
`236`	`0000000011101100`
`0`	`10000000000000000`

Se observă că ultimii `16` biți ajung toți `0`, iar bitul din față, `1` (dat de transport) este în afara zonei de memorie alocate, deci rezultatul este `0`.
Observații
- valoarea `-1` se va reprezenta prin toți biții `1` (dacă adunăm valoarea `1` se propagă mereu transportul și se obține valoarea `0`)
- valoarea cu toți biții `1` este în același timp valoarea maximă de la tipul corespunzător `unsigned`
- o configurație formată numai din biți `1` reprezintă un număr de forma *(putere de 2) - 1*
- numerele impare sunt cele care au bitul cel mai puțin semnificativ egal cu `1` (singura putere de `2` impară care apare în sumă la transformarea din baza `2` în baza `10`).

Este util de știut și următoarea regulă de transformare a unui număr din baza 10 în baza 2 (în afară de cea generală în care împărțim în mod repetat la 2 și preluăm resturile în ordine inversă):

Pentru numărul dat căutăm cea mai mare putere a lui 2 mai mică sau egală cu numărul el, apoi o scădem din număr și reluăm pentru valoarea rămasă. Repetăm asta până când se obține valoarea 0. Este corectă această abordare pentru că nu vom determina astfel de două ori aceeași putere a lui 2 (suma a două puteri de 2 este tot putere de 2 și ar însemna că la un pas anterior nu am fi găsit corect cea mai mare putere de 2 mai mică sau egală cu valoarea curentă). De asemenea, faptul că puterile de 2 care apar sunt distincte este în concordanță cu faptul că avem doar cifra 0 și cifra 1 în baza 2.

Exemplu

Dorim să scriem în baza 2 numărul 41. Căutăm cea mai mare putere de 2 mai mică sau egală cu 41 și găsim 32. Continuăm cu 9 (41 − 32). Găsim 8 și continuăm cu 1 (9−8). Acum găsim chiar 1 și ajungem deci la 1−1 = 0, oprindu-ne. Așadar $41 = 32 + 8 + 1 = 2^5 + 2^3 + 2^0$. Obținem că scrierea în baza 2 a lui 41 este: 101001.

Operatorii pe biți îi impărțim în două categorii: *operatori logici pe biți* și *operatori de deplasare*.

Operatorii logici pe biți

Ei se aplică la date întregi.
Sunt

~	*not* logic pe biți, este operator unar
&	*și* (*and*) logic pe biți, binar
|	*sau* (*or*) logic pe biți, binar
^	*sau exclusiv* (*xor*) logic pe biți, binar

Observăm că operatorii *and/or* pe biți se scriu diferit de cei logici care conțin caracterul semn dublat.
Operatorul ~ este unar, se aplică la un operand întreg iar rezultatul este cel obținut complementând biții operandului. Evident că nu are efect asupra operandului, acesta putând fi orice fel de expresie.
Exemple:

Secvență de cod	Ce se afișează	Explicație

`short a;`
`a = -1;`
`cout<<(~a);`
	0	Reprezentarea internă a lui −1 este formată doar din biți 1. După complementarea acestora se obțin doar biți 0 care reprezintă codificarea internă a valorii 0.

`short a;`
`a = -236;`
`cout<<(~a);`
	235	Scrierea internă a lui −236 este:
		1111111100010100 (vezi mai sus)
		Negarea sa:
		0000000011101011 (valoarea 235)

Operatorul & este binar, se aplică la două valori întregi iar rezultatul este ceea ce obținem aplicând "și" bit cu bit între toate perechile de biți de pe aceeași poziție. La "și" rezultatul este 1 doar dacă ambii "operanzi" sunt 1.
Exemple

Secvență de cod	Ce se afișează	Explicație

`short a, b, c;`
`a = 101;`
`b = 87;`
`c = (a&b);`
`cout<<c;`
	69	a: 0000000001100101
		b: 0000000001010111
		c: 0000000001000101

`short a, b, c;`	276	a: 0000000111110101

`a = 501;`
`b = -236;`
`c = (a&b);`
`cout<<c;`
		`b: 1111111100010100`
		`c: 0000000100010100`

Dacă ne imaginăm scrierile în baza `2` ca fiind niște mulțimi reprezentate prin vector caracteristic, putem spune că în urma aplicării operatorului `&` se obține intersecția celor două mulțimi.

Operatorul `|` este binar, se aplică la două valori întregi iar rezultatul este ceea ce obținem aplicând "sau" bit cu bit între toate perechile de biți de pe aceeași poziție. La "sau" rezultatul este `1` doar dacă este `1` cel puțin un operand.

Exemple

Secvență de cod	Ce se afișează	Explicație

`short a, b, c;`
`a = 101;`
`b = 87;`
`c = (a|b);`
`cout<<c;`
	`119`	`a: 0000000001100101`
		`b: 0000000001010111`
		`c: 0000000001110111`

`short a, b, c;`
`a = 501;`
`b = -236;`
`c = (a|b);`
`cout<<c;`
	`-11`	`a: 0000000111110101`
		`b: 1111111100010100`
		`c: 1111111111110101`

Dacă ne imaginăm scrierile în baza `2` ca fiind niște mulțimi reprezentate prin vector caracteristic, putem spune că în urma aplicării operatorului `|` se obține reuniunea celor două mulțimi.

Operatorul `^` este binar, se aplică la două valori întregi iar rezultatul este ceea ce obținem aplicând "sau exclusiv" bit cu bit între toate perechile de biți de pe aceeași poziție. La "sau exclusiv" rezultatul este `1` doar dacă este `1` exact un operand.

Exemple

Secvență de cod	Ce se afișează	Explicație

`short a, b, c;`
`a = 101;`
`b = 87;`
`c = (a^b);`
`cout<<c;`
	`50`	`a: 0000000001100101`
		`b: 0000000001010111`
		`c: 0000000000110010`

`short a, b, c;`
`a = 501;`
`b = -236;`
`c = (a^b);`
`cout<<c;`
	`-287`	`a: 0000000111110101`
		`b: 1111111100010100`
		`c: 1111111011100001`

Dacă ne imaginăm scrierile în baza `2` ca pe niște mulțimi reprezentate prin vector caracteristic, putem spune că în urma aplicării operatorului `^` se obține diferența simetrică a celor două mulțimi.

Operatorii de deplasare (shiftare)

Sunt în număr de doi, operatorul de deplasare la stânga și cel de deplasare la dreapta.

Operatorul `<<` (*de deplasare la stânga*) este binar, ambii operanzi fiind întregi. Rezultatul se obține translatând la stânga biții operandului stâng cu atâtea poziții cât este valoarea operandului drept. Valorile biților care au fost în față și ies din zona de memorie se vor pierde iar în dreapta se completează cu `0`.

Exemple

Secvență de cod	Ce se afișează	Explicație

`short a, b;`
`a = 101;`
`b = (a << 3);`
`cout<<b;`
	808	a: 0000000001100101
		b: 0000001100101000

`short a, b;`
`a = -236;`
`b = (a << 2);`
`cout<<b;`
	-944	a: 1111111100010100
		b: 1111110001010000

Iată o observație foarte importantă: operația `a << b` este echivalentă cu $a * 2^b$.

Dacă ne uităm pe primul exemplu, este ușor să probăm ce am spus anterior. Deplasând la stânga cu 3 poziții toate puterile lui 2 care compuneau reprezentarea ajung cu 3 mai mari, adică, dacă la noua reprezentare am da factor comun 2^3 am obține în paranteză chiar valoarea inițială.

$101 = 2^6 + 2^5 + 2^2 + 2^0$

$808 = 101 << 3 = 2^9 + 2^8 + 2^5 + 2^3 = 2^3 * (2^6 + 2^5 + 2^2 + 2^0) = 2^3 * 101$

Operatorul >> (*de deplasare la dreapta*) este binar, ambii operanzi fiind întregi. Rezultatul se obține translatând la dreapta biții operandului stâng cu atâtea poziții cât este valoarea operandului drept. Valorile biților care au fost în spate și ies din zona de memorie se vor pierde iar în stânga se completează cu *bitul de semn*. Deci, la numerele negative se va completa cu 1 (pe toate pozițiile necesare), iar la numerele pozitive, analog, cu 0. Astfel, semnul rezultatului va fi același cu semnul operandului stâng. Cu un raționament similar celui prezentat la operatorul de shiftare la stânga, deducem că deplasarea la dreapta este echivalentă cu împărțire la o putere de 2. Mai exect, expresia `a>>b` este echivalentă cu $a/2^b$. Aici este vorba chiar de câtul împărțirii (se pot pierde biți de 1 din dreapta când împărțirea nu este întreagă).

Exemple

Secvență de cod	Ce se afișează	Explicație

`short a, b;`
`a = 101;`
`b = (a >> 1);`
`cout<<b;`
	50	a: 0000000001100101
		b: 0000000000110010

`short a, b;`
`a = -236;`
`b = (a >> 2);`
`cout<<b;`
	-59	a: 1111111100010100
		b: 1111111111000101

Operatorii pe biți (mai puțin ~) pot fi combinați cu operatorul de atribuire obținând noi operatori de atribuire prescurtati. Astfel, se pot scrie expresii de forma:

`variabilă op= expresie`

unde op poate fi: &, |, ^, <<, >>.

Exemplu: `n = (n >> 1)` se poate scrie și `n>>=1`

Atenție la utilizarea operatorilor pe biți în expresii alături de alți operatori întrucât ei au prioritate de aplicare foarte mică și astfel se recomandă folosirea a cât mai multe perechi de paranteze rotunde pentru grupare.

Folosirea operatorilor pe biți în expresii duce la mărirea vitezei de rulare a programelor întrucât pentru fiecare dintre aceștia procesorul dispune de propriile instrucțiuni dedicate.

Iată, în continuare, câteva lucruri pe care le putem realiza elegant folosind operatorii pe biți.

`1<<k`	Această expresie este echivalentă cu valoarea 2^k. Prin folosirea operatorului obținem direct acest rezultat în comparație cu calculul clasic care necesită repetiție.
`1LL<<k`	Astfel trebuie scris dacă dorim să obținem puteri ale lui `2` mai mari ca `31`. În expresia anterioară constanta `1` era de tip `int` (așa este implicit în C/C++) deci trebuie pus sufixul care să forțeze reprezentarea lui `1` pe `64` de biți.
`n>>1`	Echivalent cu `n/2`
`n&1`	Această expresie este echivalentă cu `n%2`. Adică putem testa astfel paritatea unui număr. Iată justificarea: numărul `1` are numai biți `0`, mai puțin bitul cel mai puțin semnificativ care este `1`. Punând pe `1` ca operand lângă `&` toți biții rezultatului, mai puțin ultimul, vor fi `0`. Ultimul bit se obține ca `&` între bitul `1` (de la numărul `1`) și ultimul bit al celuilalt operand. Așadar se obține `1` doar dacă și acel bit este `1`.
`n^n`	`0`
`n^n^n`	`n` (`xor` aplicat de un număr par de ori pe aceeași valoare duce la `0`, iar `xor` aplicat de un număr impar de ori pe aceeași valoare duce la acea valoare).
`n^0`	`n`
`(n >> k) & 1`	Aceasta este expresia pe care o scriem pentru a obține valoarea bitului `k` al lui `n`. Practic, prin deplasare "ducem" acel bit pe ultima poziție, iar conform exemplului anterior, mai rămâne să aplicăm `&` cu numărul `1`

`n = n + (1<<k)`
sau
`n += (1<<k)`
	Această instrucțiune adună la n valoarea 2^k. Dacă bitul de pe poziția `k` al lui `n` este `0`, această expresie are ca efect setarea la valoarea `1` în `n` a acelui bit. Atenție, că dacă bitul de pe poziția `k` este deja `1`, el devine `0` și vor fi afectați și biții mai semnificativi întrucât apare transport la operația de adunare în baza `2`.

`n = (n | (1<<k))`
sau
`n |= (1<<k)`
	Aceasta este expresia care setează la `1` valoarea bitului de pe poziția `k` din `n` indiferent de valoarea sa anterioară, restul biților rămânând nemodificați. Dacă ne imaginam reprezentarea binară a lui `n` ca un vector caracteristic asociat unei mulțimi, prin această operație adăugăm (dacă nu există deja), la mulțime elementul `k`.
	Iată, mai detaliat ce se întâmplă:
	`k = 4`
	`n:  XXXXXXXXXXXXXXXX`
	`1:  0000000000000001`
	`1<<k 0000000000010000`
	La "sau" cu bitul `1`, pe acea poziție bitul obținut va fi `1`, iar ceilalți rămân nemodificați întrucât `0` este element neutru la această operație.

`n = (n & ~(1<<k))`
sau
`n &= ~(1<<k)`
	Acestea sunt modalități de a seta la `0` un anume bit al unei variabile, indiferent de valoarea sa anterioară și fără să afectăm valorile celorlalți biți. Dacă am fi știut sigur că bitul `k` este `1`, pentru a-l face `0` puteam folosi și expresia `n -= (1<<k)`
	Iată, detaliat, ce se întâmplă
	`k = 4`
	`n:  XXXXXXXXXXXXXXXX`
	`1:  0000000000000001`
	`1<<k  0000000000010000`
	`~(1<<k)  1111111111101111`
	În urma operației `&` cu bitul `0`, se obține `0`, iar în rest nu apar modificări întrucât `1` este element neutru la "și".

`n & -n`	Valoarea aceste i expresii este întotdeauna o putere de `2`. Ea reprezintă bitul cel mai puțin semnificativ al lui `n`. Dacă anlizăm regula complementului față de `2` pentru aflarea valorii absolute pentru un număr negativ, observăm că acela este singurul bit care rămâne `1` atât la `n` cât și la `-n` (pe restul pozițiilor avem la un maxim un operand `1`).

`for (;n;n-=(n&-n)) {`
`  cnt++;`
`}`
	Această secvență de cod numără în `cnt` câți biți de `1` are `n` scris în baza `2`. Se observă că la fiecare pas este eliminat din `n` bitul cel mai puțin semnificativ al său. Este interesent că numărul de pași realizați de repetiție este egal cu această valoare. În exemplul următor vom prezenta și varianta brut care verifică valoarea fiecărui bit în parte.
	Subliniem că secvența de cod curentă se dovedește utilă și la implementarea structurii de date numită ***arbori indexați binar***.

`for (i=15;i>=0;i--)`
`  if ((n>>i)&1)`
`  cnt++;`
	Aceasta este secvența echivalentă cu cea anterioară dar numărul de pași este mereu egal cu numărul total de biți ai reprezentării interne a lui `n`.

Probleme rezolvate.

Se dă numărul `n` și încă `n` valori nenegative și se cere să spunem pentru fiecare câți biți de `1` are în reprezentarea internă. Valorile date sunt pe `32` de biți și numărul de valori date este până la 10^6.

Soluție

Prima soluție este de a aplica algoritmul brut prezentat mai sus. Astfel, numărul de pași care se fac este de ordinul `n*32` (`32` = numărul de biți pe care se reprezintă valorile).

O altă soluție este să folosim varianta optimizată (prezentată și ea mai sus) de a obține numărul de biți `1` ai unei valori. Totuși, dacă toate numerele date au număr mare de biți `1`, timpul de executare tinde spre cel de la varianta anterioară.

Există și o a treia soluție, mult mai rapidă, pe care o vom prezenta în continuare. Ea se bazează pe precalcularea, pentru anumite numere, a numărului de biți și folosirea acestei informații. Să calculăm câți biți de `1` are fiecare valoare pe `32` de biți nu este practic, pentru că sunt de ordinul a `4` miliarde de valori și atât timpul de a realiza asta cât și memoria necesară stocării tuturor valorilor ar fi prea mari.

Vom precalcula însă numărul de biți `1` pentru fiecare număr posibil pe `16` de biți și vedem apoi cum ne folosim de această informație.

Pe `16` biți sunt `65536` valori posibile, adică efortul de memorie nu este mare și vom vedea în contiuare că nici efortul de timp. Dacă notăm `B[i]` = numărul de biți `1` din scrierea în baza `2` a lui `i`, avem:

`B[0] = 0;`
`B[1] = 1;`
`B[i] = B[i/2]` dacă `i` este par
`B[i] = 1+B[i/2]` dacă `i` este impar
Acest lucru se datorează faptului că valoarea `i/2` conține exact aceiași biți ca și `i`, mai puțin ultimul bit al lui `i` (`i/2` este echivalent cu `i>>1` și e posibil să se piardă un bit de `1`, caz în care numărul `i` ar fi impar și atunci se adună la rezultat valoarea `1`).

Așadar, precalcularea este următoarea:

```
b[0] = 0;
b[1] = 1;
```

```
for (i=2;i<=(1<<16)-1;i++)
    b[i] = b[i/2] + i%2; /// sau, mai elegant scris, b[i] = b[i>>1] + (i&1);
```

Dacă toate numerele date ar fi pe 16 biți, ne putem folosi de precalculare și rezolvăm problema astfel:

```
cin>>n;
for (i=1;i<=n;i++) {
    cin>>a;
    cout<<b[a];
}
```

Putem însă să facem un mic truc și rezolvăm problema și pentru numere de 32 de biți:

```
cin>>n;
for (i=1;i<=n;i++) {
    cin>>a;
    cout<<b[a & 0xFFFF] + b[(a>>16) & 0xFFFF];
}
```

Care este semnificația ?

Valoarea 0xFFFF reprezintă un număr cu 16 biți, toți de 1 ($2^{16}-1$). Acest lucru se datorează faptului că F(adică 15) se scrie ca 1111. Așadar, scrierea a & 0xFFFF ne oferă valoarea ce se obține luând în calcul doar ultimii 16 biți ai lui a, deci accesând în B la această poziție obținem câți de 1 sunt printre ultimii 16 biți ai lui a. Rămâne să adunăm și numărul de biți cu valoarea 1 dintre primii 16 biți ai lui a. Observăm că dacă deplasăm pe a la dreapta cu 16 poziții, biții care ne interesează rămân ultimii 16 așa că putem folosi asupra noii valori formula de la celălalt grup de 16 biți.

Putem afirma că acum răspundem direct pentru fiecare număr dat.

Temă

1. Care este cea mai mică bază în care putem considera numărul 2155 ?

2. Scrieți numărul 2155 din baza 8 în baza 10.

3. Scrieți numărul 2155 din baza 6 în baza 10.

4. Calculați scrierea numărului 1001110 în baza 10, secvența dată fiind considerată în baza 2.

5. Calculați scrierea numărului 1001110 în baza 8, secvența dată fiind considerată în baza 2.

6. Calculați scrierea numărului 1001110 în baza 16, secvența dată fiind considerată în baza 2.

7. Scrieți în baza 2 și în baza 10 numărul FF (specificat în baza 16).

8. Scrieți în baza 2 si în baza 10 numărul FFFF (specificat în baza 16).

9. Scrieți în baza 2 si în baza 10 numărul FFFFFFFF (specificat în baza 16).

10. Scrieți în baza 16 numărul 2331 (specificat în baza 4). Verificați dacă obține rezultat corect următorul algoritm: grupăm cifrele câte două (de la dreapta) și Transformăm fiecare grup de două astfel de cifre din baza 4 în baza 16. Ar trebui să funcționeze metoda pentru că $16 = 4^2$?

11. Scrieți valoarea în baza 10 pentru constanta întreagă 0xf2.

12. Afișați scrierea hexazecimală (în baza 16) pentru valoarea maximă de la tipul int.

13. Afișați scrierea hexazecimală (în baza 16) pentru valoarea maximă de la tipul unsigned int.

14. Afișați scrierea hexazecimală (în baza 16) pentru valoarea maximă de la tipul `long long`.

15. Afișați scrierea hexazecimală (în baza 16) pentru valoarea maximă de la tipul unsigned `unsigned long long`.

16. Considerăm variabila a de tip `short` care memorează valoarea `-1` și variabila `b` de tip `unsigned short` care memorează valoarea `65535`. Afișați pe ecran reprezentările lor interne pe `16` biți. Adunați (în baza `2`) valoarea `1` la fiecare dintre aceste valori.

17. Pentru o variabilă dată de tip `int`, scrieți o expresie care să ofere valoarea obținută cu ultimuii `k` biți ai săi (și valoarea `k` se dă).

18. Pentru o variabilă dată de tip `int`, scrieți o expresie care să ofere valoarea obținută cu biții săi de pe pozițiile `6`, `7`, `8`.

19. Scrieți o expresie care setează la `1` ultimii `2` biți ai valorii unei variabile de tip int numită `a`, indiferent de valoarea lor anterioară. Restul biților trebuie să își păstreze valoarea.

20. Scrieți o expresie care setează la `0` ultimii `2` biți ai valorii unei variabile de tip `int` numită `a`, indiferent de valoarea lor anterioară. Restul biților trebuie să își păstreze valoarea.

21. Fiind dată o variabilă de tip `short` notată `a`, scrieți o secvență de instrucțiuni pentru interschimbarea celor doi octeți ai săi (secvența celor mai nesemnificativi `8` biți trebuie să ajungă, în aceeași ordine pe cei mai semnificativi `8` biți, și invers).

22. Scrieți o secvență de atribuiri care interschimbă valorile a două variabile `a` și `b`, folosind operatorul pe biți `xor` și fără variabile suplimentare.

23. Rezolvați problema anterioară (ultima dintre problemele rezolvate) cu precizarea că numerele date sunt pe `64` de biți.q

24. Se citesc `n` și apoi `n` numere ce reprezentă elementele unei mulțimi. Se citește apoi `m` și `m` numere ce reprezintă elementele altei mulțimi. Se cunoaște că pentru fiecare mulțime elementele date sunt distincte și sunt numere naturale cuprinse între `0` și `60`. Afișați pe o linie a ecranului, separate prin spațiu, elementele reuniunii celor două mulțimi date și pe următoarea linie a ecranului elementele intersecției celor două mulțimi, de asemenea, separate prin spații. Numerele de pe fiecare linie a ieșirii se vor scrie în ordine crescătoare. Se garantează că `n` și `m` sunt strict pozitive și că intersecția celor două mulțimi date este nenulă. Nu este permisă folosirea tablourilor de memorie.

Exemplu

Date de intrare	Date de ieșire
4 1 19 3 12 5 11 10 12 4 19	1 3 4 10 11 12 19 12 19