Логически операции в C/C++. Побитови операции

Логика

Науката за законите, формите и начините на мислене се нарича логика. Тя е една от най-старите науки – следи от нея се забелязват и в древноиндийската, и древнокитайската философия, а също и във философията на антична Гърция в лицето на най-значителния ѝ представител – Аристотел, считан за основател на формалната логика. Дълго време логиката е била смятана за хуманитарна наука, докато Джордж Бул, не е превърне описателните ѝ методи в толкова строги закони, колкото са в другите природни науки като математиката и физиката. В основата на логиката са съжденията. Те биват са прости и сложни. Простите съжденията не могат да бъдат разделени на по-малки изрази, например: "Сега вали дъжд" или " прозореца е отворен". Сложните изрази се съставят от прости посредством логическите връзки (операции) **"И"**, **"ИЛИ"**, **"НЕ"**, **"ако ... тогава"**, **"тогава и само тогава"**. В Булева алгебра логическите изрази обикновено се обозначават с латински букви. В този вид ние се абстрахираме от конкретното съдържание и се съсредоточаваме само върху това дали израза е истина или лъжа. Например, можем да обозначим с буквата **A** твърдението "Сега вали дъжд", а с буквата **B**-твърдението "прозорецът е отворен". С тези съждения се изграждат сложни изрази от вида:

- **не A**: "Не вали дъжд"
- **не B**: "прозорецът е затворен"
- **A и B**: "Вали дъжд и прозорецът е отворен"
- **A или B**: "Вали дъжд или прозорецът е отворен"
- **ако A, тогава B**: "Ако вали дъжд, тогава прозорецът е отворен".

В допълнение към това може да се каже, че има и други изрази, които могат да бъдат получени от две твърдения. С някой от тях ще се запознаем в тази глава. Някой от операциите като **"НЕ"**, **"И"** и **"ИЛИ"** се използват по-често, отколкото други. Оказва се, че те могат да бъдат използвани за да се изрази всяка логическа операция, така че тези три операции може да се считат като базисни.

Логически операции в C++ (И, ИЛИ, НЕ)

В програмният език C++ съществуват три логически операции:

- Логическа операция `И &&;`
- Логическа операция `ИЛИ ||;`
- Логическа операция `НЕ !.`

Логическите операции образуват сложни (съставни) изрази от

няколко по-прости (две или повече) по -прости условия. Тези операции опростяват структурата на програмния код няколко пъти. Разбира се че може да се мине без тях, но тогава броят на if операторите ще се увеличи многократно. В следващата таблица е показано на кратко значението на логическите операции

Таблица 1 – Съставни логически операции в C++

Операция	Обозначение	Условие	Кратко описание
И	**&&**	a == 3 && b > 4	Стойността на логическият израз е истина ако и двете условия **a == 3** и **b > 4** са истина
ИЛИ	**||**	a == 3 || b > 4	Стойността на логическият израз е истина ако поне едно от двете условия **a == 3** или **b > 4** е истина
НЕ	**!**	**!**(a == 3)	Условието истина ако, **а не е равно на 3 (a!=3)**

Тук е мястото да кажем каква е разликата между логическите операция **И**, **ИЛИ** и **НЕ**, но преди това ще се запознаем с логическият тип за данни **bool**. Този тип може да приема две значения: **true (истина)** и **false (лъжа)**. Точно върху този тип данни действат логическите операции. Като цяло съставните логически операции съвместяват действието на няколко прости логически изрази образувани от използването на логическите оператори показани в следната таблица

Таблица 2 – Оператори за сравнение в C++

Оператор	Значение	Използване	Резултат
==	равно	(7 == 5)	**false**
!=	различно	(3 != 2)	**true**
>	по-голямо	(4 > 6)	**false**
<	по-малко	(4 < 6)	**true**
>=	по-голямо или равно	(6 >= 6)	**true**
<=	по-малко или равно	(3 <= 6)	**true**

Следната примерна програма използва три променливи от тип **bool**. Чрез тях се демонстрира работата на логическите оператори И, ИЛИ и НЕ.

Пример:

Програмен код

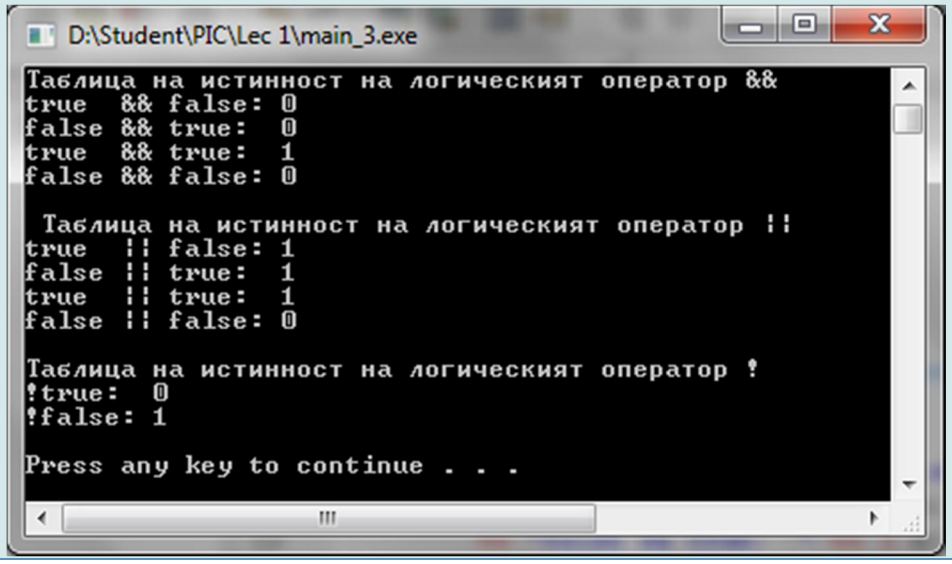
```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    bool a1 = true, a2 = false; // деклариране на логически променливи
    bool a3 = true, a4 = false;
    setlocale(LC_ALL,"Bulgarian");
```

```
    cout << "Таблица на истинност на логическият оператор &&" << endl;
    cout << "true  && false: " << ( a1 && a2 )   << endl // логическо И
         << "false && true : " << ( a2 && a1 )   << endl
         << "true  && true : " << ( a1 && a3 )   << endl
         << "false && false: " << ( a2 && a4 )   << endl;
    cout << " Таблица на истинност на логическият оператор ||" << endl;
    cout << "true  || false: " << ( a1 || a2 )   << endl // логическо ИЛИ
         << "false || true : " << ( a2 || a1 )   << endl
         << "true  || true : " << ( a1 || a3 )   << endl
         << "false || false: " << ( a2 || a4 )   << endl;
    cout << " Таблица на истинност на логическият оператор !"  << endl;
    cout << "!true : " << ( ! a1 ) << endl // логическо НЕ
         << "!false: " << ( ! a2 ) << endl;
    system("pause");
    return 0;

}
```

Резултат



На редове 7 и 8 от програмата се извършва инициализация на променливите от тип **bool**. Тук на всяка променлива се задава стойност **true** или **false**. Работният код започва от ред 7 и показва резултатът от изпълнението на всяка една логическа операция.

Може би имате един въпрос: "Какви са тези нули и единици?". След като има въпрос трябва да има и отговор. Отговорът е: "С **0** се представя стойността на **FALSE (лъжа)**, а с **1** стойността на логическа истина **TRUE** (**вярно**).". Накратко ще дадем някой разяснения относно резултата. Стойността на логическият израз използващ оператора И е истина ако и двете прости условия са истина. Ако дори и едното условие е лъжа резултатът ще бъде неистина.

Що се отнася до логическата операция ИЛИ резултатът ще бъде истина винаги когато дори и един от операторите приема стойност true. Логическото отрицание НЕ е унарна операция. При нея не се комбинират две условия, за разлика от логическите оператори И и ИЛИ, които са бинарни операции.

Логическа операция НЕ (NOT)

Операция **"НЕ"** често се нарича отрицание, или инверсия. В Булевата алгебра има само два символа `0` и `1` (`true`, `false`), така че логическото отрицание - е преходът от едната стойност в другата. От 1 в 0 или обратно. Ако операторът А е истина, следователно "не А" е лъжа, и обратно. За да се обозначи операция **"НЕ"** се използват няколко начина. Изразът "не е" в булевата алгебра се записва като $\bar{A}$ или **–А**. В програмният език С/С++ операцията **"НЕ"** се записва като:

- **!A** – за логически изрази
- **~A** – при побитови операции (Операции с отделни битове).

Табличното представяне на операцията **"НЕ"** е показано в таблицата.

A	$\bar{A}$
0	1
1	0

Тази таблица се състои от две части: в първата колона се изброяват всички възможни стойности на първоначалния израз (те са само две - 0 и 1). Във втората колона се записва резултатът от логическата операция. Тази таблица се нарича **таблица на истинност** на логическата операция. Таблицата на истинност задава **логическата функция**, тоест правилата за преобразуване на логическите стойности към изхода.

Логическата операция `"НЕ"(!)` се използва за инвертиране резултатът от даден логически израз и се записва във вида `!операнд`.

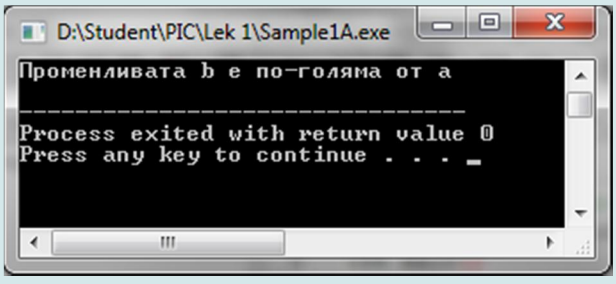
Операндът е логическа променлива или правилен логически израз за програмният език С++. Най-често операцията `"НЕ"(!)` се използва съвместно с логическият оператор `if else`.

Пример:

Програмен код

```
#include <iostream>
using namespace std;
int main()
{
  int a=2, b=3;
  setlocale(LC_ALL,"Bulgarian");
  if( !( a > b )) {
   cout << "Променливата b е по-
голяма от a" << endl;
  } else {
   cout << "Променливата a е по-
голяма от b" << endl;
  }
  return 0;
}
```

Резултат

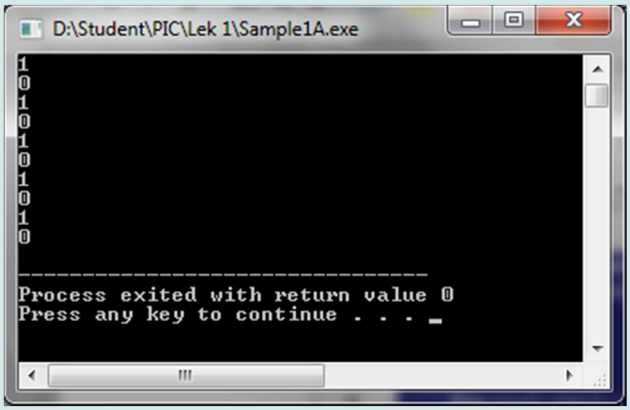


Много често в системите за автоматизация се налага дадена променлива да бъде инвертирана многократно. Например при реализиране генератор на правоъгълни импулси посредством микроконтролер. За да се изведе последователно на екрана 0 и 1 през 1s се използва следната примерна програма.

Пример:

Програмен код	Резултат

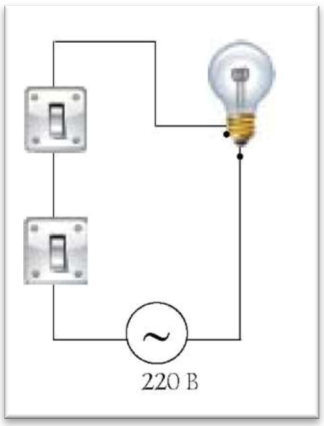
```
#include <iostream>
#include <windows.h>
using namespace std;
int main()
{
    int i, a=1;
    for(i=0;i<10;i++){
        cout << a<< endl;
        // Време закъснение от 1s
        Sleep(1000);
        a = ! a; // Инвертиране
    }
    return 0;
}
```



Логическа операция «И» (AND)

Да предположим, че има две твърдения: **A**-"Сега вали дъжд" и **B**-"прозорецът е отворен". Обединяването на тези две твърдения съставя твърдението: "Сега вали дъжд и прозорецът е отворен." Това твърдение ще бъде вярно (правилно), само ако и двете твърдения **A** и **B** са верни едновременно. За да се разбере операцията **"И"**, може да си представим един проста електрическа схема, при която за да включим една електрическа крушка използва два ключа, които са свързани последователно. За да свети крушката, трябва и двата ключа да са включени. От друга страна за да се изключи е необходимо поне единият от двата ключа да е изключен. Операция **"И"** (за разлика от **"НЕ"**) се изпълнява от две логически стойности, които определяме като **A** и **B**. Резултатът от тази операция в логическата алгебра се записва като **A** `A ^ B` или `A & B`. В програмният език C/C++ се използва записът `A & B` за реализиране на побитови операции или `A && B` при реализиране на логически изрази.

	A	B	A^B
0		0	0
1	0	1	0
2	1	0	0
3	1	1	1



В интерес на истината таблицата вече няма да бъде една колона с оригиналните данни а с две. Броят на линиите също се увеличава от 2 на 4, тъй като от два бита се получат четири различни комбинации (00, 01, 10 и 11). Тези линии са подредени в определен ред: двоични числа, получени от съединяването на A и B, като са подредени във възходящ ред. Както следва от определенията в последната колона ще има само една стойност равна на 1, за вариант **A = B = 1**. Това може лесно да се провери като се умножат стойностите на **A** и **B**. Този израз обуславя и името на операцията "И" - **логическо умножение** или конюнкция(**conjunctio**).

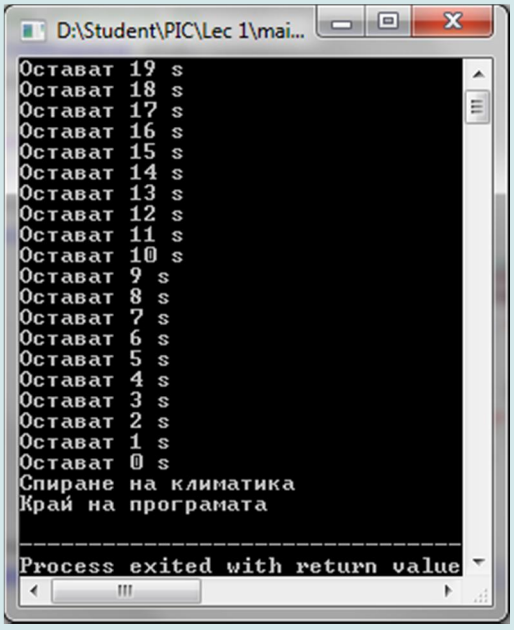
Следващия пример ще демонстрира използването на логическата операция И в реална програма. Да предположим че разполагаме с микропроцесорна система която следи за климата в една сграда. И имаме ситуация при която ако е отворен прозорец за повече от 20s, се задейства процедура по спиране на климатика. Да предположим, че променливата bWindow - указва състоянието на прозореца, т.е. 0 - прозореца е затворен а 1 е отворен. Времето за изчакване се задава от променливата nTimeOut. Когато стойността на променливата nTimeOut е 0 или отрицателна и същевременно стойността на bWindow е истина се задейства процедурата по спиране на климатика.

Пример:

Програмен код

```
#include <iostream>
#include <windows.h>
using namespace std;
int main(int argc, char* argv[])
{
  bool bWindow  = true;
  int nTimeOut  = 20 ;
  setlocale(LC_ALL,"Bulgarian");
  while(true) {
    // Време закъснение от 1s
    Sleep(1000);
     // Декрементиране на брояча
    nTimeOut--;
    cout<<"Остават "<<nTimeOut<<" s"<< endl;
    // Проверка за изход
   if ( bWindow && (nTimeOut <= 0) ) {
      cout << "Спиране на климатика" << endl;
      break; // напускане на цикъла
    }
  }
  cout << "Край на програмата" << endl;
  return 0;
}
```

Резултат

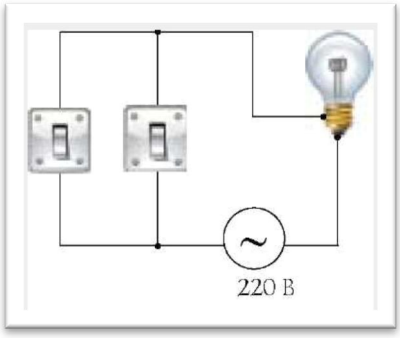


Логическа операция «ИЛИ» (OR)

Съждението **"Вали дъжд или прозорецът е отворен"** е вярно ако **"Вали дъжд"** или ако **"прозорецът е отворен"**. В булевата алгебра логическата операция "ИЛИ" се обозначава като **A+B** или **A V B**. В програмният език C/C++ логическата операция **"ИЛИ"** се означава като:

- **A || B** - за логически изрази
- **A | B** - при побитови операции.

	A	B	A+B
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	1



Операцията **"ИЛИ"** може да се представи с една проста електрическа схема, при която за да включим една електрическа крушка използва два ключа, които са свързани паралелно един на друг. За да свети крушката, трябва поне един от двата ключа да е включен. От друга страна за да се изключи е необходимо и двата ключа да бъдат едновременно изключени. Това е видно от таблицата на истинност където при **A = B = 0** резултатът е **0**. Проверката на съждението може а се направи като се сумират стойностите на **A** и **B**. Единственото отличие от математиката е че при булевата алгебра $1+1 \neq 2$, а 1+1 = 1. Операцията **"ИЛИ"** се нарича **логическа сума** или дизюкция (**disjunctio**)

Операции с отделни битове (Поразрядни логически операции) в C++

Езикът C++ притежава и някой качества на асемблерските езици. За разлика от много други езици от високо ниво той разполага и с пълен набор от операции за работа с отделни битове на числените стойности. Тези операции позволяват проверяване, установяване или преместване на отделни битове на стойностите на променливи от тип **int** (**short,long,unsigned**) или тип **char**. Поразредните операции често се използват за създаване на драйверни програми за връзка с външни устройства. Поразредните операции се означават със следните символи.

Таблица 3 – Побитови логически операции

Логическа операция	Предназначение
&	поразредно логическо И(AND)
|	поразредно логическо ИЛИ(OR)
^	поразредно логическо ИЗКЛЮЧВАЩО ИЛИ(XOR,EOR)
~	поразредно логическо ДОПЪЛВАНЕ ДО 1
>>	изместване в дясно (RIGHT SHIFT)
<<	изместване в ляво (LEFT SHIFT)
&=	поразредно логическо И(AND)
|=	поразредно логическо ИЛИ(OR) с присвояване
^=	поразредно логическо ИЗКЛЮЧВАЩО ИЛИ(XOR,EOR) с присвояване
>>=	изместване в дясно (RIGHT SHIFT) с присвояване
<<=	изместване в ляво (LEFT SHIFT) с присвояване

В C++ съществуват логическите операции: **ИЛИ** - **||**; **И** - **&&**. В много от вас ще възникне въпроса, "С какво се отличават операциите: **&** и **&&**; **|** и **||** ?". Отговора може да се получи когато се разбере принципа на работа на поразредните логически операции. Веднага може да се каже, че логическите операции **&&** и **||** се използват само за построяване на логически условия. Тогава как поразрядните логически операции се използват при двоичната аритметика? Въпросът е в това, че тези операции работят с битовете на отделните клетки от паметта. С други думи тези оператори разглеждат операндите си като подредена съвкупност от битове. Всеки бит може да има стойност 0 (false) или 1 (true). Операторите за работа с битове позволяват на програмиста да проверява и инициализира индивидуални битове или битови подмножества.

Операндите на тези оператори трябва да бъдат от целочислен тип. Въпреки че те могат да бъдат със или без знак препоръчва се използването на операнди без знак. Как точно се обработва знаковия бит при тези оператори зависи от реализацията им; програми, които работят с дадена реализация може да не работят с друга. По такъв начин използването на операнд без знак подпомага осигуряването на мобилност на програмата.

Първо ще обясним как работи всеки оператор. После ще дадем примери за използването на всеки от операторите за работа с битове. Всички оператори, с изключение на оператора **~**, са бинарни.

При това операндите могат да бъда зададени дори и в десетичен формат. Тук ще разгледаме всяка една операция поотделно.

Поразредни логически операции И (&) и ИЛИ (|)

Операторите **&** и **|** реализират съответно поразрядна конюнкция и поразрядна дизюнкция. Таблицата на истинност е показана по-горе.

За по-добро разбиране ще дадем няколко примера с числа. За опростяване на изчисленията ще работим с четири разрядни положителни числа. Първо е показан двоичния код на числата а след това и десетичния.

Поразрядно логическо И

```
  1111 = 15                    1000 = 8
&                            &
  1001 = 9                     1010 = 10
1001 = 9                     1000 = 8
```

Поразрядно логическо ИЛИ

```
  1111  = 15                   1000 = 8
|                            |
  1001 =   9                   1010 =
10
  1111 = 15                    1010 =
10
```

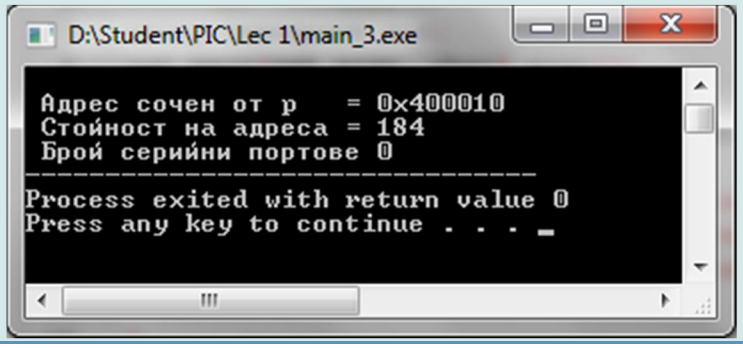
Пример:

На адрес 0040:0010 се съхранява информация за конфигурацията на компютъра. Ако желаем на определим броят на инсталираните серийни канали можем да изпълним следната операция.

Програмен код

```
#include <iostream>
#include <windows.h>
using namespace std;
int main(int argc, char* argv[])
{
  unsigned  int far *p = (unsigned int far *)0x00400010;
  setlocale(LC_ALL,"Bulgarian");
  cout << "\n Адрес сочен от p    = " << p;
  cout << "\n Стойност на адреса = " << *p;
  cout << "\n\t Брой серийни портове " << (( *p & 0x1c00) >> 10);
  return 0;
}
```

Резултат



В тази програма се използва това, че битове 9,100 и 11 съдържат броят на серийните портове. С операцията (*p & 0x1c00) се нулират всички битове с изключение на значещите, а с операцията >> се ротира информацията в резултата от предходната операция за да може значещата информация да отиде в младшите адреси.

Поразредни логически допълване до 1 (~)

Побитовата операция **"НЕ"** се записва във вида **~операнд**.

Операндът е от цял или символен тип. Поразрядно се извършва операцията отрицание. Операцията е едноместна – всеки бит 0 в операнда става 1, а всеки бит 1 – 0.

Пример:

Програмен код	Резултат

`int i = 0xAA; // 10101010`
`int c;`
`c = ~i;`
	c = 0x55
	или
	c= 01010101

Операция «изключващо ИЛИ» (XOR)

Операцията "**изключващо ИЛИ**" се отличава от операцията "**ИЛИ**" само в една комбинация при която двата оператора приемат стойност 1 (последен ред от таблицата на истинност). Тоест резултатът ще бъде 1 само тогава когато стойностите на двата оператора се различават.

Операцията "**изключващо ИЛИ**" в булевата алгебра се означава със символа "$\oplus$", а в програмният език C/C++ със символа "**^**" ("**A ^ B**"). Тази операция може да се представи посредством базовите операции ("**НЕ**", "**И**", "**ИЛИ**") със следния израз:

$$A \oplus B = \bar{A}.B + A.\bar{B}$$

Таблица на истинност

	A	B	$A \oplus B$
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	1

Пример с числа

```
   1111 = 15          1000  = 8
^                  ^
   1001 = 9           1010 = 10
   0110 = 6           0010 = 2
```

Побитови операции

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

Поразредно логическо преместване на ляво (<<) и преместване на дясно (>>) на разредни битове

Операторите **<<** и **>>** предизвикват преместване на ляво и съответно преместване надясно на разредите (битовете) на левия операнд с брой позиции, зададени като десен операнд. Например, **x << 2;** предизвиква преместване на всички битове на операнда x с две позиции наляво, като освобождаващите се битове се запълват с нули.

Преместване на ляво (<<)

Аналогично, **x >> 3;** предизвиква преместване на битовете на x с 3 позиции надясно. Когато операторът **>>** се прилага върху число без знак, освобождаващите се битове се запълват с нули.

Преместване на дясно за числа без знак (>>)

Ако числото е със знак, някои компилатори размножават знаковия бит (реализира се аритметично преместване), а други и в този случай запълват освобождаващите се битове с нули. В BORLAND C++ се реализира аритметично преместване надясно.

Преместване на дясно за числа със знак (>>)

Преместването наляво с една позиция е равносилно на умножение по две, а преместването надясно с една позиция е равносилно на деление на две . Тези свойства дават възможност за реализиране на умножение и деление по степените на две чрез операторите `<<` и `>>`. Това води до увеличаване на бързодействието, тъй като тези оператори са изключително бързи.

Пример:

```
unsigned char bits = 1; //    0 0 0 0 0 0 0 1
bits = bits << 1;       // -> 0 0 0 0 0 0 1 0
bits = bits << 2;       // -> 0 0 0 0 1 0 0 0
bits = bits >> 3;       // -> 0 0 0 0 0 0 0 1
```

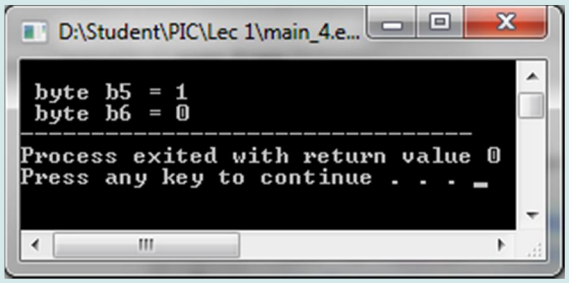
Ето някои характерни приложения на поразредните операции:

```
unsigned x;
//Нулиране на n-тия бит на x
x &= ~(1 << n );
//Установяване в единица на n-тия бит x
x |= 1<< n ;
//Извличане на n-тия бит на x
x & ( 1 << n);
//Извличане на n бита на x, започвайки от p-та позиция
( (x >> (p + 1 - n) ) & ~(~0<< n) );
```

Пример:

Програмен код	Резултат

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
// Деклариране на цели променливи a и b и
// инициализиране на a
  int a=0xAA,b;
// Идентифициране стойността на бит b5
  b = a & 0x20;
// Преместване 5 позиции на дясно
  b >>= 5;
  cout << "\n byte b5 = " << b;
// Идентифициране стойността на бит b6
 b = a & 0x40;
 b >>= 6;// Преместване 6 позиции на
дясно
  cout << "\n byte b6 = " << b;
  return 0;
}
```



Задачи и за самостоятелна работа

1. Определете дали дадено цяло число е четно или нечетно.
2. Дадено е цяло число n. На каколко е равна сумата на неговите цифри в двоична бройна система?
3. Съставете програма за цикличо ротиране на дясно на двоично число. Например: еко е дадено числото 10111 се получава числото 11011.

Контролни въпроси

1. Какво е таблица на истинност?
2. Защо в таблицата на истинност за операцията НЕ има само два реда, докато при останалите операции има повече?
3. В какъв ред обикновено се записват операндите таблицата на истинност?
4. При какви стойности на операндите А и В изразите "А и В"? "А или В" дават истина?
5. Какви електрически схеми могат да се използват за илюстриране на

операциите "И" и "ИЛИ"?

6. Какви знаци се използват за дефиниране на логическите операции "НЕ", "И", "ИЛИ"?
7. Защо операцията "И" се нарича логическо умножение?
8. Защо операцията "ИЛИ" се нарича логическо сумиране?
9. По какво се отличават операциите"изключващо ИЛИ" от "ИЛИ"?
10. Защо операцията "XOR" се нарича сума по модул 2?
11. Как се записва израз $AB\oplus$ с основен набор от операции ("НЕ", "И", "ИЛИ")?
12. Как може да се докаже или опровергае логична формула?
13. Какви интересни свойства притежава операцията "изключващо ИЛИ"?
14. Какво означава терминът "обратима операция"?
15. Какво означава термина „формализиране"?
16. В какъв ред операциите се изпълняват в логически изрази?
17. Какво може да се направи, за да се промени "естественият" начин на действие на логически израз?
18. Какви операции се наричат бинарни и унарни? Дайте примери за унарният и бинарни операции в областта на математиката.
19. Обяснете разликата между термините "логически израз" и "логика функция".
20. Можем ли да кажем, че таблицата за истинност идентифицира еднозначно

а) логически израз;

б) логиката функция.

21. Какво е изчислима Boolean израз?

22. Какво е тавтология? противоречие? Дайте примери.