ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

ИНФОРМАТИКА

част втора

Основи на програмирането на C++

лектор: гл. ас. д-р Стефан М. Панов

Катедра “Информатика”

Лекция 8 (16)

Избрано от C++

Битови оператори (Побитови оператори)

С помощта на битовите оператори (БО) може да се въздейства на отделните битове в рамките на един байт или една машинна дума. БО се използват за широк кръг задачи на ниско (системно) ниво примерно за получаване на информация за състоянието на устройство или за формиране на състоянието на устройството. **Битовите оператори могат да се прилагат само върху целочислени операнди**. (Т.е. не може върху float, double, bool или други сложни типове данни.) На следващата **Таблица 8-1** са показани шестте БО:

Оператор	Действие
&	Битово И (побитово И и т.н.)
|	Битово ИЛИ
^	Битово Изключващо ИЛИ
>>	Битово преместване надясно
<<	Битово преместване наляво
~	Битово инвертиране (унарен оператор НЕ)

Битови оператори И, ИЛИ, Изключващо ИЛИ, унарно НЕ

Изпълняват същите операции, както вече изучаваните техни логически еквиваленти. Т.е. действат съгласно същите таблици за истинност, но на битова основа.

A	B	A & B	A | B	A ^ B	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Табл.8-2 – таблици на истинност на операторите & | ^ ~

Да припомним, че изключващото ИЛИ има стойност ИСТИНА, когато точно един от операндите има стойност ИСТИНА. Унарно НЕ (както показва и името) е единственият битов оператор, който се прилага върху един операнд.

С помощта на битовото И можем да зададем 0 в желания бит (битове) на резултата. В следващия пример_85 всяка въведена от клавиатурата малка буква на латиница се преобразува в съответната ѝ главна буква. Използва се фактът, че ASCII кодът на всяка главна буква е по-малък от ASCII кода на нейната малка буква с 32, което е 0010 0000 в двоична бройна система. Т.е нулирайки шестия бит ще преобразуваме от малка в главна. За да нулираме шестият бит ще направим **битово И** между ASCII кода на малката буква и 11011111 = $223_{(10)}$. Например за буквата "a" с ASCII код 97 ще имаме:

0110 0001 = $97_{(10)}$ = "a"

& 1101 1111

0100 0001 = $65_{(10)}$ = "A"

Операндът, който определя кои битове ще се нулират, се нарича маска.

В нашият пример това е константата 223.

```
using namespace std; // nashata 85-ta programa za preobrazuvane na
#include <iostream> // malki latinski bukvi v glavni chrez pobitovo i
int main() {
	char ch;
	const char Mask = 223; // 223 v dvoichno e 1101 1111
	do {
		system("cls");// izchistva ekrana
		cout << "Vyvedi malka bukva. Za krai izberi drug simvol!\n\n";
		cout << "Bukva:  ";
		cin >> ch;
		if (ch < 97 || ch > 122) break;// ako ne e malka bukva - krai
		ch = ch & Mask; // preobrazuvane v glavna bukva
		cout << "Glavna: "<< ch;
		cin.sync();
		getchar();
	} while (1);
	return 0;
}
```

С помощта на битовото ИЛИ можем да зададем 1 в желания бит (битове) на резултата. Можем да използваме битовото ИЛИ за да преобразуваме програма_76 в нейната противоположност (програма_86), която преобразува главни букви в малки чрез установяване в единица на шестия бит на символа.

За да зададем 1 в шестият бит ще направим **битово ИЛИ** между ASCII кода на главната буква и 0010 0000 = $32_{(10)}$. Например за буквата "D" с ASCII код 68 ще имаме:

0100 0100 = $68_{(10)}$ = "D"

| 0010 0000

0110 0100 = $100_{(10)}$ = "d"

Ако в програма_77 се въведе символ различен от главна буква инструкцията **if** ще има вярно условие. Тогава инструкцията „**break**" ще сработи и ще се излезе от цикъла, което води и до край на програмата.

```
using namespace std; // nashata 86-ta programa za preobrazuvane na
#include <iostream> // glavni latinski bukvi v malki chrez pobitovo ili
int main() {
	char ch;
	const char Mask = 32; // 32 v dvoichno e 0010 0000
	do {
		system("cls");
		cout << "Vyvedi glavna bukva. Za krai izberi drug simvol!\n\n";
		cout << "Bukva: ";
		cin >> ch;
		if (ch < 65 || ch > 90) break;// ako ne e glavna bukva - krai
		ch  |= Mask; // preobrazuvane v malka bukva
		cout << "Malka: "<< ch;
		cin.sync();
		getchar();
	} while (1);
	return 0;
}
```

Унарният оператор НЕ обръща (инвертира) стойностите на всички битове на своя операнд. Ще покажем работата на оператора в програма_87, която намира обратният код на положително еднобайтово число, т.е. число от 0 до 255. Ще се извеждат на екран двоичните еквиваленти както на въведеното число, така и обратният код на числото. Това извеждане се осъществява в отделна функция disp_binary(). Да разгледаме накратко нейния код. Тъй като числото 128 в двоична бройна система е 1000 0000 то битовото И между него и аргумента на функцията ще доведе отново до резултат 128, когато старшият бит на аргумента е 1 и до резултат 0, когато старшият бит на аргумента е 0. Съответно на екрана ще се изведе 1 или 0, т.е. извежда се стойността на самия старши бит. Следващата стойност на **t** е 64 = 0100 0000, което аналогично ще доведе извеждане на стойността на следващия бит и така нататък до извеждане на всичките осем бита.

```
using namespace std; // nashata 87-a programa za namirane na
#include <iostream> // obratnia kod na chislo ot 0 do 255
void disp_binary(unsigned u) ;
int main() {
	unsigned  u;
	cout << "Vyvedete chislo mejdu 0 i 255: ";  cin>> u;
	if (u > 255){
		cout <<"Greshno chislo! Krai!"; return 1;}
	cout << "Syshtoto chislo v dvoichen kod: "; disp_binary(u);
	cout << "Obratniqt kod na chisloto:      "; disp_binary(~u);
	return 0;
}
void disp_binary(unsigned u){
	register int t;
	for(t=128; t>0; t = t/2)
		if(u & t) cout << "1 ";
		else cout << "0 ";
	cout << "\n";
}
```

Оператори за битово преместване

Операторите **">>"** и **"<<"** преместват **всички** битове в стойността надясно или наляво. Прилагат се само върху целочислени стойности. Общият им формат изглежда така:

стойност >> брой_битове

стойност << брой_битове

като **брой_битове** е цяло число указващо, на колко позиции трябва да бъде преместена стойността. При движение наляво левите „**брой_битове**“ бита се губят (изпадат), а останалите битове се преместват наляво „**брой_битове**“ позиции. В младшите „**брой_битове**“ бита се записват нули. При движение надясно десните „**брой_битове**“ бита се губят (изпадат), а останалите битове се преместват надясно „**брой_битове**“ позиции. За старшите „**брой_битове**“ бита е от значение дали стойността, което се обработва е знаково или

беззнаково число. При беззнаково число и знаково положително число в старшите „**брой_битове**“ бита се записват нули. При знаково отрицателно число в старшите „**брой_битове**“ бита се записват единици! (По тоя начин се съхранява знакът на числото). Изгубените битове, без значение от посоката, не могат да бъдат възстановени. Следващата програма_88 нагледно показва резултата от използването на двата оператора за битово преместване. Програмата трябва да бъде пусната веднъж със **signed char** формален параметър на двете функции и втори път с **unsigned char** формален параметър за да се види разликата при движение надясно.

Операторите ">>" и "<<" често се използват когато трябва да се умножи или дели на 2^n (където n е цяло число) защото вършат същата работа като умножението и делението, но много по-бързо. Намират употреба и при обработка на информацията за състояние на външни устройства. Да припомним (от лекция 7), че всички битови оператори без унарно НЕ имат и съставни форми - **&= |= ^= >>= <<=**, примерно **a <<= 2;**

```
using namespace std; // nashata 88-a programa e primer za
#include <iostream> // pobitovo premestvane >> i <<
void disp_binary(signed char i) ;void shift_and_disp(signed char i);
int main() {  signed char k=1;
      shift_and_disp(k);
      shift_and_disp(k=-1);// = 255 ako k e bezznakovo
      return 0; }
void shift_and_disp(signed char i){// probvaite i sys unsigned
      for(int t=0; t<7; t++ ) {
            disp_binary(i); i = i<< 1;
      }  disp_binary(i);cout << "\n";
      for(int t=0; t<7; t++ ) {
            disp_binary(i); i = i>> 1;
      }     disp_binary(i);cout << "\n";
}
void disp_binary(signed char i){ // probvaite i sys unsigned
      for(unsigned char t=128; t>0; t >>= 1) {
            if(i & t) cout << "1 ";  else cout << "0 ";       }
      cout << "   "<<(int)i<<"\n"; }
```

Разбиване на програмата от лекции 4 и 5 на няколко файла

В лекция 4 (12) създадохме една относително голяма програма, която по-късно в лекция 5 (13) беше увеличена с нова функционалност, позволяваща съхранение на данните във и четенето им от файл. Следващата стъпка е да разделим файла на програмата на няколко по-малки файла **(модула)**, където кодът е групиран по функционалност. Едно възможно решение е такова:

- Създаване на един заглавен файл, примерно с име **big_program_in_file.h** където са използваните готови заглавни файлове, определенията на нашите типове, константи и прототипите на нашите функции; **Този файл ще бъде включен в началото на всички останали .cpp файлове от проекта, изброени надолу**.

- Създаване на .cpp файл, примерно с име **file_proc.cpp**, в който да се намира кода на трите функции за файлови операции;

- Сорс-файл (с име **search_fun.cpp**), в който да се намира кода на функциите за търсене по фамилия и среден успех, както и двете функции-компаратори, използвани във функциите qsort и bsearch;
- Сорс-файл с име **prepare_and_print_array.cpp**, където да се намира имплементацията на функциите за извеждане на екран на основния масив с информация за студенти, както и функциите подготвящи масива за използване от другите функции.
- Още един сорс-файл **modify_array.cpp**, където ще се намира функционалността за добавяне, обновяване и премахване на записи със студенти.
- Последният сорс-файл **big_program_in_file.cpp** ще включва кода за изчисляване на средния успех, отпечатване на менюто и кода на **main()**.

Всички изброени файлове лесно се създават като се копира съответния код от голямата програма разисквана подробно в лекции 4 и 5. Следваща стъпка е да се създаде проект, който да включи посочените по-горе файлове и неговата компилация. В новия си вариант програмата е много по-удобна за работа за дебъгване, добавяне на нова функционалност, а също така и за запознаване с кода от страна на други програмисти.

Big-endian и Little-endian

В зависимост от това как стойността на многобайтовите променливи се разполага в паметта, заделена за тях, различаваме **Big-endian** и **Little-endian** компютри.

Little-endian компютрите разполагат стойността на променливата, **започвайки от най-младшия байт на паметта**, заделена за нея.

Big-endian компютрите разполагат стойността на променливата, **започвайки от най-старшия байт на паметта**, заделена за нея.

Фиг.8-1 нагледно показва това за променливата **unsigned short k = 0xABCD;**

Фиг.8-1 - Big-endian и Little-endian

Следващата проста програма 89 показва как можем да проверим каква архитектура е компютърът, на който я изпълняваме.

```
using namespace std; // nashata 89-ta programa e primer kak da
#include <iostream> // razberem dali nashata architektura e
int main() {// Little-endian ili Big-Endian
    int a = 0x12345678;
    unsigned char *c = (unsigned char*)(&a);
    if (*c == 0x78)  cout << "little-endian\n";
    else  cout << "big-endian\n";
    return 0;
}
```

Но и без приведения пример е добре да се запомни, че всички x86 процесори са Little-endian докато Моторола процесорите са Big-Endian.

Квалификатор volatile

Квалификаторът **volatile** информира компилатора, че променливата може да се измени неявно, без намесата на програмни инструкции (т.е. от хардуера).

volatile тип име-променлива;

Такава променлива не участва в оптимизацията на кода. **Например, ако между две четения на една променлива, в кода липсват програмни инструкции, които да я модифицират, компилаторът може да оптимизира кода така, че след като е прочел първия път стойността на променливата да допусне, че тя не е променена и да използва същата стойност.** Ако обаче тази променлива се промени междувременно от хардуера, това ще доведе до логическа грешка в програмата. Ако има вероятност една променлива да бъде променена извън програмата, тя трябва да се дефинира като **volatile**. Тогава компилаторът ще изключи оптимизацията.

Тъй като е трудно да се даде практически пример, ще се даде теоретичен такъв, описващ възможния проблем, ако една променлива не е квалифицирана с **volatile**.

```
int x, y;   x =  10;
.................
// Да предположим, че променливата x се използва от някое
// периферно устройство, което съхранява в нея нова стойност
// примерно 20, и това се случва след x = 10;   но преди y = x;
.................
y = x;  // на y отново се присвоява 10. Причината е, че компилаторът
// допуска, че x не се е променила от последното явно присвояване
// и не чете стойността на x, а ползва  последната стойност, в случая 10.
```

Проблемът за описания случай се решава с volatile, т.е.

```
volatile int x; int  y; ... x = 10;  … y = x;
```

Сега на **y** се присвоява 20. Причината е, че компилаторът не прави никакви допускания и чете всеки път стойността на променливата x.

inline функции

Ако една функция е обявена **inline**, то **по време на компилация** в точката на извикване се извършва заместване на извикването с тялото на функцията.

В обикновения случай (без **inline**) функцията се извиква **по време на изпълнение**. Извикването на функция предизвиква предаване на управлението към нея, а изпълнението на текущата активна функция се преустановява. Когато приключи изчислението на извиканата функция прекъснатата функция продължава изпълнението си от точката, непосредствено следваща повикването. Управлението на извикването на функции се осъществява с помощта на програмния стек, създаван по време на изпълнение.

Причина за съществуването на **inline** функциите е ефективността. При всяко извикване на обикновена функция се изпълнява цялата последователност от инструкции, свързани с обработката на самото извикване. Това включва

разполагане на аргументите на функцията в стека и инструкции, свързани със завръщане от функцията. При малки функции тая последователност е значителна в сравнение със самия код на функцията. Но ако кода на функцията се вгради в програмата по време на компилация въпросната последователност от системни инструкции просто липсва, което води до увеличаване на скоростта на изпълнение на програмата. От друга страна, когато самата функция не е малка, то общият размер на програмата съществено се увеличава, ако функцията се вика на много места. **Затова препоръката е само кратки функции да се обявяват inline**. Ключовата дума **inline** трябва да е **преди** типът на връщания резултат на функцията, пример:

```
inline  int  fun1() {…}
```

Важно е да се знае, че модификаторът inline е заявка, а не команда за компилатора да генерира вграден (**inline**) код. Съществуват различни ситуации, описани в документацията на компилатора, когато той не

удовлетворява заявката. Примерно, ако функцията е рекурсивна или съдържа статични променливи.

Следващата програма_90 е пример за употребата на **inline** функции.

```
using namespace std; // nashata 90-a programa e primer za
#include <iostream> // inline funkcii
int i;
inline int get_i(); inline void put_i(int j);
int main() {
	put_i(12); // ekvivalentno na instrukciata i = 12;
	cout << get_i(); // ekvivalentno na instrukciata cout << i;
	return 0;
}
inline int get_i(){
	return i;
}
inline void put_i(int j) {
	i = j;
}
```

Какво не е предавано в нашия лекционен курс

1. Всичко свързано с ООП.
2. Типът wchar_t;
3. Вход и изход в C.
4. Стек и опашка;
5. Списъци – едностранни и двустранни.
6. Функции за динамично управление на паметта.
7. Инструкции new и delete за динамично управление на паметта.
8. Обработка на изключения в C++.
9. Стандартната библиотека със шаблони STL.