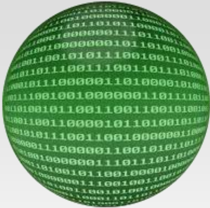


Podstawy programowania w C++

Operatory – AND, OR, NOT, XOR

Opracował: Andrzej Nowak

Bibliografia:
CPA: PROGRAMMING ESSENTIALS IN C++ https://www.netacad.com

Operatory logiczne

Komputery i ich logika

AND - &&

Spójrzmy na następujące zdanie:

Jeśli mamy trochę wolnego czasu i (ang. and) pogoda jest dobra, to pójdziemy na spacer.

Użyliśmy kombinacji "and", co oznacza, że chodzenie na spacer zależy od jednoczesnego spełnienia dwóch warunków

- mamy trochę wolnego czasu;
- pogoda jest dobra.

W języku logiki takie warunki łączenia nazywa się **koniunkcją (iloczynem zdań)**– oznaczenie w języku C++ - **&&** - operator logiczny AND

Pozwala nam kodować złożone warunki bez użycia nawiasów takich jak ten →

```
licznik > 0 && wartość == 100
```

Wynik operatora && może być określony przez tzw. Tablicę prawdy

left	right	Left&&right
false	false	false
false	true	false
true	false	false
true	true	true

OR - ||

A teraz kolejny przykład:

Jeśli jesteś w centrum handlowym lub (ang. or) jestem w centrum handlowym, jeden z nas kupi prezent dla mamy.

Użycie słowa "lub" oznacza, że zakup zależy od co najmniej jednego z tych warunków.

W logice związek taki jak ten nazywany jest **alternatywą (sumą logiczną)** – oznaczenie w języku C++ - **||** - operator logiczny OR.

Jest to operator binarny o priorytecie niższym niż && (podobnie jak "+" w porównaniu do "*"). Jego tabela prawdy wygląda tak: →

left	right	Left||right
false	false	false
false	true	true
true	false	true
true	true	true

NOT - !

Ponadto istnieje inny operator, który może zostać użyty do skonstruowania warunków. To jednoargumentowy operator dokonujący logicznej negacji.

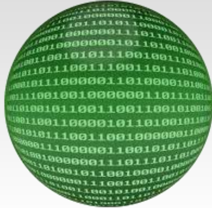
Jego działanie jest proste:

zamienia prawdę w fałsz i fałsz w prawdę.

- oznaczenie w języku C++ - **!** (wykrzyknik) - jego priorytet jest bardzo wysoki: taki sam jak operatorów inkrementacji i dekrementacji.

Tabela prawdy →

arg	! arg
false	true
true	false



Przykłady użycia operatorów logicznych

Zwróć uwagę, że następujące warunki są równoważne sobie nawzajem →

```
Zmienna> 0   !(Zmienna <= 0)
Zmienna != 0 !(Zmienna == 0)
```

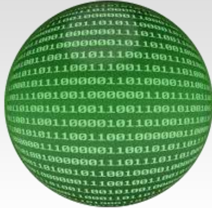
Prawa De Morgana

Negacja koniunkcji (iloczynu logicznego) jest przeciwieństwem negacji.

```
!(p && q) == !p || !q
```

Negacja przeciwieństwa jest połączeniem negacji (iloczynem logicznym negacji)

```
!(p || q) == !p && !q
```



Operatory bitowe AND, OR ,NOT, XOR

Operatory logiczne przyjmują argumenty jako całość, niezależnie od liczby bitów, które zawierają. Operatory znają tylko wartości 0 - fałsz (gdy wszystkie bity są resetowane), lub nie 0 - prawda (gdy ustawiony jest co najmniej jeden bit).

Wynikiem ich operacji jest albo wartość 0 (fałsz), albo 1 (prawda).

Operatory bitowe

& (**AND**) (ampersand) koniunkcja bitowa

wymaga dokładnie dwóch "1", aby w rezultacie uzyskać "1"

| (**OR**)(bar) rozróżnianie bitowe

wymaga co najmniej jednego "1" jako wyniku "1"

^ (**XOR**)(caret) bitowy wyłączny lub (albo)

wymaga dokładnie jednego "1", aby podać "1"

left	right	left|right	left&right	left^right
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

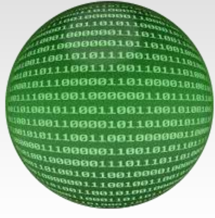
(**NOT**) (tylda) bitowa negacja

dla ”1” jako wynik poda ”0” i odwrotnie dla ”0” poda ”1”

arg	~arg
0	1
1	0

UWAGA:

argumenty tych operatorów muszą być liczbami całkowitymi (int, a także long, short lub char).



Różnica w działaniu operatorów logicznych i bitowych

1. Operatory logiczne nie wnikają do poziomu bitów argumentu. Interesuje je tylko końcowa wartość całkowita.

2. Operatory bitowe są bardziej rygorystyczne: zajmują się każdym bitem osobno. Jeśli przyjmiemy, że zmienna int zajmuje 32 bity, można sobie wyobrazić operację bitową jako 32-krotną ocenę operatora logicznego dla każdej pary bitów argumentów. Ta analogia jest oczywiście niedoskonała, ponieważ w rzeczywistości wszystkie 32 operacje wykonywane są w tym samym czasie.

Sposoby użycia operatorów logicznych i bitowych

Pokażemy teraz przykład różnicy operacji między operacjami logicznymi i bitowymi.

Załóżmy, że wykonano następującą deklarację:

```
int i = 15, j = 22;
```

Jeśli założymy, że typ zmiennej int jest przechowywany jako liczba 32 bitowa, to obraz bitowy dwóch zmiennych będzie wyglądał następująco →

I: 00000000000000000000000000001111
J: 00000000000000000000000000010110

Użycie operatora logicznego - operacja logiczna (AND)

Zapis w C++

```
int log = i && j;
```

Mamy do czynienia z iloczynem logicznym.

Prześledźmy przebieg obliczeń. Obie zmienne i i j nie są zerami, więc będą uważane za reprezentujące "prawdę".

Jeśli sprawdzimy tabelę prawdy dla operatora &&, widzimy, że wynik będzie "prawdą", dla tych wartości, gdzie liczba całkowita jest równą 1.

Oznacza to, że bitowy obraz zmiennej **log** jest następujący: →

LOG: 00000000000000000000000000000001

Użycie operatora bitowego

operacja bitowa (AND)

Zapis w C++

```
int bit = i & j;
```

Operator & będzie działał z każdą parą odpowiednich bitów osobno, tworząc wartości odpowiednich bitów wyniku.

Dlatego wynik jest taki →

bit: 00000000000000000000000000000110

Użycie operatora logicznego

operacja logiczna (NOT)

Zapis w C++

```
int logneg =! i;
```

Zmienna **`logneg`** zostanie ustawiona na 0, więc jej obraz będzie składał się tylko z zer.

Użycie operatora bitowego

operacja bitowa (NOT)

Zapis w C++

```
int bitneg = ~ i;
```

Może cię zaskoczyć fakt, że wartość zmiennej **`bitneg`** wynosi -16. Dziwne? Nie, wcale nie!

logneg: 00000000000000000000000000000000

bitneg: 11111111111111111111111111110000

Możemy użyć każdego z poprzednich dwóch argumentów operatorów w ich skróconej formie. Oto przykłady równoważnych notatek →.

x = x & y;	x &= y;
x = x | y;	x |= y;
x = x ^ y;	x ^= y

Operacje na bitach

Do czego można użyć operatorów bitowych?

PROBLEM

Wyobraź sobie, że musisz napisać ważny element systemu operacyjnego.

1. Powiedziano Ci, że musisz użyć zmiennej zadeklarowanej w następujący sposób: →

int FlagRegister;

Zmienna przechowuje informacje o różnych aspektach działania systemu.

Każdy bit zmiennej przechowuje jedną wartość tak / nie.

2. Powiedziano ci również, że tylko jeden z 32 bitów możesz wykorzystać - **bit numer trzy** (pamiętaj, że bity są ponumerowane od 0, a bit 0 jest najniższy, a najwyższy to 31).

Pozostałe bity nie mogą się zmienić, ponieważ są przeznaczone do przechowywania innych danych. Oto twój bit oznaczony literą "x" →

FlagRegister: 0000000000000000000000000000x000

PRZYKŁAD

1. Sprawdzenie stanu swojego bitu;

porównanie całej zmiennej do zera nie spowoduje niczego, ponieważ pozostałe bity mogą mieć całkowicie nieprzewidywalne wartości, ale możemy użyć następującej właściwości koniunkcji:

```
x & 1 = x

x & 0 = 0
```

Jeśli zastosujemy operację **&** do zmiennej **`FlagRegister`** wraz z następującym obrazem bitowym:

```
00000000000000000000000000001000
```

(zauważ "1" na pozycji twojego bitu), otrzymujemy jeden z następujących łańcuchów bitów:

a. jeśli twój bit był ustawiony na "1",

```
00000000000000000000000000001000
```

b. jeśli twój bit został zresetowany do "0"

```
00000000000000000000000000000000
```

UWAGA:

Ta sekwencja zer i jedynek, której zadaniem jest pobranie wartości lub zmiana wybranych bitów, nazywana jest maską bitową.

2. Tworzenie maski bitowej - aby wykryć stan swojego bitu.

Powinien wskazywać na trzeci bit. Ten bit ma wagę $2^3 = 8$.

Powinniśmy być w stanie stworzyć odpowiednią maskę poprzez deklarację:

```
int TheMask = 8;
```

Możemy również wykonać sekwencję instrukcji w zależności od stanu twojego bitu →

```
if(FlagRegister & TheMask)
{ /* mój bit jest ustawiony */
} else
{ /* mój bit jest resetowany */ }
```

3. Zresetowanie bitu

- przypisujesz zero do bitu, podczas gdy wszystkie inne bity pozostają niezmienione;

- użyjemy tej samej właściwości spójnika jak poprzednio, ale użyjemy nieco innej maski - tak jak poniżej:

```
11111111111111111111111111110111
```

Zwróć uwagę, że maska została utworzona w wyniku negacji wszystkich bitów zmiennej `TheMask`.

Resetowanie bitu jest proste i można wykonać na dwa sposoby (wybierz ten, który wolisz) →

```
FlagRegister = FlagRegister & ~TheMask;
```

```
FlagRegister &= ~TheMask;
```

4. Ustawianie swojego bitu

- przypisujesz "jeden" do swojego bitu, podczas gdy wszystkie inne bity pozostają niezmienione; użyjemy następującego operatora bitowego **|** :

```
x | 1 = 1
x | 0 = x
```

Ustaw swój bit za pomocą jednej z następujących instrukcji →

```
FlagRegister = FlagRegister | TheMask;
```

```
FlagRegister |= TheMask;
```

5. Negacji bitu

- zamienisz "jeden" na "zero" i "zero" na "jeden". Użyj interesującej właściwości operatora xor:

```
x ^ 1 =! x

x ^ 0 = x
```

Dokonaj negacji za pomocą następujących instrukcji →

```
FlagRegister = FlagRegister ^ TheMask;

FlagRegister ^= TheMask;
```

6. Przesuwanie bitu

- ma zastosowanie tylko do wartości całkowitych i nie można go używać z argumentami typu **`float`**.

Operacji tej używa się nieświadomie przez cały czas. Jak pomnożyć dowolną liczbę przez 10? Spójrz:

12345 · 10 = 123450

- jak widzisz, pomnożenie przez dziesięć to tylko przesunięcie wszystkich cyfr w lewo i dodanie "0" w prawo.

Jak podzielić przez 10? Spójrzmy:

12340 ÷ 10 = 1234

- wystarczy przesunąć cyfry w prawo.

UWAGA:

Komputer wykonuje ten sam rodzaj operacji z jedną różnicą:

- ponieważ 2 jest podstawą dla liczb binarnych (nie 10), przesunięcie wartości o jeden bit w lewo odpowiada pomnożeniu przez 2;

- odpowiednio, przesunięcie jednego bitu w prawo jest jak dzielenie przez 2 (zauważ, że najbardziej prawy bit jest tracony).

Przesunięcie bitów może być:

- **logiczne**,

  czy wszystkie bity zmiennej są przesunięte;

  tego rodzaju zmiana ma miejsce, gdy zastosujesz ją do liczb całkowitych bez znaku;

- **arytmetyczne**,

  jeśli przesunięcie pomija tak zwany bit znaku - w notacji uzupełnienia dwójkowego rolę bitu znaku pełni najwyższy bit zmiennej;

  jeśli jest równy "1", wartość jest traktowana jako wartość ujemna; oznacza to, że przesunięcie arytmetyczne nie może zmienić znaku przesuniętej wartości.

Operatorami zmian w języku "C ++" jest para dwuznaków, **<<** i **>>**, wyraźnie wskazująca kierunek, w którym będzie działać zmiana.

Lewy argument tych operatorów jest wartością całkowitą, której bity są przesunięte.

Właściwy argument określa rozmiar przesunięcia. Pokazuje to, że ta operacja z pewnością nie jest przemienna.

Value << Bits

Value >> Bits

Priorytet tych operatorów jest bardzo wysoki.

PRZYKŁAD

Załóżmy, że istnieją następujące deklaracje:

```
int Signed = -8, VarS;

unsigned Unsigned = 6, VarU;
```

spójrz na te zmiany →

```
/* równoważne dzieleniu przez 2 / -> VarS == -4 */
    VarS = Signed >> 1;
/ * odpowiednik mnożenia przez 4 -> VarS == -32 * /
     VarS = Signed << 2;
/* odpowiednik dzielenia przez 4   -> VarU == 1 */
     VarU = Unsigned >> 2;
/* odpowiednik mnożenia przez 2  -> VarU == 12 */
     VarU = Unsigned << 1;
```

Możemy użyć obu operatorów za pomocą poniższego sposobu zapisu:

```
Signed >>= 1; / * dzielenie przez 2 * /

Unsigned <<= 1; / * mnożenie przez 2 * /
```

Tabela priorytetów zawierająca wszystkie operatory

! ~ (type) ++ -- + -	unary
* / %	
+ -	binary
<< >>	
< <= > >=	
== !=	
&	
|	
&&	
||	
= += -= *= /= %= &= ^= |= >>= <<=	