

Operatory

Operatory bitowe i uzupełnienie informacji o pozostałych operatorach.

PRZYPOMNIENIE: OPERATORY

Operator przypisania			
=	przypisanie	$x = y$	$x \leftarrow y$
Operatory arytmetyczne			
*	mnożenie	$x * y$	$x \cdot y$
/	dzielenie	x / y	$\frac{x}{y}$
+	dodawanie	$x + y$	$x + y$
-	odejmowanie	$x - y$	$x - y$
%	reszta z dzielenia (modulo)	$x \% y$	$x \bmod y$
++	inkrementacja	$x++$	$x \leftarrow x + 1$
--	dekrementacja	$x--$	$x \leftarrow x - 1$
Operatory relacji			
<	mniejszy niż	$x < y$	$x < y$
>	większy niż	$x > y$	$x > y$
<=	mniejszy lub równy	$x <= y$	$x \leq y$
>=	większy lub równy	$x >= y$	$x \geq y$
==	równy	$x == y$	$x = y$
!=	różny	$x != y$	$x \neq y$
Operatory logiczne			
!	negacja (NOT)	$!x$	$\neg x$
&&	koniunkcja (AND)	$x > 1 \ \&\& \ y < 2$	$x > 1 \wedge y < 2$
	alternatywa (OR)	$x < 1 \ \ \ y > 2$	$x < 1 \vee y > 2$
Operatory wskaźnikowe			
&	referencja (pobranie adresu)	$\&x$	
*	dereferencja (dostęp pod adres)	$*x$	

operator	znaczenie	przykład
~	negacja bitowa (NOT)	~x
&	koniunkcja bitowa (AND)	x & y
	alternatywa bitowa (OR)	x y
^	alternatywa rozłączna (XOR)	x ^ y

- działają na pojedynczych bitach
- zdefiniowane dla liczb całkowitych
- najbezpieczniej używać wyłącznie dla liczb całkowitych bez znaku (unsigned)

NEGACJA

$\sim$	0	1
1	0	1

KONIUNKCJA (AND)

$\&$	0	1
0	0	0
1	0	1

ALTERNATYWA (OR)

$\mid$	0	1
0	0	1
1	1	1

ALTERNATYWA ROZŁĄCZNA (XOR)

$\wedge$	0	1
0	0	1
1	1	0

```
1  #include <stdio.h>
2
3  int main()
4  {
5      unsigned int a = 9;
6      unsigned int b = 12;
7
8      printf("a      = %u\nb      = %u\n", a, b);
9      printf("~a      = %u\n~b      = %u\n", ~a, ~b);
10     printf("a & b = %u\n", a & b);
11     printf("a | b = %u\n", a | b);
12     printf("a ^ b = %u\n", a ^ b);
13
14     return 0;
15 }
```

a	=	9
b	=	12
~a	=	4294967286
~b	=	4294967283
a & b	=	8
a b	=	13
a ^ b	=	5

SZYFROWANIE XOR

Operacja XOR jest wykorzystywana np. w funkcjach haszujących i algorytmach szyfrujących.

$$A \oplus B \oplus B \longrightarrow A$$

```
1  #include<stdio.h>
2
3  int main()
4  {
5      unsigned char klucz = 13;
6      char znak;
7
8      while( (znak=getchar()) != EOF )
9          putchar( znak ^ klucz );
10
11     return 0;
12 }
```

PRZESUNIĘCIE BITOWE

operator	znaczenie	przykład
<<	przesunięcie bitowe w lewo	x << y
>>	przesunięcie bitowe w prawo	x >> y

- pozycje wszystkich bitów są przesuwane
- bity skrajne są tracone, puste miejsca są zastępowane zerami
- odpowiada do mnożeniu/dzieleniu całkowitemu przez 2

```
1  #include <stdio.h>
2
3  int main()
4  {
5      unsigned int a = 5;
6
7      printf("a          = %u\n", a);
8      printf("a << 1 = %u\n", a << 1);
9      printf("a << 2 = %u\n", a << 2);
10     printf("a >> 1 = %u\n", a >> 1);
11     printf("a >> 2 = %u\n", a >> 2);
12
13     return 0;
14 }
```

a	=	5
a << 1	=	10
a << 2	=	20
a >> 1	=	2
a >> 2	=	1

Maska bitowa: słowo (wartość) używane w operacjach bitowych do ustawienia, zmiany lub odczytu poszczególnych bitów.

Ustawienie pojedynczego bitu na 1 operatorem OR

10011101	10010101
00001000	00001000
= 10011101	10011101

```
int zapal_bit(int x, int n)
{
    return x | 1 << n;
}
```

Włączenie wielu bitów zadaną maską

$x = x \mid \text{MASKA}$

Ustawienie pojedynczego bitu na 0 operatorem AND i negacją

10011101	10010101
& 11110111	11110111
= 10010101	10010101

```
int zgas_bit(int x, int n)
{
    return x & ~(1 << n);
}
```

Wyzerowanie wielu bitów zadaną maską

`x = x & MASKA`

Sprawdzenie wartości bitu operatorem AND

10011101	10010101
& 00001000	00001000
= 00001000	00000000

```
int sprawdz_bit(int x, int n)
{
    return x & 1 << n;
}
```

Sprawdzenie czy bity pokrywają się z maską

```
if ( (x & MASKA) == MASKA ) { ... }
```

Zamiana wartości bitu za pomocą operatora XOR

10011101	10010101
$\wedge$ 00001000	00001000
= 10010101	10011101

```
int zamien_bit(int x, int n)
{
    return x ^ 1 << n;
}
```

Zamiana wielu bitów zadaną maską

$x = x \wedge \text{MASKA}$

Rzutowanie: konwersja wartości z jednego typu na inny typ

- **niejawne**, dokonywane automatycznie

```
int a = 3.14;  
float x = a;
```

- **jawne**, wykonane przy pomocy operatora rzutowania ()

(typ)wartość

```
int a = (int)3.14;  
float b = 3/(float)2;  
char *w = (char *)malloc(10);
```

Przy rzutowaniu z typu bardziej pojemnego do mniej pojemnego możliwa utrata dokładności (nadmiar, *overflow*), taka sytuacja nie jest (zazwyczaj) sygnalizowana

OPERATORY PODSTAWIENIA

Skrócony zapis operacji podstawienia:

$$a = a \bigcirc b \quad \Leftrightarrow \quad a \bigcirc = b$$

$a += b$	$a = a + b$	$a = b$	$a = a b$
$a -= b$	$a = a - b$	$a \&= b$	$a = a \& b$
$a *= b$	$a = a * b$	$a ^= b$	$a = a ^ b$
$a /= b$	$a = a / b$	$a <<= b$	$a = a << b$
$a \% = b$	$a = a \% b$	$a >>= b$	$a = a >> b$

Uwaga na kolejność, zamiana nie zawsze powoduje błąd:

$a -= b;$	$/* \text{ OK } */$
$a = -b;$	$/* \text{ czy poprawnie? } */$

INKREMENTACJA I DEKREMENTACJA

- Jednoargumentowy operator inkrementacji ++ i operator dekrementacji --

++a	pre-inkrementacja, to samo co a=a+1
a++	post-inkrementacja
--a	pre-dekrementacja, to samo co a=a-1
a--	post-dekrementacja

- Operator post-inkrementacji i post-dekrementacji zwracają wartość poprzednią (przed zwiększeniem/zmniejszeniem). Zmiana wartości argumentu odbywa się później (po wyliczeniu całego wyrażenia).

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int a = 1;
6      int b;
7
8      b = ++a;
9      printf("a=%d, b=%d\n", a, b);
10
11     b = a++;
12     printf("a=%d, b=%d\n", a, b);
13
14     b = --a;
15     printf("a=%d, b=%d\n", a, b);
16
17     b = a--;
18     printf("a=%d, b=%d\n", a, b);
19
20     return 0;
21 }
```

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int a = 1;
6      int b;
7
8      b = ++a;
9      printf("a=%d, b=%d\n", a, b);
10
11     b = a++;
12     printf("a=%d, b=%d\n", a, b);
13
14     b = --a;
15     printf("a=%d, b=%d\n", a, b);
16
17     b = a--;
18     printf("a=%d, b=%d\n", a, b);
19
20     return 0;
21 }
```

a=2, b=2 a=3, b=2 a=2, b=2 a=1, b=2
--

PRZYKŁAD: KOPIOWANIE NAPISU

Kopiowanie napisu z tablicy a do b

```
void strcpy(char *a, char *b)
{
    while( *a != '\0' )
    {
        *b = *a;
        a = a + 1;
        b = b + 1;
    }
}
```

Dokładnie to samo

```
void strcpy(char *a, char *b)
{
    while(*b++ = *a++);
}
```

OPERATOR WYRAŻENIA WARUNKOWEGO

OPERATOR WARUNKOWY ?:

wyrażenie ? wartość 1 : wartość 2

- jeżeli *wyrażenie* jest prawdą to wynikiem jest *wartość 1*, w przeciwnym wypadku wynikiem jest *wartość 2*
- jedyny operator trzyargumentowy

PRZYKŁAD

Wyznaczanie większej wartości z 2 liczb:

$a = (b > c) ? b : c;$

Wartość absolutna:

$a = (a < 0) ? -a : a;$

- przecinek, wyrażenie rozdzielone przecinkiem wykonywane są od lewej do prawej. Wynikiem jest ostatnia instrukcja.

```
a = 1, b = 2, c = 3;
```

- dostęp do elementów tablicy []

```
a = t[i+1];
```

- wywołanie funkcji ()
- grupowanie wyrażeń ()

```
a = (b + c)/(b - c);
```

- dostęp do pól struktur . (kropka) i ->

```
a = student.nazwisko;  
b = wskaznik->nazwisko
```

PRIORYTETY I KOLEJNOŚĆ OPERATORÓW

- **Priorytet** operatora decyduje o kolejności wykonywania operacji, np. mnożenie przed dodawaniem.

```
a = 3;
```

```
x = a + a * a;
```

- **Łączność** operatora (lewostronna lub prawostronna) decyduje o kolejności wykonywania obliczeń dla operatorów o takim samym priorytecie

```
a = 1;
```

```
x = a - a - a;
```

Operator		Łączność
↑ priorytet	()	
	[] . -> () ++ --	lewostronna
	! ~ + - * & sizeof() ++ -- ()	prawostronna
	* / %	lewostronna
	+ -	lewostronna
	<< >>	lewostronna
	<<= >>=	lewostronna
	== !=	lewostronna
	&	lewostronna
	^	lewostronna
		lewostronna
	&&	lewostronna
		lewostronna
	? :	prawostronna
	= += -= *= /= %= &= = ^=	prawostronna
	,	lewostronna

```
int a, b=1, c=2;  
a = -b++ + ++c << 3 / 2 * (int)2.5 ;  
printf("a=%d, b=%d, c=%d\n", a ,b ,c);
```

Jaki będzie wynik?

```
int a, b=1, c=2;  
a = -b++ + ++c << 3 / 2 * (int)2.5 ;  
printf("a=%d, b=%d, c=%d\n", a ,b ,c);
```

Jaki będzie wynik?

a=8, b=2, c=3

- ❶ `b++` zwiększa wartość `b` na 2 ale zwraca 1
`a = -1 + ++c << 3 / 2 * (int)2.5 ;`
- ❷ `++c` zwiększa wartość `c` na 3, rzutowanie `(int)2.5` daje 2
`a = -1 + 3 << 3 / 2 * 2 ;`
- ❸ dzielenie `3/2` zwraca 1, które następnie jest mnożone przez 2 (łączność lewostronna)
`a = -1 + 3 << 2 ;`
- ❹ `a = 2 << 2 ;`
- ❺ `a = 8 ;`

W razie wątpliwości co do kolejności obliczeń używaj nawiasów grupujących (mają najwyższy priorytet)

- 🌐 Stephan Brumme, „*the bit twiddler*”,
<http://bits.stephan-brumme.com/>
- 🌐 Alex Aliain, „*Bitwise Operators in C and C++: A Tutorial*”,
http://www.cprogramming.com/tutorial/bitwise_operators.html