Operatory

Operatory bitowe i uzupełnienie informacji o pozostałych operatorach.

Ostatnia aktualizacja: 2 lutego 2026

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`	$x \mod 2$
++	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`	$x \leqslant y$
>=	większy lub równy	`x >= y`	$x \geqslant y$
==	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`	

Operatory bitowe

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 liczby w jej reprezentacji binarnej
- są zdefiniowane dla liczb całkowitych
- w praktyce używane najczęściej dla liczb całkowitych bez znaku (np. `unsigned int`)

Negacja

~	0	1
	1	0

Koniunkcja (AND)

&	0	1
0	0	0
1	0	1

Alternatywa (OR)

|	0	1
0	0	1
1	1	1

Alternatywa rozłączna (XOR)

^	0	1
0	0	1
1	1	0

Przykład

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

```c
#include <stdio.h>

int main()
{
    unsigned int a = 9;
    unsigned int b = 12;

    printf("a     = %u\nb     = %u\n", a, b);
    printf("~a    = %u\n~b    = %u\n", ~a, ~b);
    printf("a & b = %u\n", a & b);
    printf("a | b = %u\n", a | b);
    printf("a ^ b = %u\n", a ^ b);

    return 0;
}
```

☞ bitowe1.c

Przykład

```
a              9     00000000000000000000000000001001
~a    4294967286     11111111111111111111111111110110

a              9     00000000000000000000000000001001
b             12     00000000000000000000000000001100
a & b          8     00000000000000000000000000001000

a              9     00000000000000000000000000001001
b             12     00000000000000000000000000001100
a | b         13     00000000000000000000000000001101

a              9     00000000000000000000000000001001
b             12     00000000000000000000000000001100
a ^ b          5     00000000000000000000000000000101
```

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

$$\texttt{A ^ B ^ B} \longrightarrow A$$

```c
#include<stdio.h>

int main()
{
    unsigned char klucz = 13;
    char znak;

    while ((znak = getchar()) != EOF)
        putchar(znak ^ klucz);

    return 0;
}
```

☞ szyfr–xor.c

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

Przykład

```c
#include <stdio.h>

int main()
{
    unsigned int a = 5;

    printf("a       = %u\n", a);
    printf("a << 1 = %u\n", a << 1);
    printf("a << 2 = %u\n", a << 2);
    printf("a >> 1 = %u\n", a >> 1);
    printf("a >> 2 = %u\n", a >> 2);

    return 0;
}
```

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

☞ bitowe2.c

Przykład

```
a       = 5         00000000000000000000000000000101
a << 1  = 10        00000000000000000000000000001010
a << 2  = 20        00000000000000000000000000010100

a       = 5         00000000000000000000000000000101
a >> 1  = 2         00000000000000000000000000000010
a >> 2  = 1         00000000000000000000000000000001
```

Maska bitowa: słowo bitowe (liczba) 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 | 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
  ^ 00001000    00001000
  = 10010101    10011101
```

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

Zamiana wielu bitów zadaną maską

```
x = x ^ 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

Skrócony zapis operacji podstawienia:

`a = a ◯ b` $\Leftrightarrow$ `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`	`a >>= b`	`a = a >> b`

Uwaga na kolejność zapisu, zamiana nie zawsze prowadzi do błędu składniowego

```
a -=b;        /*  OK  */
a =-b;        /*  czy poprawnie? */
```

- Jednoargumentowy operator inkrementacji ++ i operator dekrementacji --

`++a`	pre-inkrementacja, to samo co `a=a+1`
`a++`	post-inkrementacja
`--a`	pre-dekrementacji, to samo co `a=a-1`
`a--`	post-dekremenatacji

- Operator post-inkrementacji i post-dekrementacji **zwracają wartość poprzednią** (sprzed zwiększenia/zmniejszenia). Zmiana wartości argumentu odbywa się później (po wyliczeniu całego wyrażenia).

Przykład

```c
#include <stdio.h>

int main()
{
    int a = 42;
    int b;

    b = ++a;
    printf("a=%d, b=%d\n", a, b);

    b = a++;
    printf("a=%d, b=%d\n", a, b);

    b = --a;
    printf("a=%d, b=%d\n", a, b);

    b = a--;
    printf("a=%d, b=%d\n", a, b);

    return 0;
}
```

Przykład

```c
#include <stdio.h>

int main()
{
    int a = 42;
    int b;

    b = ++a;
    printf("a=%d, b=%d\n", a, b);

    b = a++;
    printf("a=%d, b=%d\n", a, b);

    b = --a;
    printf("a=%d, b=%d\n", a, b);

    b = a--;
    printf("a=%d, b=%d\n", a, b);

    return 0;
}
```

```
a=43, b=43
a=44, b=43
a=43, b=43
a=42, b=43
```

Kopiowanie napisu z tablicy a do b

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

Dokładnie to samo

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

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 z 3 argumentami

Przykład

Wyznaczanie większej wartości z 2 liczb:

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

Wartość absolutna:

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

- przecinek pozwala na wykonanie kilku wyrażeń (od lewej do prawej), w miejscu gdzie oczekiwane jest jedno wyrażenie. Wynikiem jest wynik ostatniej instrukcji.

```
a = 1, b = 2, c = 3;
int a = (x = 5, y = 10, x + y);
for (i = 0, j = 10; i < j; i++, j--)
```

- dostęp do elementów tablicy []

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

- wywołanie funkcji ()
- grupowanie wyrażeń ()

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

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

```
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;
```

priorytet

Operator	Łączność
`()` wywołanie funkcji, przyrostkowe	
`[] . -> () ++ --`	lewostronna
`!` `~` `+` `-` `*` `&` `sizeof()` `++` `--` `()` jednoargumentowe, przedrostkowe, rzutowanie	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

```
int a, b=1, c=2;
a = -b++ + ++c << 3 / 2 * (int)2.5 ;
```

1. b++ zwiększa wartość b na 2 ale zwraca 1
   ```
   a = -1 + ++c << 3 / 2 * (int)2.5 ;
   ```
2. ++c zwiększa wartość c na 3, rzutowanie (int)2.5 daje 2
   ```
   a = -1 + 3 << 3 / 2 * 2 ;
   ```
3. dzielenie 3/2 zwraca 1, które następnie jest mnożone przez 2 (łączność lewostronna)
   ```
   a = -1 + 3 << 2 ;
   ```
4. ```
   a = 2 << 2 ;
   ```
5. ```
   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`