

Programowanie imperatywne

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

<http://home.agh.edu.pl/~pszwed/>

Aktualizacja: 04.05.2020

9. Operator

Operatory – wprowadzenie (1)

- Operatory są szczególnymi funkcjami, które odwzorowują wartości argumentów w odpowiednie zbiory wartości oraz mogą modyfikować wartości argumentów. Zjawisko to nosi nazwę *efektów ubocznych*.
- Czasem efekt uboczny jest zasadniczym zadaniem operatora, np.: operatory przypisania i inkrementacji są stosowane głównie ze względu na efekty uboczne.
- Wywołując operatory nie posługujemy się składnią wywołań funkcji, ale stosujemy składnię przypominającą notację matematyczną. Odpowiednio skonstruowane ciągi operatorów i argumentów nazywane są *wyrażeniami*.
- Wywołanie w odpowiedniej kolejności operatorów składających się na wyrażenie nazywane jest procesem *obliczania wyrażeń*.
- Proces ten jest wieloetapowy. Wpierw obliczane są podwyrażenia, potem wyrażenia wyższego poziomu.

Operatory – wprowadzenie (2)

Przykład

- Wyrażenie $x = y/3$ jest obliczane jako $v=y/3$; $x = v$.

Wyrażenie	Wartość	efekt uboczny
$y/3$	v ma wartość $y/3$	brak
$x = v$	v	x ma wartość v

- Pewne wyrażenia mogą być obliczane w trakcie kompilacji. Są to tzw. wyrażenia stałe.
`2.0/3`, `sizeof(double)`
- Wyrażenie może nie zawierać operatorów, a jedynie składać się z pojedynczego argumentu (identyfikatora zmiennej lub stałej). Wartością wyrażenia jest wówczas wartość argumentu.
- W C++ można definiować własne operatory i wołać je stosując składnię wywołań funkcji.
- Dodatkowo, symbole pełniące w C funkcje znaków interpunkcyjnych w C++ mogą być operatorami `[] ()`.

Klasyfikacja operatorów (1)

Ze względu na liczbę argumentów operatory można podzielić na:

- jednoargumentowe (ang. *unary*)

++ ! ~

- dwuargumentowe (ang. *binary*)

= == + - / *

- trójargumentowe (ang. *ternary*)

? :

Klasyfikacja operatorów (2)

Operatory jednoargumentowe

– ~ !	Negacja i uzupełnienie
* &	Dereferncja i pobranie adresu
sizeof	Rozmiar pamięci obiektu lub typu
+	Plus jednoargumentowy (+x)
++ --	Inkrementacja i dekrementacja (jednoargumentowy operator przypisania)
(type)	Rzutowanie (zmiana typu)

Klasyfikacja operatorów (3)

Operatory dwuargumentowe

=	Operatory przypisania
* / %	Multiplikatywne
+ -	Addytywne
<< >>	Przesunięcia bitowe
< > <= >= == !=	Relacyjne
& ^	Bitowe
&&	Logiczne
,	Operator obliczania ciągu wyrażeń

Klasyfikacja operatorów (4)

Inna klasyfikacja – podział na:

- Operatory **arytmetyczne**: dodawanie, odejmowanie, mnożenie i dzielenie. Zwracają wartości innych typów niż logiczne:

+ - * / % & | << >> ~

- Operatory **relacyjne**: porównują argumenty określonego typu i zwracają wartości logiczne:

== != < > <= >=

- Operatory **logiczne**: ich argumentami są wartości (wyrażenia) logiczne. Zwracają wartości logiczne:

! && ||

Priorytety operatorów

Operatory mają określone priorytety. W zależności od relacji priorytetów $op1$ i $op2$ wyrażenie:

$$arg1 \ op1 \ arg2 \ op2 \ arg3$$

może być obliczane jako

$$(arg1 \ op1 \ arg2) \ op2 \ arg3$$

albo

$$arg1 \ op1 \ (arg2 \ op2 \ arg3)$$

Najwyższy priorytet mają operatory

- `[]` `()` `.` `++`, `--`, `*`, `&`
- rzutowania
- operatory multiplikatywne
- operatory addytywne
- relacyjne
- bitowe
- przypisania

Kolejność jest tak dobrana, aby w typowych przypadkach można było nie używać nawiasów. W sumie 16 poziomów priorytetów. 9

Łączność operatorów

Operatory w wyrażeniach mogą łączyć się z symbolami po lewej lub prawej stronie (łączność lewo- i prawostronna):

Operator	Łączność	Operator	Łączność
Dostęp do tablicy []	L	Adres &	P
Wywołanie funkcji ()	L	Dereferencja *	P
Dostęp do pola struktury . ->	L	Unarny plus(minus) + -	P
Inkrementacja, dekrementacja postfiksowa i++ z--	L	Inkrementacja, dekrementacja prefiksowa ++i --z	P
Multiplikatywne * / %	L	Uzupełnienie logiczne ~	P
Addytywne + -	L	Rzutowanie (type)	P
Przesunięcia bitowe << >>	L	Negacja !	P
Relacyjne > >= < <=	L	sizeof	P
Równość i nierówność == !=	L	Warunkowy ?:	P
Koniunkcja (alternatywa) logiczna i bitowa && &	L	Przypisanie = *= +=	P

Kolejność obliczania wyrażeń (1)

- Priorytety i łączność operatorów mają wpływ na kolejność obliczania wyrażeń. Większość operatorów arytmetycznych ma łączność lewostronną, więc wyrażenia na ogół są obliczane od lewej do prawej.

```
int main() {
    int*p[10];
    int x;
    p[0]=&x;
    *p[0]=1234; // [] ma priorytet 1, * ma priorytet 2
    printf("%d\n",x);
    return 0;
}
```

Analiza `*p[0]=1234`

- Operator `[]` ma najwyższy priorytet 1 i łączność lewostronną. Wyznaczane jest `(p[0])`
- Operator `*` ma priorytet 2 i łączność prawostronną: wyznaczane jest `*(p[0])`

Operator `=` ma łączność prawostronną. Wpierw obliczane jest wyrażenie `1234`

Kolejność obliczania wyrażeń (2)

- Wynikający ze specyfikacji języka porządek obliczania wyrażeń może zostać zmieniony poprzez zastosowanie nawiasów.
- Określenie kolejności zgodnie ze specyfikacją priorytetów i łączności może być postrzegane, jako wstawianie nawiasów do wyrażenia.

Przykład:

- K.N. King w *C programming...* dowodzi, że:

`a = b += c++ -d + --e / -f`

jest równoważne:

`(a = (b += (((c++) - d) + ((--e) / (-f))))))`

Kolejność obliczania wyrażeń (3)

- Nawet obecność nawiasów nie pozwala jednak na jednoznaczne określenie rzeczywistej kolejności obliczania wyrażeń.
- W przypadku wyrażeń postaci :
$$\text{expr}_1 \text{ op } \text{expr}_2 \text{ op } \text{expr}_3 \text{ op } \dots \text{ op } \text{expr}_n$$
(gdzie **op** jest operatorem mnożenia lub dodawania) nie gwarantuje się, że podwyrażenia obliczane są od lewej do prawej. Teoretycznie, wynik wyrażenia (sumy, iloczynu) nie zależy od kolejności wyznaczania składników (czynników).
- Przykład: wyrażenie $(a+b) * (c+d+e)$ może ze względu na przemienność i łączność działań zostać obliczone jako:
 - $(a+b) * ((c+d) + e)$
 - $(a+b) * (c + (d+e))$
 - $((c+e) + d) * (a+b)$
 - itd..
- Nawiasy nie rozstrzygną, czy wpieryw obliczone zostanie $(a+b)$, czy $(c+d+e)$.

Nieokreślone zachowanie

- Nieokreślone zachowanie (ang. *undefined behavior*) to termin opisujący, sytuację, kiedy rezultat wykonania instrukcji nie może zostać ustalony na podstawie specyfikacji.
- Zazwyczaj oczekuje się, że dla założonych wartości zmiennych przed wykonaniem instrukcji, osiągnięte zostaną zdefiniowane wartości końcowe.

Przykład:

$a = 5;$

$c = (b = a + 2) - (a = 1);$

- Jeżeli w pierw wykonamy $(b = a + 2)$, a następnie $(a = 1)$, wówczas wynikiem będzie 6.
- Jeżeli w pierw wykonamy $(a = 1)$, wynikiem będzie 2.

Konstruując program oczekujemy jednoznacznego przepisu:

$$a = 5 \xrightarrow{\text{instrukcja}} a = 1 \wedge b = 7 \wedge c = 6$$

Nieokreślone zachowanie to niejednoznaczna specyfikacja:

$$a = 5 \xrightarrow{\text{instrukcja}} (a = 1 \wedge b = 7 \wedge c = 6) \vee (a = 1 \wedge b = 3 \wedge c = 2)$$

Zachowanie zależne od implementacji

- Standard języka C pozostawia pewne zasady interpretacji kodu programu niewyspecyfikowane, wskazując, że brakujące szczegóły powinien określić twórca kompilatora/linkera na podstawie rzeczywistej implementacji.
- Przykładem jest dzielenie liczb ujemnych. W standardzie C89 wynik dzielenia $-9/7$ jest pozostawiony implementacji (może wynosić -1 lub -2). Takie wartości zwracają rozkazy arytmetyczne procesorów.
- Mimo ewentualnych rozbieżności zawsze spełnione jest
$$a = (a / b) * b + a \% b$$
- W standardzie C99 wynik jest zaokrąglany w kierunku zera (bo tak działała większość procesorów w momencie, kiedy standard powstawał).

Punkty sekwencji

Punkt sekwencji to etap w wykonaniu programu, dla którego zagwarantowane jest, że wszystkie efekty uboczne wcześniejszych obliczeń zostały zrealizowane i nie pojawiły się efekty uboczne późniejszych obliczeń.

```
int x=0;

int add7(){
    x=x+7;
    return x;
}

int mul7(){
    x=x*7;
    return x;
}
```

```
int main(){ // P1
    int y;
    y=add7();
    y+=mul7();
    printf("%d\n",y);
}
```

```
int main(){ // P2
    int y;
    y=add7();
    y+=mul7();
    printf("%d\n",y);
}
```

Wynik wykonania programu P1 jest nieokreślony, ponieważ zależy od kolejności obliczania argumentów + w funkcji main(). Program może wypisać 56 lub 693. **Operator + nie jest punktem sekwencji.**

Program P2 wypisze 56. Średnik (koniec instrukcji) jest punktem sekwencji.

Punkty sekwencji

- Lewy operand operatora logicznego koniunkcji &&
`int x=1;`
`int z = x--&&++x; // z=1`
- Lewy operand operatora logicznego alternatywy ||
`z = (z=1) || (z=getchar()); // z=1`
- Lewy operand operatora sekwencji (przecinka)
`int y=1;`
`y++,y*=2; // y==4`
- Operator wywołania funkcji. Wszystkie argumenty zostaną obliczone przed wywołaniem funkcji. Kolejność obliczania argumentów jest nieokreślona.
- Pierwszy argument operatora warunkowego.
`int x=0;`
`int z = x++?++x:--x; // z==0`

Punkty sekwencji

- Wyrażenie służące do inicjalizacji
`int x=0;`
`int y = 1;`
`int z = ++x*(y++)*(x--), v=++x; //z=1, v=1`
- Instrukcja będąca wyrażeniem (osiągnięcie średnika)
- Selekcja, czyli wyrażenie warunkowe w `if` lub sterujące w `switch`.
Będzie ono w pełni obliczone wraz z efektami ubocznymi przed przejściem do zaleźnego kodu.
- Wyrażenie warunkowe sterujące iteracją w pętlach `while` i `do-while`.
- Każdy z trzech wyrażen pojawiających się w pętli `for` (inicjalizacja, warunek, modyfikacja)
- Wyrażenie pojawiające się po instrukcji `return`.

Najlepiej unikać zbyt złożonych wyrażen z niejasnymi efektami ubocznymi. Zwięzły zapis wyrażen nie ma wpływu na optymalizację kodu wynikowego, natomiast bardzo utrudnia debuggowanie.

Kolejność wyrażen – operatory logiczne

- Dla operatorów koniunkcji `&&` i alternatywy `||` :
 - Podwyrażenia są obliczane od lewej do prawej
 - Obliczany jest minimalny zbiór podwyrażen, na podstawie którego można określić wartość całego wyrażenia.

Przykład 1

```
int x = 0;
int y = 1;
if(x && x++); // x++ nie jest obliczane
if(!x && x++); // x++ jest obliczane, wyrażenie false
if(y || x++); // x++ nie jest obliczane
if(++x || y++); // y++ nie jest obliczane
if(x || y++); // y++ jest obliczane
```

Kolejność wyrażeń – operatory logiczne

Przykład 2

```
void isEmpty11(const char*txt){
    if(txt){
        // mozna bezpiecznie czytać txt[0]
        if(txt[0]!=0) return 0;
    }
    return 1; // txt ==0 or txt[0]==0
}

void isEmpty12(const char*txt){
    if(txt && txt[0]!=0)return 0;
    return 1; // txt ==0 or txt[0]==0
}

void isEmpty21(const char*txt){
    if(!txt)return 1;
    // mozna bezpiecznie czytać *txt
    if(!*txt)return 1;
    return 0; //
}

void isEmpty22(const char*txt){
    if(!txt || !*txt)return 1;
    return 0; //
}
```

Operatory przypisania (1)

Operatory przypisania są stosowane ze względu na efekty uboczne. Podstawowym operatorem przypisania jest operator `=`.

Jego wywołanie ma postać:

```
lvalue = rvalue
```

Operator ten kopiuje wartość wyrażenia `rvalue` do miejsca określonego przez wyrażenie `lvalue`.

- `rvalue` może być identyfikatorem zmiennej lub stałą.
- `lvalue` musi specyfikować obiekt, któremu przydzielono pamięć (np.: zmienną, element tablicy, obiekt, któremu przydzielono pamięć dynamicznie).

Jako `lvalue` nie można użyć stałej, identyfikatora tablicy, nie może też być wyrażeniem, które po obliczeniu zwraca wartość obiektu.

Operatory przypisania (2)

Przykład

```
#define V 0
v= 7; // błąd 0 = 7

int table1[]={1,2,3};
int table2[10]={4,5,6};

*table1 = 7;      // ok.
table1 = table2; // błąd nie wolno modyfikować table1

(double)table1[1]=2.7;
/* błąd, (double)table1[1] jest obliczane jako 2.0 */

table1[1]=(int)2.7;
/* ok, rzutujemy r-value na typ l-value, tracimy miejsca po
przecinku*/

table1[7] = 4; /* poprawne składniowo, niepoprawne semantycznie */

*(unsigned*)& table1[0] = 0xffffffff;
/* poprawne, ale wartością table1[0] będzie -1 */
```

Operatory przypisania

Dla zmiennych będących strukturami operator przypisania kopiuje kolejne bajty (bity).

```
typedef struct {double re,im} Complex ;  
Complex a,b;  
a.re=2.1;  
a.im=3.7;  
b=a;  
printf("( %e %e)"b.re,be.im);
```

Operatory przypisania (3)

Złożone operatory przypisania mają postać

$$lvalue \ OP = \ rvalue,$$

gdzie $OP \in \{ * / \% + - \ll \gg \& \wedge | \}$

Operator	Opis
$a += b$	Dodaje do wartości a wartość b i zapisuje wynik w a .
$a -= b$	Odejmuje b od wartości a i zapisuje wynik w a .
$a \% = b$	Dzieli wartość a modulo b i zapisuje wynik w a .
$a \ll = b$	Przesuwa bity a w lewo o b miejsc i zapisuje wynik w a .
	Itd...

Operator $a += b$ nie w każdym przypadku jest równoważny $a = a + b$.

Czasem a jest wyrażeniem, którego obliczenie wywołuje dodatkowe efekty uboczne.

Operatory przypisania (4)

Przykład

```
int main() {
    int tab[]={1,2,3,4};
    int*p=tab;
    int i;
    * (p+=1) +=1;
    printf("%d\n",p-tab);
    for(i =0;i<4;i++)printf("%d ",tab[i]);
    return 0;
}
```

```
1
1 3 3 4
```

Dodaje 1 do tab[1],
przesuwa p o 1

```
int main() {
    int tab[]={1,2,3,4};
    int*p=tab;
    int i;
    * (p+=1) =* (p+=1) +1;
    printf("%d\n",p-tab);
    for(i =0;i<4;i++)printf("%d ",tab[i]);
    return 0;
}
```

```
gcc
2
1 4 3 4
```

Zapewne gcc określa
miejsce wstawienia
analizując lvalue przed
dokonaniem przypisania?

```
Visual Studio 2010
2
1 2 4 4
```

Operator przypisania - struktury

Argumentami operatora przypisania mogą być zmienne strukturalne. W tym przypadku działanie będzie identyczne do `memcpy()` – funkcji kopiującej blok pamięci.

```
struct osoba1{
    char imie[32];
    char nazwisko[32];
    char pesel[12];
};

int main(){
    struct osoba1 jk = {"Jan","Kowalski","12070404723"};
    struct osoba1 jk2;
    jk2 = jk;
    printf("%s %s,%s\n",jk2.imie, jk2.nazwisko, jk2.pesel);
    struct osoba1 jk3;
    memcpy(&jk3,&jk,sizeof(jk));
    printf("%s %s,%s\n",jk3.imie, jk3.nazwisko, jk3.pesel);
}
```

```
Jan Kowalski,12070404723
Jan Kowalski,12070404723
```

Operator przypisania - struktury

Jeżeli struktury zawierają wskaźniki do przydzielonej dynamicznie pamięci (funkcja `malloc`), programista odpowiada za jej zwalnianie (funkcja `free`).

- Pamięć powinna być zawsze zwalniana
- Pamięć powinna być zwalniana **tylko raz**

```
struct osoba2{
    char *imie;
    char *nazwisko;
    char pesel[12];
};
void wypelnij_osobe(struct osoba2*po, const char* im,
                   const char*naz, const char*p){
    po->imie = strdup(im);
    po->nazwisko = strdup(naz);
    strcpy(po->pesel,p);
}
void zwolnij_osobe(struct osoba2*po){
    if(po->imie)free(po->imie);
    if(po->nazwisko)free(po->nazwisko);
}
```

Operator przypisania - struktury

```
int main(){
    struct osoba2 jk;
    wypelnij_osobe(&jk, "Jan", "Kowalski", "12070404723");
    struct osoba2 an = jk;
    printf("%s %s,%s\n", an.imie, an.nazwisko, an.pesel);
    zwolnij_osobe(&jk);
    zwolnij_osobe(&an);
}
```

Pola imie i nazwisko zmiennych strukturalnych jk i an wskazują tę samą pamięć. Jest ona zwalniana dwa razy. **Błąd wykonania (wyjątek).**

```
.77
.78 void zwolnij_osobe(struct osoba2*po){ po: 0x6
.79     if(po->imie)free(po->imie);
.80     if(po->nazwisko)free(po->nazwisko);
.81 }
.82
```

Funkcje zwracające struktury

Jeżeli funkcja zwraca strukturę, jej wynik może zostać użyty do przypisania wartości innej strukturze.

```
struct osoba2 utworz_osobe(const char* im, const char*naz, const char*p ){
    struct osoba2 o;
    o.imie=strdup(im);
    o.nazwisko=strdup(naz);
    strcpy(o.pesel,p);
    return o;
}
```

```
int main(){
    struct osoba2 jk = utworz_osobe("Jan", "Kowalski", "12070404723");
    struct osoba2 an = jk;
    printf("%s %s,%s\n",an.imie, an.nazwisko, an.pesel);
    zwolnij_osobe(&an);
}
```

Struktura zwracana przez funkcję nie jest lvalue. Nie można np. pobrać jej adresu. Poniższy kod jest błędny:

```
struct osoba2 *po =
&utworz_osobe("Jan", "Kowalski", "12070404723");
```

Funkcje zwracające struktury

Zwracanie zmiennych strukturalnych jest dyskusyjne.

- W języku C++ zostało zoptymalizowane i jest równie efektywne, jak przekazywanie wskaźnika do zewnętrznego obiektu.
- W języku C na ogół preferuje się funkcje zwracające wskaźniki do zmiennych strukturalnych, dla których przydzielono pamięć na stacku.

```
struct osoba2*utworz_osobe(const char* im, const char*naz, const char*p ){
    struct osoba2*po = malloc(sizeof(struct osoba2));
    po->imie=strdup(im);
    po->nazwisko=strdup(naz);
    strcpy(po->pesel,p);
    return po;
}
```

```
void zwolnij_osobe(struct osoba2*po){
    if(!po)return;
    if(po->imie)free(po->imie);
    if(po->nazwisko)free(po->nazwisko);
    free(po); // zwolnij pamięć całej struktury !!!
}
```

Funkcje zwracające struktury

```
void drukuj_osobe(const struct osoba2*po){
    printf("%s %s,%s\n",po->imie, po->nazwisko, po->pesel);
}

int main(){
    struct osoba2*jk = utworz_osobe("Jan","Nowak","12070404723");
    struct osoba2*an = jk;
    drukuj_osobe(jk);
    zwolnij_osobe(an);
}
```

Przy takim projekcie funkcji:

- Kod wygląda bardziej przejrzysto (zniknęły operatory pobierania adresu).
- Zwracanie wskaźnika z funkcji i przypisanie działają szybko

Programista dalej jest odpowiedzialny za (jednokrotne) zwalnianie pamięci.

Operatory inkrementacji i dekrementacji

Operatory inkrementacji i dekrementacji mogą być traktowane jako jednoargumentowe operatory przypisania.

Operator	Semantyka	Zwracana wartość
$x++$ (postfix)	$x' = x + 1$	x
$++x$ (prefix)	$x' = x + 1$	x'
$x--$ (postfix)	$x' = x - 1$	x
$--x$ (prefix)	$x' = x - 1$	x'

- Argumentem operatorów inkrementacji i dekrementacji musi być obiekt reprezentujący l-wartość (ponieważ dokonują przypisania)
- Wersje prefiksowe i postfixowe różnią się zwracaną wartością (przed lub po modyfikacją)
- Wartości zwracane przez operatory mogą być wykorzystane do pisania „zwięzłego kodu”, np.:

```
while (i < SIZE) a[i++] = i;
```

Kod tego typu jest jednak mało czytelny i bardziej podatny na błędy.

Operatory bitowe (1)

Operatory bitowe zwracają wartości całkowite dokonując operacji na bitach całkowitoliczbowych argumentów.

Działanie tych operatorów uzależnione jest od sposobu kodowania liczb.

- W przypadku liczb bez znaku (`unsigned`) ich wartość interpretowana jest jako

$$x = b_n * 2^n + b_{n-1} * 2^{n-1} + \dots + b_0 * 2^0$$

gdzie b_i to wartość i -tego bitu, n może przybierać wartości 7, 15, 31

- W przypadku liczb ze znakiem (`signed`), ustawiony najstarszy bit wskazuje na znak ujemny

$$x = -b_n * 2^n + b_{n-1} * 2^{n-1} + \dots + b_0 * 2^0$$

- Przykłady (ograniczające się do jednego bajtu, $n = 7$):

-128 kodowane jest jako 10000000

-16 kodowane jest jako 11110000 (-128+64+32+16)

9 kodowane jest jako 00001001

- Zachowanie operatorów bitowych dla liczb ze znakiem uzależnione jest od implementacji.

Operatory bitowe (2)

Operator	Opis
~	NOT. Operator jednoargumentowy inwersji bitów. Zamienia bit 0 na 1; 1 na 0. Operator zawsze zmienia znak liczby ze znakiem: $\sim 15 = -16$; $\sim 0 = -1$, $\sim -15 = 14$
&	AND. Operator dwuargumentowy. Wartość i -tego bitu wyjściowego jest równa 1, jeżeli oba bity wejściowe na i -tej pozycji są równe 1
	OR. Operator dwuargumentowy. Wartość bitu i -tego bitu wyjściowego jest równa 1, jeżeli jeden z i -tych bitów wejściowych jest równy 1
^	EXOR. Operator dwuargumentowy. Wartość bitu wyjściowego jest równa 1, jeżeli jeden z bitów wejściowych jest równy 1, a drugi 0.

Operatory ~ (NOT), & (AND) i | (OR) **nie powinny** być stosowane dla zmiennych pełniących rolę zmiennych logicznych.

- $\sim 0x1 = 0xffffffffffe$ ($\sim true = true$)
- $0x1 \& 0x2 = 0x0$ ($true \& true = false$)
- Dla operatora OR nie wystąpi efekt *short circuit* (obliczania minimalnej liczby argumentów wystarczających do wyznaczenia wartości wyrażenia). 34

Operatory bitowe (3)

Operator	Opis
$x \ll y$	LEFT SHIFT. Przesuwa w lewo bity x o y miejsc. „Odsłonięte” bity mają wartość 0. W przypadku braku przepełnienia, jest równoważny pomnożeniu przez 2^y .
$x \gg y$	RIGHT SHIFT. Przesuwa w prawo bity x o y miejsc. Jeżeli x jest liczbą bez znaku, „odsłonięte” bity mają wartość 0; w przeciwnym przypadku kopiowany jest bit znaku. Dla liczb nieujemnych jego działanie jest równoważne podzieleniu przez 2^y .

```
printf ("%u\n", 16<<2);           //64
printf ("%u\n", 16>>2);           //4
printf ("%u\n", 0xffffffff>>30); //3
int x=-1;
printf ("%d\n", x>>30);           //-1
printf ("%x %x\n", x, x>>30);     //ffffffff ffffffff
```

Operatory bitowe (4)

Przykłady:

Scalanie kolejnych bajtów w liczbę 2-bajtową

```
unsigned mk(unsigned lo, unsigned hi)
{
    return lo + hi<<8;
}

#define MK(LO,HI) (LO+(HI<<8))
```

Konwersja liczby do postaci binarnej

```
void printBinary(const unsigned char val)
{
    for(int i = 7; i >= 0; i--){
        if(val & (1 << i))printf("1");
        else printf("0");
    }
}
```

Operatory bitowe (5)

Przykłady:

Ustawianie flag (tu składowych koloru)

```
// wincon.h
#define FOREGROUND_BLUE      0x0001
#define FOREGROUND_GREEN    0x0002
#define FOREGROUND_RED      0x0004
#define BACKGROUND_BLUE     0x0010
#define BACKGROUND_GREEN    0x0020
#define BACKGROUND_RED      0x0040

blueOnWhite = FOREGROUND_BLUE |
BACKGROUND_BLUE | BACKGROUND_GREEN | BACKGROUND_RED;

// clear BACKGROUND_RED
blueOnCyan = blueOnWhite & (~ BACKGROUND_RED);
```

Operator warunkowy

Operator `?` : jest jedynym operatorem trójargumentowym. Jest on zdefiniowany jako:

```
cond-expression ? true-expr : false-expr
```

`cond-expression`

wyrażenie logiczne sterujące

`true-expr`

wyrażenie obliczane jeśli `cond-expression` ma wartość prawdy

`false-expr`

wyrażenie obliczane jeśli `cond-expression` ma wartość fałszu

W odróżnieniu od instrukcji `if-else` operator ten zwraca wartość jednego z wyrażeń.

```
xmin = x1 < x2 ? x1 : x2;  
xmax = x1 > x2 ? x1 : x2;  
printf( ( x < 0 ? "x=%d is less than 0" : "x=%d" ), x );
```

Operator sekwencji

Operator sekwencji, pozwala na określenie ciągu wyrażeń, które są obliczane od lewej do prawej.

`expr1 , expr2 , ..., exprn`

Rezultatem jest wartość ostatniego wyrażenia `exprn`.

- Operator ten jest stosowany głównie ze względu na efekty uboczne przy obliczaniu wyrażeń. Najczęściej jest wykorzystywany w instrukcji `for`.
- W większości przypadków operator może być zastąpiony zwykłymi instrukcjami, dlatego nie jest powszechnie stosowany.

Instrukcje będące wyrażeniami

- W języku C dowolne wyrażenie może zostać użyte jako instrukcja poprzez dodanie średnika.
- Jeżeli wyrażenie wywołuje efekty uboczne, jest to uzasadnione:
`i++;`
`x=y;`
- W takim przypadku wartość wyrażenia jest ignorowana, natomiast liczą się efekty uboczne.
- Instrukcje typu:
`a + 7; m*x;`
powodują obliczenie wartości wyrażen ale nie są one nigdzie zachowane. W semantyce C operatory te nie wywołują też efektów ubocznych.
- W języku C++ można tak przeciążyć te operatory, aby dla argumentów określonego typu wywoływały efekty uboczne, ale jest to zły wzór projektowania. Ich kod jest nieczytelny.
- Instrukcja:
`f();`
gdzie `f()` jest funkcją jest również poprawną instrukcją języka C. Jej wykonanie nie powoduje wywołania funkcji, ale obliczenie adresu funkcji `f()` i jego zignorowanie.