Języki programowania

Operatory

Operatory arytmetyczne jednoargumentowe

- Znaku: **+**, **-**
- Inkrementacji/dekrementacji: **++**, **--**
  - wersja przedrostkowa:
    - *najpierw zwiększana jest wartość zmiennej,*
    - *wartość ta staje się wartością wyrażenia*.
  - wersja przyrostkowa (końcówkowa):
    - *najpierw brana jest stara wartość zmiennej i ona staje się wartością wyrażenia,*
    - *wartość zmiennej zwiększana jest o 1.*

```
+12.7   -x   -(a*b+c)

i = i+1;   k = k-1;


++i;         --k;


i++;         k--;
```

```
int a = 5;
int b = 10;
cout << a << '\t' << ++a << '\t' <<
   a << endl;
cout << b << '\t' << b++ << '\t' <<
   b << endl;
```

Ekran po wykonaniu programu:

```
5     6     6
10    10    11
```

Operatory arytmetyczne dwuargumentowe

Uwaga: różne działanie, w zależności od typu operandów!

\+ (dodawanie)

\- (odejmowanie)

* (mnożenie)

/ (dzielenie)

% (dzielenie modulo)

```
int a = 5;    float x = 5.0;

a = a + 7;      // a = 12
x = x + 7.0;    // x = 12.0

a = 3 - a;      // a = -2
x = 3.0 - x;    // x = -2.0

a = 2 * a;      // a = 10
x = 2.0 * x;    // x = 10.0

a = a / 2;      // a = 2 (!!!)
x = x / 2.0;    // x = 2.5

a = a % 3;      // a = 2
```

Operatory przypisania

- Zwykłe podstawienie **=**

  Powoduje, że do obiektu stojącego po lewej stronie przypisana zostaje wartość wyrażenia stojącego po stronie prawej.

  Każde przypisanie jest samo w sobie wyrażeniem mającym taką wartość, jaka jest przypisywana.

- Skrótowe operatory przypisania:

+=	-=	*=
/=	%=	
>>=	<<=	
&=	|=	^=

```
x = 3.1415;
```

```
cout << (a = 2);
```

Na ekranie zostanie wypisane:

2

a przy tym do **a** zostanie pod-stawiona wartość **2**.

```
i -= 2;  // i = i - 2;
```

ale:

```
i =- 2;  // i = -2;(!!)
```

Operatory logiczne (1)

- Równość i różność (==, !=)

  Uwaga: łatwo o pomyłkę z operatorem podstawienia (=)!

- Pozostałe relacje:

  < <= > =>

- Suma i iloczyn logiczny:

  || &&

- Negacja (1-argumentowa)

  !

```
a = 5; b = 100;
if(a = b) // a nie '==' (!)
   cout << "rownosc";
else
   cout << "nierownosc";
// a = 100, b = 100 (!!!)


if(k <= 3) ...


while(a == 1 && x > 0.0) ...



if(!dodatni) ...
```

Operatory logiczne (2)

- W języku C++ są również dostępne za pomocą słów kluczowych języka:
  - `and` `&&`
  - `or` `||`
  - `not` `!`
  - `not_eq` `!=`

Operatory bitowe (1)

- Operacje wykonywane są na poszczególnych bitach lub parach bitów w zmiennych:
  - iloczyn bitowy `&`
  - suma bitowa `|`
  - bitowa różnica symetryczna `^`
  - negacja bitowa `~`
  - przesunięcie bitowe w lewo `<<`
  - przesunięcie bitowe w prawo `>>`

  Jego działanie zależy od typu zmiennej (signed/unsigned) oraz typu kompilatora!
- Przesunięcia bitowe są używane do szybkiego mnożenia lub dzielenia liczb całkowitych przez potęgi liczby 2.

```
// m    0000 1111 0000 1111
// n    0000 0010 0110 0011

m & n   0000 0010 0000 0011
m | n   0000 1111 0110 1111
m ^ n   0000 1101 0110 1100
~m      1111 0000 1111 0000



a = 0x40f2;   w = a << 3;
// a    0100 0000 1111 0010
// w    0000 0111 1001 0000

long p = 10;
p <<= 5;   // p *= 2^5 (32)
           // p = 160
```

Operatory bitowe (2)

- W języku C++ są również dostępne za pomocą słów kluczowych języka:

bitand	&
and_eq	&=
bitor	|
or_eq	|=
xor	^
xor_eq	^=
compl	~

Wyrażenie warunkowe ? :

- Uwaga: jest to wyrażenie, a nie instrukcja – tzn. coś, co ma jakąś wartość, którą można gdzieś przypisać!

```
(war) ? wart_t : wart_n
```

- Jaka jest wartość wyrażenia warunkowego?

  Jeśli warunek **war** jest spełniony, to wartość wyrażenia jest równa wartości **wart_t**, a w przypadku przeciwnym – wartości wyrażenia **wart_n**.

- Przykład:

Zamiast:

```
if(a > 0.0)   c = a;
else          c = -a;
```

można napisać krócej:

```
c = (a > 0.0) ? a : -a;
```

Operatory rzutowania (1)

Rzutowanie to jawne przekształcenie jednego typu na inny.

- Język C i C++: postać tradycyjna operatora rzutowania:

  (<nowy_typ>) <obj>

  - np.

```
int a = 0xffff;
char z;
z = (char) a;          // z = 0xff
float x = 3.14;
a = (int) x;           // a = 3
void* ptr = malloc(8);
double* q = (double*) ptr;
```

Operatory rzutowania (2)

- Język C++: postać funkcyjna operatora rzutowania:

  <nowy_typ>(<obj>)

  – w przypadku typów, które mają prostą nazwę:

```
int a = 0xffff;
char z;
z = char(a);            // z = 0xff
float x = 3.14;
a = int(x);             // a = 3
```

  – w przypadku typów, które nie mają prostej nazwy:

```
void* ptr = malloc(8);
typedef double* pDbl;
double* q = pDbl(ptr);
```

Operatory rzutowania (3)

- W przypadku niebanalnych rzutowań, notacja funkcyjna jest lepsza (B.Stroustrup).
- Używając jawnej konwersji typu do typów wskaźnikowych, można przedstawić obiekt jako wartość dowolnego typu:

```
dow_typ* p = (dow_typ*)&obj;
```

- Jeśli konwersja typu nie jest konieczna, to nie należy jej używać!
- Uwaga na niejawną konwersję typów!

Operatory rzutowania (4)

- W języku C++ pojawiła się cała rodzina nowych, dostosowanych do konkretnych potrzeb, operatorów rzutowania:
  - **`static_cast`**
  - **`const_cast`**
  - **`reinterpret_cast`**
  - **`dynamic_cast`** (używany do *downcasting*, OOP)

Operator `static_cast`

- Operator **`static_cast`** jest używany do wszystkich konwersji, które są dobrze zdefiniowane i mogą być wykonane przez kompilator (mogą być warningi), np.:
  - konwersje roszerzające (np. **`int -> float`**, **`int -> long`**),
  - konwersje zawężające (utrata informacji, np. **`float -> int`**),
  - konwersje z typu **`void*`**,
  - statyczna nawigacja w hierarchii klas,

```
double x = 3.14;
int i = static_cast<int>(x);
```

Operator `const_cast`

- Używany do konwersji typu:
  - `const` -> `nonconst`
  - `volatile` -> `nonvolatile`
- Używany TYLKO do konwersji tego typu – jeżeli równocześnie konieczna jest inna konwersja, musi być ona wykonana za pomocą oddzielnego operatora.

```
const int i = 1;
int* j = (int*)&i;          // niezalecane!
j = const_cast<int>(i);     // zalecane

long* l = const_cast<long*>(&i); // błąd!

volatile int k = 0;
int* u = const_cast<int*>(&k);
```

Operator `reinterpret_cast`

- Jest to najmniej bezpieczny mechanizm rzutowania, mogący powodować błędy.
- Traktuje obiekty jak pewien zestaw bitów, nie zwraca uwagi na znaczenie.

```
struct X { int a[100]; };

void print(X* x) {
    for(int i = 0; i < 100; ++)
        cout << x->a[i] << ' ';
}

X x;
print(&x);
int* xp = reinterpret_cast<int*>(&x);
for(int* i = xp; i < xp + 10; ++i)
    *i = 0;
print(reinterpret_cast<X*>(xp));
```

Pozostałe operatory

- Operator `sizeof(...)`

  Zwraca rozmiar danego obiektu lub typu w bajtach.

```
cout << sizeof(long);   // 4
cout << sizeof(x);
/* 8, jeśli x było typu
   double */
```

- Operator przecinek `,`

  Kilka wyrażeń oddzielonych przecinkiem stanowi wyrażenie.

  Jego wartością jest wartość wyrażenia stojącego najbardziej z prawej strony!

```
(2 + 4, a * 4, 3 < 6)
// wartosc tego wyrazenia
// wynosi 1 (czyli prawda)
```

Priorytety operatorów

::
-> [] ()
sizeof ++ -- ~ & ! + - * (1-arg) new delete (cast)
.* ->*
* / %
+ -
<< >>
< > <= >=
== !=

&
^
|
&&
||
? :
= *= += -= %= /= <<= >>= &= ^= |=
,

Łączność:

- prawostronna,
- lewostronna.

Łączność operatorów

- Operator jest prawostronnie łączny wtedy, gdy działa na argumencie stojącym po jego prawej stronie.
- Operator jest lewostronnie łączny wtedy, gdy działa na argumencie stojącym po jego lewej stronie.
- Operatory jednoargumentowe oraz operatory przypisania są prawostronnie łączne, reszta – lewostronnie.
- W wypadku operatorów dwuargumentowych łączność określa, w jaki sposób grupowane jest wykonywanie wyrażenia.
  - np. wyrażenie: `a + b + c + d`
    odpowiada wyrażeniu: `((a + b) + c) + d`
  - np. wyrażenie: `a = b = c = d`
    odpowiada wyrażeniu: `a = (b = (c = d))`