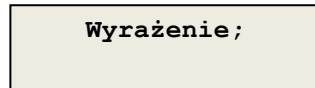
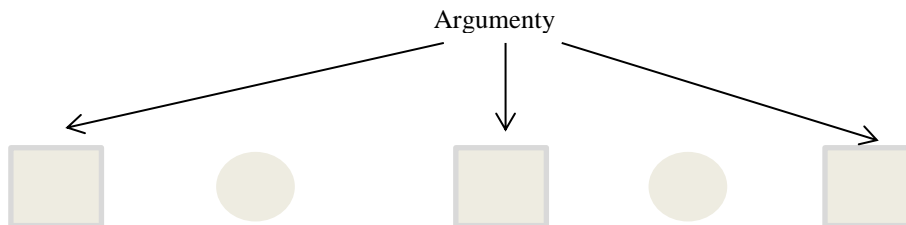


3 Wyrażenia i operatory

Instrukcja



Wyrażenie



Wyrażenia

- Odpowiednikiem czynności w C++ jest *wyrażenie* (ang. *expression*).
- Wyrażenie zakończone średnikiem nazywa się *instrukcją* (ang. *instruction*).
- W skład wyrażenia wchodzi:
 - *argumenty* (ang. *operands*) - obiekty, na których wykonuje się operacje (zmienne, stałe, wywołania funkcji),
 - *operatory* (ang. *operator*) - reprezentujące operacje (np. operator + oznacza operację dodawania, operator = przypisanie wartości zmiennej).
- Wyrażenie ma wartość, która jest określonego typu.
- Przykłady wyrażeń:

```
2 * 3 /* mnożenie dwóch stałych: wartość 6, typ int */
i + 1 /* dodawania stałej i zmiennej typu int:
      wartość zależy od i, typ int */
5 + h*6 /* h jest typu int; wartość zależy od h, typ int */
0.5 * sqrt(2.) /* sqrt - pierwiastek; typ double */
```

Operatory

- Operatory działające na jednym argumentie nazywa się operatorami *jednoargumentowymi* (ang. *unary operators*).
- Operatory działające na dwóch argumentach nazywa się operatorami *dwuargumentowymi* (ang. *binary operators*).
- W przypadku operatorów dwuargumentowych rozróżnia się argument lewostronny (ang. *left operand*) i prawostronny (ang. *right operand*).
- Ten sam symbol operatora może reprezentować różne operacje. Przykładem jest gwiazdka *. Znaczenie operatora wynika z kontekstu użycia.
- Każdy operator ma argumenty określonego typu oraz daje wynik ustalonego typu. Jeśli operator dopuszcza argumenty wielu typów, to typ wyniku określony jest pewnymi specyficznymi regułami.
- Operatory można podzielić na grupy:
 - operatory *przypisania* (ang. *assignment*)
 - operatory *arytmetyczne* (ang. *arithmetic*)
 - operatory *relacji* (ang. *relational*)
 - operatory *logiczne* (ang. *logical*)
 - operatory *bitowe* (ang. *bitwise*)
 - operatory *specjalne*

3.1 Operator przypisania = i instrukcja przypisania

- Składnia:

$$\langle arg_1 \rangle = \langle arg_2 \rangle$$

- Działanie: obliczana jest wartość wyrażenia po prawej stronie znaku równości $= (arg_2)$ i zapamiętywana w pamięci przyporządkowanej argumentowi po lewej stronie operatora $= (arg_1)$.
- Ograniczenia: argument po lewej stronie operatora przypisania (arg_1) musi być *modyfikowalną l-wartością* (ang. *lvalue*). *Modyfikowalna l-wartość* to wyrażenie określające obszar pamięci, którego wartość może być zmieniana (np. zmienna jednego z podstawowych typów)
- Przykład:

```
int x;  
x = 7; // zmienna x przyjmie wartość 7
```
- Wyrażenie z operatorem przypisania ma wartość, którą jest wartość arg_1 po wykonaniu przypisania. Typ tej wartości jest taki sam, jak typ argumentu po lewej stronie operatora przypisania (czyli arg_1).
- Przykłady:

<pre>double x; x=7.8;</pre>	<i>wartość wyrażenia: wartość zmiennej x typ wyrażenia: typ zmiennej x</i>
<pre>int a,b; a=(b=3);</pre>	<i>poprawne, ponieważ b=3 ma wartość wartość całego wyrażenia: 3, typ wyrażenia: int</i>
<pre>int n; double k; k=(n=0);</pre>	<i>wartość wyrażenia: 0 typ wyrażenia: double</i>

Wielokrotne przypisanie

- Można w tej samej instrukcji przypisywać tę samą wartość wielu zmiennym:

```
double x,y;  
x=y=0.;
```

- Jak to działa?

Operator przypisania ma wiązanie *prawostronne* (od prawej do lewej, ang. *right to left*). Oznacza to, że obliczenia w wyrażeniu wykonuje się od jego prawej strony do lewej. Powyższy zapis jest równoważny zapisowi: $x=(y=0.)$;

(Obliczana jest wartość wyrażenia $0.$; jest ona przypisywana zmiennej y ; jednocześnie stanowi ona wartość wyrażenia $y=0.$; następnie ta wartość jest przypisywana zmiennej x ; wartością całego wyrażenia jest $0.$.)

- *Warunek poprawnego działania*: wszystkie argumenty tych operatorów są zgodnego typu.

Konwersja typów w instrukcji przypisania

- *Reguła konwersji typów w instrukcji przypisania*: wartość prawej strony instrukcji jest przekształcana do typu lewej strony. Konwersja ta jest wykonywana *automatycznie*.

$arg_1 = arg_2$

Jeżeli zatem w instrukcji $arg_1 = arg_2$ typ arg_2 nie zgadza się z typem arg_1 , przed przypisaniem wykonywana jest konwersja arg_2 do typu arg_1 . Czyli wartość prawej strony instrukcji (wartość wyrażenia) jest przekształcana do typu lewej strony.

- **UWAGA: Podczas przekształcania z typu „szerszego” na „węższy” może nastąpić utrata informacji.**

Przykład:

```
int j;  
j=2.5; /* j będzie mieć wartość 2 */
```

- Inne przykłady konwersji:

```
int i;  
char c;  
float f;  
c = i; /* obcinane są najbardziej znaczące bity i */  
i = f; /* do i jest zapisywana tylko część całkowita */  
f = c; /*przekształcenie znaku (liczba całkowita) do zmiennoprzecinkowej */  
f = i; /* przekształcenie liczby całkowitej do zmiennoprzecinkowej */
```

3.2 Pierwszeństwo i łączność operatorów

- Kolejność wykonywania operacji w wyrażeniu określają reguły *pierwszeństwa* (priorytet, ang. *precedence*) i *łączności* (wiązanie, ang. *associativity*) operatorów.
- *Priorytet* operatorów stanowi o kolejności wykonywania działań określonych przez poszczególne operatory. Operatory o wyższym priorytecie określają działania, które są najpierw wykonywane.
- Przykład: operator + oznacza dodawanie, zaś operator * mnożenie, operator * ma wyższy priorytet niż +
 $2 + 3 * 4$ daje w wyniku 14; operator * ma wyższy priorytet niż +
 $3 * 4 + 2$ daje w wyniku 14; operator * ma wyższy priorytet niż +
- *Wiązanie* określa sposób łączenia operatora z argumentami. Może to być:
 - wiązanie lewostronne (ang. *left-to-right*) - obowiązujące dla większości operatorów
 - wiązanie prawostronne (ang. *right-to-left*) - obowiązujące na przykład dla operatorów jednoargumentowych i operatorów przypisania.
- Jeśli operatory mają ten sam priorytet, to kolejność działań wynika z wiązania. Przykład:

$2 + 3 - 4$ daje w wyniku 1; operatory + i - mają ten sam priorytet, kolejność wykonania operacji jest ustalana na podstawie wiązania: w tym wypadku lewostronnego

- Kolejność działań można zmienić za pomocą nawiasów. Pozwalają one wyodrębnić podwyrażenia. Obliczając wyrażenie złożone, najpierw oblicza się podwyrażenia zawarte w najbardziej wewnętrznych nawiasach.
- Przykład:

```
(2-3)*4 to -4  
2-(3*4) to -10  
4*5+6*2 to 32  
4*(5+6*2) to 68  
4*((5+6)*2) to 88
```


Priorytety podstawowych operatorów C++

Klasa	Operatory w klasie	Łączność	Przykład	Priorytet
wywołanie funkcji	()	lewostronna	<i>sqr(a);</i>	NAJ- WYŻSZY ↑
indeks tablicy	[]		<i>a[5]=2;</i>	
selektor składowej	.		<i>obiekt.składowa</i>	
selektor składowej	->		<i>obiekt->składowa</i>	
przyrostkowe zwiększanie o 1	++		<i>a++;</i>	
przyrostkowe zmniejszanie o 1	--		<i>a--;</i>	
Operatory jednoargumentowe:		prawostronna	<i>(double)a</i>	
rzutowanie (przekształcenie typu)	(typ)		<i>sizeof a</i>	
rozmiar obiektu (w bajtach)	sizeof		<i>sizeof(int)</i>	
rozmiar typu (w bajtach)	sizeof		<i>new int[10]</i>	
utworzenie obiektu	new		<i>delete [] wsk</i>	
usunięcie obiektu	delete		<i>&a</i>	
adres	&		<i>*wsk</i>	
wyłuskanie	*		<i>+a</i>	
plus jednoargumentowy	+		<i>-a</i>	
minus jednoargumentowy	-		<i>~a</i>	
negacja bitowa	~		<i>!a</i>	
negacja logiczna	!		<i>++a</i>	
przedrostkowe zwiększanie	++		<i>--a</i>	
przedrostkowe zmniejszanie	--			
wybór składowej przez wskaźnik	->*	lewostronna	<i>wsk->*wsk-do-składowej</i>	
wybór składowej przez wskaźnik	.*		<i>obiekt.*wsk-do-składowej</i>	
mnożenie	*	lewostronna	<i>a * b</i>	
dzielenie	/		<i>a / b</i>	
dzielenie modulo (reszta)	%		<i>a % b</i>	
dodawanie	+	lewostronna	<i>a + b</i>	
odejmowanie	-		<i>a - b</i>	
przesuwanie bitów w lewo	<<	lewostronna	<i>zm<<l bitów</i>	
przesuwanie bitów w prawo	>>		<i>zm>>l bitów</i>	
operacje relacji	< <= > >=	lewostronna	<i>a < b</i>	
równość, nierówność	== !=	lewostronna	<i>a == b</i>	
koniunkcja bitowa AND	&	lewostronna	<i>a & b</i>	
różnica symetryczna XOR	^	lewostronna	<i>a ^ b</i>	
alternatywa bitowa OR		lewostronna	<i>a b</i>	
iloczyn logiczny AND	&&	lewostronna	<i>a && b</i>	
suma logiczna OR		lewostronna	<i>a b</i>	
wyrażenie warunkowe	? :	nie dotyczy	<i>a ? x : y</i>	
przypisanie	=	prawostronna	<i>a = b</i>	
przypisania złożone	+= -= *= /= %= >>= <<= &= ^=		<i>a += b</i>	
przecinek	,	lewostronna	<i>a=5, b=6;</i>	NAJ- NIŻSZY

Operatory w tym samym segmencie tablicy mają ten sam priorytet, np. operator dodawania + ma ten sam priorytet co operator odejmowania -.

3.3 Operatory arytmetyczne

Składnia:
<arg_1> <operator> <arg_2> (operator dwuargumentowy)
<operator> <arg_1> (operator jednoargumentowy)

Operator	Operacja	Wiązanie	Priorytet
+	plus jednoargumentowy	od prawej do lewej	NAJWYŻSZY 
-	minus jednoargumentowy	od prawej do lewej	
*	mnożenie	od lewej do prawej	
/	dzielenie	od lewej do prawej	
%	modulo (reszta z dzielenia) (tylko dla typu całkowitego)	od lewej do prawej	NAJNIŻSZY
+	dodawanie	od lewej do prawej	
-	odejmowanie	od lewej do prawej	

- Aby operacje arytmetyczne były realizowane w sposób jednoznaczny, operatorom są nadane priorytety. W tabeli linią poziomą oddzielono operatory o różnych priorytetach. Tak więc operatory *, / i % mają wyższy priorytet niż operatory + i - . Operatory o tym samym priorytecie wykonywane są zgodnie z wiązaniem.
- Kolejność wykonywania działań można zmieniać za pomocą nawiasów. Wyodrębniają one podwyrażenia. Najpierw oblicza się podwyrażenia zawarte w najbardziej wewnętrznych parach nawiasów, później podwyrażenia w nawiasach zewnętrznych.
- Przykład:

2 + 3 * 5 daje w wyniku 17
(2 + 3) * 5 daje w wyniku 25

- Obowiązują dwie arytmetyki:
 - *arytmetyka liczb całkowitych*: wynik działania na liczbach całkowitych jest liczbą całkowitą
 - *arytmetyka liczb rzeczywistych*: wynik działania na liczbach rzeczywistych jest liczbą rzeczywistą
 - w przypadku mieszanym (liczba całkowita i liczba rzeczywista) wynik jest liczbą rzeczywistą.
- Przykład:

1/2 daje w wyniku 0
4/3 daje w wyniku 1
1./2 daje w wyniku 0.5
1./2. daje w wyniku 0.5

• Przykłady zapisów w języku C++

Wyrażenie algebraiczne	Wyrażenie w C++
$a+7$	<code>a + 7</code>
$a-b$	<code>a - b</code>
cx lub $c \cdot x$	<code>c * x</code>
x/y lub $x \div y$	<code>x / y</code>
$x \bmod y$	<code>x % y</code>
$\frac{a+b+c+d+e}{3}$	<code>(a+b+c+d+e)/3</code>
$y=mx+b$ lub $y=m \cdot x + b$	<code>y = m * x + b;</code>

• Przykład: kolejność obliczania wartości wyrażenia $y = 2x^2 + 3x + 7$ dla $x=5$:

Krok 1. $y = 2 * 5 * 5 + 3 * 5 + 7 ;$ $=10$	mnożenie najbardziej od lewej
Krok 2. $y = 10 * 5 + 3 * 5 + 7 ;$ $=50$	mnożenie najbardziej od lewej
Krok 3. $y = 50 + 3 * 5 + 7 ;$ $=15$	mnożenie przed dodawaniem
Krok 4. $y = 50 + 15 + 7 ;$ $=65$	dodawanie najbardziej od lewej
Krok 5. $y = 65 + 7 ;$ $=72$	ostatnie dodawanie
Krok 6. $y = 72 ;$	przypisanie 72 do zmiennej


Obliczenia arytmetyczne - uwagi

- W niektórych okolicznościach wynik wyrażenia arytmetycznego może być niepoprawny lub niezdefiniowany. Może to wynikać z reguł matematycznych (niedozwolone dzielenie przez zero) lub z właściwości komputera (wartość większa niż rozmiar właściwy dla danego typu).
- Wartości liczbowe przedstawiane są za pomocą skończonej liczby cyfr. W przypadku liczb całkowitych nie stanowi to problemu, liczba całkowita jest po prostu zapisywana w systemie dwójkowym. W przypadku liczb rzeczywistych często nie można liczby przedstawić dokładnie za pomocą skończonej liczby bitów. Na przykład ułamek dziesiętny 0.1 nie ma skończonego rozwinięcia w systemie dwójkowym. Dlatego liczby rzeczywiste są zaokrąglane (ang. *roundoff*), tak aby zmieściły się w obszarze przeznaczonym dla danych typu float, double lub long double, zgodnie z deklaracją typu danej wartości. Wynik działania arytmetycznego zależy od ustalonej dla danego typu dokładności obliczeń.
- Informacje na temat arytmetyki komputera, przedstawiania liczb całkowitych i rzeczywistych można znaleźć na przykład w książce: *W.Stallings, Organizacja i architektura systemu komputerowego, rozdz. 8.*

3.4 Operatory porównania, relacji oraz logiczne

Składnia:

`<arg_1> <operator> <arg_2>` (operator dwuargumentowy)
`<operator> <arg_1>` (operator jednoargumentowy)

Operator	Operacja	Wiązanie	Priorytet
negacja logiczna <code>!</code>	(zaprzeczenie)	od prawej do lewej	NAJWYŻSZY 
<code><</code>	mniejsze	od lewej do prawej	
<code><=</code>	mniejsze lub równe	od lewej do prawej	
<code>></code>	większe	od lewej do prawej	
<code>>=</code>	większe lub równe	od lewej do prawej	
<code>==</code>	równe	od lewej do prawej	
<code>!=</code>	nierówne	od lewej do prawej	
<code>&&</code>	iloczyn logiczny AND (koniunkcja)	od lewej do prawej	NAJNIŻSZY
<code> </code>	suma logiczna OR (alternatywa)	od lewej do prawej	

- Wartością wyrażen relacji lub logicznych w języku C++ jest `true` (w języku C jest to wartość 1) lub `false` (w języku C wartość 0). Jeśli jednak operator taki wystąpi w kontekście wymagającym użycia wartości całkowitej, jego wynik będzie awansowany do wartości 1 lub 0.
- Operatory porównania i relacji służą do obliczenia wartości „prawda” (`true` lub liczba 1) lub „fałsz” (`false` lub liczba 0). Warunek prawdziwy daje w wyniku wartość `true`, warunek fałszywy - `false`.
- Operator negacji logicznej `!` daje w wyniku wartość `true`, jeżeli jego argument ma wartość `false` lub 0; w przeciwnym przypadku jego wynikiem jest `false`.
- Wynikiem operatora iloczynu logicznego `&&` jest prawda wtedy i tylko wtedy, gdy oba argumenty są prawdziwe.
- Wynikiem operatora sumy logicznej `||` jest prawda wtedy i tylko wtedy, gdy którykolwiek z argumentów jest prawdziwy.
- *Uwaga:* gdy tylko określi się, że całe wyrażenie jest prawdziwe lub fałszywe, zaprzestaje się wykonywania dalszych operatorów w tym wyrażeniu. Przykłady:

```
wyr1 && wyr2; // jeśli wyr1 ma wartość false, nie będzie obliczane wyr2
wyr1 || wyr2; // jeśli wyr1 ma wartość true, nie będzie obliczane wyr2
```
- Operatory relacji i logiczne mają niższy priorytet niż operatory arytmetyczne:
wyrażenie `x + 1 < y * 2` jest interpretowane jako `(x+1) < (y*2)`

• Przykłady:

Wyrażenie	Wartość
-1 < 0	true
0 > 1	false
0 == 0	true
1 != -1	true
1 >= -1	true
1 > 10	false
int a = 1; int b = 10; bool wynik;	
wynik = a < b;	true
wynik = a == b;	false
wynik = a != b;	true
int wynik2;	
wynik2 = a < b;	1
wynik2 = a == b;	0

Prawa De Morgan’a

not (p and q) = not(p) or not(q)

not (p or q) = not (p) and not(q)

W notacji C++ prawa De Morgan’a można zapisać następująco:

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

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

3.5 Operator zwiększania ++ i zmniejszania --

- Są to operatory jednoargumentowe.
- Operatory zwiększania ++ i zmniejszania -- umożliwiają zwarty zapis dodawania lub odejmowania liczby 1. Można napisać:

```
licznik++;
```

zamiast

```
licznik=licznik+1;
```
- Inne nazwy tych operatorów to operatory *inkrementacji* i *dekrementacji*.
- Oba operatory można rozumieć jako operatory przypisania - *argument musi być modyfikowalną l-wartością* (na przykład zmienną typu arytmetycznego).
- Operatory ++ i -- mają ten sam priorytet, wyższy od operatorów arytmetycznych.

Operator	Operacja	Wiązanie	Przykład	Priorytet
++	zwiększanie o 1, operator przyrostkowy	od prawej do lewej	i++	<div>NAJWYŻSZY</div> <div>↑</div> <div>NAJNIŻSZY</div>
--	zmniejszanie o 1, operator przyrostkowy	od prawej do lewej	i--	
++	zwiększanie o 1, operator przedrostkowy	od prawej do lewej	++i	
--	zmniejszanie o 1, operator przedrostkowy	od prawej do lewej	--i	

- Operator *przedrostkowy* oznacza, że najpierw należy zwiększyć wartość o 1 a następnie użyć jej w wyrażeniu.
- Operator *przyrostkowy* oznacza, że najpierw należy użyć wartość w wyrażeniu a następnie zwiększyć ją o 1.
- Przykłady:

```
x=1;
    ++x; // x przyjmuje wartość 2
    x++; // x przyjmuje wartość 3
x=1;
y=++x; // x przyjmuje wartość 2, y przyjmuje wartość 2

x=1;
y=x++; // x przyjmuje wartość 2, y przyjmuje wartość 1

x=3;
y=2*(x++); // y przyjmie wartość 6, x wartość 4

x=3;
y=2*(++x); // y przyjmie wartość 8, x wartość 4
```

3.6 Operator pobrania rozmiaru sizeof

- Składnia:

```
sizeof(<nazwa_typu>);  
sizeof <obiekt>;
```

- Operator sizeof zwraca liczbę bajtów - rozmiar argumentu.
- Przykłady:

```
double x;  
cout << sizeof x; // nazwa obiektu (zmiennej) - nie trzeba nawiasu  
cout << sizeof(double); // identyfikator typu - w nawiasie  
  
#include <iostream>  
using namespace std;  
int main()  
{  
    cout << "Typ\t\tRozmiar" << endl;  
    cout << "char\t\t" << sizeof(char) << endl;  
    cout << "short\t\t" << sizeof(short) << endl;  
    cout << "int\t\t" << sizeof(int) << endl;  
    cout << "long\t\t" << sizeof(long) << endl;  
    cout << "float\t\t" << sizeof(float) << endl;  
    cout << "double\t\t" << sizeof(double) << endl;  
    return 0;  
}
```

3.7 Operator wyliczeniowy , (przecinek)

- Składnia:

```
<wyrażenie>, <wyrażenie>, <wyrażenie>;
```

- Przecinek jest stosowany do wiązania ze sobą większej liczby wyrażeń.
- Wyrażenia te oblicza się kolejno, poczynając od pierwszego z lewej strony.
- Przykład:

```
a=0;           a=0, b=10;  
b=10;          if(a<b)  
if(a<b)         c=a, a=b, b=c;  
{  
    c=a;  
    a=b;  
    b=c;  
}
```

- Uwaga: Operator przecinkowy należy stosować z umiarem, gdyż czasem zmniejsza czytelność programu.

3.8 Operatory bitowe

Operator	Operacja	Wiązanie	Priorytet
~	bitowa negacja (NOT)	od prawej do lewej	NAJWYŻSZY ↑
<<	przesunięcie w lewo	od lewej do prawej	
>>	przesunięcie w prawo	od lewej do prawej	
&	bitowa koniunkcja (AND)	od lewej do prawej	
^	różnica symetryczna (XOR)	od lewej do prawej	NAJNIŻSZY
	bitowa alternatywa (OR)	od lewej do prawej	

- Operatory bitowe działają osobno na każdym bicie.
- Bit może mieć wartość 0 (mówimy, że jest wyłączony lub wyzerowany) lub 1 (włączony lub ustawiony).
- Argumenty operatorów bitowych muszą być typu całkowitego.
- Operatory NOT, AND i OR działają podobnie do swoich odpowiedników logicznych.
- Dodatkowym operatorem jest operator *różnicy symetrycznej* (ang. *exclusive OR*).
- Tabela operacji logicznych na bitach:

p	q	p & q	p q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

- Tabela różnicy symetrycznej:

p	q	p ^ q
0	0	0
0	1	1
1	0	1
1	1	0

- Operatory przesunięć przesuwają wszystkie bity zmiennej odpowiednio w prawo lub w lewo.
- Bity przesunięte poza koniec bajtu lub słowa są tracone.
- Podczas przesunięcia w lewo bity, które uległy przesunięciu uzupełniane są zerami.
- Podczas przesunięcia w prawo typu `unsigned` bity, które uległy przesunięciu uzupełniane są zerami.
- Podczas przesunięcia w prawo liczby ujemnej, bity które uległy przesunięciu mogą być uzupełniane zerami lub jedynkami (powielenie znaku) w zależności od implementacji.
- Przykłady:

Wyrażenie	Lewy argument	Wynik	Wartość wyniku
5<<1	00000000 00000101	00000000 00001010	10
255>>3	00000000 11111111	00000000 00011111	31
8<<10	00000000 00001000	00100000 00000000	2 ¹³
1<<15	00000000 00000001	10000000 00000000	-2 ¹⁵

- Przesuwanie w lewo $x \ll y$ jest równoważne mnożeniu $x * 2^y$
- Przesuwanie w prawo $x \gg y$ jest równoważne dzieleniu $x / 2^y$

3.9 Skróty - złożone operatory przypisania

- Złożone operatory przypisania pozwalają uzyskać bardziej zwarty zapis wyrażenia.
- Zamiast pisać:

`<arg_1> = <arg_1> <operator> <arg_2>`

można napisać:

`<arg_1> <operator> = <arg_2>`

gdzie operator jest jednym z operatorów `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, `|`.

- Przykład: zamiast
`x = x+3;` można napisać `x += 3;`
- Złożony operator przypisania ma taki sam priorytet i łączność jak operator przypisania `=`.

3.10 Język C++ i operatory wejścia-wyjścia

- W języku C++ można definiować własne działanie operatora, o ile związany będzie on z własnym typem obiektu. Można w ten sposób zmienić działanie prawie każdego wbudowanego w język C++ operatora. Technika ta nazywana jest przeciążaniem operatora i zostanie omówiona na wykładzie "Programowanie obiektowe".
- Przykładami przeciążonych operatorów są operatory `<<` oraz `>>` używane do wprowadzania danych i wyprowadzania wyników. Ich podstawowe znaczenie to przesuwanie w lewo lub w prawo bitów zmiennej. Jednakże w bibliotece wejścia-wyjścia (dołączanej za pomocą `#include <iostream>`) przypisane zostało im inne znaczenie. W tym znaczeniu możemy mówić, że operator `<<` to *operator wyjścia*, zaś operator `>>` to *operator wejścia*.
- Aby można było nadać operatorom `<<` i `>>` własne znaczenie, trzeba było zdefiniować w bibliotece wejścia-wyjścia własne obiekty:
 - `cin` – reprezentuje wejście standardowe (ang. *standard input*) i służy do pobierania wprowadzanych danych
 - `cout` – reprezentuje wyjście standardowe (ang. *standard output*) i służy do wyświetlania wyprowadzanych wyników
- Dla obiektu `cout` przeciążono operator `<<` w ten sposób, aby drugim argumentem operacji wyjścia mógł być obiekt (zmienna) dowolnego wbudowanego typu (np. `int`, `float`, `double`) oraz napis. Wybrano ten operator ponieważ kojarzy się on z przesyłaniem danych z obiektu na wyjście. Dzięki temu rozwiązaniu możemy stosować konstrukcje:

```
cout << "Witam";  
cout << k; // k jest zmienną typu int
```
- Definiując możliwości operatora `<<` zapewniono również to, aby można było przesyłać na wyjście wiele zmiennych i tekstów:

```
int k=5;  
cout << "Wartosc k to " << k;
```
- Dla obiektu `cin` przeciążono operator `>>`, gdyż ten operator kojarzy się z przesyłaniem danych do obiektu. Podobnie jak w przypadku wyprowadzania drugim argumentem może być obiekt (zmienna) dowolnego wbudowanego typu oraz typ pozwalający na przechowanie napisu. Dzięki temu rozwiązaniu możemy pisać:

```
int i, k;  
cin >> k >> i;
```

Domyślnie operator wejścia pomija wszystkie odstępów (tj. znaki spacji, tabulacji, nowego wiersza).
- Przeciążenie operatora nie zmienia jego miejsca w tablicy priorytetów, ani nie zmienia jego łączności.

3.11 Konwersja typów

- Konwersja typu (ang. *type conversion*) ma miejsce wtedy, kiedy wspólnie korzysta się ze zmiennych lub obiektów różnych typów, na przykład w instrukcjach przypisania, w wyrażeniach, podczas przekazywania argumentów aktualnych do wywoływanej funkcji.
- Konwersja typu może być:
 - niejawna (ang. *implicit*) – wykonywana automatycznie przez kompilator,
 - jawna (ang. *explicit*) – wykonywana na życzenie użytkownika.
- Przykłady:

```
double x=3; /* konwersja niejawna */  
  
double x=(double)3; /* konwersja jawna */
```
- Konwersje zarówno jawne jak i niejawne muszą być **bezpieczne**: czyli wszędzie tam, gdzie jest to możliwe, dokonywane tak, aby nie stracić przechowywanych informacji.
- Przykładem bezpiecznej konwersji jest przekształcanie argumentów o mniejszych rozmiarach, a tym samym mniejszym zakresie wartości i dokładności, do typów o większych rozmiarach. Np. `int` do `double`.

3.11.1 Przykłady konwersji niebezpiecznych

- Konwersja typu całkowitego o rozmiarze większym do rozmiaru mniejszego (np. `int` do `char`) polega na odrzuceniu najwyższych bitów liczby i pozostawieniu niezmiennych bitów mieszczących się w zmiennej typu o mniejszym rozmiarze.
- Konwersje między typami całkowitymi o tych samych rozmiarach (np. `int` do `unsigned int`) są wykonywane za pomocą skopiowania wartości jednej zmiennej do drugiej. (Np. -1000 stanie się 64536).
- Konwersje między typami rzeczywistymi i całkowitymi odbywają się za pomocą odrzucenia części ułamkowej liczby rzeczywistej.
- Konwersje między typami rzeczywistymi o różnych rozmiarach wykonywane są za pomocą zaokrąglenia do najbliższej wartości docelowego typu.

3.11.2 Konwersja niejawna

- Konwersja *niejawna* (ang. *implicit type conversion*) jest wykonywana automatycznie przez kompilator w następujących sytuacjach:
 - jeżeli w wyrażeniu arytmetycznym występują argumenty różnych typów, to wynikiem przekształcenia będzie najszerzy typ; jest to tzw. konwersja arytmetyczna (ang. *arithmetic conversion*)
 - jeżeli wyrażenie jednego typu ma być przypisane obiektowi innego typu, to typem wyniku przekształcenia będzie typ obiektu, któremu ma być przypisana wartość wyrażenia
 - jeżeli wyrażenie jednego typu ma być przekazane jako argument do funkcji, której odpowiedni argument formalny jest innego typu, to wynikiem przekształcenia będzie typ argumentu formalnego
 - jeżeli wyrażenie zwracane przez funkcję jest innego typu niż typ wyniku funkcji, to wynikiem przekształcenia będzie typ wyniku funkcji.

3.11.3 Przekształcenia arytmetyczne

- Przekształcenie arytmetyczne ma zapewnić to, że oba argumenty operatora dwuargumentowego (np. dodawania) będą przekształcone (mówimy *awansowane* lub *promocja*) do wspólnego szerszego typu, który będzie typem wyniku.
- Przekształcenia ta podlegają dwu zasadom:
 - aby nie dopuścić do zmniejszenia dokładności, typy są zawsze *awansowane*, jeśli jest to niezbędne, do typu szerszego
 - wszystkie wyrażenia, w których występują typy całkowite mniejsze niż typ `int`, będą przed wykonaniem obliczeń *awansowane* do typu `int`.
- *Awansowaniu do typu `int`* (ang. *integral promotion*) podlegają argumenty typów całkowitowych: `char`, `signed char`, `unsigned char`, `short int`, `unsigned short int`, typ wyliczeniowy oraz typ logiczny.
- Przypadki szczególne:
 - Argumenty są przekształcane w typ `int`, o ile typ `int` w danym komputerze jest dostatecznie szeroki aby mógł reprezentować wszystkie wartości tego typu, w przeciwnym wypadku są *awansowane* do typu `unsigned int`.
 - Argumenty typu `bool` są *awansowane* do wartości typu `int`; `false` staje się 0, zaś `true` staje się 1.
- W większości kompilatorów *awansowanie* do typu `int` przebiega następująco:

Typ	Typ po konwersji	Metoda konwersji
<code>char</code>	<code>int</code>	powielenie bitu znaku dla typu <code>signed char</code> lub wypełnienie zerami dla typu <code>unsigned char</code>
<code>unsigned char</code>	<code>int</code>	starszy bajt wypełniony zerami
<code>signed char</code>	<code>int</code>	powielenie bitu znaku
<code>short</code>	<code>int</code>	ta sama wartość
<code>unsigned short</code>	<code>unsigned int</code>	ta sama wartość
<code>enum</code>	<code>int</code>	ta sama wartość

- Jeśli po tej konwersji nadal argumenty różnią się typami, wykonywane są przekształcenia typu węższego na szerszy. Dla wszystkich par argumentów wykonywane są przekształcenia:
 1. Jeśli jeden z argumentów jest `long double`, drugi jest przekształcany do typu `long double`.
 2. W przeciwnym razie jeśli jeden z argumentów jest `double`, drugi jest przekształcany do typu `double`.
 3. W przeciwnym razie jeśli jeden z argumentów jest `float`, drugi jest przekształcany do typu `float`.
 4. W przeciwnym razie oba argumenty są typu całkowitego, następuje promocja typów całkowitych.
 5. Jeśli jeden z argumentów jest typu `unsigned long int`, drugi jest przekształcany do typu `unsigned long int`.
 6. W przeciwnym razie jeśli jeden z argumentów jest `signed long int`, konwersja będzie od względnej wielkości tych typów. Jeśli `long` może reprezentować wszystkie wartości `unsigned long`, `unsigned long` jest przekształcany w `long`.
 7. W przeciwnym razie oba argumenty są przekształcane w `unsigned long`.
 8. W przeciwnym razie, jeśli jeden z argumentów jest `long`, oba są przekształcane do `long`.
 9. W przeciwnym razie, jeśli jeden z argumentów jest `unsigned int`, drugi jest przekształcany do typu `unsigned int`.
 10. W przeciwnym razie oba argumenty są typu `int`.

- Przykład:

```
char c;  
int i;  
float f;  
double d, wynik;
```

```
wynik = (c/i) + (f*d) - (f+i)  
      int  double double  
      |    |  
      +----+  
      |  
      double  
      |  
      +----+  
      |  
      double
```

3.11.4 Konwersja jawna - rzutowanie typów

- *Konwersja jawna* (ang. *explicit conversion*) wymuszana jest za pomocą operatora konwersji (*rzutowania*, ang. *cast*) i nosi nazwę rzutowania (ang. *cast*).
- *Operator konwersji* jest to operator jednoargumentowy i ma taki sam priorytet jak inne operatory jednoargumentowe.
- *Składnia*:

```
(<nazwa_typu>) <wyrażenie>
```

Jest to tzw. *notacja rzutowania w starym stylu*, przejętym z języka C.

```
nazwa_typu (wyrażenie)
```

Jest to *notacja rzutowania stylu C++*.

Oprócz powyższych standard języka C++ ma nowy sposób rzutowania, który będzie przedstawiony w wykładzie "*Programowanie obiektowe*".

- Przykład

```
int n=2;  
double wynik;  
wynik = 1/(double)n; // bez rzutowania otrzymalibyśmy 0.
```