

Podstawy Programowania

Wykład nr 4: Pozostałe operatory, arytmetyka, konwersje pomiędzy typami

dr hab. inż. Dariusz Dereniowski

Katedra Algorytmów i Modelowania Systemów
Wydział ETI, Politechnika Gdańska

Operatory bitowe & ^ | ~

- & realizuje operację logicznej koniunkcji (AND)
- ^ realizuje operację sumy modulo 2 (EXOR)
- | realizuje operację logicznej alternatywy (OR)
- ~ realizuje operację logicznej negacji (NOT)

Definicja spójników logicznych:

a	b	AND	EXOR	OR	NOT a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	0	1	0

Przykłady (przyjmujemy tutaj, że typ int jest 32 bitowy):

```
a = 13 & 5;           /*a przyjmuje wartość 5*/  
a = 9 | 37;           /*a przyjmuje wartość 45*/  
a = 11 ^ 18;          /*a przyjmuje wartość 25*/  
a = 21 & (~4);         /*a przyjmuje wartość 17*/
```

Przesunięcia bitowe \gg oraz \ll

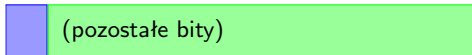
- Wyrażenie: $a \gg b$ (operandy całkowite) zwraca liczbę, której reprezentacja bitowa powstaje poprzez przesunięcie bitów liczby a o b pozycji w **prawo**.
- Wyrażenie: $a \ll b$ (operandy całkowite) zwraca liczbę, której reprezentacja bitowa powstaje poprzez przesunięcie bitów liczby a o b pozycji w **lewo**.

Uwagi:

- Operand b nie może być ujemny.
- Jeśli a jest typem ze znakiem, ma nieujemną wartość i $a \cdot 2^b$ jest reprezentowalne w wynikowym typie, to jest to wynikowa wartość operacji $a \ll b$. W innym przypadku rezultat jest niezdefiniowany dla ujemnych a .
- Jeśli a jest typem ze znakiem i ma nieujemną wartość, to wynik $a \gg b$ jest zależny od implementacji.
- Operacja $a \gg 1$ jest równoważna $a/2$.
- Operacja $a \ll 1$ jest równoważna $a*2$.
- Operatory przesunięć bitowych nie zmieniają wartości operandów (dotyczy sytuacji, gdy są to zmienne).

Reprezentacja liczb całkowitych

Bity numerujemy od zera (skrajnie prawy bit jest bitem zerowym). Bit znaku ma numer 31 dla 4-bajtowych typów (np. int na niektórych architekturach), 15 dla 2-bajтового typu (np. short int na niektórych architekturach) oraz numer 7 dla typu char.



↑
bit znaku (=0 dla liczby nieujemnej oraz =1 dla liczby ujemnej)

```
#include <iostream>
using namespace std;
/* Program ilustrujący reprezentację binarną liczby typu int */
int main() {
    int a = -1, b;

    for (b=8*sizeof(int)-1; b>=0; b--)
        cout << ((a & (1U<<b)) > 0);
    cout << endl;

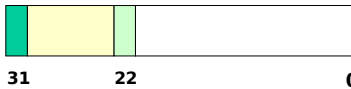
    a = 13;
    for (b=8*sizeof(int)-1; b>=0; b--)
        cout << ((a & (1U<<b)) > 0);
    cout << endl;

    return 0;
}
```

```
11111111111111111111111111111111
000000000000000000000000000000001101
```

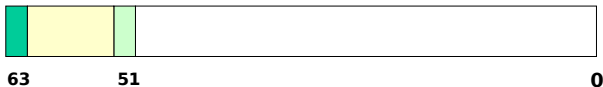
Reprezentacja liczb zmiennoprzecinkowych (IEEE 754)

float



$$\text{liczba} = \pm \text{mantysa} \cdot 2^{\pm \text{wykładnik}}$$

double



long double



- znak
wykładnika



- wykładnik



- znak liczby



- mantysa

Pozostałe operatory

- Operator wyliczeniowy:

`a, b, c;`

`a, b` i `c` są dowolnych typów. Opracowanie wyrażenia jest zgodne z kolejnością (od lewej do prawej). Typowe zastosowanie: wszędzie tam, gdzie nie jest możliwe użycie separatora `;`. Na przykład: `for (i=0, j=0; i < n; i++, j+=2)`

- Operatory wyboru składowych (`.` oraz `->`): omówimy później.
- Operator indeksowania `[]`: omówimy później.
- Operator pobrania adresu `&`: omówimy później.
- Operator dostępu do zmiennej wskazywanej `*`: omówimy później.

Opracowywanie wyrażeń

Opracowywanie wyrażenia to wszystkie czynności (obliczenia i inne) wykonywane podczas przetwarzania wyrażenia.

Kolejność wykonywania działań:

- W przypadku operatorów o różnych priorytetach decyduje priorytet,
np.: $a + b * c \mapsto a + (b * c)$
- W przypadku operatorów dwuargumentowych o identycznych priorytetach decyduje wiązanie:
 - **od lewej do prawej**: $a \circ b \circ c \mapsto (a \circ b) \circ c$.
Dotyczy wszystkich operatorów z wyjątkiem przypisania i operatora warunkowego.
 - **od prawej do lewej**: $a = b = c \mapsto a = (b = c)$.
Dotyczy operatora przypisania i warunkowego.
- Dla szeregu operatorów, kolejność wartościowania (obliczania operandów) nie jest ściśle określona i zależy od implementacji.

Priorytety operatorów

Im wyżej operator znajduje się w tabeli, tym wyższy jego priorytet.

Typ	Operatory	Wiązanie*
Wyrażenia	() [] . -> wyrażenie++ wyrażenie--	L⇒R
Unarne	* & + - ! ~ ++wyrażenie --wyrażenie (rzutowanie) sizeof	R⇒L
Binarne	* / % + - >> << > < <= >= == != & ^ && 	L⇒R
Warunkowy	?:	R⇒L
Przypisania	= += -= *= /= %= >>= <<= &= ^= =	R⇒L
Wyliczeniowy	,	R⇒L

* L⇒R (od lewej do prawej); R⇒L (od prawej do lewej)

Uwaga: używamy nawiasów, aby poprawić czytelność wyrażień.

Kolejność wykonywania obliczeń – punkty sekwencyjne

- Wyrażenia rozdzielone średnikami są wartościowane po kolei.
- Wyrażenia rozdzielone przecinkami są wartościowane po kolei (dotyczy przecinka rozdzielającego definicje zmiennych, nie dotyczy przecinka rozdzielającego argumenty funkcji).
- Wyrażenia w blokach `if(...)`, `switch(...)`, `while(...)` i `for (...; ... ; ...)` są wartościowane w odpowiednim momencie, zgodnie ze specyfikacją działania warunku/pętli.
- Argumenty przekazywane do funkcji są obliczane przed jej wykonaniem, a wartość zwracana (`return`) przed jej zakończeniem.
- Lewy operand operatora `&&` jest zawsze wartościowany przed prawym, a jeśli jego wartość jest zerowa, prawy operand nie jest wartościowany.
- Lewy operand operatora `||` jest zawsze wartościowany przed prawym, a jeśli jego wartość jest niezerowa, prawy operand nie jest wartościowany.
- W operatorze `? :`, warunek jest wartościowany przed odpowiednim (wybrany) spośród następujących po nim wyrażen.

Typowe błędy programisty

- `if (0 < a < 5) ...` Wyrażenie w nawiasie jest zawsze prawdziwe, ponieważ kompilator traktuje je następująco: $(0 < a) < 5$. Wówczas wyrażenie w nawiasie przyjmuje wartość 0 lub 1, więc otrzymujemy $0 < 5$ lub $1 < 5$, co jest zawsze prawdziwe.
- `if (i = 5) ...` Wyrażenie w nawiasie jest zawsze prawdziwe: następuje przypisanie liczby 5 do zmiennej i, w wyniku czego wartością wyrażenia w nawiasie jest 5 (czyli "prawda").
- `if (i != 5) ...` Wyrażenie jest traktowane jako `i!=(5)`. Czyli, najpierw obliczana jest negacja 5, a następnie rezultat jest przypisywany do zmiennej i.
- `cout << 011;` Instrukcja wypisze na ekran liczbę 9.
- `if (a || b++) ...` Programista nie może liczyć na to, że inkrementacja zmiennej b nastąpi. (Nie nastąpi jeśli $a \neq 0$.)
- `if (a && b++) ...` Programista nie może liczyć na to, że inkrementacja zmiennej b nastąpi. (Nie nastąpi jeśli $a=0$.)
- `if (6 & 2 == 2) ...` Wyrażenie fałszywe, gdyż jest ewaluowane jako `6 & (2 == 2)` (ze względu na priorytety)

Konwersja arytmetyczna – zasady

W języku C/C++ wyrażenia arytmetyczne mogą obejmować zarówno operandy całkowite, jak i zmiennoprzecinkowe. Konwersja typów następuje przede wszystkim w następujących sytuacjach:

- **konwersja wymuszona przez operator rzutowania bądź sufix przy stałej:**
(int)(a+3.5), (float)x, (unsigned)y, (double)5, 5F, 5U
- **konwersja przy przypisaniu:** double x = -7.7; int a = x; Uwaga: konwersja typu zmiennoprzecinkowego na całkowity zawsze odbywa się pośrednio poprzez typ long, poprzez odrzucenie części liczby po kropce (w powyższym przykładzie: a = -7)
- **konwersja przy wykonywaniu operacji arytmetycznych:**
 - **long double** o **wyrażenie** – obydwa operandy traktowane jako long double
 - w przeciwnym razie: **double** o **wyrażenie** – obydwa jako double
 - w przeciwnym razie: **float** o **wyrażenie** – obydwa jako float
 - w przeciwnym razie: **unsigned long** o **wyrażenie** – obydwa jako unsigned long
 - w przeciwnym razie: **unsigned int** o **long** – obydwa jako unsigned long
 - w przeciwnym razie: **long** o **wyrażenie** – obydwa jako long
 - w przeciwnym razie: **unsigned int** o **wyrażenie** – obydwa jako unsigned int
 - w każdym innym przypadku: obydwa operandy traktowane jako int

Konwersja arytmetyczna – przykłady

```
#include <iostream>
using namespace std;
/* Program ilustrujący potencjalne
   kłopoty związane z konwersją typów */
int main() {
    int a = -1;
    unsigned int b = 1;
    short c=100;
    long long l;
    char znak = '0';

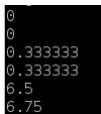
    cout << (unsigned) a << endl;
    cout << (a < b) << endl;      // niejawna konwersja a do typu unsigned int
    cout << a+1 << endl;          // wszystko dobrze (oba operandy typu int)
    cout << a+ (unsigned int)1 << endl; // wszystko dobrze... przypadkiem
    a = -2;
    cout << a+ (unsigned int)1 << endl; // niejawna konwersja a do unsigned int
    cout << c*c*c*c << endl; // wszystko dobrze: mnożenie po konwersji do int
    l = ((long long)1)<<35; // wszystko dobrze
    cout << l << endl;
    l = 1<<35; // możliwy błąd gdy int jest 32-bit. Kompilator może ostrzegać!
    cout << l << endl;
    cout << znak << endl; // wypisujemy znak
    cout << (int) znak << endl; // oraz jego kod ASCII
    return 0;
}
```

```
deren@linux-nb6o:~> g++ -o w04p02 w04p02.cpp
w04p02.cpp: In function 'int main()':
w04p02.cpp:21:10: warning: left shift count >
= width of type [enabled by default]
deren@linux-nb6o:~> ./w04p02
4294967295
0
0
0
4294967295
1000000000
34359738368
0
0
48
```

Konwersja arytmetyczna – przykłady

```
#include <iostream>
using namespace std;
/* Program ilustrujący potencjalne
   kłopoty związane z konwersją typów
   (zaokrąglenia) */
int main() {
    double d = 1/3; // dzielenie całkowitoliczbowe!
    int i = 1/3;

    cout << i << endl;
    cout << d << endl;
    d = 1/3.0; // operandy zmiennoprzecinkowe
    cout << d << endl;
    d = (1.0F)/3; // operandy zmiennoprzecinkowe
    cout << d << endl; // j.w.
    // Uwaga: dwa poniższe wyrażenia dają
    // inne wyniki. Powód: opracowywanie
    // wyrażenia "od lewej do prawej"
    d = 27/2/2.0;
    cout << d << endl;
    d = 27.0/2/2;
    cout << d << endl;
    return 0;
}
```



```
0
0
0.333333
0.333333
6.5
6.75
```