Wykład 3

3. Operatory i wyrażenia

3.1. Pojęcia podstawowe

3.2. Klasyfikacja operatorów

- standardowe konwersje,
- zmiana typu i operator konwersji (rzutowanie),
- operatory arytmetyczne,
- operatory inkrementacji i dekrementacji,
- operatory przypisania,
- operatory bitowe,
- operatory relacji i wyrażenia logiczne,
- operator warunku,
- operator sizeof,
- operator wyliczeniowy (przecinek),
- operator wywołania funkcji,
- wyrażenia kontrowersyjne,
- priorytet i wiązanie operatorów.

3.3. Znaki w C++

3.1. Pojęcia podstawowe

- *Operatory* - określają jakie operacje mają być wykonane i w jakiej kolejności (np. operatory arytmetyczne +, *, /).

- *Wyrażenie* - ciąg (kombinacja) operatorów, operandów i znaków przestankowych, które określają sposób (regułę) wykonywania obliczeń (np. a=2 , -a, x>4) i wyznaczania pewnej wartości.

- *Instrukcja* – kompletne polecenie wydane komputerowi w danym języku; w języku C/C++ instrukcje zawsze kończą się *średnikiem*, np. instrukcje wyrażeniowe b=3; 7; a=(x>2);

- *Opracowywanie wyrażenia* – wykonywanie obliczeń podczas analizy wyrażenia w celu wyznaczenia jego wartości (wykonywanie operacji na argumentach - operandach, tj. stałych lub zmiennych); zazwyczaj są to proste operacje jedno-, dwu- lub trzy-argumentowe (np. -a, a+b, c=b=a).

- *Priorytet operatorów* – określa kolejność wykonywania operacji przez operatory (uwzględniany jeśli dwa operatory mają wspólny operand, np. y = 2 * 10 + 3*7; w tym przypadku 10 jest wspólnym operandem dla * i +; wyższy priorytet ma * (mnożenie), stąd y = 41).

- *Wiązanie* – określa sposób łączenia operatora z operandami (operandem) w przypadku, gdy priorytety operatorów są jednakowe wiązanie „lewe" oznacza, że operacje są wykonywane od lewej do prawej, natomiast wiązanie „prawe" oznacza, że od prawej do lewej; np. y = 20 / 2 *5; operatory / i * dzielą operand 2 i mają ten sam priorytet; ze względu na „lewe" wiązanie y = 10*5 = 50;

- W języku C/C++ większość operatorów ma wiązanie lewostronne; do wyjątków należą m.in. operatory :: (widoczności), ++ i -- (poprzedzające), ?: (warunkowy), = (przypisania), które mają wiązanie prawostronne.

Operacje związane z operatorami o wyższym priorytecie są wykonywane jako pierwsze. Jeżeli priorytety operatorów są równe dla danego operandu, to kolejność wykonania operacji jest określona przez wiązanie.

Np. Rozpatrzmy wyrażenie: x = b + c ∗ d.

Wyrażenie zawiera trzy operatory: (=), (+) i (∗). Najwyższy priorytet ma operator mnożenia, potem dodawania i na końcu przypisania. W takiej taż kolejności zostaną wykonane operacje, tzn. x = (b + (c∗d)).

Np. Rozpatrzmy wyrażenie: x = a + b – c.

Priorytety operatorów (+) i (-) są jednakowe. Ponieważ wiązanie jest od lewej do prawej x = ((a + b) – c).

- Operatory mogą być jednoargumentowe lub dwuargumentowe (np. !a; -a; a-b; a/b).

- L-wartość (l-wyrażenie) – wyrażenie reprezentujące zmienną, mogące znaleźć się z lewej strony operatora przypisania (np. x=10; w=x+y); nie jest poprawnym wyrażeniem, np. (x-y)++.

- Modyfikowalna l-wartość – określa obszar pamięci (zmienną), którego zawartość może być zmieniana, np. const x = 10; jest l-wartością, ale nie jest modyfikowalną l-wartością.

W języku C/C++ każde wyrażenie ma wartość. Jest ona obliczana przez wykonanie wszystkich działań zgodnie z priorytetami i kierunkami wiązania operatorów.

Przykładowe wyrażenia:

2 + 4	wartość	6
a = 2 – 3	wartość	-1
3 > 0	wartość	1
3 > 5	wartość	0
2 + (a=3+4)	wartość	9
(2 > 1) + (5>4)	wartość	2

3.2. Klasyfikacja operatorów

Standardowe konwersje

Jeżeli w wyrażeniu występują operandy różnych typów, to wykonywane są standardowe konwersje arytmetyczne mające na celu sprowadzanie różnych typów danych do jednego typu. Konwersje takie są przeprowadzane automatycznie przy obliczaniu wyrażeń.

Ogólna zasada – operandy typów o mniejszych rozmiarach (mniejszy zakres i dokładność) są przekształcane do typów o większych rozmiarach.

Np. int a; float b; double c;

c = 2∗a + b∗a

literał 2 jest typu int, wyrażenie 2∗a jest typu int; w wyrażeniu b∗a zmienna a jest przekształcana do typu float, wyrażenie 2∗a przed dodaniem do b∗a jest przekształcane do typu float; suma ma typ float, a więc przed podstawieniem na zmienną c jest przekształcana do typu double.

- Operandy typów *char*, *unsigned char*, *signed char*, *short*, *unsigned short* i wyliczeniowych (*enum*) są przekształcane do typu *int* lub *unsigned int* wg. następującej zasady: jeżeli liczba jest ze znakiem to w całym bajcie powielany jest bit znaku, w przeciwnym przypadku starszy bajt jest wypełniany zerami (np. char c = -3; 1 1111101 → 11111111 11111101).

- Konwersja z typu całkowitego o rozmiarze mniejszym do większego zachowuje wartość zmiennej, natomiast konwersja odwrotna niekoniecznie.

- Konwersja z typu całkowitego o rozmiarze większym do mniejszego polega na odrzuceniu najwyższych bitów liczby i pozostawieniu bitów odpowiadających mniejszemu typowi.
  Np.
  char a = 2; int b = 3; int w = (30000∗a) / b; (w = -1845)

Przypisanie a = 2 dokonuje konwersji literału int do typu char.
W wyrażeniu 30000∗a zmienna a jest przekształcana do typu int, ponieważ wynik wyrażenia jest typu int wartość 60000 będzie traktowana jako liczba ujemna (przekroczony zakres). Aby temu zapobiec można zastosować modyfikator unsigned, np. 30000U.

- Konwersje między typami całkowitymi o tych samych rozmiarach (np. int i unsigned int) są wykonywane bez zmiany reprezentacji. Należy jednak pamiętać, że w wyniku podstawienia unsigned int na int możemy otrzymać liczbę ujemną, np. 60000U (- 5536).

- Konwersje między typami rzeczywistymi, a całkowitymi polegają na *odrzuceniu części ułamkowej* liczby rzeczywistej. W przeciwną stronę przez nadanie liczbie rzeczywistej wartości liczby całkowitej.

- Konwersje między typami rzeczywistymi o różnych rozmiarach wykonywane są przez zaokrąglenie do najbliższej wartości docelowego typu.

Zmiana typu i operator konwersji (rzutowanie)

W programach możemy wymusić na kompilatorze użycie określonego (właściwego) typu. Język C/C++ udostępnia następujące sposoby zmiany typu (rzutowania typów):

(nazwa_typu) wyrażenie lub nazwa_typu (wyrażenie).

```
Np.  int a  = 2;
printf("%f ", (float) a);   printf("%f ", float (a));   //  wynik 2.000000

float x = 2.7;
printf("%d ", (int) x );    printf("%d ", int (x) );  // wynik 2
```

Operatory

- Operatory arytmetyczne
  - plus, minus jednoargumentowy (np. +a, -b),
  - dwuargumentowe addytywne: a + b, a - b,
  - dwuargumentowe multiplikatywne:
    mnożenie (a ∗ b), dzielenie (a / b), reszta z dzielenia (a % b).

Jeżeli a i b należą do typów całkowitych, to wartością wyrażenia a / b jest:
- największa liczba całkowita mniejsza niż rzeczywisty iloraz, gdy a i b są tego samego znaku (np. 7 / 3 = 2),
- najmniejsza liczba całkowita większa niż rzeczywisty iloraz, gdy a i b mają różne znaki (np. –7 / 3 = -2).

```
Analogicznie:     7 % 3 = 1,   -7 % 3 = -1
                  int a=2;  float x = a / 3;        // x = 0.0
                  float y = (float) a / 3;    // y = 0.6666
```

Podczas wykonywania operacji arytmetycznych należy pamiętać o tym, aby nie został przekroczony zakres wykorzystywanych zmiennych.

Operatory inkrementacji i dekrementacji

Pozwalają zwiększyć lub zmniejszyć wartość zmiennej o jeden.

- Operator zwiększania: ++ .

- Operator zmniejszania: -- .

Operatory mogą być stosowane tylko do modyfikowalnych l-wyrażeń, oznaczających obszar pamięci, w którym jest przechowywana wartość typu skalarnego (arytmetyczna lub wskazująca).

Forma przyrostkowa (następująca):

- zmienna++ // zwiększenie wartości zmiennej po użyciu w wyrażeniu,
- zmienna-- // zmniejszenie wartości zmiennej po użyciu w wyrażeniu.

Forma przedrostkowa (poprzedzająca):

- ++zmienna // zwiększenie wartości zmiennej przed użyciem w wyrażeniu,
- --zmienna // zmniejszenie wartości zmiennej przed użyciem w wyrażeniu.

Operatory ++ i – mają bardzo wysoki priorytet. Forma *przyrostkowa* należy do grupy operatorów o najwyższym priorytecie. Forma *przedrostkowa* ma priorytet o jeden poziom niższy (w wyrażeniach najpierw uwzględniamy zasadę działania operatorów a potem ich priorytet). Operatory działają dla zmiennych całkowitych i rzeczywistych.
Np.

```
int a,  b,  c = 4;
double x=0, y=1;
    c++;                    // c = 5
    --c;                    // c = 4
    a = 2∗c++;              //  a =2∗4=8;   c = 5
    a = 2∗++c;              // c=6;   a = 12
    x++;                    // x=1
    y = ++y + x;            // y=3
    b = (a--) + (++c);      // c=7;  b = 12 + 7 = 19;  a = 11
```

Wartość b jest obliczana w sposób następujący: najpierw c jest zwiększane o jeden, potem a jest sumowane z c i wynik jest podstawiany do b.
Na koniec wartość a jest zmniejszana o jeden.

Operatory przypisania

Realizujemy za ich pomocą podstawienia pod zmienne.

- Prosty operator przypisania: =

np. int a = 2; przypisuje wartość po prawej stronie zmiennej o nazwie a znajdującej się po lewej stronie.

W przypadku wyrażenia a = b, operand a musi być modyfikowalną l-wartością. Operand a jest zastępowany wartością b. Operand b jest konwertowany do typu a. Wartość a jest modyfikowana.

Wyrażenie z operatorem przypisania posiada wartość, która może być wykorzystywana, np. w = (a=x) - (b=y), odpowiada: a=x, b=y, w = a-b.

Kilku zmiennym można nadać jednocześnie taką samą wartość:

```
float  a, b, c;

a = b = c = 2.15;
```

Operator przypisania (=) jest wiązany z operandami z prawej do lewej.

```
float x;     int k;
x = k = 1.4;         // otrzymamy  x = (k = 1.4) = 1
```

- Złożone operatory przypisania: operator=

Operatory arytmetyczne:

```
x += y       // x = x + y
x -= y       // x = x – y
x ∗= y       // x = x ∗ y
x /= y       // x = x / y
x %= y       // x = x % y
```

Np. Zamiast c = c + a + b można napisać c += a + b;

```
    float x, y;   x += 5.0;   x -=y;
```

Wyrażenie ze złożonym operatorem przypisania posiada wartość, która może być wykorzystywana, np. w = (a += x) - (b∗ = y), tj. a=a + x, b=b ∗ y, w = a – b.

Operatory bitowe

Można za ich pomocą realizować funkcje logiczne (typu „bit do bitu”).

```
c = a & b       - iloczyn logiczny (AND),
c = a ^ b       - suma modulo 2 (EXOR),
c = a | b       - alternatywa (OR),
c = ~a          - negacja (NOT),
c = a << b      - przesunięcie w lewo a o b bitów (SHL),
c = a >> b      - przesunięcie w prawo a o b bitów (SHR).
```

Każdy z operandów wszystkich wymienionych typów bitowych musi być typu całkowitego.

Wykorzystując złożone operatory przypisania możemy napisać:

```
c &= a;          c = c & a;
c |= a;          c = c | a;
c ^= a;          c = c ^ a;
c <<= a;         c = c << a;
c >>= a;         c = c >> a;
```

W przypadku przesunięcia bitowego w lewo (<<) młodsze (zwalniane bity) są zastępowane zerami (operacja mnożenia przez 2)

Przy przesunięciu w prawo (>>) zwalniane bity dla typów bez znaku są uzupełniane zerami, natomiast dla typów ze znakiem (signed) są zastępowane bitem znaku, co zapewnia zachowanie znaku liczby przy przesuwaniu (operacja dzielenia przez 2).
Np.

```
    unsigned char c = 1; // c = 0000 0001
    c << 1          // c = 00000010 = 2
    c << 2          // c = 00000100 = 4

    char c = 127 ;  // c = 01111111
    c >> 1          // c = 00111111 = 63
    c >> 2          // c = 00011111 = 31

    signed char c = -64 ;   // c = 11000000
    c >> 1                  // c = 11100000  (-32)
    c >> 2                  // c = 11110000  (-16)
```

Przykład 3.1. Operacje bitowe

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>

void main (void)
{
  int a, b, c;
  signed char z = -64;
  char u;

  clrscr();
  a = 0x0F;      // 00001111 = 15
  b = 0x17;      // 00010111 = 23

  c = a & b;
  printf("%3d AND %3d = %3d \n",a,b,c);       // c = 7
  c = a ^ b;
  printf("%3d EXOR %3d = %3d \n",a,b,c);     // c = 24 = 00011000
  c = a | b;
  printf("%3d OR %3d = %3d \n",a,b,c);        // c = 31 = 00011111
  c = ~a;
  printf(" NOT %3d = %3d \n",a,c);             // c = -16 = 11110000
  c = a << 2;
  printf("%3d SHL 2 = %3d \n",a,c);            // c = 60
  c = a >> 2;
  printf("%3d SHR 2 = %3d \n\n",a,c);         // c = 3

  u = z;     // u = z = -64
  u >>= 1; printf("%d SHR 1 = %d \n",z,u);   // u = -32
  u = z;
  u >>= 2; printf("%d SHR 2 = %d \n",z,u);   // u = -16
  u = z;
  u >>= 3; printf("%d SHR 3 = %d \n",z,u);   // u = -8

  getch();
  }
```

Operatory bitowe są najbardziej przydatne do ustawiania i testowania zawartości pewnych obszarów pamięci, w których przechowywane są flagi-znaczniki. W zależności od ustawienia określonych bitów flag mogą zachodzić różne zdarzenia.
Np.

```
  enum {                          // stałe testowania pozycji bitowych
        flg1 = 0x01,
        flg2 = 0x02,
        flg3 = 0x04  };

char flaga = 0;

flaga = flaga | flg2;            // ustawienie drugiego bitu
flaga = flaga | (flg1 | flg3);   // ustawienie pierwszego i trzeciego bitu
```

Operatory bitowe wykorzystywane są również do wyodrębniania starszego lub młodszego półbajtu zmiennej typu char.
Np.

```
// wprowadzamy znak

char c = getchar();

// wartość zapisana w 4 młodszych bitach

printf("\n%d", c & 0x0f);

// wartość zapisana w 4 starszych bitach

printf("\n%d", (c >> 4) & 0x0f );
```

Gdyby c było typu unsigned char, to przy przesuwaniu c >> 4 bity zwalniane w starszym półbajcie zapełniane byłyby zerami, a więc bitowa koniunkcja & 0x0f nie byłaby potrzebna. Ponieważ jednak zmienna typu char może być ujemna starsze bity są eliminowane, gdyż mogą zawierać bit znaku.

Operatory relacji i wyrażenia logiczne

Operatory porównania – pozwalają stwierdzić, czy między operandami zachodzi określona relacja.

Operatory relacyjne:

```
a < b       // a mniejsze niż b
a > b       // a większe niż b
a <= b      // a mniejsze równe b
a >= b      // a większe równe b
a == b      // a równe b
a != b      // a różne b
```

Należy zwrócić uwagę, że sprawdzanie równości realizowane jest za pomocą ==, a przypisanie za pomocą =.

Jeżeli operandy a i b spełniają relację określoną przez operator, to wartością wyrażenia jest 1, w przeciwnym razie 0.

Wyrażenia relacyjne:

```
int a=2;   int b=3;   int w1, w2;
w1 = (a > b);    w2 = (a <= b);              // w1=0;     w2=1;
```

Operatory logiczne:

```
a && b              // koniunkcja   a && b = 1, tylko wtedy gdy a ≠ 0 i b ≠ 0
a || b              // alternatywa  a || b = 0,    tylko wtedy gdy a=0 i b=0
!a                  // zaprzeczenie logiczne; dla a=0 jest !a=1, dla a≠0 jest !a=0
                    // np. !0.0 = 1;  !'\0'=1; !-1=0;  !5=0; !'0'=0
```

Operatory relacji posiadają wyższy priorytet niż operatory logiczne.

W wyrażeniu: (a < 0) || (a > 5)
można pominąć nawiasy, tzn. możemy napisać a < 0 || a > 5.

Wyrażenia logiczne są obliczane od lewej do prawej. Proces obliczeń kończy się w momencie, gdy znany jest wynik całego wyrażenia
(np. a && (b >0 || c < 0) jeśli a = 0, to wartość wyrażenia równa się 0).

W wyrażeniach: (++i < 10); // zwiększ i; porównaj
                (i++ < 10); // porównaj; zwiększ i
W wyrażeniu: (a++ < 5 && a + b > 10) najpierw obliczone zostanie w całości wyrażenie (a++ < 5), a więc obliczona zostanie wartość a<5 i zostanie zwiększona zmienna a, następnie obliczone zostanie wyrażenie a + b > 10.

Operator warunku

Wyrażenie z użyciem operatora warunkowego ma następująca postać:

```
w1 ? w2 : w3
```

Wyrażenie w1 musi być typu prostego, natomiast typy wyrażeń w2 i w3 muszą być takie same lub zgodne, tzn. dopuszczające wzajemne przekształcenie.

```
Np.  min = (a < b) ? a : b;          // obliczanie min(a,b)
```

Obliczanie wyrażenia. Najpierw jest obliczane wyrażenie w1.
Jeśli wartość wyrażenia w1 jest niezerowa (prawda), to obliczane jest wyrażenie w2, natomiast w3 jest ignorowane. Jeżeli w1=0, to obliczana jest wartość wyrażenia w3, natomiast w2 jest ignorowane.

Rezultat wyrażenia warunkowego jest wartością. Można więc budować wyrażenia postaci:

```
d = (a < 5) ? (b=10) : (c=20);      //  wynik d = b = 10  lub   d = c = 20
d = (a < 5) ? b=10 : c   = 20;      //   wynik d = b = 20   lub  d = c = 20

int x = (a > b ? 1 : -1);
printf("\nWartosc zmiennej x jest  %sujemna", (x >= 0) ? "nie" : "");
```

Operator sizeof

Sposób użycia: sizeof(nazwa_zmiennej) lub

sizeof(typ_danej).

W pierwszym przypadku podawany jest rozmiar w bajtach pamięci zajmowanej przez zmienną, w drugim rozmiar podanego typu.

Np.

```
char a;   int   b;   long c;   double d;

sizeof(char) = 1;          sizeof(a) = 1;
sizeof(int) = 2;           sizeof(b) = 2;
sizeof(long) = 4;          sizeof(c) = 4;
sizeof(double) = 8;        sizeof(d) = 8;
```

Przykład 3.2. Wykorzystanie wyrażeń relacyjnych i logicznych

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>

const MIN = 10;
const MAX = 99;


void main (void)
{
   int a = 5;   int b = 5;    int c = 50;
   int zn1, zn2, w_zakresie, rowne;

   clrscr();

   // sprawdzenie zakresu [MIN, MAX]

    zn1 = (MIN <= a);
    zn2 = a <= MAX;    //priorytet = mniejszy niz porownania
    w_zakresie = zn1 && zn2;

    printf("\n%d miesci sie w zakresie od %d do %d: %s",
           a, MIN, MAX, (w_zakresie) ? "TRUE" : "FALSE"); // FALSE

   // sprawdzanie rownosci dwoch lub wiekszej ilosci liczb
    rowne =  a == b || a == c || b == c;
    printf("\nco najmniej dwie liczby sa rowne: %s",
           rowne ? "TRUE" : "FALSE");       // TRUE

   // inne testy
   printf("\n%d != %d: %s", a,b, (a!= b) ? "TRUE": "FALSE");  // FALSE

   printf("\n%d < %d: %s", a,b, (a < b) ? "TRUE": "FALSE");   // FALSE

   printf("\n%d <= %d: %s", a,b, (a <= b) ? "TRUE": "FALSE"); // TRUE

   printf("\n(%d = %d) AND (%d <> %d) : %s", c, a, b, c,
           (c == a && b != c) ? "TRUE": "FALSE");          // FALSE

   getch();
}
```

Priorytet i wiązanie operatorów

	Operatory	Wiązanie
1.	() [] -> :: . ++ i -- (przyrostkowe)	od lewej do prawej
2.	! ~ + - ++ -- & * (typ) sizeof new delete	*od prawej do lewej*
3.	.* ->* (dostęp do składowych obiektów)	od lewej do prawej
4.	* / %	od lewej do prawej
5.	+ - (dwuargumentowe)	od lewej do prawej
6.	<< >>	od lewej do prawej
7.	< <= > >=	od lewej do prawej
8.	== !=	od lewej do prawej
9.	&	od lewej do prawej
10.	^	od lewej do prawej
11.	|	od lewej do prawej
12.	&&	od lewej do prawej
13.	||	od lewej do prawej
14.	?:	*od prawej do lewej*
15.	= *= /= %= += -= &= ^= |= <<= >>=	*od prawej do lewej*
16.	,	od lewej do prawej

Im wyżej w tabeli tym wyższy priorytet.

W języku C++ wszystkie operatory oprócz następujących:

. - bezpośredni dostęp do składowej,
.* - dereferencja,
:: - widoczność,
?: - warunek,

mogą być przeciążane (zmiana znaczenia w danej klasie).

Operator wyliczeniowy (przecinek)

Wyrażenie wyliczeniowe ma postać:

(wyr_1, wyr_2, ..., **wyr_n**)

gdzie wyr_i (i=1,...,n) są operandami dowolnych typów.

Operandy są opracowywane od lewej do prawej. Typ i wartość całego wyrażenia określa operand wyr_n.

Wynik wyrażenia wyliczeniowego jest l-wartością,

np. (x=2, y=3, z = x+y)++ // z =6

```
int  a = 6,  b = 2,  c = 4,  wynik;
wynik = (b += a,  c -= 2,  a *= c);
            \\  b = 8,  c = 2,  a = 12,  wynik = 12)
```

Operator wywołania funkcji ()

Nazwa_funkcji(arg1, arg2, ..., argN);

Nazwa funkcji jest wyrażeniem wskazującym funkcję, natomiast arg1, ..., argN są aktualnymi parametrami wywołania funkcji (np. y = sin(6);).

Wyrażenia kontrowersyjne

Nie należy konstruować wyrażeń, które budzą wątpliwości.

Np. a=1; b=2; x = a+++b; // na ogół interpretowane jako a++ +b
// przyrostkowy ++ znajdowany jako najdłuższy ciąg znaków
a=1; x = 3*a + (2 + a++)*4;
W wyrażeniu tym w zależności od interpretacji 3*a może być wyliczone jako pierwsze (x = 3*1 + (2+1)*4 = 15; a=2;) lub jako drugie ((2+1)*4; a=2; x = 3*2 + 3*4 = 18).

printf("%d %d", a, a*a++); // a, a*a, a++ lub a*a, a++, a

Podczas przekazywania parametrów do funkcji nie wiadomo, które z wyrażeń będzie obliczone jako pierwsze. Należy unikać wyrażeń tego typu a w razie wątpliwości stosować nawiasy lub rozłożyć wyrażenie na prostsze podwyrażenia. W przypadku: int b= --(a++); // błąd - nie jest l-wyrażeniem

3.3. Znaki w C++

W języku C++ znaki są interpretowane w sposób dwoisty:

- jako znaki,
- jako liczby całkowite.

Każdy znak ma swój numer określony przez kod ASCII (od 0 do 255).

W wyrażeniach znaki są zastępowane ich kodami, np.

int i = 'A' + 'B' + 5; // i = 65 + 66 + 5

```
int ofs_ASCII = 'a' – 'A';
char ch;
```

Zamiana małych liter na duże:

ch = ch – ofs_ASCII;

Zamiana dużych liter na małe:

ch = ch + ofs_ASCII;

W niektórych systemach przesyłania informacji istnieje możliwość przesyłania danych binarnych w postaci tekstu o alfabecie 64 znakowym. Alfabet obejmuje małe i duże litery, cyfry oraz znaki + i -. Wówczas powstaje problem jak zamienić informację binarną na informację zapisaną w kodzie 64 znakowym. Najczęściej postępuje się w taki sposób, że każde 3 bajty 8 bitowej informacji oryginalnej są zamieniane na 4 bajty informacji 6 bitowej.

```
|1111 1100 | 1111 1100 | 1111 1100 |
 |--------| |----------| |----------| |--------|
```

Otrzymujemy 4 bajty informacji: d1 = 00111111, d2 = 00001111,
d3 = 00110011, d4 = 00111100.