Wykład 3

- Operatory
- Przekształcenia typów
- Wprowadzanie / wyprowadzanie danych

dr inż. Marcin Radom

Operatory języka C

- Arytmetyczne, logiczne, bitowe
  Priorytety operatorów

Operatory arytmetyczne

- **Dwuargumentowymi operatorami arytmetycznymi są: * / + – %**
- **Dzielenie całkowite** obcina część ułamkową wyniku.
- **Operator dzielenia modulo %** (reszta z dzielenia). Wyrażenie *x % y* daje w wyniku resztę z dzielenia *x* przez *y*, czyli 0 jeśli *x* jest podzielne przez *y*, np. 9 % 4 to 1

```
int  liczba,  nieparzystosc;
nieparzystosc = liczba % 2; // 0 - p, 1 – np
```

- Operatora **%** nie można stosować do danych typu **float** oraz **double**.
- Kierunek zaokrąglania oraz znak wyniku są zależne od znaku argumentów operacji arytmetycznej.
- **+ i – (dwuargumentowe)** mają priorytet, który jest niższy niż priorytet operatorów * , / oraz %. Te trzy z kolei mają niższy priorytet niż jednoargumentowe – i + (określające znak).
- Operatory arytmetyczne są lewostronnie łączne.

Operatory arytmetyczne

- Np. dla wyrażenia:

$$\frac{3x^2+5x-1}{7x[(2x+3)(1-x)+5]+3}$$

```
(3 * x * x + 5 * x - 1) /
(7 * x ((2 * x + 3) * (1 - x) + 5) + 3)
```

- Akcje podejmowane po wystąpieniu nadmiaru (występuje, gdy wynik operacji arytmetycznej jest większy niż górny zakres danego formatu liczb binarnych) lub niedomiaru (występuje, gdy wynik operacji arytmetycznej jest mniejszy od dolnego zakresu formatu liczb binarnych) zależą od maszyny. Np.:

```
int   a = 1700000000,   b = 1900000000, suma;
suma = a + b;   // nadmiar stałopozycyjny
float   x = 0.5e35,   y = 0.2e5,   z;
z = x * y;      // nadmiar zmiennopozycyjny
```

Operatory arytmetyczne – Przykład1

```
int main(){
        // dobierz wartość zmiennej A tak,
        // aby uzyskać nadmiar stałopozycyjny
        int W, A = 2100000000;
        W = A + A;
        printf("%d\n\n", W); // liczba ujemna

        // dobierz wartość zmiennej X tak,
        // aby uzyskać nadmiar zmiennopozycyjny
        double Z, X = 9E307;
        Z = X + X;
        printf("%32.15le\n\n", Z);

        // dzielenie całkowite odcina część ułamkową
        int p;
        p = 9/5;
        printf("\n%d\n\n", p); // 1
```

Operatory arytmetyczne – Przykład1 cd.

```
// Może wystąpić utrata dokładności (precyzji). Jest to wynikiem
// tego, że ta sama liczba bitów (obydwa typy są 4-bajowe) musi
// w przypadku typu float przechowywać cechę i mantysę, więc nie
// jest możliwe pamiętanie pełnego zakresu liczb z typu int z
// dokładnością co do jedynki.
int k, m = 2111333444;
float f;
//double f;

f = m;
k = (int)f;
printf("\n%d\t%d\n\n", m, k); // m = 2111333444 k = 2111333504

// najpierw doda do siebie 2 liczby typu int (wystąpi nadmiar),
// później dokona konwersji wyniku do long long. Gdyby jedna ze
// zmiennych dodawanych była typu long long wynik byłby prawidłowy
int a = 2000222333, b = 2000222333;
long long n;
n = a + b;
printf("\n%lld\n\n", n); // liczba ujemna
// gdy: n = a + (long long) b; to n = 40004444666
```

Operatory arytmetyczne – Przykład1 cd.

```
// Sumowanie dużej liczby z małą (w systemie  zmiennoprzecinkowym)
// może być niedokładne ze względu na utratę precyzji. Należy zawsze
// sumować od wartości najmniejszych do największych.
// Dobierz wartość dużej liczby i wartość małej liczby tak, aby
// suma S1 była równa dużej liczbie a suma S2 była różna od sumy S1
const long N = 10000000;   // liczba dodawań

double big = 1.8E18;    // wartość dużej liczby
double little =  1.2;   // wartość małej liczby

double S1, S2;
int i;
S1 = big;

// sumowanie liczby dużej i małej
for (i = 0 ; i < N ; ++i)
     S1 = S1 + little;
printf("S1 = %28.15E\n", S1); // S1 = 1.8E18
```

Operatory arytmetyczne – Przykład1 cd.

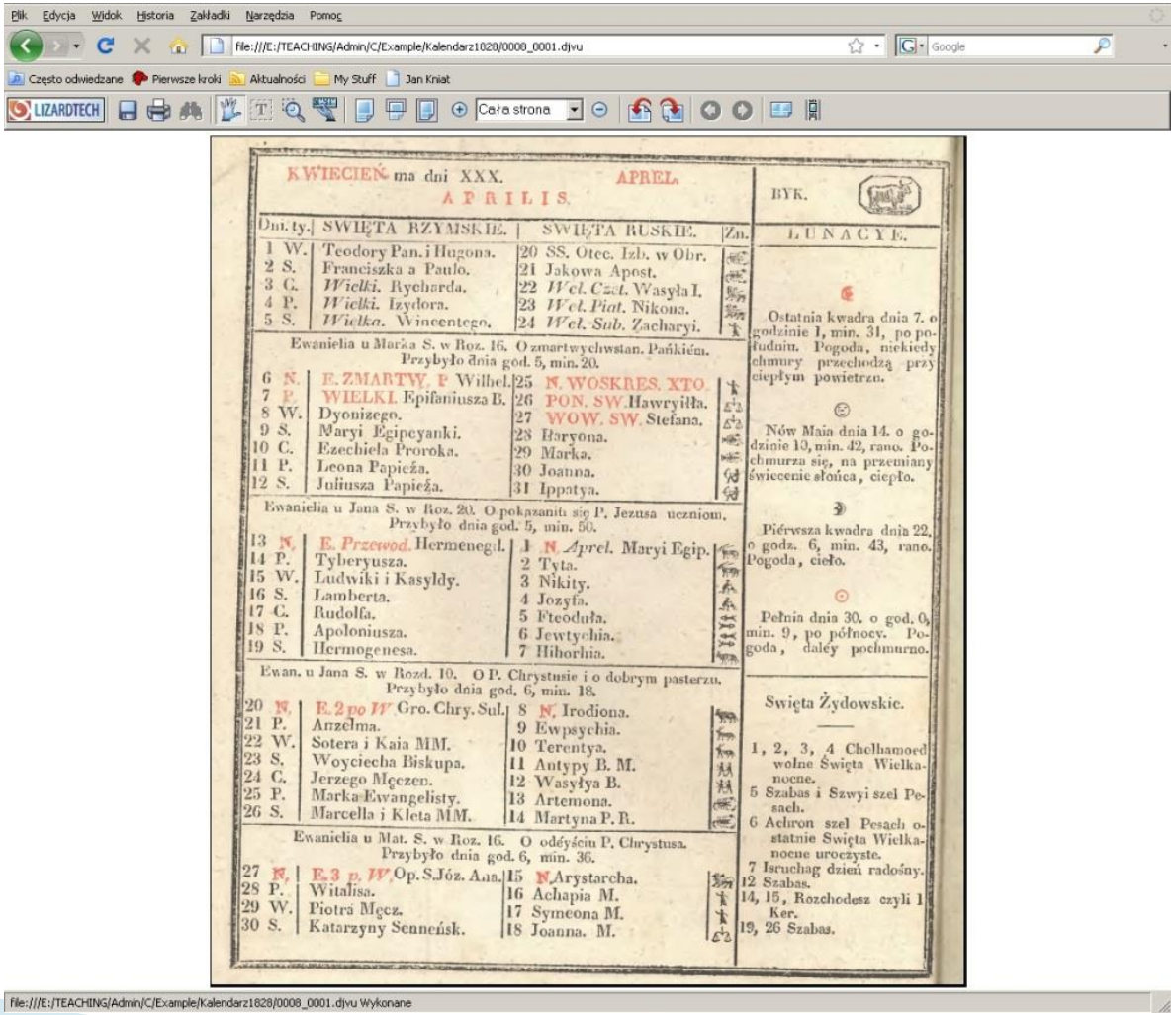
```
        S2 = 0;
        for (i = 0 ; i < N ; ++i)
           S2 = S2 + little;
        S2 = S2 + big;
        printf("S2 = %28.15E\n", S2); // zsumowana mała liczba, S2 = 1.8E18
        // różnica wartości zsumowanej małej liczby i dużej liczby
        printf("S2 - S1 = %23.15E\n\n", S2 - S1); // S2 - S1 = 1.2E7

        // 10 wykonań pętli - pętla nieskończona. Dlaczego? – odpowiedź
        // kryje się w reprezentacji maszynowej liczb. Liczba 0.1 (1,6*2-4)
        // reprezentowana jest jako 0.10000000149011612 (podwójna precyzja)
        // (http://www.h-schmidt.net/FloatConverter/IEEE754.html)
        double X = 0;
        int N = 0;
        while (X != 1.0){
             X += 0.1;
             ++N;
        }
        printf("Koniec. N = %d\n", N);
        return 0;
}
```

Operatory arytmetyczne – Przykład2

- **Algorytm Jeana Meeusa ustalania daty Wielkanocy** (Wynik to dzień i miesiąc. Nie ma od niego wyjątków, wystarczy podać rok).

```
void main(){
	int a, b, c, d, e, f, g, h, i, k, l, m, p, n, y;
	printf("Podaj rok : ");
	scanf("%d", &y);
	a = y % 19;
	b = y / 100;
	c = y % 100;
	d = b / 4;
	e = b % 4;
	f = ((b + 8) / 25);
	g = (b - f + 1) / 3;
	h = (19 * a + b - d - g + 15) % 30;
	i = c / 4;
	k = c % 4;
	l = (32 + 2 * e + 2 * i - h - k) % 7;
	m = (a + 11 * h + 22 * l ) / 451;
	p = (h + l - 7 * m + 114) % 31 + 1;
	n = (h + l - 7 * m + 114) / 31;
	printf("%\nW roku %d Wielkanoc przypada %d.%d .\n\n", y, p ,n);
}
```

Operatory arytmetyczne – Przykład2, rok 1828



KWIECIEŃ ma dni XXX. APREL.
APRILIS.

BYK.

Dni. ty.	SWIĘTA RZYMSKIE.	SWIĘTA RUSKIE.	Zn.
1 W.	Teodory Pan. i Hugona.	20 SS. Otcc. Izb. w Obr.	
2 S.	Franciszka a Paulo.	21 Jakowa Apost.	
3 C.	*Wielki.* Ryeharda.	22 *Wel. Czet.* Wasyła I.	
4 P.	*Wielki.* Izydora.	23 *Wel. Piat.* Nikona.	
5 S.	*Wielka.* Wincentego.	24 *Wel. Sub.* Zacharyi.	

Ewanielia u Marka S. w Roz. 16. O zmartwychwstan. Pańskiém. Przybyło dnia god. 5, min. 20.

6 N.	E. ZMARTW. P. Wilhel.	25 N. WOSKRES. XTO.	
7 P.	WIELKI. Epifaniusza B.	26 PON. SW. Hawryiła.	
8 W.	Dyonizego.	27 WOW. SW. Stefana.	
9 S.	Maryi Egipcyanki.	28 Haryona.	
10 C.	Ezechiela Proroka.	29 Marka.	
11 P.	Leona Papieża.	30 Joanna.	
12 S.	Juliusza Papieża.	31 Ippatya.	

Ewanielia u Jana S. w Roz. 20. O pokazaniu się P. Jezusa uczniom. Przybyło dnia god. 5, min. 50.

13 N.	E. *Przewod.* Hermenegil.	1 N. *Aprel.* Maryi Egip.	
14 P.	Tyberyusza.	2 Tyta.	
15 W.	Ludwiki i Kasyldy.	3 Nikity.	
16 S.	Lamberta.	4 Jozyfa.	
17 C.	Rudolfa.	5 Fteodula.	
18 P.	Apoloniusza.	6 Jewtychia.	
19 S.	Hermogenesa.	7 Hiborhia.	

Ewan. u Jana S. w Rozd. 10. O P. Chrystusie i o dobrym pasterzu. Przybyło dnia god. 6, min. 18.

20 N.	E. 2 po *W.* Gro. Chry. Sul.	8 N. Irodiona.	
21 P.	Anzelma.	9 Ewpsychia.	
22 W.	Sotera i Kaia MM.	10 Terentya.	
23 S.	Woyciecha Biskupa.	11 Antypy B. M.	
24 C.	Jerzego Męczen.	12 Wasyłya B.	
25 P.	Marka Ewangelisty.	13 Artemona.	
26 S.	Marcella i Kleta MM.	14 Martyna P. R.	

Ewanielia u Mat. S. w Roz. 16. O odéyściu P. Chrystusa. Przybyło dnia god. 6, min. 36.

27 N.	E. 3 p. *W.* Op. S. Józ. Ana.	15 N. Arystarcha.	
28 P.	Witalisa.	16 Achapia M.	
29 W.	Piotra Męcz.	17 Symeona M.	
30 S.	Katarzyny Senneńsk.	18 Joanna. M.	

LUNACYE.

Ostatnia kwadra dnia 7. o godzinie 1, min. 31, po południu. Pogoda, niekiedy chmury przechodzą przy ciepłym powietrzu.

Nów Maia dnia 14. o godzinie 10, min. 42, rano. Pochmurza się, na przemiany świecenie słońca, ciepło.

Piérwsza kwadra dnia 22, o godz. 6, min. 43, rano. Pogoda, cieło.

Pełnia dnia 30. o god. 0, min. 9, po północy. Pogoda, daley pochmurno.

Swięta Żydowskie.

1, 2, 3, 4 Chelhamoed wolne Swięta Wielkanocne.
5 Szabas i Szwyi szel Pesach.
6 Achron szel Pesach ostatnie Swięta Wielkanocne uroczyste.
7 Isruchag dzień radośny.
12 Szabas.
14, 15, Rozchodesz czyli 1 Ker.
19, 26 Szabas.

Operatory relacji oraz logiczne

- Operatorami relacji są: **<, >, >=, <=**, wszystkie mają ten sam priorytet. Po nich, ze względu na priorytet są == oraz !=
- Te operatory mają niższy priorytet niż operatory arytmetyczne, dlatego i < lim – 1 to inaczej: i < (lim – 1)

- **Warunki logiczne i relacji są również wartościami (całkowitymi)**
  - Warunek jest **spełniony**, jeśli ma wartość różną od zera
  - Warunek jest **niespełniony**, jeśli ma wartość zero
  - Np. 0>1 ma wartość 0, a warunek 0<=1 ma wartość różną od zera.

- **Przykład:**

```
#include <stdio.h>
int main() {
   printf("(0>1) == %d\n", (0>1));
   printf("(0<=1) == %d\n", (0<=1));
   return 0;
}
```

wypisze:

(0>1) == 0

(0<=1) == 1

Operatory relacji oraz logiczne

- **Przykład** dla operatorów relacji:

```
enum boolean { false, true } ;
enum boolean  mniejsze,  rowne,  nierowne, wiekszerowne;
int  i = 5;
float  x = 12.3;
mniejsze     = i < x;          // true
rowne        = i == x;         // false
nierowne     = i != x;         // true
wiekszerowne = i >= x;         // false
```

- Operatory logiczne to: **||** (lub, OR), **&&** (oraz, i, AND), **!** (nie, NOT).

- Wyrażenia połączone tymi operatorami || i && są obliczane **od strony lewej do prawej**, aż do momentu ustalenia wartości prawda lub fałsz. Jeśli wartość jest już pewna, wtedy program nie oblicza już dalszej części wyrażenia, skoro nie ma ono na nic wpływu.

Operatory relacji oraz logiczne

- **Przykład:**

```
for(i = 0; i < lim-1 && ((c = getchar()) != '\n' && c != EOF; ++i){
      //…
}
```

  - Pierwszym warunkiem w pętli jest ***i < lim-1*** i jest on sprawdzany jako pierwszy. Dopiero potem sprawdzane są kolejne warunki połączone operatorem ***&&.*** Jeśli pierwszy warunek nie jest spełniony to pozostałe nie będą sprawdzane.
  - Z definicji numeryczną wartością wyrażenia logicznego lub relacyjnego jest **1** - jeżeli jest ono prawdziwe lub **0** – jeśli jest fałszywe.
  - Jeśli warunek ***i < lim–1*** nie będzie spełniony, to jego wynikiem będzie wartość **0**, a jedna taka wartość wystarcza, aby całe to wyrażenie stało się fałszywe.

- Jednoargumentowy operator negacji **!** zmienia argument różny od zera na wartość 0, a argument równy zero na wartość 1. Np.:

```
if (!cos)  { /* … */ }
if (cos == 0) { /* … */ } // dokładnie to samo co wyżej
```

  - Stosowanie jednej czy drugiej formy zależy od preferencji programisty.

Operatory relacji oraz logiczne

- **Przykład** dla operatorów logicznych:

```
int a, b, c;
enum boolean z;

z = a < b && b < c;          // jeżeli a < b < c to z = 1

int rok = 2000;
enum boolean przestepny;
przestepny = !(rok % 4)  &&  rok % 100 || !(rok % 400);
int f = 0, k;

// jeżeli a < b to warunek dalej nie jest sprawdzany i wartość f
// pozostaje niezmieniona. Jeśli a > b to drugi warunek jest
// sprawdzany i wartość f jest zwiększana

( a > b ) && ( k <= f++ );          // optymalizacja

// jeżeli k > 5 to drugi warunek nie jest już sprawdzany i
// wartość b pozostaje niezmieniona. Jeżeli k < 5 to drugi
// warunek jest sprawdzany i wartość b zmieni się na 7

( k > 5 ) || (c < ( b = 7 ) );
```

Operatory zwiększania i zmniejszania

- W języku C dostępne są dwa operatory zmniejszania i zwiększania wartości zmiennych.
- Operator zwiększania **++** dodaje 1 do swojego argumentu, a operator zmniejszania **--** odejmuje 1 od swojego argumentu.
- Operatory **++** i **--** mogą być używane zarówno jako **przedrostkowe** (przed zmienną np. *++n*) jak i **przyrostkowe** (po zmiennej np. *n++*).
  - Wyrażenie *++n* zwiększa *n* przed użyciem jej wartości, a wyrażenie *n++* zwiększa *n* po użyciu jej poprzedniej wartości.
- **Przykład:**

```
++alfa     --beta         // przed obliczeniem wyrażenia
alfa++     beta--          // po obliczeniu wyrażenia

float  x = 2.5,  y;
x ++ ;          // równoważne x = x + 1;
++ x ;          // równoważne x = x + 1;
x -- ;          // równoważne x = x - 1;
-- x ;          // równoważne x = x - 1;
// błąd!, operatory zmniejszania i zwiększania można stosować
// tylko do zmiennych
y = ++(2 * x);
```

Operatory zwiększania i zmniejszania

- W kontekście w którym istotna jest także wartość (a nie tylko efekt) zwiększenia, wyrażenia *++n* oraz *n++* są różne. Np.:

```
int n = 5, x;
x = n++; // nadaje x wartość 5
n = 5;
x = ++n; // nadaje x wartość 6

int i = 3,  j = 4,  s;
s = j++ + i;    // s == 7   j == 5
j = 4;
s = ++j + i;    // s == 8   j == 5
```

- W kontekście, w którym nie jest istotna wartość, a tylko sam efekt zwiększania użycie operatora przedrostkowego i przyrostkowego jest równoważne, np.

```
if (c == '\n') n++;
```

- W niektórych sytuacjach specjalnie stosuje się jeden z tych operatorów.

Przekształcenia typów

- **Przykład -** dołącz tekst z tablicy *t* na koniec tekstu w tablicy *s*. Tablica *s* musi być dostatecznie duża, aby pomieścić dołączony tekst z tablicy *t*:

```
void strcat(char s[], char t[]) {
     int i, j;
     i = j = 0;
     while (s[i] != '\0') i++; // znajdź koniec ciągu znaków s
     // zastosowanie operatora przyrostkowego do obu zmiennych i oraz
     // j zapewnia, że wskażą one odpowiednie pozycje w następnym
     // przebiegu pętli
     while ((s[i++] = t[j++]) != '\0') ; //kopiuj t na koniec s
}
```

- Druga pętla **while** może być zapisana równoważnie:

```
while (t[j] != '\0') {
     s[i] = t[j];
     i++; j++;
}
```

Operatory bitowe

- Język C oferuje sześć operatorów pozwalających na manipulację bitami:
  - **&** - bitowa koniunkcja (**AND**)
  - **|** - bitowa alternatywa (**OR**)
  - **^** - bitowa różnica symetryczna **XOR** (eXclusive OR : 'jedno lub drugie, ale nie oba' – daje 1 jeśli jeden i tylko jeden z dwóch argumentów ma wartość 1, czyli dla dwóch jedynek: 0)
  - **<<** - przesunięcie bitów w lewo
  - **>>** - przesunięcie bitów w prawo
  - **~** - dopełnienie jedynkowe – zamienia 1 na 0 i na odwrót (operator 1 -argumentowy (unarny))

- Można je stosować tylko dla argumentów całkowitych: **char**, **short**, **int**, **long** zarówno ze znakiem liczby jak i bez znaku.

- Przykład:

```
int  i = 35,  przeciwna,  nieparzystosc;
przeciwna =  ~ i + 1;          // równoważne  - i; kod uzupełnień do dwóch
nieparzystosc = i & 1;         // jeśli liczba jest nieparzysta to wynik będzie 1
                               // i:                  0000000000100011
                               // 1:                  0000000000000001
                               // nieparzystosc:      0000000000000001
```

Operatory bitowe: &

- Bitowy operator koniunkcji **&** jest stosowany często do **„zasłaniania" pewnego zbioru bitów**, np.:

```
n = n & 0177;
```

wyzeruje wszystkie oprócz 7 najmłodszych bitów zmiennej n (ponieważ 0177 (ósemkowo) dotyczy właśnie 7 bitów młodszych – tych „od prawej")

- 0177 – zapis ósemkowy liczby 127 (dziesiętnej)
- 127 to bitowo 01111111 (czyli 7 najmłodszych bitów jest ustawionych na 1, tj. mających wartości od 1(2^0) do 64(2^7), bit nr 8, czyli najstarszy, koduje wartość 128 (2^8 = 128))
- & jest bitowym AND, to znaczy, że da wynik 0, jeśli przynajmniej jeden z argumentów to 0.

np.	n:	0 0 1 1 0 1 0 1 1 0 **1 1 1** 0 **1 1**	$(13755)_{10}$
	0177:	0 0 0 0 0 0 0 0 0 **1 1 1 1 1 1 1**	$(127)_{10}$
	nowe n:	0 0 0 0 0 0 0 0 0 0 **1 1 1** 0 **1 1**	$(59)_{10}$

Operatory bitowe: |

- Bitowego operatora alternatywy **|** używa się do **„ustawiania” bitów**, np.:

```
x = x | SET_ON;
```

gdzie SET_ON to jakaś liczba mająca pewne bity 'zapalone' (jedynki), a pewne 'zgaszone' (zera)

- Powyższa operacja bitowego OR (|) ustawi wszystkie bity w x na 1 (nie ważne jaka była poprzednia wartość), jeżeli są one 'zapalone' w SET_ON. Bity, które w SET_ON mają wartość 0, pozostaną w zmiennej x takie, jakie były.

- Np.

x:	0 1 1 1 0 1 1 0 0 0 1 0 1 1 0 0
SET_ON:	0 1 0 0 1 0 1 0 0 1 1 1 1 1 1 1
wynik:	**0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1**

Operatory bitowe: ^

- Operator bitowej różnicy symetrycznej **^**: ustawia 1 tylko tam, gdzie bity w obu argumentach mają różne wartości, a 0 tam, gdzie są takie same. Np.:

```
nowa_liczba = liczba ^ SET_XOR;
```

  - Np.: liczba: 0 1 0 1 0 1 0 1
    SET_XOR: 1 1 1 1 0 0 0 0
    nowa_liczba: **1 0 1 0 0 1 0 1**

- Inne przykłady:

```
char a = 'r', b = 8;
a = a & 0x5F;    // zamiana małych liter ASCII na duże
// a:             01110010 ('r', 114_10)
// 0x5F:          01011111 (95_10)
// wynik:         01010010 ('R', 82_10)
b = b | 0x30;     // zamiana wartości binarnej 0-9 na kod znaku ASCII
// b:             00001000  (8_10)
// 0x30:          00110000 (48_10)
// wynik:         00111000 ('8', 56_10)
```

Operatory bitowe: << >>

- Operatory **<<** i **>>** służą do **przesunięcia bitów** argumentu stojącego po lewej stronie o zadaną liczbę pozycji w lewo **<<** lub w prawo **>>** o liczbę pozycji określoną przez argument stojący po prawej stronie (**wartość dodatnia!**). Bity 'opuszczane' z prawej/lewej strony są zerowane.

- Np. ***x << 2*** przesunie bity w *x* o 2 pozycje w lewo, jest to równoważne **pomnożeniu *x* przez 4**.

- Przykład dla *x*:

x:	0 0 0 0 0 1 1 1	(7 = 4 + 2 + 1)
nowe_x:	0 0 0 1 1 1 0 0	(28 = 4 + 8 + 16)

- Analogicznie, **>>** jest przesunięciem w prawo i odpowiednikiem **całkowitego dzielenia bez reszty**.

- Przy **przesuwaniu w prawo** wielkości bez znaku (**unsigned**) 'opuszczone' bity zawsze wypełniane są zerami. W przypadku wielkości ze znakiem na niektórych maszynach wypełniane takie miejsca są zerami (przesunięcie „logiczne”) a na innych bitem znaku (przesunięcie „arytmetyczne”).

```
int i = 35, r, s;
r = i << 2;                    // równoważne r = i * 4;
s = i >> 3;                    // równoważne s = i / 8;
```

Złożone operatory przypisania

- Wyrażenie takie jak *a = a + 20* (gdzie ta sama zmienna występuje po lewej i natychmiast po prawej stronie przypisania) można zapisać skrótowo jako *a += 20*. Operator += nazywany jest **operatorem przypisania.**

- Dla większości operatorów dwuargumentowych istnieje w języku C odpowiedni operator przypisania ***op=***, gdzie *op* jest jednym z operatorów:

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

- **Uwaga**, wszystko po prawej stronie znaku równości jest traktowane jako całość, na której wykonywana jest operacja!, czyli:

  wyr1 op= wyr2 //wyr1 i wyr2 to wyrażenia

  jest równoważne

  wyr1 = (wyr1) op (wyr2)

  przy czym wyrażenie *wyr1* wyliczane jest tylko raz.

  Np.:

```
a *= b - c + 1;
```

to w rzeczywistości:

```
a = a * (b - c + 1);
```

a nie:

```
a = a * b - c + 1; (!!!)
```

Złożone operatory przypisania

- **Przypisanie ma wartość i może znaleźć się w wyrażeniach.** Najczęściej spotykanym przykładem jest:

```
int c;
// wczytuje z klawiatury dane znak po znaku; Gdy wejście jest
// puste funkcja getchar zwraca EOF (end of file)
// c musi być typu int, żeby pomieścić stałą symboliczną EOF
while ((c = getchar()) != EOF)
```

- **Przykład:**

```
double  kurs,  zwyzka;
kurs += zwyzka;   // kurs = kurs + zwyzka;

int x = 10;
x -= 5 – 2 ;  //  x = x – (5 - 2)

int i, j, k;
i = (j = 5) + 1; // równoważne j = 5; i = j + 1;
i = j = k = 0;   // równoważne i = 0; j = 0; k = 0;
```

Operator warunkowy

- Załóżmy, że chcemy do zmiennej c zapisać większą z liczb przechowywanych w zmiennych *a* i *b*. Można to zrobić następującą instrukcją:

```
if (a > b)
     c = a;
else
     c = b;
```

- Można to jednak zrobić jednym wyrażeniem warunkowym za pomocą trzyargumentowego operatora „?”:

```
c = ( a > b ) ? a : b;
```

- Wyrażenie warunkowe utworzone za pomocą operatora „?” ma ogólną postać:

  wyrazenie1 ? wyrazenie2 : wyrazenie3

  Działanie: Najpierw obliczane jest wyrażenie *wyrazenie1*. Jeśli *wyrazenie1* ma **wartość różną od zera („prawda”)**, to wtedy obliczane jest *wyrazenie2* i ta wartość będzie wartością całego wyrażenia warunkowego ('? :'). W przeciwnym przypadku (gdy *wyrazenie1* jest **równe 0 („fałsz”)**), wtedy obliczane jest *wyrazenie3* i to jego wartość będzie wartością wyniku.

Operator warunkowy

- Nawiasy okrągłe otaczające pierwsze wyrażenie w wyrażeniu warunkowym nie są konieczne, ponieważ priorytet operatora **?:** jest bardzo niski, tuż nad priorytetem operatora przypisania.

- Wyrażenie warunkowe często wpływa na zwięzłość programu.

- **Przykład:**

```
float x, y, max;
// wynik wyrażenia warunkowego zostanie przypisany zmiennej max
max = x > y ? x : y; // nawiasy okrągłe nie są wymagane
```

- **Operator rozmiaru *sizeof***
  - Zwraca ilość bajtów zajmowanych przez argument (dany typ, zmienną, tablicę, itd.)
  - **Np.:**

```
long liczba_1;
dlugosc_1 = sizeof liczba_1;      //  == 4
dlugosc_1 = sizeof (long);        //  == 4
```

Operator wyliczenia (przecinek)

- Wyrażenia oddzielone przecinkiem oblicza się od lewej do prawej (**łączność lewostronna**), przy czym typem i wartością wyniku jest typ i wartość prawego argumentu.

- Zastosowanie tego operatora w kodzie programu jest ograniczone. Generalnie rzecz biorąc lepiej jest rozdzielać instrukcje średnikami.

- Stosuje się go w pętli ***for***.

- **Przykłady:**

```
long a1, a2, a3, s;
// a3 == 84000  s == 84000
s = ( a1 = 52700, a2 = 31300, a3 = a1 + a2 );
s = a1 = 52700, a2 = 31300, a3 = a1 + a2; // s == 52700


float x, y ,z;
z = ( x = 5.3, y = 2.5, y++ );  // y == 3.5  z == 2.5
```

- **Operator przypisania**

  - Operator przypisania ma **łączność prawostronną** tzn. obliczanie przypisań odbywa się od prawej do lewej i zwraca przypisaną wartość, dzięki czemu może być użyty kaskadowo:

```
a1 = a2 = a3 = 123;
```

Priorytety i łączność operatorów

- W języku C z każdym operatorem związany jest jego **priorytet** (określa kolejność wykonywania działań, np. dla wyrażenia: *a+b*c* najpierw wykonuje się mnożenie a potem dodawanie) oraz **łączność** (od której strony wykonywane jest działanie w przypadku połączenia operatorów o tym samym priorytecie, np. odejmowanie ma łączność lewostronną, więc *3-3-3* da w wyniku : -3).

- **Przykład:**

```
char a, b, c, d, e;
// lewostronnie łączne czyli (((a + b) - c) - d) + e;
a + b - c - d + e;
// prawostronnie łączny czyli a ? b : (c ? d : e);
a ? b : c ? d : e;
```

Priorytety i łączność operatorów – Przykład 1

```
int main ( ) {
     int nPierwszy, nDrugi = 5, nTrzeci;
     nPierwszy = 25;
     nTrzeci = nDrugi + nPierwszy;
     printf("\nWynik obliczen wartosci wyrazenia \n"
          "Trzeci = Drugi + Pierwszy \n"
          "dla Drugi = 5 i Pierwszy = 25 \n"
          "-----------------------------\n"
          "          Trzeci = %d\n\n", nTrzeci);
     return 0;
}
```

Priorytety i łączność operatorów – Przykład 2

```
int main ( ) {
     double dbA, dbB;

     printf("\nObliczenie wartosci wyrazenia a*a+b+1\n"
          "---------------------------------------\n"
          "Podaj wartosc a : ");
     // scanf_s dostępna w Visual C++ różni się od scanf tym, że
     // sprawdza czy jest odpowiednia ilość miejsca w buforze na
     // zapis podanej wartości
     scanf("%lf", &dbA);

     printf("\nPodaj wartosc b : ");

     scanf("%lf", &dbB);

     printf("\nWynik : a*a+b+1  = %.2lf\n\n", dbA * dbA + dbB + 1);
     return 0;
}
```

Priorytety i łączność operatorów – Przykład 3

```
/*
  k =
          1 + x   dla x > 0
          37      dla x == 0
          -x - 1  dla x < 0
*/
int main ( ) {
     double dbK, dbX;
     char* Tekst = "\nPodaj wartosc x : ";
     printf("%s", Tekst);

     scanf("%lf", &dbX);
     dbK = dbX > 0 ? 1 + dbX : dbX == 0 ? 37 : -dbX - 1;
     printf("\n Wynik : k = %5.2lf\n", dbK);
     return 0;
}
```

Priorytety i łączność operatorów

Operatory	Łączność
Wywołanie funkcji: () *jednoargumentowe przyrostkowe*: [] ->	lewostronna
(typ) sizeof *jednoargumentowe przedrostkowe*: ! ~ ++ -- + - * &	prawostronna
* / %	lewostronna
+ -	lewostronna
<< >>	lewostronna
< <= > >=	lewostronna
== !=	lewostronna
&	lewostronna
^	lewostronna
|	lewostronna
&&	lewostronna
||	lewostronna
?:	prawostronna
= += -= *= /= %= ^= |= <<= >>=	prawostronna
,	lewostronna

Podsumowanie operatorów

- Opieranie się na kolejności i priorytecie różnych operatorów (bez dodatkowych nawiasów okrągłych) czyni program bardziej zwięzłym, jednak za cenę tego, że trudniej go zrozumieć i łatwiej o błędy.

- Dobrą zasadą na początek jest stosowanie nawiasów częściej niż rzadziej, nawet, jeśli teoretycznie nie są wymagane, bo np. kolejność operatorów jest (teoretycznie) wszystkim znana.

- W języku C **nie określa się kolejności obliczania wartości argumentów operatora** (wyjątek to operatory: **&& || ?: ,**), np.:

```
X = fun1() + fun2();
```

  - Funkcja *fun2()* może być wykonana przed funkcją *fun1()* lub odwrotnie.

- Podobnie **kolejność obliczania wartości argumentów funkcji nie jest określona**, np.

```
moja_funkcja( ++n, odwrotnosc( n ) );
```

  - Tak naprawdę nie wiadomo z góry (zależy to od kompilatora) czy najpierw *n* będzie powiększone, czy może jednak najpierw uruchomi się funkcja *odwrotnosc(n)*, a dopiero później *n* zostanie zwiększone.

Przekształcenia typów danych

- Garść prostej teorii

Przekształcenia typów

- Jeżeli argumentami operatora są zmienne różnych typów, to są one **przekształcane do jednego, wspólnego typu wg kilku reguł**:
  - Automatycznie wykonywane jest tylko przekształcenie, w którym typ 'mniejszy' jest zamieniany w 'większy' bez utraty informacji (**konwersja rozszerzająca,** w wyniku której następuje zwiększenie liczby bajtów), np. liczba całkowita w zmiennoprzecinkową. Np.:

```
int li32 = 21212345;
long long li64 = li32;      // konwersja rozszerzająca
```

  - Niedozwolone są wyrażenia, które z samej idei nie mają sensu, np. indeks tablicy jako zmienna zmiennopozycyjna: *tablica[3.14]*

  - Wyrażenia, w których może nastąpić utrata informacji (**konwersja zawężająca,** w wyniku której następuje zmniejszenie liczby bajtów) np. przypisanie wartości o dłuższym typie całkowitym zmiennej krótszego typu, mogą powodować wypisanie komunikatu ostrzegawczego, ale nie są zabronione. Np.:

```
int li32 = 21212345;
short li16 = li32;        // konwersja zawężająca
                          // strata danych li16 == -21319
```

Przekształcenia typów

- Obiekty typu **char** to niewielkie wartości liczbowe od 0 do 127 (0-7F w systemie szesnastkowym), więc można ich używać w wyrażeniach arytmetycznych.
- Daje to dużą elastyczność przy różnego rodzaju przekształceniach.
- Przykładem jest poniższy program, zamieniający ciąg cyfr na liczbę typu **int**:

```
int liczba(char s[]) {
	int i, n;
	n = 0;
	for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)    {
		// jeżeli s[i] zawiera cyfrę, to odjęcie od niej kodu cyfry '0',
		// pozwala uzyskać wartość tej cyfry
		n = 10 * n + (s[i] – '0');
	}
	return n;
}
// np. dla "123"
// i = 0, n = 10 * 0 + ('1' – '0') = 1
// i = 1, n = 10 * 1 + ('2' – '0') = 12
// i = 2, n = 12 * 1 + ('3' – '0') = 123
```

Przekształcenia typów

- Inny przykład - program zmieniający wszystkie wielkie (i tylko wielkie) litery na małe:

```
int zmniejsz(int c){
     if (c >= 'A' && c <= 'Z')
        return c + 'a' – 'A';
     else
        return c;
}
// np. dla 'D'
// kod ASCII 'D' == 68, 'a' – 'A' jest równy 32,
// więc 68+32=100, co daje kod ASCII 'd'
```

- Program działa, ponieważ w tablicy znaków ASCII odległość między literami a-z oraz A-Z jest taka sama oraz ponieważ oba alfabety są ciągłe (między literami A i Z występują jedynie litery).

Przekształcenia typów

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Przekształcenia typów

- W standardowym pliku nagłówkowym **<ctype.h>** zdefiniowane są funkcje, które realizują podobne sprawdzenia i przekształcenia niezależnie od zbioru znaków.
- Jedną z nich jest funkcja **tolower(c)** – zwraca wartość małej litery określonej przez *c,* jeżeli *c* jest dużą literą.
- Przykładowo wyrażenie *if(c > '0' && c < '9')* które sprawdza, czy *c* jest cyfrą, można zastąpić ogólniejszą funkcją **isdigit(c)**, już istniejącą w bibliotece *<ctype.h>*
- Język C nie określa czy zmienne typu **char** są wielkościami ze znakiem liczby czy bez, więc gdy zamieniamy typ **char** na **int** to czy może powstać liczba ujemna? Np.:

```
char c = 255;
int a = c;                    // a == ?
```

Wynik jest zależny od maszyny.

- W niektórych znak, którego skrajnie lewy bit jest równy 1 zostanie przekształcony na liczbę ujemną (tzw. „powielenie bitu znaku”),
- w innych awansuje do typu **int** przez dodanie zer z lewej strony i będzie zawsze dodatni.

Przekształcenia typów

- Z definicji **wyrażenia relacyjne oraz logiczne** połączone operatorami **||** i **&&** będą miały wartość **1** – jeżeli są prawdziwe i **0** – jeżeli są fałszywe.
  - Łączenie wyrażeń elementarnych (złożonych z liczb, znaków, zmiennych, stałych czy wywołań funkcji) przy pomocy operatorów (jedno-, dwu- lub trójargumentowych o ustalonych priorytetach) pozwala na budowę dowolnie skomplikowanych warunków (wyrażeń złożonych).
  - W wyrażeniach stosujemy nawiasy okrągłe dla ustalenia kolejności obliczeń.

- W związku z powyższym, operacja:

```
int d = c >= '0' && c <= '9'; // d = 1 jeśli c jest cyfrą
```

  - Przypisze do zmiennej *d* wartość *1* wtedy i tylko wtedy gdy *c* jest cyfrą (kod ASCII pomiędzy 48 (zero) a 57 (dziewięć)).

- W przypadku instrukcji takich jak **if**, **for** lub **while** 'prawda' w ich części warunkowej oznacza po prostu DOWOLNĄ wartość numeryczną inną niż ZERO („nie zero”).

- Innymi słowy: **while**(0) nigdy ani razu się nie wykona, natomiast **while**(1) jest pętlą nieskończoną.

Przekształcenia typów

- Ogólnie jeśli argumenty operatora dwuargumentowego (np. * czy -) są różnego typu to typ „mniejszy” jest promowany do „większego” przed wykonaniem operacji.

- Całkowicie upraszczając sprawę przekształceń niejawnych (wykonywanych przez kompilator na zmiennych różnych typów), **w wyrażeniu typ mniejszy jest chwilowo 'powiększany' do większego, a wynik jest (teoretycznie) zapisywany jako typ większy**.

- **Reguły dotyczące przekształceń arytmetycznych:**
  - Jeśli jeden z dwóch argumentów jest typu **long double**, to drugi zostanie przekształcony (powiększony) do **long double**
  - W przeciwnym przypadku, jeśli typem któregokolwiek argumentu jest **double**, to drugi argument zostanie przekształcony do typu **double**
  - W przeciwnym przypadku, jeśli typem któregokolwiek argumentu jest **float**, to drugi argument zostanie przekształcony do typu **float**
  - W przeciwnym przypadku, wszystkie obiekty typu **char** i **short** są przekształcane do typu **int**
  - Następnie, jeśli którykolwiek z argumentów ma kwalifikator **long**, to ten drugi zostanie przekształcony do **long**.

Przekształcenia typów

- Reguły są bardziej skomplikowane dla argumentów **unsigned**.
- Wynik porównania dwóch obiektów, z których jeden jest liczbą ze znakiem (**signed**), a drugi bez (**unsigned**) zależy od maszyny (ponieważ zależą od długości reprezentacji różnych typów całkowitych – dla wielkości o tym samym rozmiarze można otrzymać nieoczekiwane wyniki).

- Zmienne typu **unsigned** mogą mieć większe wartości niż **signed**, ponieważ w przeciwieństwie do **signed**, żaden bit nie 'marnuje' się na zakodowanie znaku liczby. Z drugiej strony, zmienne **unsigned** są z tego powodu wyłącznie dodatnie.

- **Reguły dotyczące przekształceń arytmetycznych dla sytuacji gdy choć jeden z argumentów jest typu unsigned:**
  - Jeśli jeden z dwóch argumentów jest typu **unsigned long int**, to drugi zostanie przekształcony (powiększony) do **unsigned long int**
  - W przeciwnym przypadku, jeśli typem któregokolwiek argumentu jest **long int**, a drugiego **unsigned int**, wynik zależy od tego, czy typ **long int** jest w stanie wyrazić wszystkie wartości typu **unsigned int**. Jeśli tak to argument typu **unsigned int** jest przekształcany do **long int**. Jeśli nie to oba są przekształcane do typu **unsigned long int**.
  - W przeciwnym przypadku, jeśli typem któregokolwiek argumentu jest **unsigned int**, to drugi jest przekształcany do **unsigned int**.

Przekształcenia typów

- Arytmetykę zmiennopozycyjną można wykonywać w pojedynczej precyzji (nie jest wymagana podwójna precyzja) – zmiana od C99.

- Krótsze typy całkowite bez znaku w kombinacjach z dłuższymi ze znakiem nie przenoszą właściwości braku znaku arytmetycznego na typ wyniku – zmiana od C99.

- **Przykład:**

```
int i; char c;
i = c;          // mniejsza wartość przypisywana do większej, OK, nie
                // będzie straty informacji
c = i;          // Uwaga – dla operatora podstawienia konwersja na typ lewej
                // strony może spowodować stratę informacji: np. i == 258, c == ?
```

- *c = i;* - jeśli *i* ma większą wartość niż 255, nastąpi utrata danych

- Jeżeli *x* jest typu **float** a zmienna *y* typu **int** to wtedy nastąpią odpowiednie przekształcenia typów. Przy czym *y = x* (czyli zamiana **float** na **int**) spowoduje oczywiście obcięcie części ułamkowej.

- Problem dotyczy też przekazywania wartości zmiennych do funkcji, jeśli przekazujemy różne typy (jeśli nie podano prototypu funkcji to typy **char** i **short** stają się **int**, a typ **float** staje się **double**).

Przekształcenia typów - rzutowanie

- W dowolnym wyrażeniu można jawnie wymusić przekształcenie typów za pomocą jednoargumentowego operatora zwanego rzutem (ang. *cast*).

- Jest to konwersja typu na życzenie programisty, nazywana też **rzutowaniem**.

- **Konstrukcja:**

  (**typ**) wyrażenie

  Powoduje przekształcenie *wyrażenia* do typu określonego przez *typ*.

- Rzutowanie działa tak, jakby wartość wyrażenia *wyrażenie* przypisano zmiennej wskazanego typu *typ*, a następnie użyto tej zmiennej zamiast całej konstrukcji.

- Np. funkcja biblioteczna **sqrt()** (biblioteka: *math.h*) obliczająca pierwiastek kwadratowy z liczby podanej jako argument, oczekuje (i wymaga), aby ta liczba była typu **double**. Jeśli więc *n* jest typu **int,** należy wykonać rzutowanie:

```
sqrt( (double)n );
```

aby przekształcić wartość *n* do **double** przed przekazaniem jej do funkcji **sqrt**.

Przekształcenia typów - rzutowanie

- **UWAGA**: rzutowanie tworzy **tymczasową wartość** o odpowiednim typie. Sama zmienna (np. *n* z przykładu) wciąż pozostaje niezmieniona tzn. ma typ taki, jak w deklaracji!

- **Przykład:**

```
int ile = 27;
float tyle = 1.4;

ile = ile + tyle;
/* konwersja 27 na  27.0, dodawanie zmiennopozycyjne,
   konwersja 28.4 na 28, przypisanie */

ile = ile + (int) tyle;
/* konwersja 1.4 na 1, dodawanie całkowitoliczbowe,
   przypisanie */
```

- Operator rzutowania ma taki sam priorytet jak inne operatory jednoargumentowe.

- Ogólnie rzecz ujmując, rzutowanie stanowi wskazówkę dla kompilatora, że programista wie co robi mieszając zmienne różnych typów. Czy w rzeczywistości się to sprawdza w danym kodzie, to już kwestia zdolności tegoż programisty.

Przekształcenia typów - rzutowanie

- Jeżeli typy argumentów są zadeklarowane w **prototypie funkcji** (informuje on kompilator o tym, jaki typ zwraca dana funkcja oraz jakiego typu parametry przyjmuje), np.

```
double pierwiastek(double);
```

wtedy jej wywołanie:

```
double wynik = pierwiastek(2);
```

spowoduje automatyczną zamianę liczby całkowitej 2 na zmiennopozycyjną wartość 2.0 (podwójnej precyzji) i taka wartość zostanie wysłana do funkcji *pierwiastek*, bez potrzeby wykonywania rzutowania przez programistę.

- Powyższe będzie działało nawet jeśli funkcja nie będzie miała prototypu, pod warunkiem, że funkcja *pierwiastek* będzie znajdować się w tym samym pliku co funkcja z której zostanie ona wywołana (tutaj funkcja *main*) oraz jej definicja znajdować się będzie nad funkcją wywołującą (tutaj nad funkcją *main*).

- Jeśli natomiast funkcja *pierwiastek* znajdzie się poniżej funkcji *main* oraz nie będzie prototypu funkcji, to kompilator wyświetli błędy i ostrzeżenia.

- **Jeśli nie ma prototypu funkcji to funkcja taka jest deklarowana w pierwszym miejscu w którym występuje** (czyli tak jakby jej argumentem była wartość typu **int**, a nie **double**).

Nazywanie typów danych

- W języku C jest mechanizm zwany **typedef** do tworzenia nowych nazw typów danych. Nie tworzy nowego typu, tylko nadaje nazwę dla już istniejącego.

- **Ogólna postać:**

```
typedef     typ     nowy_identyfikator
```

- **Przykłady:**

```
// tworzy dla typu char* synonim string. Z typu string można
// korzystać w deklaracjach, rzutowaniach etc. tak samo jak z typu char*
typedef   char*  string;
string  S1,  S2,  S3 = "napis";

typedef   int   num;
num  k;
int  l = 5;
k = l; // typ num jest równoważny z typem int

typedef   long    BIG;
unsigned   BIG  ww;    // błąd: long w type to tak naprawdę signed long
```

Wprowadzanie i wyprowadzanie danych w programie

- Funkcje **putchar**, **getchar**, **printf**, **scanf**

Wyprowadzanie danych: *putchar*

- Funkcja z biblioteki **`stdio.h`** , deklaracja:

```
int  putchar ( int c ) ;
```

- Wysyła znak *c* na standardowe wyjście (***stdout***), którym domyślnie jest ekran.

- Zwraca wartość wypisanego znaku jeśli wszystko dobrze poszło lub zwraca kod EOF jeżeli wystąpił jakiś błąd.

- **Przykład:**

```
char cc = 'R';
putchar ( cc );  // 'R' na ekran
```

- Wyniki można także kierować do pliku używając symbolu **>** Jeśli program *prog* wykorzystuje funkcję *putchar* to polecenie:

```
prog > outfile
```

spowoduje, że *prog* będzie pisał do pliku *outfile*, a nie na standardowe wyjście.

- Wywołanie *putchar(c)* jest równoważne wywołaniu *putc(c, stdout).*

Wyprowadzanie danych: *puts*

- Funkcja z biblioteki **`stdio.h`** , deklaracja:

```
int  puts ( char  *napis );
```

- Wypisuje tekst zawarty w tablicy *napis* oraz znak nowego wiersza do standardowego strumienia wyjściowego (***stdout***), którym domyślnie jest ekran.

- Zwraca liczbę nieujemną lub EOF w przypadku błędu.

- **Przykład:**

```
#include <stdio.h>
int main( )
{
  char *nn = "Napis ćwiczebny.";
  puts(nn);
       return 0;
      }
```

Wyprowadzanie danych: *printf*

- Funkcja z biblioteki **`stdio.h`**, deklaracja:

```
int  printf ( const char *format,  wyrażenie, wyrażenie, ... );
```

- Funkcja przekształca i formatuje swoje argumenty zgodnie z poleceniami wewnątrz bloku 'format', a następnie wypisuje je na standardowym wyjściu (***stdout***). Zwraca liczbę wypisanych znaków.
- Wewnątrz *format* znajdują się znaki przesyłane bezpośrednio do **`stdout`** oraz **wzorce konwersji** (specyfikacje przekształceń, wskazujące sposób przekształcenia i wypisania kolejnego argumentu).

- Każdy wzorzec konwersji rozpoczyna się znakiem **%** a kończy pewnym charakterystycznym znakiem dla danego przekształcenia (liczby, ciągi znaków, etc.).

- **Przykład:**

```
int liczba_kolorow = 256;
printf( "%d", liczba_kolorow );
```

Wyprowadzanie danych: *printf*

```
double objetosc = 15.72;
printf( "%lf", objetosc );

char *Tekst = "Dokumentacja.";
printf ( "%s", Tekst);
```

- **Wzorzec konwersji :**

  % [opis] [szerokość] [.precyzja] [prefiks] znak_konwersji

- Pomiędzy % a znakiem końcowym (konwersji) mogą wystąpić:
  - Opis:
    - **-** (minus) - nakazujący dosunięcie przekształconego argumentu do lewego krańca jego pola (uzupełnianie znakami spacji z prawej strony)
    - **+** (plus) – wyprowadzenie znaku liczby np.+35
    - spacja – znak spacji zamiast znaku plus np. 35
  - Szerokość:
    - Liczba określająca minimalny rozmiar pola. Przekształcony argument będzie wpisany do pola o co najmniej takim rozmiarze. Jeśli trzeba pole zostanie uzupełnione do pełnego rozmiaru z lewej strony (lub z prawej jeśli tego zażądano, np. gdy wystąpi minus).

Wyprowadzanie danych: *printf*

- Szerokość pola można zastąpić znakiem *, co oznacza, że żądaną liczbę należy obliczyć przekształcając kolejny argument funkcji (musi być typu **int**), np.:

```
printf ("Width trick: %*d \n", 5, 10);
// Width trick:     10
```

  - Precyzja:
    - kropka rozdzielająca rozmiar pola od precyzji.
    - liczba określająca precyzję - np. liczbę cyfr części ułamkowej, maksymalną liczbę znaków dla tekstu lub minimalną liczbę cyfr dla wartości całkowitej (uzupełni zerami do wymaganej liczby cyfr).
  - Prefiks:
    - litera **h** jeśli argument całkowity należy wypisać jako **short** (*s* jest zajęte dla łańcuchów znaków), litera **l** – jeśli jako **long**.

- **Przykład:**

```
printf ("Wypisz liczbe w roznych systemach: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
// Wypisz liczbe w roznych systemach: 100 64 144 0x64 0144

printf ("Dodaj spacje na poczatek: %5d \n", 1977);
// Dodaj spacje na poczatek: 1977
```

Wyprowadzanie danych: *printf*

Tablica 7.1. Podstawowe przekształcenia funkcji **printf**

Znak	Typ argumentu	Dana wyjściowa
d, i	int	liczba dziesiętna
o	int	liczba ósemkowa bez znaku (bez wiodącego zera)
x, X	int	liczba szesnastkowa bez znaku (bez wiodących 0x lub 0X) z literami abcdef lub ABCDEF dla 10, ..., 15
u	int	liczba dziesiętna bez znaku
c	int	jeden znak
s	char *	ciąg znaków wypisywany do napotkania '\0' lub wyczerpania liczby znaków określonej przez precyzję
f	double	[−]*m.dddddd*, gdzie liczbę cyfr *d* określa precyzja (domyślnie 6)
e, E	double	[−]*m.ddddde±xx* lub [−]*m.ddddd*E±*xx*, gdzie liczbę cyfr *d* określa precyzja (domyślnie 6)
g, G	double	wypisana w formacie %e lub %E, jeśli wykładnik jest mniejszy niż −4 albo większy lub równy precyzji; w przeciwnym przypadku wypisana w formacie %f; nie wypisuje się nieznaczących zer i kończącej kropki dziesiętnej
p	void *	wskaźnik (postać zależna od implementacji)
%		nie ma przekształcenia argumentu; wypisany znak %

Wyprowadzanie danych: *printf*

- **Przykłady:**

```
int Alfa = 5;
float Beta = 12.45;
printf ( "Wynik: \n Alfa = %d,\t Beta = %f\n", Alfa, Beta + 500);
// Wynik: Alfa = 5,    Beta = 512.450000

char opcja = 'X';
char *Napis = "Opis programu.";
printf("Wybrano opcję %c : %31s", opcja, Napis );
// Wybrano opcję X :           Opis programu.

int koty = 2, *wsk_k = &koty;
float test = 23.345678;
double wynik = -0.01234567;
printf("Liczba kotów : %d", *wsk_k );
// Liczba kotów : 2
printf("\nWynik testu = %12.3f\n Razem = %.5lf\n", test + 5, wynik );
// Wynik testu = 28.345
// Razem = -0.01234
```

Wyprowadzanie danych: *printf*

```
void main ( ) {
     int alfa = 105000;
     printf("%d", alfa);
     //%hd – short -> przekroczenie zakresu, ujemna
     printf("\n\n");

     float wynik = 187.457f;
     printf("\nWynik obliczen : %f\n\n", wynik);  //  zaokr.

     long long a = 31112223331l;
     int  i = 3111222333;
     printf("\nDane typu long long: a = %lld\nDane typu int  : i = %d\n", a, i);

     // wartość poza zakresem, ujemna
     double moc = 125.4567890123456;
     printf("%lf", moc);      // .2 , 12.2 – dodatkowe spacje
     printf("\n............\n\n");
```

Wyprowadzanie danych: *printf*

```
// wzorzec konwersji zapisany w tablicy znakowej
     long a = 5, b = 7;
     int  i = 1, j = 3;
     char Wzorzec[ ] = "\nDane typu long : a = %ld\tb = %ld"
                              "\nDane typu int  : i = %d\tj = %d\n";
printf(Wzorzec, a, b, i, j);

// przykład formatowania tekstu za pomocą funkcji printf
printf("\nTabela liczb\n\n"
       " ------------------------\n"
       "| 1.  Jeden    %10.4lf |\n"
       "| 2.  Dwa      %10.4lf |\n"
       "| 3.  Trzy     %10.4lf |\n"
       " ------------------------\n\n",
                0.0234, 1272.23, 15432.2349321);
}
```

```
Tabela liczb

 ------------------------
| 1.  Jeden       0.0234 |
| 2.  Dwa      1272.2300 |
| 3.  Trzy    15432.2349 |
 ------------------------
```

Wyprowadzanie danych: *printf*

- Precyzję można również zastąpić znakiem *, co oznacza, że żądaną liczbę należy obliczyć przekształcając kolejny argument funkcji (musi być typu **int**), np.:

```
printf ("%.*s", max, s);
```

wypisuje maksymalnie *max* znaków z ciągu znaków *'s'*.

- Poniższe zestawienie prezentuje działanie różnych specyfikacji (precyzji używa się do kontroli maksymalnej długości tekstu) podczas wypisywania tekstu "ahoj przygodo" (14 znaków):

```
:%s:          :ahoj, przygodo:
:%10s:        :ahoj, przygodo:
:%.10s:       :ahoj, przy:
:%-10s:       :ahoj, przygodo:
:%.20s:       :ahoj, przygodo:
:%-20s:       :ahoj, przygodo      :
:%20.10s:     :          ahoj, przy:
:%-20.10s:    :ahoj, przy          :
```

```
//min rozmiar pola == 10
//kontroluje max rozmiar pola
//uzupełnienie spacjami z prawej

//strony (dosunięcie do lewej)


//rozmiar pola + dodatkowe

//spacje
```

Wyprowadzanie danych: jak działa *printf*

- Deklaracja funkcji *printf* wewnątrz biblioteki **`stdio.h`** ma postać:

```
int printf(char *fmt, …);
```

- **…** (trzy kropki) oznaczają zmienną liczbę argumentów (liczba i typy pozostałych argumentów nie są znane), mogą być w nagłówku tylko na końcu listy zdefiniowanych argumentów.

- Biblioteka **`stdarg.h`** zawiera makra umożliwiające poruszanie się po liście argumentów o nieokreślonej (w momencie definiowania funkcji) długości.

- Makro **va_start** inicjuje pewną zmienną (w przykładzie *ap*), aby wskazywała na pierwszy nienazwany argument z listy …

-

- Należy utworzyć zmienną typu **va_list** aby odwoływać się do nie nazwanych argumentów.

```
// ap – argument pointer czyli wskaźnik do argumentów
va_list ap;  // wskazuje po kolei każdy nienazwany argument
```

- Makro **va_start** jako pierwszy argument przyjmuje zmienną typu *va_list*, a drugim jest ostatni nazwany parametr, w naszym przypadku jest to *fmt*.

```
va_start(ap, fmt)  // wskazuje 1-szy nienazwany argument
```

Wyprowadzanie danych: jak działa *printf*

- Każde wywołanie makra **va_arg** udostępnia jeden argument i przesuwa *ap* na następny *va_arg(ap, <typ>)*. Nazwa typu potrzebna jest do określenia typu szukanej wartości oraz rozmiaru kroku (o ile przesunąć zmienną *ap*). Np.:

```
va_arg(ap, int);
```

- Po zakończeniu działań należy użyć makra **va_end**, które czyści wszystko to co związane było z poprzednimi wywołaniami (musi być wywołane przed zakończeniem działania funkcji).

- **Przykład uproszczonej funkcji printf:**

```
#include <stdarg.h> /* minimalna printf ze zmienną liczbą argumentów */
void minprintf(char *fmt, …){
va_list ap; /* wskazuje po kolei każdy nienazwany argument */
char *p, *sval;
int ival;
float dval;
/* ap wskazuje 1, nienazwany argument; fmt – ostatni nazwany argument */
va_start(ap, fmt);
for (p = fmt; *p; p++) { // szuka w fmt początku wzorca konwersji
     if (*p != '%') {      // znaki nienależące do wzorca wypisuje na ekranie
             putchar(*p);
             continue;
     }
```

Wyprowadzanie danych: jak działa *printf*

```
        switch (*++p) {
            case 'd':       // nienazwany argument jest typu int
                ival = va_arg(ap, int); // pobiera argument
                printf("%d", ival);
                break;
            case 'f':       // nienazwany argument jest typu float
                dval = va_arg(ap, float);      // pobiera argument
                printf("%f", dval);
                break;
            case 's':                 // wypisuje tekst znak po znaku

                for (sval = va_arg(ap, char *) ; *sval; sval++)
                    putchar(*sval);
                break;
            default:
                putchar(*p);
                break;
        }
    }
    va_end(ap); /* po pracy wyczyść co trzeba */
}
```

//the C library macro **void va_end(va_list ap)** allows a function with variable arguments which used the //**va_start** macro to return. If **va_end** is not called before returning from the function, the result is undefined

Wprowadzanie danych - *getchar*

- Funkcja z biblioteki **`stdio.h`** , deklaracja:

```
int getchar(void);
```

- Standardowa funkcja, czyta po jednym znaku ze standardowego wejścia (zwykle klawiatury), podając za każdym razem znak z wejścia lub stałą symboliczną EOF jako wartość **int**.

- Wczytywanie kończy się w momencie natrafienia na kod znaku specjalnego **EOF** (End Of File). Zazwyczaj ma on wartość liczbową
-1, ale wszelkie porównania lepiej robić ze stałą znakową EOF, aby uniezależnić się od takiej specyficznej wartości, czyli:

```
if (getchar() != EOF)  { /* … */ }
```

lepiej niż:

```
if (getchar() != -1) {  /* … */ }
```

- W wielu środowiskach można zastąpić klawiaturę plikiem korzystając z symbolu **<**, np.:

```
prog < infile
```

spowoduje, że *prog* będzie czytał znaki z pliku *infile,* a nie z klawiatury.

Wprowadzanie danych – *getch, getche*

- Funkcja z biblioteki **`conio.h`** , deklaracja:

```
int  getch ( );
```

- Niestandardowa funkcja (czyni program mniej przenośnym), która oprócz oczekiwania na znak, pozwala również na odczytanie kodu wciśniętego klawisza na klawiaturze.
- Czyta z bufora wtedy gdy coś w nim jest, gdy bufor jest pusty wywołuje funkcję *getchar.* Bez echa.
- Zwraca kod ASCII lub 0.

- **Przykład**:

```
char nowy;
nowy = getch();
```

- **getche**
  - **Jak `getch` + echo** (zwraca znak (kod znaku) pobrany z bufora klawiatury)

Wprowadzanie danych - *scanf*

- Funkcja z biblioteki **`stdio.h`**, deklaracja:

```
int scanf ( const char *format,  wskaźnik, wskaźnik, ... );
```

- Składnia jak w przypadku **printf**, z tym, że funkcja wczytuje znaki ze standardowego wejścia (np. klawiatury). Interpretuje je zgodnie ze specyfikacjami zawartymi w argumencie *format* i zapamiętuje wyniki w miejscach określonych przez pozostałe argumenty. Dlatego też każdy z argumentów musi być wskaźnikiem (adresem zmiennej będącej argumentem).

- Zatrzyma się gdy zinterpretuje wszystkie znaki formatu lub gdy dana nie pasuje do wzorca konwersji (specyfikacji przekształcenia).

- Zwraca liczbę pomyślnie wczytanych i przypisanych danych wejściowych.

- Kolejne wywołanie **scanf** wznawia szukanie bezpośrednio za ostatnim wczytanym znakiem.

- Wczytuje kolejne pola (ciągi znaków do spacji lub znaku nowej linii) ze ***stdin***.
- **Format to tekst będący ciągiem wzorców konwersji.**

Wprowadzanie danych - *scanf*

- **Wzorzec konwersji:**

  % [*] [szerokość] [prefiks] znak_konwersji

Tablica 7.2. Podstawowe przekształcenia funkcji **scanf**

Znak	Dana wejściowa	Typ argumentu
d	liczba całkowita dziesiętna	int *
i	liczba całkowita; może wystąpić w postaci ósemkowej (z wiodącym 0) lub szesnastkowej (z wiodącymi 0x lub 0X)	int *
o	liczba całkowita w postaci ósemkowej (razem z wiodącym 0 lub bez)	int *
u	liczba całkowita dziesiętna bez znaku	unsigned int *
x	liczba całkowita w postaci szesnastkowej (z wiodącymi 0x lub 0X, albo bez)	int *
c	znaki; następne znaki z wejścia (domyślnie 1) umieszcza się we wskazanej tablicy; nie obowiązuje zwykła zasada pomijania białych plam; aby przeczytać najbliższy czarny znak, należy użyć %1s	char *
s	tekst (ale nie napis, tj. ciąg znaków występujący bez znaków cudzysłowu); argument powinien wskazywać na tablicę znakową o rozmiarze wystarczającym do przyjęcia tekstu wraz z dodanym na końcu znakiem '\0'	char *
e, f, g	liczba zmiennopozycyjna z opcjonalnym znakiem, opcjonalną kropką dziesiętną i opcjonalnym wykładnikiem	float *
%	literalnie znak %; nie będzie żadnego przypisania	

Wprowadzanie danych - *scanf*

- Należy pamiętać o stosowaniu **właściwych znaków konwersji**, w przeciwnym wypadku uzyskamy dziwne/nieoczekiwane wyniki:

Typ argumentu	Znak konwersji
char	%c
short	%hd
int	%d, %i
long	%ld
long long	%lld
float	%f, %e
double	%lf, %le
long double	%Lf, %Le
char*	%s

Wprowadzanie danych - *scanf*

- **Przykłady:**

```
int liczba_sztuk;
scanf ( "%d", &liczba_sztuk );

double szerokosc;
scanf( "%lf", &szerokosc );
// powyższe można zapisać za pomocą jednej instrukcji
scanf( "%lf%d", &szerokosc, &liczba_sztuk );

int lampy, krzesla, *wsk = &krzesla;
float temp;
double cena;
char opcja;

scanf( "%d%d%f%lf", &lampy, wsk, &temp, &cena );
//Wpisujemy: 1 5 SP 3 4 7 Enter - 2 5 . 4 Enter
// 3 . 9 9 Enter
// Wynik: lampy == 15  krzesla == 347  temp == -25.4   cena == 3.99
```

Wprowadzanie danych - *scanf*

```
// wczytywanie pojedynczych znaków
fflush(stdin);  // oczyszczenie bufora klawiatury
scanf( "%c", &opcja );
// A Enter
// opcja == 'A'

// wczytywanie tekstów
char Tekst[16]; // tablica 16 elementowa
scanf ( "%15s", Tekst ); // wczyta tekst do spacji lub do max. 15 znaków
// A l f a  Enter
// Tekst == "Alfa"

/* W nawiasach kwadratowych podano przedział znaków ASCII (od znaku spacji do znaku
tyldy), który będzie akceptowany i wprowadzany. Pierwszy znak spoza tego przedziału
(czyli np. znak nowej linii) kończy wprowadzany tekst. */

scanf ( "%15[ -~]", Tekst ); // wczyta tekst rozdzielony spacjami
// A l a  SP  m a  SP  k o t a . Enter
// Tekst == "Ala ma kota."
```

Wprowadzanie danych - *scanf*

- **Przykłady:**

```
int main ( ) {
        // traktuje wczytywaną liczbę jako zmiennopozycyjną i zapisuje ją w
        //odpowiednim formacie, nie dokonuje konwersji do int
        int alfa1 =  10;
        scanf("%f", &alfa1); // błędnie

        char znak1, znak2;
        int alfa;
        scanf("%d", &alfa); // poprawnie
        // Funkcja fflush wymusza zapis wszystkich buforowanych danych dla
        // danego strumienia wyjściowego stream poprzez podległą strumieniowi
        // funkcję zapisu. Stan strumienia nie jest zmieniany, jest on nadal
        // otwarty.
        // aby usunąć spację/znak nowej linii z bufora
        // fflush(stdin); // rozwiązanie niewłaściwe, zależne od
        // implementacji - fflush() nie zawsze czyści bufor
        // znak2 = getch(); // wczyta znak po wciśnięciu klawisza enter
        scanf("%c", &znak1); //wczyta spację/znak nowej linii
        // lepiej: scanf("%1s", &znak1);
```

Wprowadzanie danych - *scanf*

```
        int alfa2 = 3, *wsk_alfa2 = & alfa2;
        scanf("%d", wsk_alfa2);                 // ok

        // %f w scanf() nakazuje przechowywać daną w pamięci o adresie podanym
        // przez &miara jako zmienną pojedynczej precyzji. scanf() NIE zna

        // typu tej zmiennej, tylko jej ADRES
        double miara = 5.5;
        scanf("%f", &miara);                    //       %lf


        int a = 5;
        char NN[5];
        //        scanf("%s", NN); // nie sprawdza dostępnej pamięci
        scanf("%4s", NN);
        return 0;
}
```

Pytania?