Operatory, wyrażenia i instrukcje

Operator przypisania =

W języku C znak równości nie oznacza „równa się", jest natomiast operatorem przypisującym wartości.
Instrukcja: `bmw = 2002;`
przypisuje wartość 2002 zmiennej o nazwie `bmw`.
Pozycja po lewej stronie znaku **=** jest nazwą zmiennej, a pozycja po prawej - wartością przypisywaną tej zmiennej. Symbol **=** nazywamy **operatorem przypisania**.

Obiekty danych, l-wartości, r-wartości i operandy

Obiekt danych jest ogólnym pojęciem oznaczającym obszar pamięci, który może zostać użyty do przechowania wartości; np. fragment pamięci, w którym przechowywana jest zmienna lub tablica.
Nazwa lub wyrażenie określające obiekt danych nosi nazwę **lvalue** (l-wartość); l-wartością jest na przykład nazwa zmiennej.
Termin obiekt oznacza zatem rzeczywiste miejsce w pamięci, a l-wartość - identyfikującą to miejsce etykietę.
Ponieważ nie wszystkie obiekty danych mogą zmieniać wartość, do określania takich obiektów używa się terminu ***modifiable lvalue*** (modyfikowalna l-wartość). Właśnie dlatego mówimy, że po lewej stronie operatora przypisania powinna znajdować się modyfikowalna l-wartość. Litera l pochodzi od słowa left (lewy).

Termin **rvalue** (r-wartość) odnosi się do wartości, które mogą zostać przypisane modyfikowalnym l-wartościom; r-wartości mogą być stałymi, zmiennymi lub dowolnymi innymi wyrażeniami posiadającymi jakaś wartość.

Prawidłowym terminem określającym to, co do tej pory nazywaliśmy „pozycją" (np. „pozycja po lewej stronie znaku ="), jest **operand**.
Operandy są tym, czym posługują się operatory. Na przykład, zjedzenie hamburgera można opisać jako zastosowanie operatora ‚jeść" do operandu „hamburger".

```
/* golf.c — tablica wyników turnieju golfa */
#include <stdio.h>
int main(void)
{
int jane, tarzan, gepard;
gepard = tarzan = jane = 68;      // instrukcja przypisania
printf("                          gepard  tarzan   jane\n");
printf("Wynik pierwszej rundy %4d %8d %8d\n",  gepard,  tarzan,  jane);
return 0;
}
```

Dane wyjściowe programu:

	gepard	tarzan	jane
Wynik pierwszej rundy	68	68	68

Operator dodawania: +

Operator dodawania powoduje dodanie do siebie wartości znajdujących się po jego obydwu stronach.
Na przykład instrukcja

```
printf("%d",   4 + 20);
```

powoduje wyświetlenie liczby 24, a nie wyrażenia 4 + 20
Dodawane wartości (operandy) mogą być zarówno stałymi, jak i zmiennymi. Stąd instrukcja

```
dochód = pensja + łapówki;
```

sprawia, że komputer sprawdza wartości zmiennych pensja i łapówki, dodaje je, a otrzymany wynik przypisuje zmiennej dochód.

Operator odejmowania: -

Operator odejmowania powoduje odjęcie wartości następującej po znaku - od wartości znajdującej się przed tym znakiem. Instrukcja

```
dochód = 224.00 - 24.00;
```

przypisuje zmiennej dochód wartość 200.0.

Operatory **+** i **-** nazywane są operatorami dwuargumentowymi lub dwuelementowymi (*ang. binary operators*), co oznacza, że wymagają one dwóch operandów.

Operatory znaku: - i +

Symbol - (minus) może również służyć do zmiany algebraicznego znaku wartości. Na przykład ciąg instrukcji

```
bolek = -12;
lolek = -bolek;
```

nadaje zmiennej lolek wartość 12.

Gdy znak minus jest stosowany w ten sposób, występuje on w roli operatora jednoargumentowego (ang. *unary operator*).

Operator jednoargumentowy + nie zmienia wartości ani znaku operandu; pozwala on po prostu używać instrukcji, takich jak:

```
tuzin = +12;
```

bez narażania się na błąd kompilacji.

Operator mnożenia: *

Mnożenie w języku C wyrażamy symbolem *. Instrukcja

```
cm = 2.54 * cale;
```

mnoży zmienną cale przez 2.54 i przypisuje wynik zmiennej cm.

Potężny władca chciał wynagrodzić pewnego mędrca, który oddał mu ogromną przysługę. Gdy przyszedł czas wyboru nagrody, mędrzec wskazał na szachownicę i poprosił o jedno ziarno pszenicy na pierwszym polu, dwa ziarna na drugim polu, cztery na trzecim, osiem na czwartym, i tak dalej. Władca, będąc najwyraźniej niezbyt biegłym w matematyce, był zaskoczony skromnością tej prośby - był bowiem przygotowany na oddanie wielkich kosztowności …

```
/* pszenica.c — wzrost wykładniczy */
#include <stdio.h>
#define POLA 64            // liczba pol na szachownicy
#define PLON 9E14          // roczna produkcja pszenicy w USA (w ziarnach)
int main(void)
  {
 double bieżące, suma;
 int licznik = 1;
 printf("pole     dodane ziarna       suma ziaren        czesc rocznej     \n ");
 printf("                                                  produkcji w USA\n ");
 suma = bieżące = 1.0;      // zaczynamy od jednego ziarna
 printf("%3d %15.2e %14.2e %14.2e\n", licznik,  bieżące,  suma,  suma / PLON);
 while (licznik < POLA)
   {
   licznik = licznik + 1;
   bieżące = 2.0 * bieżące;    // podwójna liczba ziaren na następnym polu
   suma = suma + bieżące;    // aktualizacja sumy
   printf("%3d %15.2e %14.2e %14.2e\n",  licznik,  bieżące,  suma,  suma / PLON);
   }
return 0;
}
```

Dane wyjściowe:

pole	dodane ziarna	suma ziaren	część rocznej produkcji w USA
1	1.00e+000	1.00e+000	1.11e-015
2	2.00e+000	3.00e+000	3.33e-015
3	4.00e+000	7.00e+000	7.78e-015
4	8.00e+000	1.50e+001	1.67e-014
5	1.60e+001	3.10e+001	3.44e-014
6	3.20e+001	6.30e+001	7.00e-014
7	6.40e+001	1.27e+002	1.41e-013
8	1.28e+002	2.55e+002	2.83e-013
9	2.56e+002	5.11e+002	5.68e-013
10	5.12e+002	1.02e+003	1.14e-012
. . .			
50	5.63e+014	1.13e+015	1.25e+000
. . .			
64	?	?	?

Operator dzielenia: /

Dzielenie w języku C wyrażamy za pomocą symbolu **/** .

cztery = 12.0/3.0;

Dzielenie przebiega inaczej w przypadku typów całkowitych niż w przypadku typów zmienno-przecinkowych.
Dzielenie wartości zmiennoprzecinkowych daje bowiem wynik zmiennoprzecinkowy, a dzielenie wartości całkowitych - wynik całkowity.
Np. dzielenie 5 przez 3, daje liczbę całkowitą a wynik tego działania nie jest taką liczbą.
W języku C każda część ułamkowa będąca wynikiem dzielenia wartości całkowitych jest odrzucana. Proces ten nosi nazwą obcinania (ang. *truncation*).

```
/* dziel.c — rożne rodzaje dzielenia */
 #include <stdio.h>
int main(void)
{
printf("dzielenie całkowite:  5/4       to %d\n”, 5/4);
printf("dzielenie całkowite:  6/3       to %d\n”, 6.3);
printf("dzielenie całkowite:  7/4       to %d\n”, 7/4);
printf("dzielenie zmiennoprz.:  7./4. to %1.2f\n”, 7./4.);
printf("dzielenie mieszane:  7./4       to %1.2f\n”, 7./4);
return 0;
}
```

Wynik:

```
dzielenie całkowite:  5/4       to 1
dzielenie całkowite:  6/3       to 2
dzielenie całkowite:  7/4       to 1
dzielenie zmiennoprz.:  7./4.   to 1.75
dzielenie mieszane:  7./4       to 1.75
```

Przy dzieleniu całkowitym wynik pozbawiany całej części ułamkowej.

Operator sizeof

Operator **sizeof** zwraca rozmiar operandu w bajtach.

```
/* sizeof.c — wykorzystuje operator sizeof */
 #include <stdio.h>
int main(void)
{
int n = 0;
printf("n = %d, n ma dlugosc %d bajtów”,  n ,  sizeof n );
printf("wszystkie zmienne int maja dlugosc %d bajtów.\n ",   sizeof (int) );
return 0;
}
```

Operator modulo: %

Operator modulo jest wykorzystywany wyłącznie w arytmetyce całkowitej.
Zwraca on resztę z dzielenia liczb całkowitych.

```
/* min_sek.c — przelicza sekundy na minuty i sekundy */
 #include <stdio.h>
#define   SEK_W_MIN   60         //  liczba sekund w jednej minucie
 int main(void)
{
int sek, min, reszta;

printf("Przelicz sekundy na minuty i sekundy!\n"};
printf("Podaj liczbę sekund, która chcesz przeliczyć.\n");
scanf("%d", &sek);                    // pobranie liczby sekund
min = sek / SEK_W_MIN;           // liczba minut z obcięta częścią ułamkową
reszta = sek  %  SEK_W_MIN;    // pozostała liczba sekund
printf("%d sekund to %d minut, %d sekund.\n",   sek,   min,   reszta);
return 0;
}
```

Dane wyjściowe:

```
Przelicz sekundy na minuty i sekundy!
Podaj liczbę sekund, która chcesz przeliczyć.
496
496 sekund to 8 minut, 16 sekund.
```

Operatory inkrementacji i dekrementacji: ++ i --

Operator **inkrementacji ++** wykonuje prostą czynność polegającą na zwiększeniu operandu o 1.
Operator ten dostępny jest w dwóch odmianach.
W pierwszej z nich symbol **++** znajduje się przed operandem; jest to tzw. ***tryb przedrostkowy***.
Druga odmiana nosi nazwę ***trybu przyrostkowego*** i dwa znaki plus następują w niej po operandzie.
Tryby przedrostkowy i przyrostkowy różnią się momentem, w którym dokonywane jest zwiększenie wartości operandu.

q = 2 * ++a; ⇔ q = 2 * (a + 1);
q = 2 * a++; ⇔ {q = 2 * a; a = a + 1;}

Dla każdej postaci operatora inkrementacji istnieje odpowiednia postać operatom **dekrementacji**.
Aby ją uzyskać, wystarczy zamiast **++** użyć symbolu **--**:

```
--licznik;      // odmiana przedrostkowa operatora dekrementacji
 licznik--;     // odmiana przyrostkowa operatora dekrementacji
```

q = 2 * --a; ⇔ q = 2 * (a - 1);
q = 2 * a--; ⇔ {q = 2 * a; a = a - 1;}

Potencjalne problemy.

```
odp = num/2 + 5*(1+ num++);

y = n++ + n++;

printf("%10d  %10d\n", num,  num * num++); }
```

► Nie stosuj operatora inkrementacji lub dekrementacji do zmiennej, która jest częścią więcej niż jednego argumentu funkcji.
► Nie stosuj operatora inkrementacji lub dekrementacji do zmiennej, która w wyrażeniu pojawia się więcej niż jeden raz.

Wyrażenia i instrukcje

Wyrażenie (ang. *expression*) jest kombinacją operatorów i operandów. Najprostsze możliwe wyrażenie składa się z samego operandu.

W języku C każde wyrażenie ma wartość.

Wyrażenie	***Wartość***
-4 + 6	2
c = 3 + 8	11
5 > 3	1
2 > 5	0
6 + (c = 3 + 8)	17

Instrukcje (ang. *statements*) są głównymi elementami, z których zbudowane są programy.
Program można zdefiniować jako **ciąg instrukcji**.
Instrukcja jest kompletnym poleceniem wydawanym komputerowi.
W języku C instrukcje kończą się zawsze znakiem średnika.

```
nogi = 4  // wyrażenie, które może być częścią większego wyrażenia
nogi = 4; // jest instrukcją.

Int suma, różnica;        // instrukcja deklaracji
x = 25;                   // instrukcja podstawienia
++x;                      // instrukcja inkrementacji
printf("x = %d\n",  x);   // instrukcja wywołania funkcji
```

Instrukcją złożoną lub **blokiem** nazywamy przynajmniej dwie instrukcje scalone ze sobą przez zawarcie ich w klamrach **{ }**.

Instrukcja złożona jest składniowo równoważna instrukcji pojedynczej.

Operator rzutowania (typ)

W przypadku, gdy w wyrażeniu występują operandy różnych typów kompilator dokonuje automatycznej konwersji do tego samego typu (o ile to możliwe).

Ranga typów w kolejności rosnącej:

char, unsigned char -> int -> unsigned int -> long -> unsigned long -> float -> double -> long double

Podstawowe zasady konwersji:
1. Konwersję dokonuje się do występującego w wyrażeniu typu o najwyższej randze.
2. W instrukcji przypisania ostateczny wynik obliczeń jest przetwarzany na typ zmiennej, której nadawana jest wartość.

Programista ma możliwość wymuszenia konwersji za pomocą **rzutowania** (ang. *cast*).
Rzutowanie polega na poprzedzeniu operandu nazwą żądanego typu podaną w nawiasie.
Nazwa typu razem z nawiasem stanowi **operator rzutowania**. Jego ogólna postać to: (typ), np.:
(int), (unsigned int), (double), …

Przykład:

```
int m;
float a = 1.6, b = 1.7;
m = a + b;              // 1.6 + 1.7 -> 3.3 -> (konwersja automatyczna) -> 3
m = (int)a + (int)b;   // 1 + 1 -> 2
```

Instrukcja pętli while

```
// dodaj.c oblicza sumę pierwszych 20 liczb całkowitych
#include <stdio.h>
int main(void)
  {
   int licznik = 0, suma = 0;
  while (licznik++  <  20) suma = suma + licznik;
  printf("suma = %d\n", suma);
  return 0;
  }
```

Instrukcja while składa się z trzech odrębnych części.

Pierwszą z nich jest słowo kluczowe **while**,

drugą - podany w nawiasie **warunek**,

trzecią - **instrukcja** wykonywana (powtarzana) tak długo, jak długo warunek jest spełniony.

W pętli może znajdować się tylko jedna instrukcja. Może być to ***instrukcja prosta*** lub ***złożona***.

Program – autor tekstów piosenek biesiadnych.

```
/* butelki.c — odliczanie w dół */
#include <stdio.h>
#define MAX 100
int main(void)
  {
  int licznik = MAX + 1;
  while {--licznik > 0)          // instrukcja pętli
    {
    printf("%d butelek piwa na stole, %d butelek piwa!\n",  licznik,  licznik);
    printf("Weź jedną i puść ją w krąg,\n");
    printf("%d butelek piwa!\n\n",  licznik - 1);
    }
  return 0;
  }
```

Podsumowanie: wyrażenia i instrukcje

Wyrażenia:

Wyrażenie jest kombinacją operatorów i operandów.

Najprostszym wyrażeniem jest po prostu stała lub zmienna bez operatora, np. **22** lub **beebop**.

Przykładami bardziej skomplikowanych wyrażeń są **55 + 22** oraz **vap = 2 * (vip + (vup = 4))**.

Instrukcje:

Instrukcja jest rozkazem wydawanym komputerowi.

Wyróżniamy ***instrukcje proste*** i ***instrukcje złożone***, instrukcje proste kończą się średnikiem, co widać poniżej:

Instrukcja deklaracji:	int palce;
Instrukcja przypisania:	palce = 12;
Instrukcja wywołania funkcji:	printf ("%d\n", palce);
Instrukcja strukturalna:	while (palce < 20) palce = palce + 2;
Instrukcja pusta:	; // nie robi nic

Instrukcje złożone lub inaczej bloki składają się z kilku instrukcji (które same mogą być złożone) zawartych pomiędzy klamrami **{ }**.

Instrukcję złożoną zawiera na przykład następująca pętla while:

```
while (lata < 100)
  {
  wiedza = wiedza + 1;
  printf("%d %d\n", lata, wiedza);
  lata = lata +1;
  }
```

Podsumowanie: operatory w C.

Operator przypisania:

=	Przypisuje wartość po prawej stronie zmiennej po lewej stronie.

Operatory arytmetyczne:

+	Dodaje wartość po prawej do wartości po lewej.
-	Odejmuje wartość po prawej od wartości po lewej. Jako operator jednoargumentowy zmienia znak wartości po jego prawej stronie.
*	Mnoży wartość po prawej stronie przez wartość po lewej stronie.
/	Dzieli wartość po lewej stronie przez wartość po prawej stronie. Wynik jest obcinany, jeśli oba operandy są liczbami całkowitymi.
%	Zwraca resztę z podzielenia wartości po lewej stronie przez wartość po prawej stronie (tylko liczby całkowite).
++	Dodaje 1 do zmiennej po prawej stronie (tryb przedrostkowy) lub po lewej stronie (tryb przyrostkowy).
--	Działa jak ++, ale odejmuje 1.

Inne operatory:

sizeof	Zwraca rozmiar w bajtach operandu znajdującego się po jego prawej stronie. Operand może być nazwą typu podaną w nawiasie, np. sizeof (float) lub nazwą konkretnej zmiennej, tablicy, itp., np. sizeof foo lub sizeof (foo).
(typ)	Jako operator rzutowania zamienia następującą po nim wartość na podany w nawiasie typ. Na przykład, (float) 9 przetwarza liczbę całkowitą 9 na liczbę zmiennoprzecinkową 9.0.

Funkcje z argumentami

```
/* hash.c — definiuje funkcje z argumentem */
#include <stdio.h>

void hash(int n);          // prototyp funkcji

int main(void)
{
   int     razy = 5;
   char  ch =  '!';          // kod ASCII wykrzyknika to 33
   float  f  = 6.0;

   hash(razy);               // argument typu int
   hash(ch);                 // char jest automatycznie przetwarzany na int
   hash((int)  f);           // rzutowanie wymusza konwersje float do int
   return 0;
}

// definicja funkcji
void hash(int n)             // nagłówek funkcji
{
 while   (n --  >  0} printf("#");
 printf ("\n");
 }
```

Operatory i wyrażenia relacyjne

Wyrażenia porównujące wartości noszą nazwę **wyrażeń relacyjnych**, a występujące w nich operatory są zwane **operatorami relacyjnymi**.

*Operatory relacyj*ne.

Operator	***Znaczenie***
<	jest mniejszy niż
<=	jest mniejszy lub równy
==	jest równy
>=	jest większy lub równy
>	jest większy niż
!=	jest różny od

Czym jest prawda?

```
/* p_i_f.c — wartości prawdziwe i fałszywe w C */
#include <stdio.h>
 int main(void)
   {
   int prawda, fałsz;
   prawda = (10 > 2);    // wartość prawdziwej relacji
   fałsz = (10 == 2);      // wartość fałszywej relacji
   printf("prawda = %d; fałsz = %d \n", prawda, fałsz);
   return 0;
   } //
```

```
/* prawda.c — które wartości są prawdziwe? */
#include <stdio.h>
int main(void)
   {
   int n = 3;
   while (n) printf ("%d\n", n--) ;
   n = -3;
   while (n) printf("%2d\n", n++);
   return 0;
   }
```

WIOSEK: w języku C prawda ⇔ != 0 oraz fałsz ⇔ == 0

Priorytet operatorów relacyjnych

Priorytet operatorów relacyjnych jest niższy niż operatorów arytmetycznych, ale wyższy niż operatorów przypisania.

x > y + 2 ⇔ x > (y + 2)

x = y > 2 ⇔ x = (y > 2)

Operatory relacyjne dzielą się na dwie grupy o różnym priorytecie:

grupa o wyższym priorytecie: < <= > >=

grupa o niższym priorytecie: == ! =

x < 2 == y > 2 ⇔ (x < 2) == (y > 2)

Pętla for

```
/* zycz2.c — pętla licząca z wykorzystaniem for */
#include <stdio.h>
#define   LICZBA   22
int main(void)
   {
   int licznik;

  for (licznik = 1;  licznik <= LICZBA;  licznik++)    printf("Wesołych Świat!\n");

  return 0;
  }
```

Nawias po słowie kluczowym **for** zawiera trzy wyrażenia rozdzielone dwoma średnikami.

Pierwszym z nich jest ***inicjalizacja*** licznika. Jest ona wykonywana jeden raz, zaraz po pierwszym uruchomieniu pętli for.

Drugim wyrażeniem jest ***warunek***, obliczany przed każdym potencjalnym wykonaniem pętli. Jeśli wyrażenie to jest fałszywe, pętla zostaje zakończona.

Trzecie wyrażenie, ***zmiana*** (aktualizacja), jest obliczane na końcu każdej iteracji.

Pętla for kończy się jedną instrukcją prostą lub złożoną.

Każde z trzech wyrażeń sterujących jest pełnym wyrażeniem, a więc wszelkie skutki uboczne wywołane przez jedno z nich są realizowane przed przejściem do kolejnego wyrażenia.

Elastyczność pętli for

```
/* odliczanie w dół  */
#include <stdio.h>
 int main(void)
{
int sek;
for {sek = 5; sek > 0; sek—)
       printf("%d sekund!\n", sek) ;
printf("Start!\n");
return 0;
}
```

dane wyjściowe:

```
5 sekund!
4 sekund!
3 sekund!
2 sekund!
1 sekund!
Start!
```

```
/* liczenie co 13 */
#include <stdio.h>
int main(void)
{
 int n;
for   (n = 2;  n < 60;  n = n + 13)  printf("%d \n",   n);
return 0;
}
```

wynik:

```
2
15
28
41
54
```

```
/* znaki zamiast liczb: */
#include <stdio.h>
int main(void)
{
char ch;
for  (ch =  'a';  ch <=  'z';  ch++)
    printf("Wartość ASCII znaku %c wynosi %d.\n",  ch,  ch);
return 0;
}
```

dane wyjściowe:

```
Wartość ASCII znaku a wynosi 97.
Wartość ASCII znaku b wynosi 98.
…
Wartość ASCII znaku x wynosi 120.
Wartość ASCII znaku y wynosi 121.
Wartość ASCII znaku z wynosi 122.
```

```
#include <stdio.h>
int main(void)
{
double dług;
for(dług = 100.0; dług < 150.0; dług = dług * 1.1)
   printf("Twój dług jest teraz równy %.2f zl.\n", dług);
 return 0;
}
```

Dane wyjściowe:

```
Twój  dług jest teraz równy 100.00 zl.
Twój  dług jest teraz równy 110.00 zl.
Twój  dług jest teraz równy 121.00 zl.
Twój  dług jest teraz równy 133.10 zl.
Twój  dług jest teraz równy 146.41 zl.
```

```
#include <stdio.h>
int main(void)
{
int x;
int y = 55;
for (x = 1; y <= 75; y = (++x * 5) + 50)
     printf("%10d %10d\n", x, y);
 return 0;
}
```

Dane wyjściowe

1	55
2	60
3	65
4	70
5	75

```
/* puste wyrażenie sterujące */
#include <stdio.h>
int main(void)
{
 int odp,  n;
odp = 2;
for  (n = 3;  odp <= 25;   )
odp = odp * n;
printf("n = %d;  odp = %d.\n",  n,  odp);
return 0;
}
```

dane wyjściowe:

n = 3; odp = 54.

```
/* Pierwsze wyrażenie sterujące  nie musi inicjalizować zmiennej */.
#include <stdio.h>
int main(void)
{
int num;

for ( printf("Wpisuj liczby!\n");   num != 6;   )  scanf("%d", &num);

printf("Właśnie o nia to chodziło!\n");
return 0;
}
```

Powyższy program wyświetla jeden raz tekst *Wpisuj liczby!,*
a następnie pobiera liczby, dopóki nie wpiszesz szóstki:

Podsumowanie: instrukcja for

Instrukcja **for** do sterowania pętlą wykorzystuje trzy wyrażenia sterujące rozdzielone średnikami.

Wyrażenie **inicjalizacja** (patrz postać instrukcji for poniżej) jest wykonywane raz, przed pozostałymi instrukcjami pętli.

Następnie obliczane jest wyrażenie **test** - jeśli jest ono prawdziwe (różne od zera), wykonany zostaje jeden cyki pętli.

Na końcu cyklu wykonane zostaje wyrażenie **aktualizacja**.

Kolejna iteracja rozpoczyna się od obliczenia wyrażenia test.

Instrukcja for jest pętlą z warunkiem wejścia - decyzja o wykonaniu kolejnej iteracji zapada bowiem na początku pętli.

Może się zdarzyć, że pętla nie zostanie wykonana ani razu.

Część oznaczona instrukcja może być instrukcją prostą lub złożoną.

Postać:

for (inicjalizacja; test; aktualizacja) instrukcja

Pętla jest powtarzana do momentu, gdy wyrażenie test stanie się fałszywe {równe zeru).

Operatory przypisania: +=, -=, *=, /=, %=

```
punkty += 20        ⇔   punkty = punkty + 20
monety -= 2         ⇔   monety = monety - 2
zające  *= 2        ⇔   zające = zające  * 2
czas  /= 2.73       ⇔   czas = czas / 2.73
redukcja  %= 3      ⇔   redukcja = redukcja  % 3
x *= 3 * y + 12     ⇔   x = x *  (3 * y + 12)
```

Operator przecinkowy: ,

Operator przecinkowy (,) łączy dwa wyrażenia w jedno, gwarantując, że wyrażenie lewostronne zostanie obliczone jako pierwsze.

Jest on zwykle używany w pętlach for, gdzie umożliwia zmieszczenie większej ilości informacji w jednym wyrażeniu sterującym.

Wartością całego wyrażenia jest wartość jego prawostronnej części.

Przykłady:

```
for (krok = 2,  licznik = 0;  licznik < 1000;  krok *= 2)   licznik += krok;
x =   (y = 3,    (z = ++y + 2)   + 5);
pi = 3,14;        // nie jest to błąd składniowy !
pi = (3,14);
```

Pętla: do while

```
/* dowhile.c — pętla z warunkiem wyjścia */
#include <stdio.h>
 int main(void)
   {
  char ch;
  do
    {
    scanf("%c", &ch);
    printf("%c", ch);
    }
  while (ch != '#');
  return 0;
  }
```

Program odczytuje wpisywane znaki i wyświetla je do momentu napotkania znaku #.

Wybieram drzwi #3! // tekst wprowadzony z klawiatury

Wybieram drzwi # // tekst na ekranie

Ogólna postać pętli ***do while***:

do *instrukcja* **while (***wyrażenie* **);**

przy czym *instrukcja* może być instrukcją prostą lub złożoną.

```
/* rzedy2.c -- zastosowanie zależnych od siebie pętli zagnieżdżonych */
#include<stdio.h>
#define RZĘDY 6
#define ZNAKI 6
int main(void)
  {
  int rząd;
  char ch;
  for (rząd = 0;   rząd < RZĘDY;   rzad++)
     {
     for (ch = ('A’ + rząd);    ch < ('A' + ZNAKI);   ch++)   printf("%c", ch);
     printf("\n");
     }
   return 0;
  }
```

dane wyjściowe:

```
ABCDEF
BCDEF
CDEF
DEF
EF
F
```

```c
/* potęga.c — podnosi liczby do potęg naturalnych */
#include <stdio.h>
double potęga(double a, int b);  // prototyp funkcji potęga()
int main(void)
 {
 double x, xpot;
 int n;
 printf("Podaj liczbę oraz wykładnik naturalny, do której podniesiona zostanie liczba.\n");
 printf("Wpisz q, aby zakończyć program.\n ");
 while (scanf("%lf%d", &x, &n) == 2)
  {
  xpot = potęga(x,n);            // wywołanie funkcji
  printf("%.3g do potęgi %d to %.5g\n", x, n, xpot);
  printf("Podaj kolejna parę liczb lub wpisz q, aby zakończyć.\n");
  }
 printf("Bye!\n");
 return 0;
 }

double potęga(double a, int b)  // definicja funkcji
 {
 double pot = 1.0;
 int i;
 for(i =1; i <= b; i++) pot *= a;
 return (pot);                  // zwrot wartości pot
 }
```

Tablice

Tablica jest ciągiem wartości tego samego typu.
Cała tablica nosi jedną nazwę, a dostęp do poszczególnych jednostek - elementów tablicy uzyskujemy przez podanie ***indeksu***, który jest zawsze liczbą całkowitą nieujemną.

float długi[20];

deklaruje tablicę 20 elementów, z których każdy może przechować wartość typu float.
Pierwszy element tablicy nosi nazwę długi [0], drugi - długi [1], i tak dalej, aż do długi [19].
Numerowanie rozpoczyna się od zera.

```
/* rzedy2.c – wersja tablicowa */
#include<stdio.h>
#define RZĘDY 6
#define ZNAKI 6
int main(void)
  {
  int rząd;
  char ch, znaki[ZNAKI];    // tablica znakowa
  for (ch = 0;   ch < ZNAKI;   ch++)   znaki[ch] = ('A' + ch);          // wypelnienie tablicy
  for (rząd = 0;   rząd < RZĘDY;   rząd++)
     {
     for (ch = rząd;    ch < ZNAKI;   ch++)   printf("%c", znaki[ch]); // wypisanie linii znaków
     printf("\n");             // przejście do następnego wiersza
     }
   return 0;
  }
```

Instrukcja if

Ogólna postać instrukcji **if**:

```
if  (wyrażenie) instrukcja
```

Jeśli `wyrażenie` jest prawdziwe (niezerowe), wykonywana jest `instrukcja`. W przeciwnym wypadku jest ona pomijana. Instrukcja może być instrukcją prostą lub złożoną (blokiem).

Instrukcja if else

Ogólna postać instrukcji **if else**:

```
if (wyrażenie) instrukcja_1  else  instrukcja_2
```

Jeśli `wyrażenie` jest prawdziwe (niezerowe), wykonywana jest `instrukcja_1`. Jeśli `wyrażenie` jest fałszywe (równe zeru), wykonywana jest `instrukcja_2` następująca po słowie `else`. Obie instrukcje mogą być proste lub złożone.

```c
/* liczby.c – zliczanie liczb o różnych znakach */
#include <stdio.h>
int main(void)
 {
 int dodatnie, ujemne, zera;
 float x;
 dodatnie = ujemne = zera = 0;   // wyzerowanie liczników
 printf("Podaj ciag liczb, zakończ znakiem #.\n");
 while (scanf("%f", &x) == 1)
  {
  if (x > 0.0) dodatnie++;
  else if (x < 0.0) ujemne++;
  else zera++;
  }
 printf("Podałeś %d liczb dodatnich, %d liczb ujemnych i %d zer\n", dodatnie, ujemne, zera);
 return 0;
 }
```

Funkcje getchar() i putchar()

Funkcja `getchar()` nie pobiera argumentów, a zwraca kolejny znak z łańcucha wejściowego

`ch = getchar();` ⇔ `scanf("%c", &ch);`

Funkcja `putchar()` wyświetla na ekranie przekazany jej argument.

`putchar(ch);` ⇔ `printf("%c", ch);`

```c
/* szyfr.c — zmienia dane wejściowe zachowując odstępy */
#include <stdio.h>
#define ODSTĘP ' '            // apostrof-spacja-apostrof
int main(void)
 {
 char ch;
 while ((ch = getchar()) != '\n')     // dopóki nie ma końca wiersza
  {
  if  (ch == ODSTĘP)  putchar(ch); // pozostaw znak spacji bez zmian
  else   putchar(ch + 1);             // zmień pozostałe znaki
  }
 putchar(ch);                         // wyświetl znak nowej linii
 return 0;
 }
```

Przykładowy przebieg działania programu:

MOW MI HAL.
NPX NJ IBM/

```
/*Program elektr. c. */
#include <stdio.h>
#define STAWKA1 0.2401        // stawka dla pierwszych 240 kWh
#define STAWKA2 0.2881        // stawka dla następnych 300  kWh
#define STAWKA3 0.3457        // stawka powyżej 540 kWh
#define PROG1 240.0           // pierwszy próg
#define PROG2 540.0           // drugi próg
#define PODSTAWA1 (STAWKA1 * PROG1)     // cena pierwszych 240 kWh
#define PODSTAWA2 (STAWKA2 * (PROG2 - PROG1))     // cena pierwszych 540 kWh
int main(void)
 {
 double kwh;           // ilosc zużytych kilowatogodzin
 double oplata;        // ich cena
 printf {"Podaj ilosc zużytych kWh\n");
 scanf("%lf", &kwh);
 if (kwh <= PROGI) oplata = STAWKA1 * kwh;
 else if (kwh <= PROG2)                    // pomiędzy 240 a 540 kWh
   oplata = PODSTAWA1 + (STAWKA2 * (kwh - PROG1));
 else                                      // ponad 540 kWh
   oplata = PODSTAWA2 + (STAWKA3 * (kwh - PROG2));
 printf("Oplata za %.lf kwh wynosi %1.2f zl.\n", kwh, oplata);
  return 0;
 }
```

Rodzina funkcji znakowych ctype.h

Funkcje testujące znaki z rodziny ctype.h.

Nazwa	***Prawdziwa, jeśli argument jest***
isalnumf)	Alfanumeryczny (jest literą lub cyfrą)
isalpha ()	Alfabetyczny (jest literą)
iscntrl ()	Znakiem sterującym, np. Ctrl-B
isdigit ()	Cyfrą
isgraph ()	Jakimkolwiek znakiem drukowanym
islower()	Małą literą
isprint ()	Znakiem drukowanym lub odstępem
ispunct ()	Znakiem przestankowym (jakimkolwiek znakiem drukowanym niealfanumerycznym)
isspace()	Znakiem niedrukowanym (whitespace): odstępem, znakiem nowej linii, znakiem wysuwu strony, znakiem powrotu karetki, tabulatorem pionowym, tabulatorem poziomym lub innym znakiem w zależności od implementacji
isupper ()	Wielką literą
isxdigit()	Znakiem będącym cyfrą szesnastkową

Funkcje odwzorowujące znaki z rodziny ctype.h.

Nazwa	***Działanie***
tolower()	Jeśli argument jest wielką literą, zwraca jego maty odpowiednik; w przeciwnym wypadku, zwraca pierwotny argument.
toupper()	Jeśli argument jest małą literą, zwraca jego wielki odpowiednik; w przeciwnym wypadku, zwraca pierwotny argument.

Np.:

```
ch = tolower(ch); // zamienia ch na mała literę
```

Operatory logiczne

Operator	***Znaczenie***
&&	i (koniunkcja)
||	lub (alternatywa)
!	nie (negacja)

```
a > b  &&  b > c  ||  b > d    <=>    ((a > b)   &&   (b > c))   ||   (b > d)
```

Operator warunkowy: ?:

Ogólna postać wyrażenia warunkowego:

`wyrażenie_1 ? wyrażenie_2 : wyrażenie_3`

Jeśli `wyrażenie_1` jest prawdziwe (niezerowe), to całe wyrażenie warunkowe ma wartość taką samą jak `wyrażenie_2`.

Jeśli `wyrażenie_1` jest fałszywe (równe zeru), całe wyrażenie warunkowe otrzymuje wartość `wyrażenia_3`.

Typowe przykłady:

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

```
abs = (a > 0) ? a : -a;
```

```c
/* farba.c — wykorzystuje operator warunkowy */
#include <stdio.h>
#define POKRYCIE 18   // ilosc m.kw. pokrywana przez jedna puszkę
int main(void)
  {
  int m_kw;
  int puszki;
  printf("Podaj liczbę metrów kwadratowych do pomalowania:\n")
  while (scanf("%d", &m_kw) = 1)
    {
    puszki = m_kw / POKRYCIE;
    puszki += (m_kw % POKRYCIE == 0) ? 0 : 1;
    printf("Potrzeba %d %s farby.\n", puszki, puszki == 1 ? "puszki" : "puszek");
    printf("Podaj kolejna wartość (q kończy program):\n");
    }
  return 0;
  }
```

Wynik działania programu:

```
Podaj liczbę metrów kwadratowych do pomalowania:
18
Potrzeba 1 puszki farby.
Podaj kolejna wartość (q kończy program):
21
Potrzeba 2 puszek farby.
Podaj kolejna wartość (q kończy program):
q
```

Instrukcja continue

Instrukcji tej można używać w ramach każdej z trzech pętli.
Powoduje ona pominięcie pozostałej części cyklu i natychmiastowe rozpoczęcie kolejnej iteracji.
Instrukcja continue w strukturze zagnieżdżonej wpływa tylko na pętlę, w której się bezpośrednio znajduje (nie wpływa na pętle zewnętrzne).

```
licznik = 0;
while (licznik < 10)
  {
  ch = getchar();
  if (ch == '\n') continue;
  putchar(ch);
  licznik++;
  }

for (licznik = 0; licznik < 10; licznik++)
  {
  ch = getchar();
  if (ch == '\n') continue;
  putchar(ch);
  }
```

Instrukcja break

Instrukcja break powoduje natychmiastowe opuszczenie pętli i przejście do kolejnego etapu programu.

```
/* break.c — wykorzystuje break do wyjścia z pętli */
#include <stdio.h>
int main(void)
 {
 float dlugosc, szerokość;
 printf("Podaj dlugosc prostokąta:\n");
 while (scanf("%f", &dlugosc) == 1)
  {
  printf("Dlugosc = %0.2f:\n", dlugosc);
  printf("Podaj szerokość prostokąta:\n");

  if (scanf("%f", &szerokosc) != 1) break;

  printf("Szerokość = %0.2f:\n", szerokość);
  printf("Pole = %0.2f:\n", dlugosc * szerokość);
  printf("Podaj dlugosc prostokąta:\n");
  }
 return 0;
 }
```

Instrukcja switch

```
switch (wyrażenie)
  {
  case etykieta_1 : instrukcja_1 /* break powoduje zakończenie */
  case etykieta_2 : instrukcja_2
  …
  case etykieta_n : instrukcja_n
  default         : instrukcja_n1
  }
```

Instrukcja ***switch*** wykonuje skok do *etykiety* ***case*** odpowiadającej wartości wyrażenia, a następnie wykonuje wszystkie instrukcje do końca bloku, chyba że napotkana zostanie instrukcja ***break***.

Zarówno wyrażenie, jak i etykiety ***case*** muszą mieć wartości całkowite (dozwolony jest typ char), a etykiety muszą ponadto być stałymi lub wyrażeniami złożonymi wyłącznie ze stałych.

W przypadku, jeśli żadna z etykiet nie pasuje do wartości wyrażenia, program przechodzi w miejsce oznaczone ***default***.

Jeśli etykieta ***default*** nie istnieje, wykonany zostaje skok do pierwszej instrukcji następującej po strukturze ***switch*** (cały blok zostaje pominięty).

Przykład

```
switch (wybór)
 {
 case 1 :
 case 2 : printf("Brawo!\n"); break;
 case 3 : printf("Tak trzymać!\n");
 case 4 : printf("Bardzo dobrze!\n"); break;
 default : printf("Miłego dnia.\n");
 }
```

```c
/* samoglos.c — zliczanie samogłosek */
#include <stdio.h>
int main{void}
 {
 char ch;
 int a_licz, e_licz, i_licz, o_licz, u_licz, y_licz;          // deklaracja liczników
 a_licz = e_licz = i_licz = o_licz = u_licz = y_licz = 0;   // wyzerowanie liczników
 printf("Wpisz jakiś tekst; # kończy program.\n");
 while ((ch = getchar()) != '#')
  {
  switch(toupper(ch))
   {
   case ‘A’:  a_licz++; break;
   case ‘E’:  e_licz++; break;
   case ‘I’:   i_licz++; break;
   case ‘O’: o_licz++; break;
   case ‘U’: u_licz++; break;
   case ‘Y’:  y_licz++; break;
   default:  break;
   }
  }
 printf("liczba samogłosek:  A   E   I   O   U   Y\n");
 printf("                    %4d %4d %4d %4d %4d %4d\n", a_licz, e_licz, i_licz, o_licz, u_licz, y_licz);
 return 0;
 }
```

```
Wpisz jakiś tekst; # kończy program.
Hanka modrooka szła po wodę do potoka.#
liczba samogłosek:  A   E   I   O   U   Y
                    5   1   0   8   0   0
```

□