## Stałe

- *Stałe* (*literały*, ang. *literal constant*) reprezentują ustalone wartości, które nie mogą być zmienione podczas działania programu.
- Z każdą stałą związany jest jej typ i wartość.

### Stałe liczbowe - całkowite i zmiennopozycyjne

- Stała całkowita to liczba całkowita ewentualnie poprzedzona znakiem.
- Domyślnym typem *stałych całkowitych* jest typ `signed int` . Przykłady:
  `0  1234  -1234`
- Stała zmiennopozycyjna to liczba zawierająca kropkę dziesiętną, wykładnik, może być poprzedzona znakiem kropki dziesiętnej.
- Domyślnym typem *stałych* zmiennopozycyjnych jest typ `double`. Przykłady:
  `3.14  3.  0.14  .14  1.2e5  2e-5  0.0  0.`
  gdzie:
  `1.2e5` oznacza $1,2*10^5$
  `2e-5` oznacza $2*10^{-5}$

- Pozostałe typy wymagają wpisania odpowiedniego specyfikatora.

| Stała | Typ stałej |
|---|---|
| `1000L` | `long int` - litera `L` za liczbą całkowitą |
| `1024l` | `long int` - litera `l` za liczbą całkowitą |
| `128u` | `unsigned int` - litera `u` za liczbą całkowitą (mała lub wielka) |
| `1010LU` | `unsigned long int` - litery `LU` za liczbą (w dowolnej kolejności) |
| `3.14f` | `float` - litera `f` za liczbą z kropką dziesiętną (mała lub wielka) |
| `1.0L` | `long double` - litera `L` za liczbą z kropką dziesiętną (mała lub wielka) |

#### Różne zapisy stałej całkowitej

- Stałe całkowite można zapisywać na trzy sposoby:
  - dziesiętnie: `20`
  - ósemkowo: `024` - liczba jest poprzedzona cyfrą 0 (dziesiętnie ma wartość 20)
  - szesnastkowo: `0x14` - liczba jest poprzedzona znakami 0x (lub 0X)
- Dotyczy to *wszystkich* typów stałych całkowitych. Przykład: `0x12345LU`

### Stałe logiczne

Stałe logiczne mają dwie wartości: `true` i `false`.

### Stałe znakowe

- Stała znakowa (ang. *character constant*) to znak umieszczony w *apostrofach*: `'a'`
- Typem stałej znakowej jest `char`.
- Wartość stałej znakowej to *kod liczbowy* znaku. Do reprezentacji znaków używany jest kod ASCII.
- Przykłady:
  'a' (kod 97) - kompilator zapamiętuje znak `'a'` jako liczbę 97
  'A' (kod 65)
  `'0'` (kod 48)
  `'8'` (kod 56)
  `' '` (kod 32)
- Znak można podawać również w postaci liczby całkowitej - kodu ASCII znaku:

```
char znak;      // deklaracja zmiennej typu char
znak='a';       // przypisanie wartości kodu znaku 'a'
znak=97;        // to samo - kod podany dziesiętnie
```

- Znaki niedrukowalne wymagają poprzedzenia znakiem odwróconego ukośnika (ang. *backslash*).
- Przykłady:
  `'\n'` - koniec linii (ang. *new line*), kod ASCII 10
  `'\r'` - przesunięcie do początku wiersza (ang. *carriage return*), kod ASCII 13
  `'\a'` - sygnał dźwiękowy (ang. *alert*), kod ASCII 7
  `'\0'` - znak pusty (ang. *null*), kod ASCII 0
- Znak ukośnika można użyć wraz z ósemkowym lub szesnastkowym przedstawieniem kodu znaku.
- Przykład: różne sposoby przypisania znaku `'a'` do zmiennej `znak`

```
char znak;
znak='a';       // przypisanie wartości kodu znaku 'a'
znak=97;        // to samo - kod podany dziesiętnie
znak=0141;      // to samo - kod podany ósemkowo
znak=0x61;      // to samo - kod podany szesnastkowo
znak='\141';    // to samo - kod ósemkowy
znak='\x61';    // to samo - kod szesnastkowy
```

- Jeśli chcemy przedstawić ukośnik to musimy użyć sekwencji `'\\'` .

### Stałe napisowe

- Stała napisowa (*łańcuch*, *literał*, *tekst* ang. *string constants*, *string literals*) jest to ciąg znaków umieszczony w cudzysłowach:

```
"Witamy"
```

- Typem stałej znakowej jest *tablica* typu `char` (ciąg bajtów przylegających do siebie).
- Stała napisowa jest pamiętana jako sekwencja znaków zakończona znakiem pustym (`'\0'`).
- Więcej na temat stałych znakowych podczas omawiania tablic i wskaźników.
- Przykłady wykorzystania stałych napisowych:

```
cout << "Witamy!";
cout << "Witamy!\n";
cout << "Witamy\nw szkole\n");
cout << "Lp.\tNazwisko";
```

## Modyfikator typu `const`

- Modyfikator `const` przekształca obiekt w *stałą* (ang. *constant*). Wartość takiego obiektu określana jest podczas definiowania i nie może później ulec zmianie. Jest przeznaczona tylko do czytania (ang. *read only*).
- Często tego typu obiekt nazwa się stałą symboliczną (ang. *named constant*), w odróżnieniu od stałej zapisywanej dosłownie (ang. *literal constant*) np. w postaci liczby.
- Przykład:

```
const int id=12345;
const int max=100;
...
id=10000;  // BŁĄD! Próba przypisania nowej wartości
int tab[max]; // Poprawne użycie: stała może być
              // używana do określenia wymiaru tablicy
```

- Wartość stałej może być określana za pomocą innej stałej.

```
const int odlegloscMile = 3959;
const int odlegloscKm = 1.609*odlogloscMile;
```

- Zaletą stosowania stałych symbolicznych jest łatwiejsza pielęgnacja kodu programu. W razie zmiany wartości stałej wystarczy zmienić wiersz z jej definicją.
- Eliminujemy w ten sposób istnienie w programie magicznych liczb, reprezentujących jakieś założenie odnośnie programu, np. jakieś przeliczniki lub rozmiary tablic. Jeśli założenia te będziemy zapisywać za pomocą dobrze skomentowanych stałych symbolicznych, łatwiej będzie zmieniać czy weryfikować program.
- Z informacji o tym, że pewna wartość nie zmienia się podczas wykonywania programu może również korzystać kompilator budując bardziej efektywny kod.

## Definiowanie nazw typów - `typedef`

- Za pomocą konstrukcji `typedef` można nadać nową nazwę (synonim) istniejącemu typowi. Jeśli na przykład wpiszemy następującą definicję:
  `typedef float realType`
  oznacza to, że można teraz wymiennie używać `float` i `realType`

### Zastosowania

- Wprowadzenie własnej nazwy czyni program łatwiejszym do modyfikacji. Załóżmy, że w programie działania na liczbach rzeczywistych zaprogramowano z użyciem typu `float`. Podczas użytkowania programu stwierdzono, że wymagana jest większa precyzja i potrzebny jest typ `double`. Oznacza to, że trzeba wszystkie wystąpienia zmiennych typu `float` należy zamienić na `double`. Zmiany te można uprościć, jeśli użyto by instrukcji `typedef`. Na przykład:

```
#include <iostream.h>
int main()
{
  typedef float real; // real jest synonimem float

  const real PI=3.14159;
  real promien;
  cout << "Wpisz promien:";
  cin >> promien;
  real obwod=2*PI*promien;
  cout << "Obwod=" << obwod << endl;
  return 0;
}
```

  Zmiana typu zmiennych używanych do obliczeń sprowadzi się do wymiany instrukcji
  `typedef float real`
  na:
  `typedef double real;`

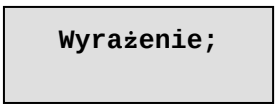
- Nowa nazwa może być wygodnym skrótem dla określenia typu o długiej nazwie:
  `typedef unsigned char uchar;`
  Odtąd zamiast `unsigned char` można pisać `uchar`.

- Standardowe biblioteki C++ bardzo często korzystają z tej techniki w przypadku deklaracji, które zależą od systemu. Łatwiej jest wtedy przenosić oprogramowania między różnymi komputerami.

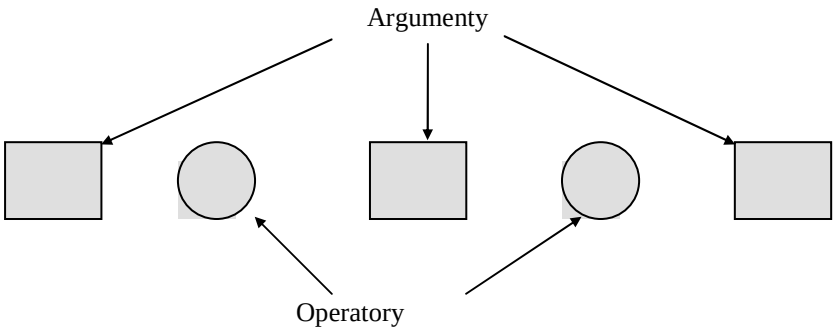
  *Przykład:*
  Załóżmy, że w pewnych zastosowaniach sieciowych potrzebujemy posługiwać się liczbami umieszczonymi na 32 bitach (przykładem jest adres IP). Moglibyśmy skorzystać z typu `unsigned int`, ale typ `int` może zajmować 2 (czyli 16 bitów) lub 4 bajty (32 bity) w zależności od komputera. Chcemy napisać program, który łatwo będzie można przenosić. Załóżmy, że nasz komputer przeznacza na typ `unsigned int` 32 bity. Możemy zatem zdefiniować typ o nazwie `uint32` jako:
  `typedef unsigned int uint32`
  Tym typem będziemy się posługiwać wszędzie tam, gdzie będziemy definiowali zmienne wymagające 32 bitów. Jeśli teraz program zostanie przeniesiony na komputer, na którym `unsigned int` zajmuje 2 bajty, zaś `unsigned long int` zajmuje 4 bajty, wystarczy w odpowiednim pliku nagłówkowym zmienić jedną definicję:
  `typedef unsigned long int uint32`

# Wyrażenia i operatory

**Instrukcja**




**Wyrażenie**




## Wyrażenia

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

```
2 * 3            /* mnożenie dwóch stałych: wartość 6, typ int */
i + 1            /* dodawania stałej i zmiennej typu int:
                    wartość zależy od i, typ int */
5 + h*6         /* h jest typu int; wartość zależy od h, typ int */
0.5 * sqrt(2.) /* sqrt - pierwiastek; wartość 0.5*1.41, typ double */
```

## Operatory

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

## Operator przypisania = i instrukcja przypisania

- Składnia:

```
arg_1 = arg_2
```

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

```
int x;
x=7;    // zmienna x przyjmie wartość 7
```

- Wyrażenie z operatorem przypisania ma wartość, którą jest wartość *arg_1* po wykonaniu przypisania.
- Typ tej wartości jest taki sam, jak typ argumentu po lewej stronie operatora przypisania (czyli *arg_1*).
- Przykłady:

```
double x;
x=7.8;
```
*wartość wyrażenia: wartość zmiennej x*
*typ wyrażenia: typ zmiennej x*

```
int a,b;
a=(b=3);
```
*poprawne, ponieważ b=3 ma wartość*
*wartość całego wyrażenia: 3, typ wyrqżenia: int*

```
int n;
double k;
k=(n=0);
```
*wartość wyrażenia: 0  typ wyrażenia: double*

### Wielokrotne przypisanie

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

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

- Jak to działa?
  Operator przypisania ma wiązanie *prawostronne* (od prawej do lewej, ang. *right to left*). Oznacza to, że obliczenia w wyrażeniu wykonuje się od jego prawej strony do lewej. Powyższy zapis jest równoważny zapisowi: `x=(y=0.);`
  (Obliczana jest wartość wyrażenia `0.`; jest ona przypisywana zmiennej `y`; jednocześnie stanowi ona wartość wyrażenia `y=0.`; następnie ta wartość jest przypisywana zmiennej `x`; wartością całego wyrażenia jest `0.`)
- *Warunek poprawnego działania*: wszystkie argumenty tych operatorów są zgodnego typu.

### Konwersja typów w instrukcji przypisania

- *Reguła konwersji typów w instrukcji przypisania*: wartość prawej strony instrukcji jest przekształcana do typu lewej strony. Konwersja ta jest wykonywana *automatycznie*.
  Jeżeli zatem typ *arg_2* nie zgadza się z typem *arg_1*, przed przypisaniem wykonywana jest konwersja *arg_2* do typu *arg_1*. Czyli wartość prawej strony instrukcji (wartość wyrażenia) jest przekształcana do typu lewej strony.
- ***Podczas przekształcania z typu „szerszego" na „węższy" może nastąpić utrata informacji.***
- Przykład:

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

- Inne przykłady konwersji:

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

## Pierwszeństwo i łączność operatorów

- Kolejność wykonywania operacji w wyrażeniu określają reguły *pierwszeństwa* (*priorytet*, ang. *precedence*) i *łączności* (*wiązanie*, ang. *associativity*) operatorów.
- *Priorytet* operatorów stanowi o kolejności wykonywania działań określonych przez poszczególne operatory. Operatory o wyższym priorytecie określają działania, które są najpierw wykonywane.
- Przykład: operator + oznacza dodawanie, zaś operator * mnożenie, operator * ma wyższy priorytet niż +

  **2 + 3 * 4** daje w wyniku 14; operator * ma wyższy priorytet niż +

  **3 * 4 + 2** daje w wyniku 14; operator * ma wyższy priorytet niż +
- *Wiązanie* określa sposób łączenia operatora z argumentami. Może to być:
  - wiązanie lewostronne (ang. *left-to-right*) - obowiązujące dla większości operatorów
  - wiązanie prawostronne (ang. *right-to-left*) - obowiązujące na przykład dla operatorów jednoargumentowych i operatorów przypisania.
- Jeśli operatory mają ten sam priorytet, to kolejność działań wynika z wiązania. Przykład:

  **2 + 3 - 4** daje w wyniku 1; operatory + i - mają ten sam priorytet, kolejność wykonania operacji jest ustalana na podstawie wiązania: w tym wypadku lewostronnego
- Kolejność działań można zmienić za pomocą nawiasów. Pozwalają one wyodrębnić podwyrażenia. Obliczając wyrażenie złożone, najpierw oblicza się podwyrażenia zawarte w najbardziej wewnętrznych nawiasach.
- Przykład:

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

**Priorytety wybranych operatorów C++**

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

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

## Operatory arytmetyczne

Składnia:

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

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

- Aby operacje arytmetyczne były realizowane w sposób jednoznaczny, operatorom są nadane priorytety. W tabeli linią poziomą oddzielono operatory o różnych priorytetach. Tak więc operatory `*`, `/` i `%` mają wyższy priorytet niż operatory `+` i `-` .
- Operatory o tym samym priorytecie wykonywane są zgodnie z wiązaniem.
- Kolejność wykonywania działań można zmieniać za pomocą nawiasów. Wyodrębniają one podwyrażenia. Najpierw oblicza się podwyrażenia zawarte w najbardziej wewnętrznych parach nawiasów, później podwyrażenia w nawiasach zewnętrznych.
- Przykład:
  `2 + 3 * 5` daje w wyniku `17`
  `(2 + 3) *5` daje w wyniku `25`
- Obowiązują dwie arytmetyki:
  - *arytmetyka liczb całkowitych*: wynik działania na liczbach całkowitych jest liczbą całkowitą
  - *arytmetyka liczb rzeczywistych*: wynik działania na liczbach rzeczywistych jest liczbą rzeczywistą
  - w przypadku mieszanym (liczba całkowita i liczba rzeczywista) wynik jest liczbą rzeczywistą.
- Przykład:
  `1/2` daje w wyniku `0`
  `4/3` daje w wyniku `1`
  `1./2` daje w wyniku `0.5`
  `1./2.` daje w wyniku `0.5`

- Przykłady zapisów w języku C++

| Wyrażenie algebraiczne | Wyrażenie w C++ |
|---|---|
| $a+7$ | `a + 7` |
| $a-b$ | `a - b` |
| $cx$ lub $c \cdot x$ | `c * x` |
| $x/y$ lub $\frac{x}{y}$ lub $x \div y$ | `x / y` |
| $x \bmod y$ | `x % y` |
| $m = \frac{a+b+c+d+e}{5}$ | `m = (a+b+c+d+e)/5;` |
| $y=mx+b$ lub $y=m \cdot x + b$ | `y = m * x + b` |

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

Krok 1. `y = 2 * 5 * 5 + 3 * 5 + 7 ;`
`=10` mnożenie najbardziej od lewej



Krok 2. `y = 10 * 5 + 3 * 5 + 7 ;`
`=50` mnożenie najbardziej od lewej



Krok 3. `y = 50 + 3 * 5 + 7 ;`
`=15` mnożenie przed dodawaniem



Krok 4. `y = 50 + 15 + 7 ;`
`=65` dodawanie najbardziej od lewej



Krok 5. `y = 65 + 7 ;`
`=72` ostatnie dodawanie



Krok 6. `y = 72;` przypisanie `72` do zmiennej `y`

- Do zmiany kolejności wykonywania działań można używać nawiasów.

### Obliczenia arytmetyczne

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

## Operatory porównania, relacji oraz logiczne

Składnia:

*arg_1 operator arg_2* (operator dwuargumentowy)
*operator arg_1* (operator jednoargumentowy)

| Operator | Operacja | Wiązanie | Priorytet |
|---|---|---|---|
| ! | negacja logiczna NOT (zaprzeczenie) | od prawej do lewej | NAJWYŻSZY |
| < | mniejsze | od lewej do prawej | |
| <= | mniejsze lub równe | od lewej do prawej | |
| > | większe | od lewej do prawej | |
| >= | większe lub równe | od lewej do prawej | |
| == | równe | od lewej do prawej | |
| != | nierówne | od lewej do prawej | |
| && | iloczyn logiczny AND (koniunkcja) | od lewej do prawej | |
| \|\| | suma logiczna OR (alternatywa) | od lewej do prawej | NAJNIŻSZY |

- Wartością wyrażeń relacji lub logicznych w języku C++ jest `true` (w języku C jest to wartość 1) lub `false` (w języku C wartość 0). Jeśli jednak operator taki wystąpi w kontekście wymagającym użycia wartości całkowitej, jego wynik będzie awansowany do wartości 1 lub 0.
- Operatory porównania i relacji służą do obliczenia wartości „prawda" (`true` lub liczba 1) lub „fałsz" (`false` lub liczba 0). Warunek prawdziwy daje w wyniku wartość `true`, warunek fałszywy - `false`.
- Operator negacji logicznej `!` daje w wyniku wartość `true`, jeżeli jego argument ma wartość `false` lub 0; w przeciwnym przypadku jego wynikiem jest `false`.
- Wynikiem operatora iloczynu logicznego `&&` jest prawda wtedy i tylko wtedy, gdy oba argumenty są prawdziwe.
- Wynikiem operatora sumy logicznej `||` jest prawda wtedy i tylko wtedy, gdy którykolwiek z argumentów jest prawdziwy.
- *Uwaga*: gdy tylko określi się, że całe wyrażenie jest prawdziwe lub fałszywe, zaprzestaje się wykonywania dalszych operatorów w tym wyrażeniu. Przykłady:
  ```
  wyr1 && wyr2; // jeśli wyr1 ma wartość false, nie będzie obliczane wyr2
  wyr1 || wyr2; // jeśli wyr1 ma wartość true, nie będzie obliczane wyr2
  ```
- Operatory relacji i logiczne mają niższy priorytet niż operatory arytmetyczne:
  wyrażenie `x + 1 < y * 2` jest interpretowane jako `(x+1) < (y*2)`
- Przykłady:

| Wyrażenie | Wartość | Wyrażenie | Wartość |
|---|---|---|---|
| `-1 < 0` | `true` | `int a=1;` | |
| `0 > 1` | `false` | `int b=10;` | |
| `0 == 0` | `true` | | |
| `1 != -1` | `true` | `bool wynik; // w stylu C++` | |
| `1 >= -1` | `true` | `wynik = a<b;` | `true` |
| `1> 10` | `false` | `wynik = a == b;` | `false` |
| | | `wynik = a != b;` | `true` |
| | | | |
| | | `int wynik; // w stylu C` | |
| | | `wynik = a<b;` | `1` |
| | | `wynik = a == b;` | `0` |

## Operator zwiększania ++ i zmniejszania - -

- Są to operatory jednoargumentowe.
- Operatory zwiększania `++` i zmniejszania `--` umożliwiają zwarty zapis dodawania lub odejmowania liczby 1. Można napisać:
  ```
  licznik++;
  ```
  zamiast
  ```
  licznik=licznik+1;
  ```
- Inne nazwy tych operatorów to operatory *inkrementacji* i *dekrementacji*.
- Oba operatory można rozumieć jako operatory przypisania - *argument musi być modyfikowalną l-wartością* (na przykład zmienną typu arytmetycznego).
- Operatory `++` i `--` mają ten sam priorytet, wyższy od operatorów arytmetycznych.

| **Operator** | **Operacja** | **Wiązanie** | **Priorytet** |
|---|---|---|---|
| `++` | zwiększanie o 1, operator przyrostkowy | od prawej do lewej | NAJWYŻSZY |
| `--` | zmniejszanie o 1, operator przyrostkowy | od prawej do lewej | ↑ |
| `++` | zwiększanie o 1, operator przedrostkowy | od prawej do lewej | |
| `--` | zmniejszanie o 1, operator przedrostkowy | od prawej do lewej | NAJNIŻSZY |

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

Przykłady:

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

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

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

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

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

- Pytanie: Jak myślisz, dlaczego język C++ nie nazywa się ++C ? (*S.Lippman*, *Podstawy języka C++*)

## Operator pobrania rozmiaru `sizeof`

- Składnia:

```
sizeof(nazwa_typu);
sizeof obiekt;
```

- Operator `sizeof` przekazuje liczbę bajtów oznaczającą rozmiar jego argumentu.
- Przykłady:

```
double x;
cout << sizeof x;        // nazwa obiektu (zmiennej) - nie trzeba nawiasu
cout << sizeof(double);  // identyfikator typu - w nawiasie

#include <iostream.h>
int main() {
  cout << "Typ\t\tRozmiar" << endl;
  cout << "char\t\t"  << sizeof(char) << endl;
  cout << "short\t\t" << sizeof(short) << endl;
  cout << "int\t\t" << sizeof(int) << endl;
  cout << "long\t\t" << sizeof(long) << endl;
  cout << "float\t\t" << sizeof(float) << endl;
  cout << "double\t\t" << sizeof(double) << endl;
  return 0;
}
```

## Operator wyliczeniowy , (przecinek)

- Składnia:

```
wyrażenie, wyrażenie, wyrażenie;
```

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

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

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

## Operatory bitowe

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

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

| p | q | p & q | p \| q | !p |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |

- Tabela różnicy symetrycznej:

| p | q | p^q |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

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

- Przykłady:

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

- Przesuwanie w lewo `x<<y` jest równoważne mnożeniu $x*2^y$
- Przesuwanie w prawo `x>>y` jest równoważne dzieleniu $x/2^y$

## Skróty - złożone operatory przypisania

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

```
arg_1 = arg_1 opera tor arg_2
```

  można napisać:

```
arg_1 operator= arg_2
```

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

- Przykład:
  Zamiast `x = x+1;` można napisać `x += 1;`

- Złożony operator przypisania ma taki sam priorytet i łączność jak operator przypisania =.

## Język C++ i operatory wejścia-wyjścia

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

```
cout << "Witam";
cout << k;  // k jest zmienną typu int
```

  Definiując możliwości operatora `<<` zapewniono również to, aby można było przesyłać na wyjście wiele zmiennych i tekstów:

```
int k=5;
cout << "Wartosc k to " << k;
```

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

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

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

## Konwersja typów

- Konwersja typu (ang. *type conversion*) ma miejsce wtedy, kiedy wspólnie korzysta się ze zmiennych lub obiektów różnych typów, na przykład w instrukcjach przypisania, w wyrażeniach, podczas przekazywania argumentów aktualnych do wywoływanej funkcji.
- Konwersja typu może być:
  - niejawna (ang.*implicite*) – wykonywana automatycznie przez kompilator,
  - jawna (ang. *explicite*)– wykonywana na życzenie użytkownika.
- Przykład:
  - `double x=3;  /* konwersja niejawna */`
  - `double x=(double)3; /* konwersja jawna */`
- Konwersje zarówno jawne jak i niejawne muszą być *bezpieczne*: czyli wszędzie tam, gdzie jest to możliwe, dokonywane tak, aby nie stracić przechowywanych informacji.
- Przykładem bezpiecznej konwersji jest przekształcanie argumentów o mniejszych rozmiarach, a tym samym mniejszym zakresie wartości i dokładności, do typów o większych rozmiarach. Np. `int` do `double.`

### Przykłady konwersji niebezpiecznych

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

### Konwersja niejawna

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

### Przekształcenia arytmetyczne

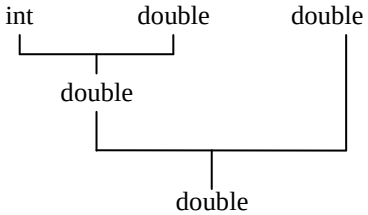
- Przekształcenie arytmetyczne ma zapewnić to, że oba argumenty operatora dwuargumentowego (np. dodawania) będą przekształcone (mówimy *awansowane* lub *promocja*) do wspólnego szerszego typu, który będzie typem wyniku.
- Przekształcenia ta podlegają dwu zasadom:
  - aby nie dopuścić do zmniejszenia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

| Typ | Typ po konwersji | Metoda konwersji |
|---|---|---|
| `char` | `int` | powielenie bitu znaku dla typu `signed char` lub wypełnienie zerami dla typu `unsigned char` |
| `unsigned char` | `int` | starszy bajt wypełniony zerami |
| `signed char` | `int` | powielenie bitu znaku |
| `short` | `int` | ta sama wartość |
| `unsigned short` | `unsigned int` | ta sama wartość |
| `enum` | `int` | ta sama wartość |

- Jeśli po tej konwersji nadal argumenty różnią się typami, wykonywane są przekształcenia typu węższego na szerszy. Dla wszystkich par argumentów wykonywane są przekształcenia:
  **IF** jeden z argumentów jest `long double`
  **THEN** drugi jest przekształcany do typu `long double`
  **ELSE IF** jeden z argumentów jest `double`
  **THEN** drugi jest przekształcany do typu `double`
  **ELSE IF** jeden z argumentów jest `float`
  **THEN** drugi jest przekształcany do typu `float`
  **ELSE IF** jeden z argumentów jest `unsigned long int`
  **THEN** drugi jest przekształcany do typu `unsigned long int`
  **ELSE IF** jeden z argumentów jest `signed long int`
  **THEN** drugi jest przekształcany do typu `signed long int`
  **ELSE IF** jeden z argumentów jest `unsigned int`
  **THEN** drugi jest przekształcany do typu `unsigned int`
  **ELSE** oba argumenty są typu `int`.

- Przykład:

```
char c;
int i;
float f;
double d, wynik;
wynik = (c/i) + (f*d) - (f+i)
```



### Konwersje jawne - rzutowanie typów

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

  ```
  (nazwa_typu) wyrażenie
  ```

  Jest to tzw. *notacja rzutowania w starym stylu*, przejętym z języka C. Standard języka C++ ma nowy sposób rzutowania, który będzie przedstawiony w wykładzie "*Programowanie obiektowe*".

- Przykład

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


#include <iostream.h>
int main() {
  int i=0;
  while (i<=10) {
    cout << i <<"   " <<(float) i/2 << endl;
    i++;
  }
  return 0;
}
```

# Instrukcje

- *Instrukcja* (ang. *statement*) opisuje czynności wykonywane w programie. W języku naturalnym analogiczną konstrukcją jest zdanie.
- W języku C++ wyróżnia się instrukcje pojedyncze (proste) i złożone.
- Instrukcja *pojedyncza* jest zakończona średnikiem.
- Ciąg instrukcji umieszczony w nawiasach klamrowych tworzy instrukcję *złożoną* (ang. *compound statement*). Nazywana jest ona również *blokiem*. Instrukcji złożonej nie trzeba kończyć średnikiem.
- Jeżeli nie powiedziano inaczej, instrukcje są wykonywane *sekwencyjnie* czyli w takiej kolejności, w jakiej występują w programie.
- Kolejność wykonywania obliczeń można zmieniać za pomocą instrukcji *sterujących* (ang. *control statements*). W języku C++ są to instrukcje: *wyboru*, *powtarzania* i *skoku*.
- Instrukcje *wyboru* (ang. *decision*) służą do wybrania jednej z kilku ścieżek wykonania programu.
- Instrukcje *powtarzania* (*pętli*, ang. *loop*) służą do iteracyjnego wykonywania fragmentu programu.
- Instrukcje *skoku* służą do bezwarunkowego przekazania sterowania do innego miejsca w programie.
- Instrukcje języka C++ przedstawiono w poniższej tabeli.

| Typ instrukcji | Składnia |
|---|---|
| pusta | `;` |
| wyrażeniowa (prosta) | *wyrażenie*`;`<br>*wywołanie_funkcji*`();` |
| złożona | `{`<br>`  `*instrukcja*<br>`  `*instrukcja*<br>`  `*instrukcja*<br>`}` |
| instrukcje wyboru | **`if (`***wyrażenie***`)`** *instrukcja*<br>**`if (`**wyrażenie**`)`** *instrukcja* **`else`** *instrukcja*<br>**`switch (`***wyrażenie***`)`**<br>`{`<br>`  `*instrukcja*<br>`}` |
| instrukcje powtarzania (pętli) | **`while (`***wyrażenie***`)`** *instrukcja*<br>**`do`** *instrukcja* **`while (`***wyrażenie***`);`**<br>**`for (`***wyrażenie***`;`***wyrażenie***`;`***wyrażenie***`)`** *instrukcja* |
| instrukcje skoku | **`break;`**<br>**`continue;`**<br>**`return`** *wyrażenie*`;`<br>**`goto`** *identyfikator*`;` |

- Najprostszą instrukcją jest *instrukcja pusta*. Ma ona postać:
  ```
  ;
  ```
  Instrukcja ta nie powoduje wykonania żadnej czynności i jest wprowadzona przede wszystkim do ujednolicenia opisu pewnych konstrukcji.
- Zapis *`instrukcja`* bez średnika oznacza, że może być w tym miejscu użyta dowolna instrukcja: pusta, prosta lub złożona.
- Wszędzie tam, gdzie składnia wymaga *`instrukcji`*, można użyć instrukcji pustej, prostej lub złożonej.

## Instrukcja prosta i złożona

- Instrukcja *prosta* (*wyrażeniowa*) to wyrażenie lub wywołanie funkcji zakończone średnikiem.
- Przykłady:
  - wyrażenie - zwiększenie wartości zmiennej `i` o `1`:
    ```
    i++;
    ```
  - wyrażenie - instrukcja przypisania:
    ```
    y=2*i;
    ```
  - wywołanie funkcji:
    ```
    drukuj("Pomiary",t); // funkcja drukuje wyniki pomiarów
    ```
    (Uwaga: funkcja nie zwraca wartości lub pomijamy zwracaną wartość, ponieważ nas ona nie interesuje.)
  - wyrażenie zawierające wywołanie funkcji:
    ```
    a=sqrt(x); // funkcja zwraca wartość określonego typu
    ```

- Instrukcja *złożona* (*blok instrukcji*) to ciąg instrukcji umieszczonych w nawiasach klamrowych.
- Po instrukcji złożonej nie używa się średnika.
- Składniowo instrukcja złożona jest równoważna jednej instrukcji prostej - można ją umieścić wszędzie tam, gdzie może się pojawić prosta instrukcja.
- Przykład: składnia instrukcji warunkowej `if` pozwala aby z `if` związana była tylko jednia instrukcja do wykonania; w praktyce często chcemy wykonać więcej instrukcji - wówczas musimy użyć instrukcji złożonej:

```
// zamiana wartości zmiennych a i b
if (a > b)
{
  int tymcz = a;
  b=a;
  a=tymcz;
}
```

## Instrukcja przypisania

- Instrukcja przypisania (podstawienia) pozwala zmienić wartość zmiennej podczas wykonywania programu.
- Ogólna postać instrukcji przypisania (ang. *assignment statement*):

```
nazwa_zmiennej = wyrażenie;
```

- Przykłady:

```
i=1;
i=(2+3)*5;
i=2+3*5;
k=i+j;
```

- Instrukcje przypisania pozwalają tylko na realizację algorytmów najprostszej postaci: algorytmów liniowych, które polegają na wykonaniu ciągu podstawień.

- Przykład:
Dane są boki trójkąta. Oblicz jego pole ze wzoru Herona: $s=\sqrt{(p(p-a)(p-b)(p-c)}$ , gdzie *a*,*b*,*c* to boki trójkąta, zaś *p* to połowa obwodu.

```
// Rozwiązanie 1
#include <iostream.h>  // dla cin i cout
#include <math.h>      // dla obliczania pierwiastka kwadratowego sqrt
int main()
{
  double a,b,c;
  double p,s;
  cin >> a >> b >> c; // zakładamy, że długości a,b,c tak są dobrane,
                      // że można z nich zbudować trójkąt
  p=(a+b+c)/2;
  s=sqrt(p*(p-a)*(p-b)*(p-c));
  cout << "Pole: " << s << endl;
  return 0;
}

// Rozwiązanie 2
#include <iostream.h>
#include <math.h>
int main()
{
  double a,b,c;
  cin >> a >> b >> c; // zakładamy, że długości a,b,c tak są dobrane,
                      // że można z nich zbudować trójkąt
  cout << "Pole: " << sqrt((a+b+c)*(b+c-a)*(c+a-b)*(a+b-c)/4) << endl;
  return 0;
}
```

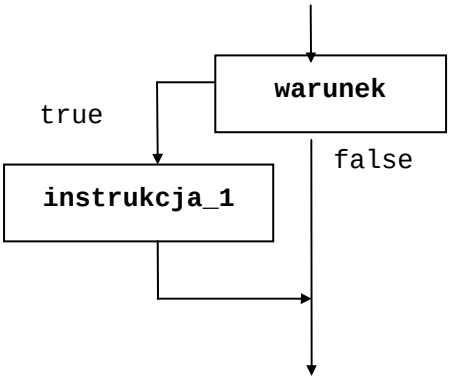
- *Komentarz*
Algorytm 1: zmienna pomocnicza *p* i *s*, 5 dodawań i odejmowań, 4 mnożenia i dzielenia, jedno pierwiastkowanie
Algorytm 2: nie ma zmiennych pomocniczych, 8 dodawań i odejmowań, 4 mnożenia i dzielenia, jedno pierwiastkowanie.
Algorytm 1 jest oszczędniejszy ze względu na liczbę wykonywanych operacji, algorytm 2 zaś ze względu na oszczędność pamięci.

## Instrukcja wyboru - `if-else`

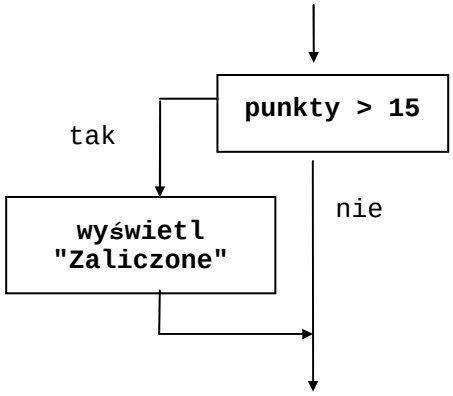
- Instrukcja `if` służy do podejmowania decyzji. Umożliwia wykonywanie instrukcji lub bloku instrukcji w zależności od tego, czy sprawdzany warunek jest prawdziwy.

**Przykład 1: Pojedynczy wybór `if`**

```
if (warunek)
  instr_1; // Wykonaj, gdy prawda
```

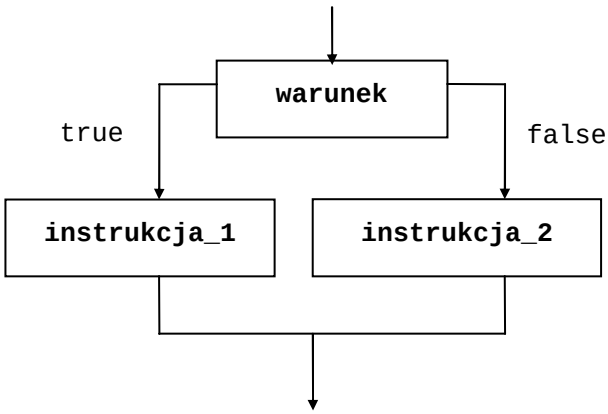



```
if (punkty >= 15 )
  cout >> "Zaliczone\n";
```




**Przykład 2: Podwójny wybór `if-else`**

```
if (warunek)
  instr_1;   // Wykonaj, gdy prawda
else
  instr_2;   // Wykonaj, gdy fałsz
```




```
if (punkty >= 15)
  cout << "Zaliczone\n";
else
  cout << "Nie zaliczone\n";
```

**Przykład 3. Różne zapisy warunku w instrukcji `if`**

```
if (x > 0) cout << "x jest dodatnie" << endl;

if (x != 0) cout << "x nie jest równe zeru" << endl;
if (x) cout << "x nie jest równe zeru" << endl; // krócej

if (x==0) cout << "x jest równe zero" << endl;
if (!x) cout << "x jest równe zero" << endl; // krócej

if (a>2 && a<5) cout << "a należy do przedziału" << endl;

if (!(a>2 && a <5)) cout << "a nie należy do przedziału" << endl;
// czytelniej
if (a <= 2 || a >=5) cout << "a nie należy do przedziału" << endl;
```

**Przykład 4. Podwójny wybór `if-else`**

```
// Program sprawdza, czy wpisano literę
#include <iostream.h>
#include <ctype.h>  // dla isalpha()
int main()
{
  char znak;
  cout << "Wpisz litere: ";
  // zamiast cin >> znak uzyjemy funkcji cin.get,
  // rezygnujemy z "inteligencji" operatora >>
  znak=cin.get();
  if (isalpha(znak))
    cout << "Wpisales litere: <" << znak << ">" << endl;
  else
    cout << "Znak \<" << znak << "\> o kodzie ASCII <"
         << (int)znak <<"> to nie litera" << endl;
  return 0;
}
```

Uwagi:

- Funkcja `int isalpha(int c)` zwraca wartość różną od zera, jeżeli `c` jest literą. Nie uwzględniane są tutaj polskie litery.
- Poniższe zapisy są równoważne:

```
if (isalpha(znak))
if (isalpha(znak) != 0)
```

- Chcemy wczytać jeden dowolny znak, musimy więc zrezygnować z "inteligencji" operatora >> (na przykład pomijania spacji) i użyć odpowiedniej funkcji. Zamiast `cin >> znak` wywołujemy funkcję `cin.get()`.
- Operator << jest operatorem "inteligentnym" - rozpoznaje typ wyprowadzanej wartości. Chcąc wydrukować znak jako liczbę (czyli odpowiadający mu kod ASCII) musimy powiedzieć o tym operatorowi << za pomocą operatora rzutowania `(int)`.

**Przykład 5. Użycie instrukcji złożonych**

```
if (warunek)
{                    /* instrukcja złożona */
   instrukcja_1;
   instrukcja_2;
}
```

Poprawny fragment programu:

```
if (punkty < 15)
  cout << "Zaliczone\n";
else {
  cout << "Nie zaliczone\n";
  cout << "Termin egzaminu poprawkowego: \n";
}
```

Niepoprawny fragment programu:

```
if (punkty < 15)
  cout << "Zaliczone" << endl;
else
  cout << "Nie zaliczone" << endl;
  cout << "Termin egzaminu poprawkowego:" << endl;
```

Komentarz: składnia jest poprawna i kompilator skompiluje program. Jednakże w obydwu wypadkach wyświetlony zostanie komunikat: `Termin egzaminu poprawkowego:`, ponieważ z instrukcją `else` związano tylko jedną instrukcję: `cout << "Nie zaliczone" << endl;`

**Przykład 6: Stopniowanie `if-else-if`**

```
if (wyrażenie_1)
  instr_1;
else if (wyrażenie_2)
   instr_2;
...
else
   instr_n;
```

A. Zlicz liczby dodatnie, ujemne i zera:

```
if (x>0)
  l_dod++;
else if (x<0)
  l_uj++;
else        // czyli  x == 0
  l_zer++;
```

B. Wyświetl oceny

```
if ( punkty >= 28)
  cout << "bdb" << endl;
else if (punkty >= 25)
  cout << "db" << endl;
else if (punkty >= 15)
  cout << "dst" << endl;
else
  cout << "ndst" << endl;
```

**Przykład 7: Zagnieżdżone struktury `if-else`**

```
if (warunek_1)
{
  if (warunek_2)
    instrukcja;
}
else
  instrukcja;

/* drukuj 1 jeśli x ≠ 0 i y ≠ 0,
   drukuj 2, jeśli x = 0,
   w przeciwnym razie nic nie drujkuj */
if (x)
  {
    if (y)
      cout << "1" << endl;
  }
else
  cout "2" << endl;
```

**Przykład 8: Program zgadywanka**

```
// Program zgadywanka wersja 1
#include <iostream.h>
#include <stdlib.h>   // dla rand()
#include <time.h>     // dla time()
int main () {
   int liczba;      // liczba z generatora
   int odp;         // liczba podana przez zgadującego
// generowanie liczby losowej
   liczba = rand()%10; // program ma generować liczby z zakresu 0-9
// odgadywanie
   cout << "Podaj liczbę z zakresu od 0 do 9: ";
   cin >> odp;
   if (liczba == odp)
     cout << "* * Gratulacje. Odgadłeś * *" << endl;
   else
     cout << "* * Nie zgadłeś. Moja liczba to " << liczba <<" * *" <<endl;
   return 0;
}
```

Uwagi:

- Funkcja `rand()` generuje liczby pseudolosowe z zakresu 0 i RAND_MAX (stała zdefiniowana w `stdlib.h`).
- Sekwencja liczb generowanych przez funkcję `rand()` jest zawsze taka sama. W przypadku naszego programu oznacza to, że za każdym razem użytkownik musiałby odgadnąć tę samą liczbę. Jeśli chcemy, żeby w każdym uruchomieniu programu generowana była inna liczba, trzeba dostarczyć jej innej wartości bazowej, na przykład za pomocą funkcji `srand()`.

```
/* program zgadywanka wersja 2 */
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
int main ()
{
   int liczba;      // liczba z generatora
   int odp;         // liczba podana przez zgadującego
// generowanie liczby losowej
   srand((unsigned) time(0));
   liczba = rand()%10;
// odgadywanie
   cout << "Podaj liczbę: ";
   cin >> odp;
   if (liczba == odp)
       cout << "* * Gratulacje. Odgadłeś. * *" << endl;
   else {
       cout << "* * Nie zgadłeś. * *" << endl;
       if (odp > liczba) cout << "Twoja liczba jest zbyt duża\n" << endl;
       else cout << "Twoja liczba jest zbyt mała\n" << endl;
   }
   cout << "Moja liczba to " << liczba << endl;
   return 0;
}
```

Uwagi:

- Funkcja `srand()` pobiera argument całkowity typu `unsigned` i używa go do zainicjalizowania generatora liczb losowych.
- Jeśli nie chcemy wprowadzać za każdym razem liczby bazowej inicjalizującej generator liczb losowych, możemy wykorzystać funkcję `time()` z argumentem 0, która zwraca wtedy bieżący czas w sekundach.

## Skrócony zapis instrukcji `if-else` czyli operator warunkowy ?

Składnia:

```
wyrażenie_1 ? wyrażenie_2 : wyrażenie_3
```

- Jest to operator trójargumentowy. Zastępuje instrukcję `if-else` postaci:

```
if (warunek) instrukcja;
else instrukcja;
```

- Działanie:
  1. Obliczana jest wartość *wyrażenia_1*;
  2. Jeśli tą wartością jest `true`, to obliczane jest *wyrażenie_2*, zaś *wyrażenie_3* jest ignorowane;
  3. W przeciwnym razie (wartością jest `false`) obliczane jest *wyrażenie_3*, zaś *wyrażenie_2* jest pomijane; uzyskana wartość *wyrażenia_2* lub *wyrażenia_3* staje się wartością wyrażenia warunkowego

- Przykład 1

```
x=10;
znak = (x < 0) ? -1 : 1; /* zmiennej znak przypisane zostanie 1 */
```

To samo zapisane za pomocą instrukcji `if`:

```
x=10;
if (x < 0) znak =-1;
else znak = 1;
```

- Przykład 2

```
cout << "Większa z liczb to: " << ( (a>b) ? a : b );
```

To samo zapisane za pomocą instrukcji `if`:

```
if (a>b)
  cout << "Większa z liczb to: " << a;
else
  cout << "Większa z liczb to: " << b;
```

- Przykład 3

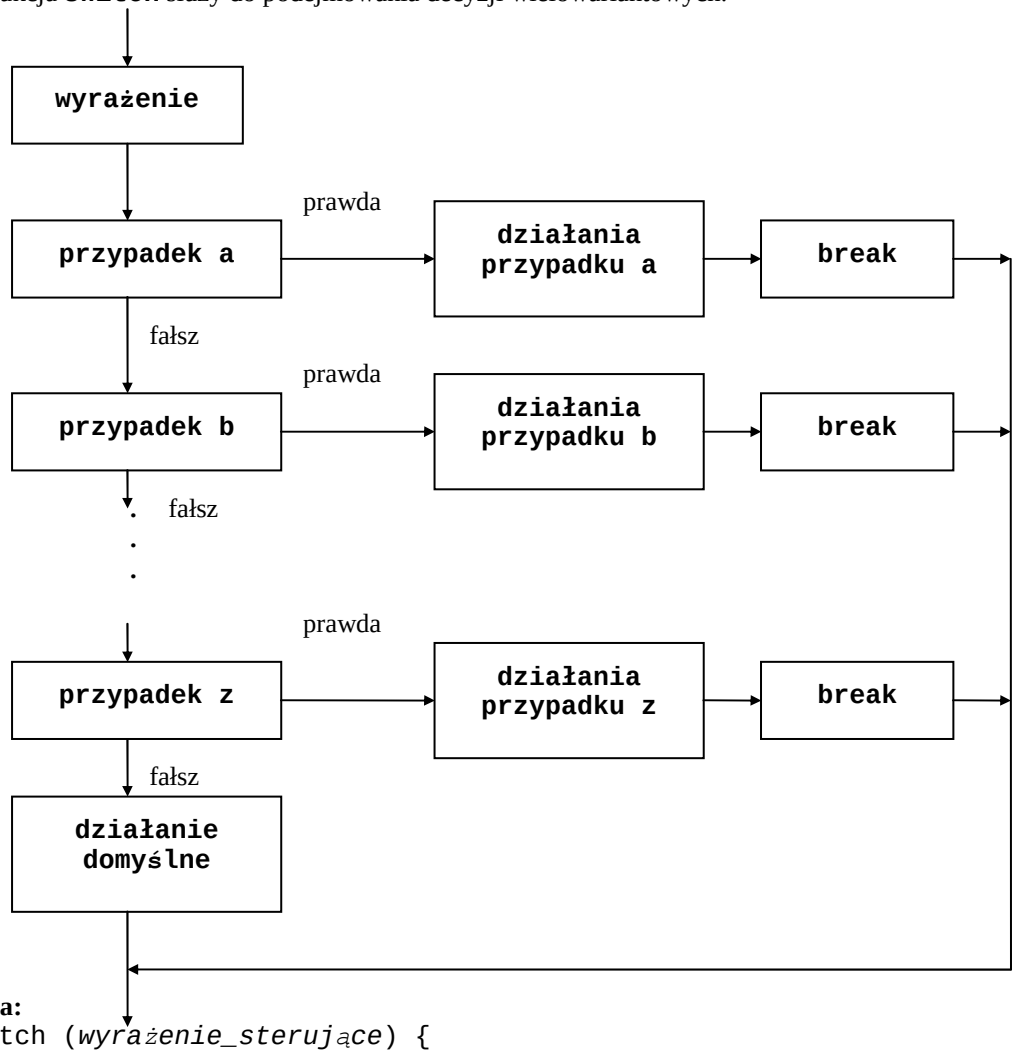
```
 int min (int a, int b)
{ return ( a<b) ? a : b; }
```

Zamiast

```
int min (int a, int b)
{
  if (a<b)
    return a;
  else
    return b;
}
```

## Instrukcja wyboru `switch`

- Instrukcja **`switch`** służy do podejmowania decyzji wielowariantowych.




**Składnia:**

```
switch (wyrażenie_sterujące) {
  case wyrażenie_stałe: instrukcje
  ...
  case wyrażenie_stałe: instrukcje
  default: instrukcje
}
```

- Działanie:
  - Oblicz wartość wyrażenia sterującego, które występuje po słowie `switch`.
  - Porównaj kolejno tę wartość z wyrażeniami stałymi występującymi po słowie `case` (etykietami wariantów).
  - Jeżeli napotkana zostanie etykieta wariantu równa wartości wyrażenia, wykonaj instrukcje poczynając od pierwszej instrukcji występującej po tej etykiecie. Instrukcja `break` powoduje zakończenie instrukcji `switch` i przejście do pierwszej instrukcji za `}` zamykającym `switch`.
  - Jeżeli wartość wyrażenia `switch` nie odpowiada żadnej etykiecie `case`, a istnieje etykieta `default`, wykonaj instrukcje występujące po tej etykiecie. (Uwaga: nie musi to być ostatnia etykieta).
- *Uwaga*:
  - Wartość wyrażenia występującego po słowie `switch` musi być typu całkowitego
  - Instrukcje są wykonywane od pierwszej instrukcji występującej po wybranej etykiecie do końca instrukcji `switch` (nawias `}` ), stąd potrzeba instrukcji `break`, która powoduje zakończenie wykonywania instrukcji wybranego przypadku i przejście do instrukcji umieszczonej po `}` zamykającym `switch`.
  - Przypadek `default` nie jest obowiązkowy. Jeśli nie występuje i wartość wyrażenia nie pasuje do innych przypadków, to nie podejmuje się żadnej akcji.

**Przykład 1:** Wyświetlenie opisu słownego otrzymanej oceny. Ocena wczytywana jest w postaci liczby.

```
cin >> ocena;
switch (ocena)
{
   case 5: cout << "bardzo dobry\n";
           break;
   case 4: cout << "dobry\n";
           break;
   case 3: cout << "dostateczny\n";
           break;
   case 2: cout << "mierny\n";
           break;
   case 1: cout << "niedostateczny\n";
           break;
   default:cout << "Ocena spoza zakresu\n";
           break; /* dobry zwyczaj programowania, to również break
                     po ostatniej instrukcji case
                     nie zapomni się wtedy o break w przypadku
                     rozszerzenia o dodatkowy przypadek */
}
```

**Przykład 2:** Proste menu.

```
#include <iostream.h>

int main() {
  char wybor;
  cout << "1. Dodaj rekord\n";
  cout << "2. Usuń rekord\n";
  cout << "3. Szukaj rekord\n";
  cout << " Wpisz numer polecenia: ";
  cin >> wybor;
  switch (wybor)
  {
    case '1': dodaj_rekord();
              break;
    case '2': usun_rekord();
              break;
    case '3': szukaj_rekord();
              break;
    default:  cout << "Niewłaściwy numer polecenia\n";
              break;
   }
   return 0;
}
```

**Przykład 3.** Porównanie dwóch liczb i wyświetlenie odpowiedniego komunikatu

```
switch (x>y)
{
   case 0: cout << "x nie jest większe od y" << endl;
           break;
   case 1: cout << "x jest większe od y" << endl;
           break;
}
```

**Przykład 5.** Prosty kalkulator

```
double oblicz (double arg1, double arg2, char op) {
   switch (op) {
     case '+': return arg1 + arg2;
     case '-': return arg1 - arg2;
     case '*': return arg1 * arg2;
     case '/': return arg1 / arg2;
     default:  cout << "Błąd: Nieznany operator" << endl;
               exit(1);
   }
}
```

**Przykład 6.** Wyświetlenie dni w wybranym miesiącu

```
switch (Miesiac)
{
  // 30 dni ma Kwiecień, Czerwiec, Wrzesien, Listopad
  case 4: case 6: case 9: case 11:
    Dni=30;
    break;
  // pozostale miesiace mają 31 dni
  case 1: case 3: case 5: case 7: case 8:
  case 10: case 12:
    Dni=30;
    break;
  // z wyjątkiem Lutego
  case 2:
    if (RokPrzestepny)
      Dni=29;
    else
      Dni=28;
    break;
  default:
    cout << "Nieprawidlowy numer miesiaca\n";
}
```