Strumienie standardowe C

Lista standardowych strumieni w języku C składa się z trzech elementów: `stdin` (strumień wejściowy), `stdout` (strumień wyjściowy) oraz `stderr` (standardowe wyjście błędów). W niektórych implementacjach wyróżnia się dodatkowo `stdprn` (strumień wyjściowy związany z drukarką), `stdaux` (strumień wyjściowy związany z ekranem). Najczęściej używanymi funkcjami operującymi na tych strumieniach są: `printf`, `puts`, `scanf`, `gets` (zdefiniowane w `stdio.h`).

`printf` jest funkcją służącą do wysyłania sformatowane danych do `stdout`, zdefiniowaną w `stdio.h`. Zazwyczaj funkcja ta używana jest do wypisywania danych na ekranie komputera (jeśli `stdout` jest skojarzony z ekranem komputera, a nie z np. plikiem). Funkcja `printf` zwraca liczbę wypisanych znaków lub wartość ujemną, jeśli wystąpił błąd. Funkcja ta może mieć zmienną liczbę parametrów. Jej składnia wygląda następująco:

```
int printf(const char * ciąg_formatujący[,argument_1, argument_2, ...]);
```

ciąg_formatujący w wywołaniu `printf` musi pojawić się zawsze, natomiast obecność kolejnych argumentów jest uzależniona od postaci *ciągu_formatującego*.

ciąg_formatujący jest stałą łańcuchową, tj. ciągiem znaków ujętych w podwójne cudzysłowie. Może on zawierać: zwykłe znaki (widoczne znaki ASCII z pominięciem niektórych z nich), sekwencje znaków kontrolnych (składające się ze znaku `\` i pojedynczej litery), oraz specyfikację formatowania kolejnych argumentów (zaczynające się znakiem `%`). Kiedy *ciąg_formatujący* nie zawiera żadnej specyfikacji formatowania, funkcja `printf` ma tylko jeden argument – *ciąg_formatujący* właśnie. Jeśli nie występują w nim sekwencje kontrolne, to *ciąg_formatujący* jest zwykłym tekstem, bez zmian wypisywanym na `stdout`. Gdy *ciąg_formatujący* zawiera sekwencje kontrolne, ich znaczenie będzie takie, jak podano w poniższej tabelce:

sekwencje znaków kontrolnych	
sekwencja	Znaczenie
`\a`	dzwonek (alarm)
`\b`	cofnięcie (backspace)
`\n`	znak nowej linii
`\t`	znak tabulacji
`\\`	ukośnik (backslash)
`\?`	Pytajnik
`\'`	pojedynczy apostrof
`\"`	podwójny apostrof

Przykład:

kod źródłowy

```
#include <stdio.h>
void main(){
      printf( "\n\nSekwencja\tZnaczenie" );
      printf( "\n========\t=======" );
      printf( "\n\\a\t\tdzwonek" );
      printf( "\n\\b\t\tbackspace" );
      printf( "\n...\t\t...");
}
```

wynik

```
Sekwencja       Znaczenie
========        =======
\a              dzwonek
\b              backspace
...             ...
```

Obecność w *ciągu_formatującym* specyfikacji formatowania świadczy o tym, że funkcja `printf` ma wypisać na `stdout` wartość jakiegoś argumentu. Sposób wypisania tej wartości zależy od specyfikacji formatowania. Składnia specyfikacji formatowania jest następująca (choć może być bogatsza):

`%[`*flagi*`] [`*szerokość*`] [.`*precyzja*`] [{| l | h | L|}]znak_typu`

gdzie `znak_typu` jest parametrem obowiązkowym, pozostałe zaś parametry specyfikacji są opcjonalne.

flagi (tj. jeden znak lub parę znaków) definiują sposób wyrównania oraz sposób wypisywaniu znaków, pól pustych, przecinków dziesiętnych, przedrostów dla liczb dziesiętnych, ósemkowych i szesnastkowych. Ich znaczenie i wartości domyślne przedstawia poniższa tabela.

Flaga	Znaczenie	Wartość domyślna
-	wyrównanie do lewej wewnątrz danego pola	wyrównanie do prawej
+	przedrostek określający znak (+ albo −) dla liczby ze znakiem	znak występuje tylko dla ujemnych liczb ze znakiem
0	jeśli jest przedrostkiem szerokości, zera są dodawane aż do wypełnienia pola o zadanej minimalnej szerokości. 0 jest ignorowane, jeśli: występuje razem ze znakiem − ; towarzyszy któremuś ze znaków typu (i, u, x, X, o, d).	nie uwzględniane
' ' (spacja)	jeśli wartość wyjściowa jest ze znakiem i jest dodatnia, to dodawana jest przed nią spacja. Flaga ta jest ignorowana, jeśli wystąpi ona razem z flagą +	spacje nie są dodawane
#	towarzysząc znakom typu o, x lub X powoduje, odpowiednio, dopisanie 0, 0x lub 0X do każdej niezerowej wartości wyjściowej	nie występują znaki puste
	towarzysząc znakom typu e, E lub f wymusza wystąpienie przecinka dziesiętnego w każdym przypadku	przecinek dziesiętny, jeśli są jakieś liczby po przecinku
	towarzysząc znakom typu g lub G wymusza pojawienie się przecinka dziesiętnego we wszystkich przypadkach i zabezpiecza przed obcinaniem zer. Jest ignorowana, kiedy występuje razem z c, d, i, u lub s.	przecinek dziesiętny, jeśli są jakieś liczby po przecinku. Zera są obcinane

szerokość jest nieujemną, dziesiętną liczbą całkowitą, określającą minimalną liczbę wypisywanych znaków. Jeśli liczba wypisywanych znaków wartości wyjściowej jest mniejsza niż zadeklarowana *szerokość*, do wypisywanej wartości dodawane są znaki puste do lewej lub do prawej strony (zależnie od obowiązującego sposobu wyrównania) aż do osiągnięcia minimalnej liczby znaków. Jeśli *szerokość* poprzedzona jest znakiem 0, puste pola wypełniane są zerami aż do osiągnięcia minimalnej liczby znaków. Jeśli parametr *szerokość* jest zadeklarowany jako gwiazdka (*), wtedy wartość dziesiętna określająca minimalną liczbę wypisywanych znaków pobierana jest z listy argumentów. Wartość ta musi poprzedzać argument, który aktualnie będzie formatowany. Zadeklarowanie małej *szerokości* nigdy nie powoduje obcinania wartości. Jeśli liczba znaków wartości wyjściowej przekracza *szerokość* lub gdy *szerokość* nie jest zdefiniowana, to wypisywane są wszystkie znaki (zgodnie z zadeklarowaną *precyzją*).

precyzja jest nieujemną, dziesiętną liczbą całkowitą określającą liczbę znaków, które mają być wypisane, liczbę miejsc dziesiętnych lub liczbę cyfr znaczących. Deklaracja *precyzji* może powodować obcinanie wypisywanych wartości wyjściowych lub zaokrąglanie wartości zmiennoprzecinkowych (zależnie od znaku_typu). Jeśli *precyzja* określona jest jako 0 i wartość do wypisania równa jest 0, to żaden znak nie jest wypisywany (jak w przypadku printf("%.0d", 0);). Jeśli parametr *precyzja* jest zadeklarowany jako gwiazdka (*), jego wartość pobierana jest z listy argumentów. Wartość ta musi poprzedzać argument, który aktualnie będzie formatowany. Domyślna wartość *precyzji* dla liczb całkowitych równa jest 1, zaś dla liczb zmiennoprzecinkowych równa jest 6.

znak_typu określa sposób, w jaki potraktowany zostanie odpowiadający mu argument funkcji printf. W ogólności liczba specyfikacji formatowania (tj. fragmentów *ciągu_formatującego* zaczynających się od %) powinna odpowiadać liczbie argumentów funkcji printf (tj. liczbie argumentów występujących za *ciągiem_formatującym*). Jeśli argumentów w wywołaniu printf jest więcej niż specyfikacji formatowania, nadmiarowe argumenty nie zostaną wypisane. Jeśli argumentów jest mniej, wypisane zostaną śmiecie. Ponadto argumentem funkcji printf może być wyrażenie, którego wartość jest wyliczana. znak_typu i typ argumentu musza ze sobą korelować. Na przykład: specyfikacji formatowania %c powinien odpowiadać argument typu char, %d – argument typu int lub short, %ld – argument typu long, %s – argumentu typu tablica znaków, itd. Poniższa tabela zawiera listę znaków typu oraz ich znaczenie.

Znaki formatujące i ich znaczenie	
znak	Znaczenie
d, i	liczba całkowita ze znakiem w kodzie dziesiętnym
u	liczba całkowita bez znaku w kodzie dziesiętnym
o	liczba całkowita w kodzie ósemkowym
x, X	liczba całkowita (bez znaku) w kodzie szesnastkowym (x dla małych liter, X dla dużych liter).
c	pojedynczy znak (odpowiadający argument powinien być kodem ASCII znaku)
e, E	liczba zmiennoprzecinkowa w notacji naukowej, zgodnie z podaną precyzją, gdzie e stosuje się dla wypisania małej litery wykładnik, E dla litery dużej. Dla domyślnej wartości *precyzji* 123.45 wyświetlone będzie jako 1.234500e+002.
f	liczba zmiennoprzecinkowa w kodzie dziesiętnym, zgodnie z podaną *precyzją*. Dla domyślnej wartości *precyzji* 123.45 wyświetlone będzie jako 123.450000.
g, G	g działa jak e, lub f, zaś G jak E lub f, wybierając format dający bardziej zwarty zapis dla bieżącej wypisywanej wartości. Format e użyty zostanie, jeśli wykładnik wartości jest mniejszy niż –4 albo większy lub równy zadeklarowanej *precyzji*, w przeciwnym razie użyty zostanie format f. Zera na końcu są obcinane, przecinek wystąpi, jeśli istnieje przynajmniej jedna cyfra po przecinku.
n	Nic nie wypisuje. Odpowiadający argument powinien być wskaźnikiem do int. Funkcja printf wpisze pod ten wskaźnik liczbę dotychczas wyprowadzonych znaków (np. przy deklaracji int licznik; wykonaniu funkcji printf("1234%n567",&licznik); sprawi, że licznik będzie równy 5).
s	łańcuch znaków (argument powinien być wskaźnikiem do char). Znaki są

	wypisywane aż do napotkania znaku null (‘\0’) lub przekroczenia limitu wypisanych znaków określonego przez *precyzję* (standardowo jest to 32767). Kończący ciąg znak null nie jest wypisywany.
l	przedrostek (long) stosowany przed: d u x o (nie odpowiada mu żaden argument)

Jeśli po znaku % wystąpi jakiś znak nie odgrywający roli w formatowaniu, znak ten zostanie wypisany bez żadnych zmian. Dlatego printf("%%"); wypisuje pojedynczy znak %.

Szerokość i *precyzja* decydują o wielkości pola przeznaczonego na wypisywaną liczbę oraz ilości cyfr uwzględnianych po przecinku. Na przykład specyfikacja %4d mówi, że argument będzie wypisany jako liczba dziesiętna na czterech pozycjach, %4f – jako liczba rzeczywista na 4 pozycjach, %5.3 – jako liczba rzeczywista na 5 pozycjach z dokładnością do 3 miejsc po przecinku, itd.
Przykład:

kod źródłowy

```
#include <stdio.h>
int a = 2, b = 10;
char c = '$';
float f = 1.05, g = 25.5, h = -0.1;

main()
{
  printf("1:a = %d, f = %f\n", a, f);
  printf("2:\t%d \t%c \t%s\n", a, c, "wyraz");
  printf("3:\t%f\t%f\n", f, g, h);
  printf("4:\t%f\n\t%f\n", f, g);
  printf("%f%%, b/a = %d", f, b/a);
  return 0;
}
```

wynik

```
1:a = 2, f = 1.050000
2:      2       $       wyraz
3:      1.050000        25.500000
4:      1.050000
        25.500000
5:1.050000%, b/a = 5
```

scanf jest funkcją służącą do wczytywania pod zadany adres pamięci sformatowane danych ze strumienia stdin, zdefiniowaną w stdio.h. Funkcja ta może mieć zmienną liczbę parametrów. Jej składnia wygląda następująco:

int scanf(const char * *ciąg_formatujący*[,*argument*]…);

ciąg_formatujący w wywołaniu scanf musi pojawić się zawsze, natomiast obecność kolejnych argumentów jest uzależniona od postaci *ciągu_formatującego*.
ciąg_formatujący jest stałą łańcuchową, tj. ciągiem znaków ujętych w podwójne cudzysłowie, definiującą sposób interpretacji danych w strumieniu wejściowym.
ciąg_formatujący przeglądany jest od lewej strony do prawej. Podobnie jak w przypadku funkcji printf, może on zawierać: znaki puste (spacje (' '); tabulatory ('\t'); znaki nowej linii ('\n')); znaki niepuste (ale bez %); specyfikacje formatowania.
Pojedynczy znak pusty w ciągu formatującym powoduje, że scanf pomija w strumieni wejściowym wszystkie pojawiające się znaki puste, aż do wystąpienia znaku niepustego. Mówiąc inaczej, jednemu znakowi pustemu w ciągu formatującym odpowiada dowolna liczba (włącznie z 0) pustych znaków lub ich kombinacji w strumieniu. Znaki niepuste (jak np. 'a', ..., '9') powodują, że scanf czyta odpowiadające im niepuste znaki ze strumienia

wejściowego (choć i w tym wypadku nigdzie ich nie zapisuje). Jeśli któryś z niepustych znaków w strumieniu wejściowym nie pokrywa się z niepustym znakiem w ciągu formatującym, scanf kończy swoje działanie, pozostawiając ten znak w strumieniu wejściowym.
Obecność specyfikacji *formatowania* w *ciągu_formatującym* świadczy o tym, że funkcja scanf ma: wczytać znaki ze standarowego wejścia, przekonwertować je zgodnie z podanym formatem, zapisać otrzymane dane pod adres podany w argumencie.
Argumenty powinny być adresami w pamięci, pod które wartości danych mają być wczytywane. Kolejne *specyfikacje formatowania* odpowiadają kolejnym argumentom.
Składnia specyfikacji formatowania jest następująca:

```
%[*] [szerokość] [{h | l | L}]znak_typu
```

Jeśli po % występują jakieś znaki, które nie decydują o formatowaniu, znaki te (aż do następnego znaku %) muszą wystąpić w strumieniu wejściowym. Jeśli w strumieniu wejściowym spodziewany jest znak %, można go „przeskoczyć" używając %%.
gwiazdka (*) występująca zaraz za znakiem % nie pozwala przypisać wartości pod adres podany w bieżącym argumencie, choć wartość ta jest wczytana zgodnie z podaną specyfikacją. Umożliwia to „przeskoczenie" wybranej wartości w strumieniu wejściowym.
Pole w strumieniu wejściowym zdefiniowane jest jako ciąg znaków, które czytane są: aż do pierwszego wystąpienia znaku pustego (spacji, tabulatora, znaku nowej linii); lub aż do pierwszego znaku, który nie może być skonwertowany zgodnie z podaną specyfikacją; lub aż do osiągnięcia limitu zdefiniowanego parametrem *szerokość* (tj. maksymalnej liczby znaków dla danego argumentu).
znak_typu mówi o tym, jakiego typu danych należy spodziewać się w strumieniu. Lista znaków typu funkcji scanf jest podobna do znaków typu funkcji printf:

Znaki formatujące scanf i i ich znaczenie	
znak	Znaczenie
d, i	liczba całkowita ze znakiem w kodzie dziesiętnym
u	liczba całkowita bez znaku w kodzie dziesiętnym
o	liczba całkowita w kodzie ósemkowym (bez znaku)
x, X	liczba całkowita (bez znaku) w kodzie szesnastkowym
c	pojedynczy znak
s	łańcuch znaków
f, e	liczba zmiennoprzecinkowa

Przedrostki znaku typu (l, h, L) są modyfikatorami typu argumentu. l (przed d u x o) oznacza, że będzie on traktowany jako long int; l (przed f e) - jako double; h - jako short int; L (przed: f e) – jako long double.
Funkcja scanf zwraca liczbę poprawnie wczytanych danych, tj. liczbę wartości, które zostały poprawnie skonwertowanych i przypisanych argumentom. Jeśli jakaś dana została wczytana, a nie została przypisana argumentowi, nie jest ona zliczana. Jeśli funkcja zwróci 0, to znaczy, że nie została wykonana żadna operacja przypisania. W przypadku wystąpienia błędu wartością zwracaną jest EOF. Wartość EOF jest również zwracana, jeśli napotkany został znak końca pliku lub znak końca ciągu znaków podczas próby czytania pierwszego znaku. Jeśli argumentów w wywołaniu scanf jest zbyt wiele (więcej niż specyfikacji formatowania), nadmiarowe argumenty są ignorowane. Jeśli argumentów jest za mało, wynik działania scanf jest nieprzewidywalny.
Przykład:

Kod źródłowy

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

wynik

```
> Podaj liczby typu: int, float, double,
```

```
void main(void){
int a; float b;
double c;
char d;
printf("Podaj liczby typu: int, float, double, char:" );
scanf( "%d %f %le %c", &a, &b, &c, &d);
printf(("Wynik wczytywania: a=%d b=%f c=%e d=%c", a, b, c, d);
}
```

```
char:
> 123 (Enter)
> 456.789 12e-10 (Enter)
> z (Enter)
> Wynik wczytywania: a=123, b=456.789, c=1.200000e-009 znak == 'a'
```

Strumienie standardowe C++

Język C++ dostarcza czterech predefiniowanych strumieni. Są to: `cin` (standardowe wejście – połączone z klawiatura), `cout` (standardowe wyjście – połączone z ekranem monitora), `cerr` (standardowe wyjście – połączone z monitorem), `clog` (podobnie do `cerr`, tylko, że jest to strumień buforowany).
Klasą odpowiedzialną za automatyczne tworzenie i inicjalizowanie predefiniowanych strumieni (tj. tworzenie i inicjalizowanie odpowiednich obiektów) jest statyczna klasa `Iostream_init`. Normalnie użytkownik nie tworzy żadnego obiektu klasy `Iostream_init`. Obiekt tej klasy tworzony jest niejawnie przy pierwszym pojawieniu się referencji do któregoś z predefiniowanych strumieni.

Predefiniowane strumienie są obiektami klas `istream` (`cin`) oraz `ostream` (`cout`, `cerr`, `clog`). Dysponują one wszystkimi możliwościami tych klas, a więc dysponują również możliwościami odziedziczonymi z klasy bazowej `ios`.

Wewnątrz klasy `ios` zdefiniowanych jest szereg parametrów, typów i metod, dzięki którym można sterować własnościami klasy, wykonywać pewne operacje, monitorować osiągane stany. Bity niektórych parametrów interpretowane są jako flagi określające np. sposób formatowania danych, czy też flagi informujące o błędach. W ogólności użytkownik nie musi wcale wiedzieć, za co odpowiadają poszczególne bity parametrów. Wystarczy, jeśli znać będzie funkcje operujące na tych bitach oraz typy wyliczeniowe zdefiniowanych w klasie (o nazwach odpowiadających roli bitów). Najważniejsze funkcje, manipulatory, flagi i stałe dostępne w klasie `ios` przedstawione są poniżej.

Funkcje dostępu do flag i funkcje definiujące sposób formatowania (metody publiczne `ios::`)	
`long flags( long lFlags );` `long flags() const;`	- ustawia flagi formatowania i zwraca stare flagi - odczytuje flagi formatowania

`long setf( long lFlags );`
 	
`long setf( long lFlags, long lMask );`
	- ustawia te flagi formatowania, którym odpowiada 1 w argumencie *`lFlags`* i zwraca cały stare flagi;
	- zmienia te wartości flag, którym odpowiada 1 w argumencie *`lMask`*, przy czym nowa wartość flagi zdefinowana jest wartością bitu na odpowiednim miejscu w argumencie *`lFlags`*; oraz zwraca stare flagi

`long unsetf( long lFlags );`	- czyści te flagi formatowania, którym w argumencie *`lFlags`* odpowiada 1 i zwraca stare flagi
`char fill( char cFill );` `char fill() const;`	- ustawia nowy znak wypełnienia i zwraca stary znak - odczytuje znak wypełnienia

`int precision( int np );`
 	
`int precision() const;`
	- ustawia dokładność wyświetlania liczb zmiennoprzecinkowych i zwraca starą dokładność
	- odczytuje dokładność wyświetlania liczb zmiennoprzecinkowych. Domyślnie dokładność ustawione jest na 6 cyfr. Jeśli formatem wyświetlania jest format `scientific` lub `fixed`, dokładność określa liczbę cyfr po przecinku. Jeśli formatem jest `automatic`, `precision` określa ogólną liczbę cyfr znaczących.

`int width( int nw );`
 	
`int width() const;`
	- ustawia szerokość pola w strumieniu wyjściowym zwracając starą wartość
	- odczytuje szerokość pola w strumieniu wyjściowym. Jeśli szerokość wynosi 0 (wartość domyślna), do strumienia wstawiane będą tylko znaki konieczne do reprezentowania wstawianej wartości. Kiedy szerokość różna jest od 0, wolne pola uzupełniane są znakami wypełnienia. Deklaracja szerokości mniejszej niż liczba wyprowadzanych znaków danej wartości nie powoduje ich obcięcia. Parametr `nw` jest szerokością minimalną.

Maski (statyczne parametry publiczne `ios::`)	
`static const long basefield;`	używana do otrzymania flagi podstawy konwersji (`dec`, `oct`, lub `hex`)
`static const long adjustfield`	używana do ustawienia flagi wyrównania w polu (`left`, `right`, lub `internal`)
`static const long floatfield`	używana do otrzymania flagi formatu numerycznego (`scientific` lub `fixed`)

Przykład (warunek sprawdzający, czy liczby wypisywane będą szesnastkowo):

```
extern ostream os;
if( ( os.flags() & ios::basefield ) == ios::hex ) .....
```

Funkcje testujące status (metody publiczne ios::)	
`int good() const;`	zwraca wartość różną od zera, jeżeli nie było błędu (wszystkie bity błędu są zerami)
`int bad() const;`	zwraca wartość różną od zera, aby pokazać wystąpienie poważnego błędu we/wy (co jest równoważne z ustawieniem flagi błędu `badbit`)
`int eof() const;`	zwraca wartość różną od zera, jeśli osiągnięty został koniec pliku (co jest równoważne z ustawieniem flagi błędu `eofbit`)
`int fail() const;`	zwraca wartość różną od zera, jeśli wystąpił błąd we/wy (ale nie koniec pliku) (co jest równoważne do ustawienia flagi błędu `badbit` lub `failbit`). Jeśli funkcja bad zwróci wartość 0, prawdopodobnie wystąpił naprawialny błąd formatowania lub konwersji.
`int rdstate() const;`	zwraca aktualny stan błędu (wartość flagi konkretnego błędu uzyskać można przez zastosowanie operatora `&` (`AND`) do wartości zwracanej i którejs z masek: `ios::goodbit`, `ios::eofbit`, `ios::failbit` lub.`ios::badbit`)
`void clear( int nState = 0 );`	ustawia lub czyści flagi błędu strumienia zgodnie z wartościami bitów *`nState`* (bity te można ustawić stosując operator `|` (`OR`) z maskami `ios::goodbit`, `ios::eofbit`, `ios::failbit` lub.`ios::badbit`)

Status błędu operacji we/wy przechowywany jest wewnątrz klasy ios. Istnieją dwie metody na sprawdzenie statusu błędu we/wy:

1. Można wywołać metodę `rdstate` (zwracającą aktualny stan błędu) i odczytać status błędu przez zastosowanie masek `ios::goodbit` (nie ma błędu), `ios::eofbit` (osiągnięto koniec pliku), `ios::failbit` (wystąpił błąd we/wy), `ios::badbit` (wystąpił poważny błąd we/wy).

Przykład: `if(File.rdstate() & ios::eofbit) { // koniec pliku}`

2. Można użyć funkcji testujących status: `bad`, `eof`, `fail`, `good`

Inne funkcje (metody publiczne `ios::`)	
`delbuf`	steruje powiązaniem usuwania `streambuf` z destrukcja `ios`
`rdbuf`	pobiera obiekt `streambuf` związany ze strumieniem
`sync_with_stdio`	synchronizuje predefiniowane obiekty `cin`, `cout`, `cerr`, oraz `clog` ze standardowymi strumieniami wejścia/wyjścia systemu (`stdin`, `stdout`, `stderr`). Jeśli w programie występują operacje we/wy zaimplementowane zgodnie ze standardem języka C i języka C++, wtedy należy użyć tej metody.
`tie`	wiąże podany `ostream` z bieżącym strumieniem.

Operatory (metody publiczne `ios::`)	
`operator void*`	dokonuje konwersji strumienia do wskaźnika, który może być użyty jedynie do sprawdzania błędów
`operator !`	zwraca wartość różną od 0 jeśli wystąpił błąd we/wy.

Manipulatory formatowania (publiczne stałe wyliczeniowe `ios::`)	
`dec`	powoduje, że kolejne pola interpretowane są w formacie dziesiętnym (domyślnie)
`hex`	powoduje, że kolejne pola interpretowane są w formacie szesnastkowym
`oct`	powoduje, że kolejne pola interpretowane są w formacie ósemkowym
`skipws`	opuszczanie znaków pustych (spacji, tabulatorów, znaków końca wiersza) na wejściu
`left`	wyrównanie do lewej
`right`	wyrównanie do prawej
`internal`	dodaje znak wypełnienia za znakiem prowadzącym lub znakiem podstawy, ale przed wartością
`showbase`	wyświetla stałą numeryczną w formacie, który może być przeczytany przez kompilator C++
`showpoint`	pokazuje przecinek dziesiętny oraz pozostałe zera dla liczb zmiennoprzecinkowych
`uppercase`	wyświetla duże litery `A` do `F` w zapisze szesnastkowym oraz duże `E` w zapisie naukowym
`showpos`	pokazuje znak + dla wartości pozytywnych
`scientific`	wyświetla liczby zmiennoprzecinkowe w notacji naukowej
`unitbuf`	powoduje, że `ostream::osfx` opróżnia bufory strumienia po każdej operacji wstawiania (tj. operacje nie są buforowane)
`stdio`	powoduje, że `ostream::osfx` opróżnia bufory strumieni `stdout` i `stderr` po każdej operacji wstawiania (tj. operacje nie są buforowane)

Manipulatory parametryzowane (wymagają `iomanip.h`)	
`long setiosflags( long lFlags );`	ustawia flagi formatujące strumienia (flagi mogą być połączone operatorem `|` (`OR`))
`long resetiosflags( long lFlags );`	czyści flagi formatujące strumienia

`int setfill( int nFill );`	ustawia znak wypełnienia strumienia
`int setprecision( int np );`	ustawia precyzję wypisywania liczb
`int setw( int nw );`	ustawia szerokość pola (ważne tylko dla następnego pola)

Manipulatory trybów otwarcia strumienia (publiczne stałe wyliczeniowe `ios::`)	
`in`	dozwolony odczyt ze strumienia
`out`	dozwolone zapis do strumienia
`app`	wskaźnik przesuwany na koniec strumienia przed wykonaniem każdej operacji zapisu
`ate`	wskaźnik przesuwany na koniec strumienia po stworzeniu obiektu obsługującego strumień
`trunc`	obcięcie istniejącego strumienia do rozmiaru 0 po stworzeniu obiektu obsługującego strumień
`binary`	ustawia tryb otwarcia pliku na tryb binarny (domyślnym trybem jest tryb tekstowy, zobacz `ifstream::setmode`)
`nocreate`	Jeśli plik nie istnieje, otwieranie pliku skończy się niepowodzeniem (zobacz `ifstream::open`)
`noreplace`	Jeśli plik istnieje, otwieranie pliku skończy się niepowodzeniem, chyba, że plik otwierany jest do dopisywania lub jego wskaźnik jest ustawiany od razu na koniec (zobacz `ifstream::open`)

Manipulatory określające pozycję odniesienia (publiczne stałe wyliczeniowe `ios::`)	
`beg`	względem początku pliku
`cur`	względem pozycji aktualnej
`end`	względem końca pliku

Klasa `istream`, oprócz metod odziedziczonych z klasy `ios`, zawiera metody służące odczytywaniu danych wspólne dla wszystkich strumieni wejściowych. Metody te pozwalają odczytywać dane w postaci binarnej i tekstowej. Definicja klasy `istream` zawarta jest w pliku `iostream.h`. Do najważniejszych metod tej klasy należą:

Metody klasy istream	
`istream& get( char& znak)`	wczytuje jeden znak ze strumienia
`istream& getline( char* bufor, int max_dług, char delim = '\n' );`	wczytuje do bufora znaki ze strumienia dopóki: albo nie napotka znaku_końca, albo wczytanych zostało już `max_dług-1` znaków, albo napotkany został znak końca pliku. Znak `delim` jest wczytywany ze strumienia, ale nie jest wpisywany do bufora.
`read( char* bufor, int ilość_bajtów )`	wczytuje ciąg bajtów do bufora
`istream& seekg( streampos pos )` `istream& seekg( streamoff off, ios::seek_dir dir )`	ustawia pozycję czytania ustawia pozycję czytania względem odniesienia
`istream& ignore( int dł_skok = 1, int delim = EOF );`	wczytuje ze strumienia i nigdzie nie zapisuje *dł_skok* znaków. Kończy działanie, jeśli natrafi na znak *delim* albo na koniec pliku.

	Jeśli *delim* = EOF (domyślne ustawienie), wtedy dochodzi do końca pliku. Znak delim jest pobierany ze strumienia.
int gcount() const;	zwraca liczbę wczytanych symboli dla ostatniej niesformatowanej operacji wejścia
int peek();	zwraca bieżący znak w strumieniu nie przesuwając wskaźnika położenia
void eatwhite();	przeskakuje znaki puste w strumieniu
istream& putback(char *ch*);	zwraca znak do strumienia, przy czym musi to być znak, który uprzednio został wczytany

W klasie tej zdefiniowany jest operator:

Operator klasy istream	
>>	operator pobrania/odczytu danych ze strumienia tekstowego

Klasa ostream, oprócz metod odziedziczonych z klasy ios, zawiera metody służące zapisywania danych wspólne dla wszystkich strumieni wyjściowych. Metody te pozwalają zapisywać dane w postaci binarnej i tekstowej. Definicja klasy ostream zawarta jest w pliku iostream.h. Do najważniejszych metod tej klasy należą:

Metody klasy ostream	
put(char& *znak*)	wysyła jeden znak do strumienia
write(char* *bufor*, int *ilość_bajtów*)	wysyła ciąg bajtów z bufora do strumienia
ostream& seekp(streampos *pos*) ostream& seekp(streamoff *off*, ios::seek_dir *dir*)	ustawia położenie pisania ustawia względne położenie pisania

W klasie ostream zdefiniowany jest operator

Operator klasy ostream	
<<	operator wysłania/zapisu danych do strumienia tekstowego

oraz manipulatory:

Manipulatory klasy ostream	
endl	wstawia znak nowej linii i opróżnia bufory
ends	wstawia znak null kończący ciąg znaków
flush	opróżnia bufory strumienia

W C++ można definiować własne manipulatory formatujące. Wystarczy zdefiniować funkcję, której argumentem będzie strumień przekazany przez referencję, i która zwracać będzie referencję do strumienia. Poniżej pokazany jest przykład manipulatora gwiazdki, który zastosowany do strumienia klasy ostream wypisuje do niego 4 gwiazdki.

```
ostream &gwiazdki(ostream &wy)
{
        ostream << "****";
        return wy;
}
cout << "Tu mamy 4 gwiazdki:" << gwiazdki;
```

Operatory wstawiania i czytania ze strumienia (pobierania i umieszczania)

Operatory wstawiania i czytania ze strumienia zaimplementowane w C++ potrafią radzić sobie ze wszystkimi standardowymi typami danych. Pozwalają przy tym użytkownikowi elastycznie formatować dane za pomocą funkcjami składowych klasy `ios` i manipulatorów (zdefiniowanymi w `iomanip.h`).

Przykład:

kod źródłowy

```
float f = 12.345

cout << f << "\n";
cout.setf(ios::scientific);
cout << f << "\n";
cout.setf(ios::uppercase);
cout << f << "\n";
```

wynik

```
12.345
1.2345e+01
1.2345E+01
```

Przykład: wczytywanie danych z klawiatury i wydruk na ekranie

```
// podejście proceduralne
# include <stdio.h>
void main( void )
 {
  char znak;
  int x;
  long y;
  double z;
  char tekst[ 20 ];
  scanf( "%c", &znak );
  scanf( "%d", &x );
  scanf( "%ld", &y );
  scanf( "%lf", &z );
  scanf( "%19s", tekst );

  printf( "znak   = %c \n" , znak );
  printf( "int       = %d \n" , x );
  printf( "long    = %d \n" , y );
  printf( "double = %f \n" , z );
  printf( "tekst    = %s \n" , tekst );
}
```

```
// podejście obiektowe
# include <iostream.h>
void main( void )
{
 char znak;
 int x;
 long y;
 double z;
 char tekst[ 20 ];
 cin >> znak;   // cin.get(znak);
 cin >> x;
 cin >> y;
 cin >> z;
 cin >> tekst;  //cin.getline(tekst,19)

 cout << "znak =" << znak << "\n";
 cout << "int     =" << x << "\n";
 cout << "long  =" << y << "\n";
 cout << "double = " << z << "\n";
 cout << "tekst = " << tekst << "\n";
}
```