

# Kurs języka C++

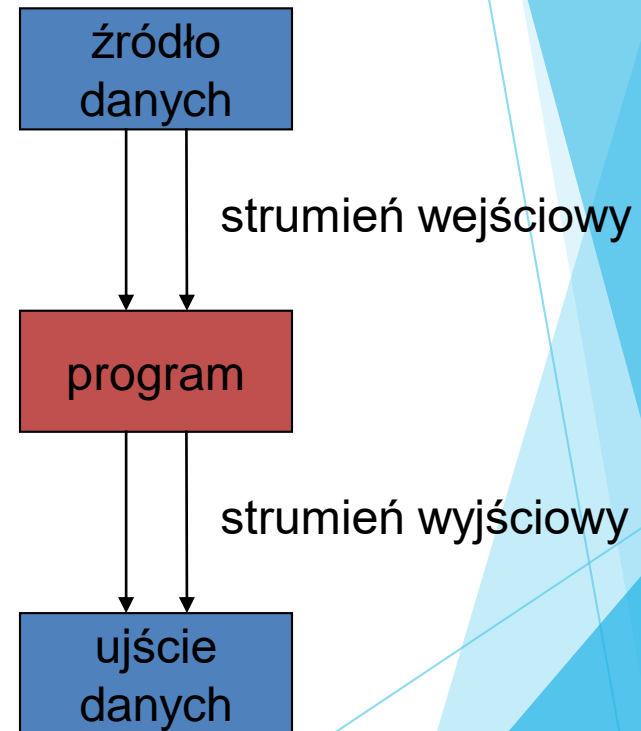
## 11. Strumienie

# SPIS TREŚCI

- Pojęcie strumienia
- Strumienie w bibliotece standardowej
- Operatory strumieniowe  $\gg$  i  $\ll$
- Hierarchia klas strumieni
- Sterowanie formatem
- Manipulatory
- Nieformatowane operacje we/wy
- Błędy w strumieniach
- Strumienie związane z plikami
- Strumienie związane z łańcuchami
- Synchronizacja strumieni

# STRUMIENIE

- Strumień to obiekt kontrolujący przepływ danych.
- Strumień wejściowy transportuje dane do programu.
- Strumień wyjściowy transportuje dane poza program.
- Strumienie dzielimy na:
  - wejściowe i wyjściowe,
  - binarne i tekstowe.



# OBIEKTY STRUMIENI W BIBLIOTECE STANDARDOWEJ

- Klasy zdefiniowane w bibliotece `<iostream>` są szablonami.
- Klasa `istream` to strumień wejściowy będący instancją szablonu klasy `basic_istream<char>`.
- Klasa `ostream` to strumień wyjściowy będący instancją szablonu klasy `basic_ostream<char>`.

# STRUMIENIE W BIBLIOTECE STANDARDOWEJ

- Biblioteka ze strumieniami we/wy ogólnego przeznaczenia to `<iostream>`.
- Biblioteka ze strumieniami we/wy przeznaczona do operacji na plikach to `<fstream>`.
- Biblioteka ze strumieniami we/wy przeznaczona do operacji na obiektach klasy `string` to `<sstream>`.
- Strumienie tekstowe zdefiniowane w `<iostream>` (pracujące na danych typu `char`) związane ze standardowym we/wy to:
  - `cin` – standardowe wejście (zwykle klawiatura),
  - `cout` – standardowe wyjście (zwykle ekran),
  - `clog` – standardowe wyjście dla błędów (zwykle ekran),
  - `cerr` – niebuforowane wyjście dla błędów,
- `wcin`, `wcout`, `wclog`, `wcerr` – strumienie analogiczne do powyższych, ale pracujące na danych typu `wchar_t`.

## OPERATORY >> I << (WYJMOWANIA ZE I WSTAWIANIA DO STRUMIENIA)

- Dla strumieni wejściowych pracujących w trybie tekstowym został zdefiniowany operator >> wyjmowania danych ze strumienia.
- Dla strumieni wyjściowych pracujących w trybie tekstowym został zdefiniowany operator << wstawiania danych do strumienia.
- Operatory >> i << zawsze zwracają referencję do strumieni na których pracują, dlatego operatory te można łączyć kaskadowo przy czytaniu lub pisaniu.
- Operatory >> i << automatycznie dokonują konwersji z danych tekstowych na binarne i na odwrót.
- Należy pamiętać o priorytecie operatora >> i << gdy używa się wyrażeń. Przykład:  

```
cerr << (a*x+b) << endl;
```

# OPERATOR STRUMIENIOWY << DO PISANIA

- `int i = 7;`  
`std::cout << i << endl;`  
`// wyjście: 7`
- `std::string s = "Abecadło";`  
`std::cout << s << endl;`  
`// wyjście: Abecadło`
- `std::bitset<10> flags(7);`  
`std::cout << flags << endl;`  
`// wyjście: 0000000111`
- `std::complex<float> c(3.1, 2.7);`  
`std::cout << c << endl;`  
`// wyjście: (3.1, 2.7)`

# OPERATOR STRUMIENIOWY >>

## DO CZYTANIA

- `int i = 0;`  
`std::cin >> i;`  
`// wejście: odczytanie wartości int`
- `std::string s;`  
`std::cin >> s;`  
`// wejście: odczytanie napisu`  
`string`
- `double d = 0.0;`  
`std::complex<double> c;`  
`std::cin >> d >> c;`  
`// wejście: sekwencyjne odczytanie`  
`// liczby rzeczywistej i zespolonej`

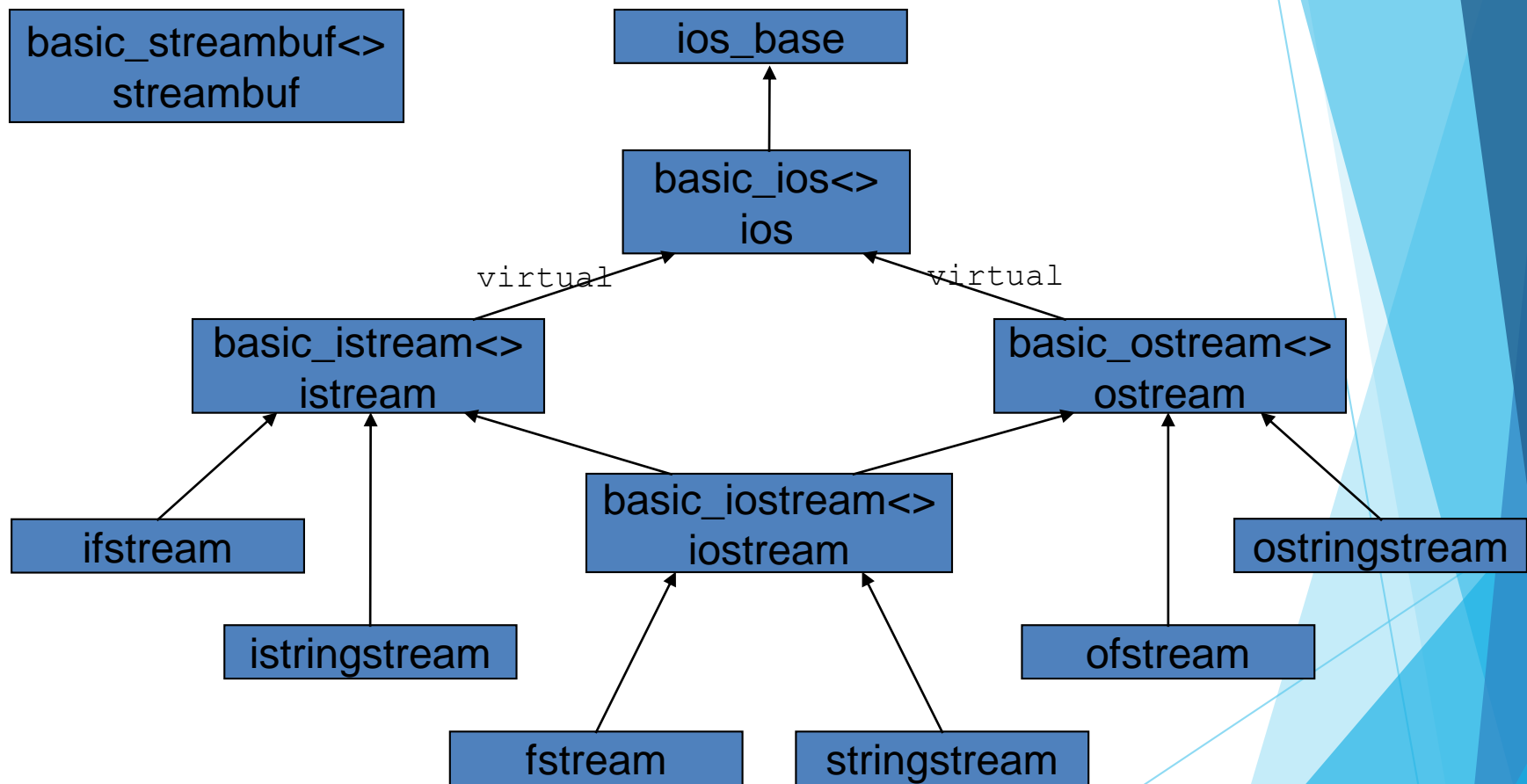


# OPERATORY STRUMIENIOWE >> | << ZDEFINIOWANE PRZEZ UŻYTKOWNIKA

- Dla typów zdefiniowanych przez użytkownika można zdefiniować własne operatory wstawiania do i wyjmowania ze strumienia:  

```
class Typ {...};  
// operator wyjmowania ze strumienia  
istream& operator >> (istream &os, Typ &x);  
// operator wstawiania do strumienia  
ostream& operator << (ostream &os, const Typ &x);
```
- Należy pamiętać o zwróceniu referencji do strumienia, na którym się pracuje.
- Operatorów wstawiania do i wyjmowania ze strumienia nie dziedziczy się.
- Operatory wstawiania do i wyjmowania ze strumienia nie mogą być wirtualne.

# HIERARCHIA KLAS STRUMIENIOWYCH REALIZUJĄCYCH OPERACJE WE/WY



# STEROWANIE FORMATEM

- Podczas operacji na strumieniu używane są pewne domniemania dotyczące formatu danych – domniemania te zapisane są w strumieniu we fladze stanu formatowania.
- Klasa w której umieszczono flagę stanu formatowania to `ios_base` – typ takiej flagi to `fmtflags`.

# STEROWANIE FORMATEM

- Flagi odpowiadające za sposób formatowania:
  - ignorowanie białych znaków `skipws`;
  - justowanie `left`, `right`, `internal` (maska `adjustfield`);
  - pełne nazwy boolowskie `boolalpha`;
  - reprezentacja liczb całkowitych `dec`, `oct`, `hex` (maska `basefield`);
  - uwidocznienie podstawy reprezentacji `showbase`;
  - kropka dziesiętna `showpoint`;
  - duże litery w liczbach `uppercase`;
  - znak + w liczbach dodatnich `showpos`;
  - reprezentacja liczb rzeczywistych `scientific`, `fixed` (maska `floatfield`);
  - buforowanie `unibuf`.

# STEROWANIE FORMATEM

- Zmianę reguł formatowania dokonuje się następującymi metodami:

```
fmtflags flags () const;
fmtflags flags (fmtflags fls);
fmtflags setf (fmtflags fl);
fmtflags setf (fmtflags fl, fmtflags mask);
fmtflags unsetf (fmtflags fl);
streamsize width () const;
streamsize width (streamsize w);
streamsize precision () const;
streamsize precision (streamsize p);
```

- Uwaga – metoda `width(w)` ma działanie jednorazowe.

- Przykłady:

```
fmtflags f = cout.flags();
cout.unsetf(ios::boolalpha);
cout.setf(ios::showpos|ios::showpoint);
cout.setf(ios::hex,ios::basefield);
...
cout.flags(f);
```

# MANIPULATORY

- Manipulatory, zdefiniowane w pliku zagłówkowym `<iomanip>` to specjalne obiekty, które można umieścić w strumieniu za pomocą operatorów `>>` albo `<<`, które powodują zmianę reguł formatowania lub inne efekty uboczne na strumieniu.
- Standardowe manipulatory bezargumentowe:  
endl, ends,  
hex, dec, oct,  
fixed, scientific,  
left, right, internal,  
skipws, noskipws, ws,  
boolalpha, noboolalpha,  
showpoint, noshowpoint,  
showpos, nowhowpos,  
showbase, noshowbase,  
uppercase, nouppercase,  
unitbuf, nounitbuf,  
flush.

# MANIPULATORY

- Standardowe manipulatory sparametryzowane:

```
setw(int),  
setprecision(int),  
setfill(char), setfill(wchar_t),  
setiosflags(fmtflags),  
resetiosflags(fmtflags).
```

- Przykłady:

```
cout << setiosflags(ios_base::boolalpha);
```

# MANIPULATORY

- Własne manipulatory bezparametrowe definiuje się w postaci funkcji.

- Przykład:

```
inline ostream& tab (ostream &os)
{
    return os << "\t";
}
```

...

```
cout << "x:" << tab << tab << x << endl;
```



# MANIPULATORY

- Własne manipulatory sparametryzowane definiuje się w postaci klas:
  - klasa ta musi posiadać konstruktor tworzący chwilowy obiekt manipulatora,
  - oraz zaprzyjaźniony operator strumieniowy >> albo << używający obiektu naszej klasy.

- Przykład:

```
struct liczba {
    int wart, podst;
    friend ostream& operator <<
        (ostream &os, const liczba &licz)
    { /* ... */ }
public:
    liczba (int wart, int podst)
        : wart(w), podst(p)
    { /* ... */ }
};

...
cout << "y = " << liczba(y, 7) << endl;
```

# NIEFORMATOWANE OPERACJE WE/WY

- Formatowane operacje we/wy przeprowadzane są za pośrednictwem operatorów `>>` i `<<`, które przekształcają dane z postaci tekstowej na binarną (czytanie) albo z postaci binarnej na tekstową (pisanie).
- Są jednak sytuacje, gdy formatowanie nie jest nam potrzebne...
- Nieformatowane operacje we/wy są umieszczone w klasach `istream` i `ostream` (oraz uzupełnione kilkoma funkcjami składowymi w klasie `iostream`).

# NIEFORMATOWANE CZYTANIE (WYJMOWANIE ZE STRUMIENIA)

- Funkcje składowe wyjmujące po jednym znaku:  
`istream& get(char&);` – w przypadku końca strumienia strumień przechodzi w stan błędu  
`int get();` – w przypadku końca strumienia funkcja zwraca wartość EOF (o wartości -1).

- Przykłady użycia:

```
char a, b, c;  
cin.get(a).get(b).get(c);
```

...

```
char z;  
while (cin.get(z)) {...}
```

...

```
char z;  
while ((z=cin.get()) != EOF) {...}
```

# NIEFORMATOWANE CZYTANIE (WYJMOWANIE ZE STRUMIENIA)

- Funkcje składowe wyjmujące wiele znaków:  
`istream& get(char *gdzie, streamsize ile, char ogr='\n');` – gdy w trakcie czytania znaków zostanie napotkany ogranicznik, to czytanie będzie przerwane (znak ogranicznika pozostanie w strumieniu)  
`istream& getline(char *gdzie, streamsize ile, char ogr='\n');` – gdy w trakcie czytania znaków zostanie napotkany ogranicznik, to czytanie będzie przerwane (znak ogranicznika zostanie usunięty ze strumienia)
- Po zakończeniu czytania powyższe funkcje dopiszą na końcu danych bajt zerowy `'\0'` poprawnie kończący C-string (wczytanych zostanie więc maksymalnie `ile-1` znaków).

# NIEFORMATOWANE CZYTANIE (WYJMOWANIE ZE STRUMIENIA)

- Funkcje zewnętrzna wyjmująca wiele znaków to:  
`istream& std::getline(istream &we, string &wynik, char ogr='\n');` – funkcja ta nie ma limitu na liczbę wczytywanych znaków (znak ogranicznika zostanie usunięty ze strumienia).

- Przykład użycia:

```
string s;  
while (getline(cin, s)) {...}
```

# NIEFORMATOWANE CZYTANIE (WYJMOWANIE ZE STRUMIENIA)

- Do binarnego czytania danych służą funkcje składowe:
  - `istream & read(char *gdzie, streamsize ile)` – funkcja wczytuje blok znaków (gdy brakuje danych strumień przechodzi w stan błędu)
  - `streamsize readsome (char *gdzie, streamsize ile)` – funkcja wczytuje blok znaków (gdy brakuje danych strumień nie zmienia stanu)
  - `istream & ignore(streamsize ile=1, int ogr=EOF)` – funkcja pomija blok znaków
  - `streamsize gcount()` – funkcja mówi, ile znaków zostało wyciągniętych za pomocą ostatniej operacji czytania nieformatowanego
  - `int peek ()` – funkcja pozwala podglądnać następny znak w strumieniu
  - `istream & putback(char)` – funkcja zwraca do strumienia jeden znak
  - `istream & unget()` – funkcja zwraca do strumienia ostatnio przeczytany znak

# NIEFORMATOWANE PISANIE (WSTAWIANIE DO STRUMIENIA)

- Wstawianie do strumienia realizuje się za pomocą dwóch funkcji składowych:

`ostream & put(char)` – funkcja ta wstawia do strumienia jeden znak

`ostream & write(const char *skąd, streambuf ile)` – funkcja ta wstawia do strumienia wiele znaków

- Przykłady użycia:

```
char napis[] = "jakiś napis";
```

```
for (int i=0; napis[i]; ++i)
```

```
    cout.put(i?' ':'-').put(napis[i]);
```

```
...
```

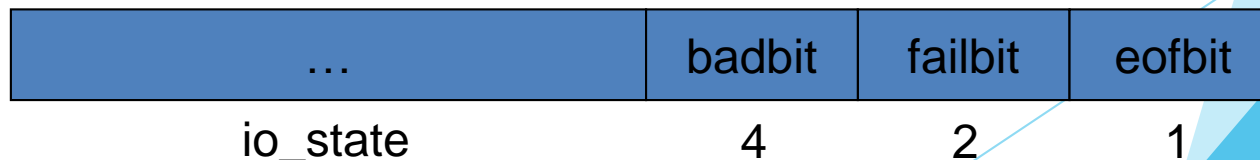
```
ofstream plik = ...;
```

```
double e = 2.718281828459;
```

```
plik.write(reinterpret_cast<char*>(&e), sizeof(e));
```

# BŁĘDY W STRUMIENIACH

- W klasie `ios` mamy zdefiniowane narzędzia do kontrolowania poprawności operacji na strumieniach i sprawdzania stanu strumienia.
- W każdym strumieniu znajduje się flaga stanu strumienia (zdefiniowana w klasie `ios_base`).
- Flaga stanu strumienia składa się z trzech bitów:
  - `eofbit` – flaga ta jest ustawiana, gdy osiągnięto koniec strumienia
  - `failbit` – flaga ta jest ustawiana, gdy nie powiodła się operacja `we/wy`
  - `badbit` – flaga ta jest ustawiana, gdy nastąpiło poważne uszkodzenie strumienia





# BŁĘDY W STRUMIENIACH

- Funkcje do pracy z flagami błędów w strumieniach:
  - `bool good()` – zwraca `true`, gdy żadna flaga błędu nie jest ustawiona
  - `bool eof()` – zwraca `true`, gdy został osiągnięty koniec strumienia i jest ustawiona flaga `ios::eofbit`
  - `bool fail()` – zwraca `true`, gdy strumień jest w stanie błędu, czyli jest ustawiona flaga `ios::failbit` lub `ios::badbit`
  - `bool bad()` – zwraca `true`, gdy strumień jest poważnie uszkodzony i jest ustawiona flaga `ios::badbit`

# BŁĘDY W STRUMIENIACH

- W obsłudze błędów w strumieniach przydatne są też operatory zdefiniowane w klasie `ios`:

`operator bool() const` – operator ten zwraca wartość niezerową, gdy `!fail()`

`bool operator ! () const` – operator ten zachowuje się tak jak funkcja `fail()`

- Przykłady użycia:

```
if (! cin) cout << "błąd" << endl;
```

...

```
if (cin) cout << "ok" << endl;
```

# BŁĘDY W STRUMIENIACH

- Istnieje kilka funkcji składowych do ustawiania i kasowania flag błędu:
  - `io_state rdstate()` – funkcja zwraca flagę błędu strumienia
  - `void clear (io_state = ios::goodbit)` – funkcja zastępuje flagę błędu strumienia inną wartością
  - `void setstate (io_state)` – funkcja dopisuje flagę błędu do flagi strumienia
- Przykłady użycia:

```
if (plik.rdstate() & ios::failbit)
    cout << "failbit jest ustawiona" << endl;
...
cin.clear(ios::eofbit);
...
cin.setstate(ios::failbit);
```

# BŁĘDY W STRUMIENIACH

- Strumień można zmusić do zgłaszania wyjątków w pewnych sytuacjach za pomocą funkcji `exceptions()`:  
`void exceptions (io_state)`
- Argument funkcji `exceptions()` określa, flagi dla których ma być zgłoszony wyjątek `ios_base::failure`.
- Gdy chcemy sprawdzić na jakie flagi strumień będzie reagował wyjątkiem, należy użyć innej funkcji `exceptions()`:  
`io_state exceptions (void) const`
- Przykład użycia:  
`plik.exceptions(ios::failbit | ios::badbit);`

# STRUMIENIE ZWIĄZANE Z PLIKAMI

- Typy strumieni plikowych: `ifstream`, `ofstream`, `fstream`.
- Strumienie te są zadeklarowane w pliku nagłówkowym `<fstream>`.
- Strumień plikowy należy na początku otworzyć metodą `open()` a na końcu zamknąć metodą `close()`.
- Strumień plikowy można otworzyć w konstruktorze.  
Przykład:  

```
ifstream plik("dane.txt");
```

# STRUMIENIE ZWIĄZANE Z PLIKAMI

- Przy otwieraniu strumienia należy podać tryb otwarcia.

- Możliwe tryby otwarcia strumienia to:

`in` – do czytania

`out` – do pisania

`ate` – ustawienie głowicy na końcu pliku

`app` – do dopisywania

`trunc` – skasowanie starej treści

`binary` – tryb binarny (domyślnie jest tryb tekstowy)

- Przykład:

```
string nazwa = "dane.txt";
```

```
ofstream plik(nazwa.c_str(), ios::app|ios::bin);
```

# STRUMIENIE ZWIĄZANE Z ŁAŃCUCHAMI

- Typy strumieni łańcuchowych: `istream`, `ostream`, `stringstream`.
- Strumienie te są zadeklarowane w pliku nagłówkowym `<sstream>`.
- Strumienie łańcuchowe przechowują jako składową obiekt klasy `string`.

# STRUMIENIE ZWIĄZANE Z ŁAŃCUCHAMI

- Strumień łańcuchowy do pisania `ostream` gromadzi dane w łańcuchu znakowym.
- Można go zainicjalizować jakąś wartością początkową i trybem. Przykłady:  

```
ostream wy1;  
ostream wy2("jakiś tekst",  
ios::app);
```
- Ze strumienia łańcuchowego do pisania `ostream` można wyciągnąć bieżącą zawartość łańcucha za pomocą funkcji `str()`:  

```
string str() const
```



# STRUMIENIE ZWIĄZANE Z ŁAŃCUCHAMI

- Strumień łańcuchowy do czytania `istream` udostępnia dane z łańcucha znakowego.
- Można go zainicjalizować jakąś wartością początkową. Przykłady:  

```
istream we1;  
istream we2("jakiś tekst");
```
- Strumień łańcuchowy do czytania `istream` można reinicjalizować nowym łańcuchem za pomocą funkcji `str()`:  

```
void str (const string &)
```

# SYNCHRONIZACJA STRUMIENI

- Synchroniczną pracę strumieni uzyskuje się dzięki wiązaniu strumieni za pomocą funkcji składowej `tie()` zdefiniowanej w `ios_base`:  

```
ostream* ios_base::tie (ostream*)  
ostream* ios_base::tie ()
```
- Można wiązać dowolny strumień z jakimś jednym strumieniem wyjściowym.
- Efektem wiązania jest opróżnienie bufora związanego strumienia wyjściowego przed operacją na danym strumieniu.
- Aby zerwać dotychczasowe powiązanie należy na strumieniu wywołać metodę `tie(nullptr)`.
- Strumienie standardowe `cin` i `cerr` są powiązane ze strumieniem `cout`.

# PROBLEMY Z BIBLIOTEKĄ <CSTDIO>

- Wady funkcji `printf()` i `scanf()` z biblioteki `<cstdio>`:
  - zmienna liczba argumentów (kompilator tego nie kontroluje),
  - mało czytelna semantyka tych funkcji (przynajmniej na początku),
  - brak eleganckiego sposobu na wczytywanie i wypisywanie obiektów typów zdefiniowanych przez użytkownika,
  - analiza wzorca i zawartych w nim znaczników (rozpoczynających się od znaku procenta) jest wykonywana dopiero w trakcie działania programu.
- Strumienie `cin`, `cout`, `clog` i `cerr` nie mają żadnych powiązań ze strumieniami `stdin`, `stdout` i `stderr` (za wyjątkiem tych samych deskryptorów plików).
- Aby strumienie standardowe z biblioteki `<iostream>` dobrze współdziałały ze strumieniami standardowymi z biblioteki `<cstdio>` należy wywołać funkcję `sync_with_stdio()`:  

```
bool ios_base::sync_with_stdio(bool sync=true)
```

# BUFORY

- Bufor to magazyn na dane, do którego można pisać i z którego można czytać określone wartości sekwencyjnie.
- Bufory są wykorzystywane przez obiekty strumieniowe do transferu danych do przedmiotowych magazynów.
- Bufor znakowy `streambuf` jest zdefiniowany w bibliotece `<streambuf>`.
- Funkcja `basic_streambuf<>* rdbuf ()` pozwala na pobranie adresu obiektu bufora związanego ze strumieniem a ustanowienie nowego bufora w strumieniu jest możliwe za pomocą funkcji `basic_streambuf<>* rdbuf (basic_streambuf<>*)`.
- Wskaźnik na bufor w strumieniu nie może być pusty.