WYDZIAŁ FIZYKI
UNIWERSYTET WARSZAWSKI

Programowanie zaawansowane FM i NI

Wykład 12

Strumienie: plikowe, napisowe i standardowe

Krzysztof Piasecki

Semestr letni roku akad. 2023-24

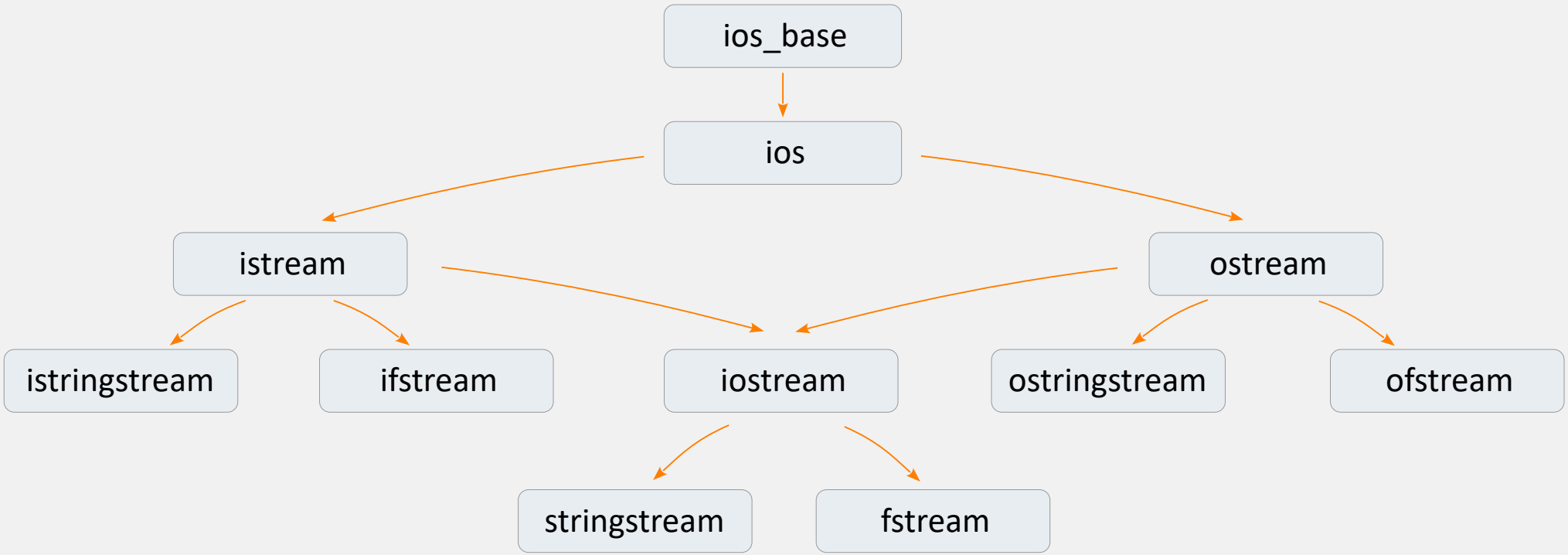

krzysztof.piasecki@fuw.edu.pl

- **Strumienie.** Strumień to obiekt, który zarządza przekazywaniem treści ze źródła do ujścia. Zwykle mieści w sobie bufor, w którym tymczasowo przechowuje treść.

⊙ Główne rodzaje strumieni: **plikowy**, **napisowy** i **standardowy** (np. `cin`, `cout`).

⊙ Strumienie mogą być: **wyjściowe** (do zapisu) , **wejściowe** (do odczytu) lub **we/wy** (do obu funkcji).

⊙ W C++ klasy strumieni ułożone są w hierarchię dziedziczenia. Wiodący trzon tej hierarchii:



- **Klasa `ios_base`:** klasa bazowa dla wszelkich strumieni. Nie tworzy się jej obiektów.
  Zarządza cechami wspólnymi, np. stan trybu formatowania strumienia itd.

⊙ **Stan trybu formatowania**: zestaw flag sygnalizujących włączenie (lub nie) danego trybu.
Tryby ustawia się przez metody: `{strumień}.setf ( .. )` i `.unsetf( .. )`
Typowo używane flagi:

```
ios_base::boolalpha     :   wartości typu bool będą wypisywane jako true / false, a nie 1 / 0.
          left / right  :   w polach treści będą wypisywane od lewej / od prawej
          dec / oct / hex :  liczby będą wypisywane w systemie 10 / 8 / 16 -tkowym.
          fixed         :   liczby będą wypisywane z ilością cyfr po kropce ustaloną przez precision
          scientific    :   liczby będą wypisywane w notacji naukowej, np. 6.63e-34
```

Np. :

```
cout.setf ( ios_base::hex );  cout.unsetf ( ios_base::dec ) ;   cout << 100 ;
```

⊙ **Ilość znaków dla następnego pola**: `{strumień}.width(..)` , np: `cout.width ( 5 )`

⊙ **Precyzja wyświetlania liczby** : `{strumień}.precision(..)` , np: `cout.precision ( 5 );`

```
streamsize myPrecision = cout.precision();
```

⊙ W `ios_base` zdefiniowane są też **flagi trybu otwarcia** strumienia (**opening mode**) :

```
ios_base::in    :   strumień jest w trybie  czytania
          out   :                           zapisu   (zwykle: wpierw skasowanie starej treści)
          trunc:                            zapisu            (wpierw skasowanie starej treści)
          app   :                           zapisu    z dopisywaniem nowej treści na koniec starej
```

- **Klasa `ios`:** klasa dziedzicząca cechy ios_base i wspólna dla wszystkich strumieni we i/lub wy, operujących na znakach ASCII. Nie tworzymy jej obiektów.

⊙ Ważna własność tej klasy: **stan błędu**. Są 4 metody sprawdzające, czy zachodzi dana cecha tego stanu:

```
bool ios::good ()  :      true ,  gdy wszystko ok.
       ::eof   ()  :      true ,  gdy czytanie napotkało znak EOF  (koniec treści do wczytania)
       ::fail  ()  :      true ,  gdy zaszedł błąd logiczny we/wy.
       ::bad   ()  :      true ,  gdy próba we/wy napotkała „popsucie”
```

Przykłady `fail`: ▷ gdy `cin` ma wczytać liczbę do zmiennej typu `int` , ale natrafił na literę.
▷ gdy strumień chce otworzyć plik do odczytu, ale dostęp jest wzbroniony.

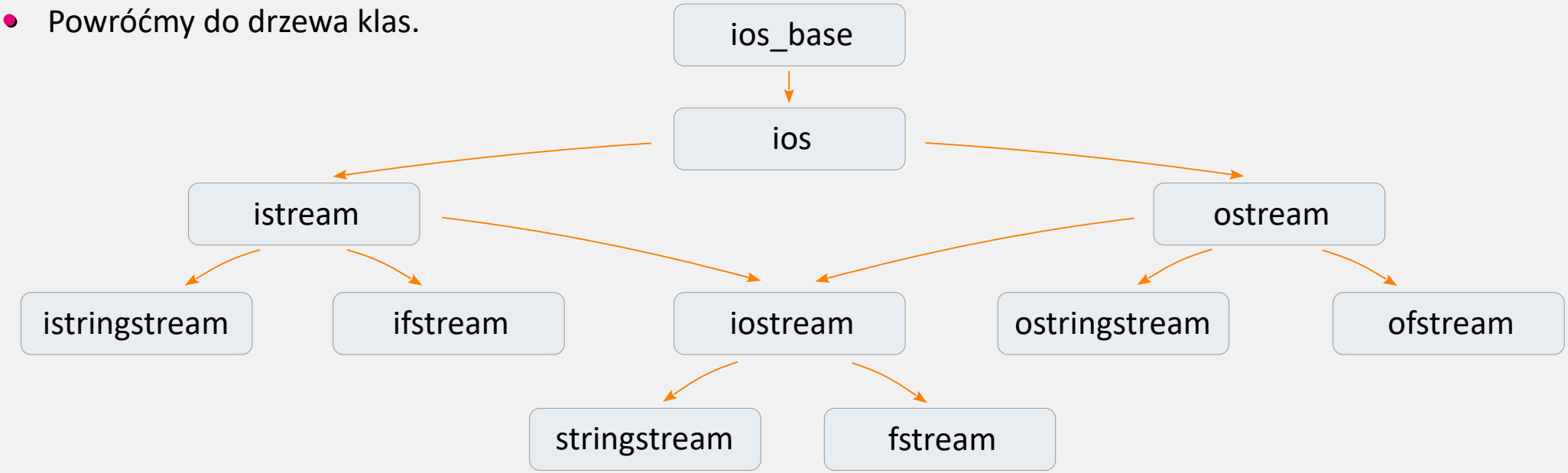
Przykład `bad` : ▷ gdy strumień próbuje wczytać kolejny znak z pliku, ale plik się popsuł na dysku.

⊙ Metoda `clear()` : czyści status błędu do stanu `good` .

[Link]

```
4  int main () {
5    cout << "Wpisz słowo i naciśnij Enter. \n"
6         << "Następnie wciśnij Ctrl-D, aby wygenerować EOF: \n";
7    char c;
8    do {
9      cin >> c;
10     cout << c << ' ' << cin.good() << ' ' << cin.fail()
11          << ' ' << cin.eof () << ' ' << cin.bad () << endl;
12   }
13   while ( ! cin.eof() );
14 }
```

- Powróćmy do drzewa klas.



- Załączenie biblioteki **<iostream>** powoduje utworzenie kilku strumieni standardowych, m.in.

  cin : strumień do odczytu , klasy `istream`. Odczyt „typowo” z klawiatury.
  cout: strumień do wypisania „zwykłych treści” , klasy `ostream`. Zapis „typowo idzie na ekran”
  cerr: strumień do wypisania błędów , klasy `ostream`. Zapis „typowo idzie na ekran”

- Załączenie biblioteki **<fstream>** umożliwia tworzenie strumieni **plikowych**.
  <sstream> **napisowych**.

- Dany strumień może być typu: **do odczytu** (`istream / ifstream / istringstream`)
  do zapisu (`ostream / ofstream / ostringstream`)
  lub **do obu funkcji na raz**: (`iostream / fstream / stringstream`)

- **Strumienie plikowe – podstawowe działania**. W kodzie trzeba załączyć bibliotekę `<fstream>`.

⊙ **Utworzenie** strumienia do **odczytu** z podpięciem pod plik:

```
ifstream myStr ("nazwa.txt") ;
```

▻ Można to rozbić na dwa etapy:

```
ifstream myStr ;
myStr.open ("nazwa.txt") ;
```

⊙ **Utworzenie** strumienia do **zapisu** z podpięciem pod plik:

```
ofstream myStr ("nazwa.txt") ;
```

▻ Jeśli chcemy wskazać **tryb otwarcia**, to np.:

```
ofstream myStr ("nazwa.txt",ios::app);
```

⊙ Zawsze warto sprawdzić, **czy** otwarcie się **powiodło**.
Służy do tego metoda `is_open()`,
zwracająca `true`/`false`:

```
if ( ! myStr.is_open () ) {
  cerr << "Error opening file. \n";
  exit (0);
}
```

⊙ **Odpięcie** strumienia od pliku („**zamkniecie pliku**”): `close()`

```
myStr.close() ;
```

▻ Można ten strumień podpiąć ponownie pod jakiś (np. inny) plik.

▻ Jest „w dobrym tonie” zamknąć plik przed końcem kodu.
Jeśli zapomnimy, to program na końcu zrobi to za nas (dokładniej: podczas destrukcji strumienia) .

- **Jak czytać ze strumienia i wpisywać do niego?** Robimy to, działając na `cin` i `cout`, od pierwszych zajęć ☺

Operatory `>>` i `<<`. Na każdym strumieniu klasy `istream` i dziedziczącej - można wykonać:

```
i_strumien  >>  zmienna_liczbowa ;
i_strumien  >>  C-string ;
i_strumien  >>  string ;        (po załączeniu biblioteki <string>)
```

Podobnie, na każdym strumieniu klasy `ostream` i dziedziczącej - można wykonać:

```
o_strumien  <<  zmienna_liczbowa ;
o_strumien  <<  C-string ;
o_strumien  <<  string ;        (po załączeniu biblioteki <string>)
```

[Link]

- Możemy też zdefiniować `operator<<` i `operator>>` dla obiektów naszej klasy ☺

W tym przykładzie tworzymy strukturę `Complex` i jej obiekt `Z1`. Następnie przeciążamy `operator<<` i `operator>>`, aby w `main` móc napisać:

```
cin  >> Z1 ;
cout << Z1 ;
```

⊙ *Uwaga*: jeśli pola są prywatne, to trzeba dopuścić do nich operator (poprzez `friend`)

```
5  struct Complex {  double re, im;  };
6
7  ostream& operator<< (ostream& str, Complex& Z)      {
8    return str << '[' << Z.re << ',' << Z.im <<"] ";  }
9
10 istream& operator>> (istream& str, Complex& Z)      {
11   return str >> Z.re >> Z.im ;                      }
12
13 int main ()                                        {
14   Complex Z1;
15   cout << "Podaj dwie liczby: ";
16   cin >> Z1;
17   cout << Z1 << endl;                              }
```

- **Czytanie seryjne z pliku danych**

  Gdy dane w pliku mają ustaloną (niedużą) liczbę kolumn, to polecenie czytania każdego wiersza:

  `strumien >> zm1 >> zm2 >> … ;`

  można wstawić w warunek pętli, np. pętli `while`.

⊙ Pętla ta będzie trwać tak długo, jak kolejne wiersze w pliku będą istnieć (więc dadzą się odczytać). Gdy zabraknie danych, w warunku pojawi się `false`.

(trick polega na sposobie konwersji `istream` → `bool`)

```
5  int main () {
6   ifstream myStr ("dane.txt");
7
8   if ( ! myStr.is_open() ) {
9     cerr << "<E> Error opening file.\n";
10    return -1;
11  }
12
13  double x, y;
14  while ( myStr >> x >> y )  {
15    cout << x << '\t' << y << endl;
16  }
17
18  myStr.close ();
19 }
```

[Link]

```
5  int main () {
6   ifstream myStr ("Akerman.txt");
7   string linia;
8
9   while ( getline (myStr , linia) ) {
10    cout << linia << endl;
11  }
12
13  myStr.close ();
14 }
```

[Link]

- Gdy chcemy przeczytać plik tekstowy w paczkach po całej jednej linii, przydaje się polecenie:

  `getline (strumień, C-string);` albo
  `getline (strumień, string );`

  Zwraca ono nasz strumień.

⊙ Tu też można zastosować trick ze wstawieniem `getline` do warunku pętli, np. `while`.

- **Zapis do pliku: przykład**

[Link]

Jak wspominaliśmy, do pliku wpisujemy tekst poprzez:

```
strumien << zmienna / napis ;
```

⊙ W tym miejscu przykładu testujemy wpisywanie różnych wariantów.

⊙ Tu przykład, jak w pliku ułożyć tabelę z dwoma kolumnami jakichś danych

⊙ Komenda **system** z biblioteki <cstdlib> pozwala wywołać polecenie systemowe. Tu polecenie cat wypisze zawartość pliku.

```
6  int main () {
7    ofstream myStr ("output.txt");
8
9    if ( ! myStr.is_open() ) {
10     cout << "<E> error opening file.\n";
11     return -1;
12   }
13
14   double x = 12.34;
15   string jakis_string = "jakis_string" ;
16
17   myStr << "Tu kawalek tekstu. Stala liczbowa: "
18         << 9 << ", zmienna double: "
19         << x << " i string: "
20         << jakis_string << endl;
21
22   for (double x = 1; x <= 5; x++)  {
23     myStr << x << '\t' << -x*5. << endl;
24   }
25   myStr.close ();
26
27   cout << "Sprawdzimy zapisany plik: " << endl;
28   system ("cat output.txt");
29 }
```

- **Strumienie napisowe.**
  - to obiekty klas **`istringstream`** (**do odczytu**) i **`ostringstream`** (**do zapisu**), w bibliotece **`<sstream>`**. Posiadają bufor do przechowywania napisu.

⊙ Pełnią bardzo praktyczne role ☺ :

[Link]

- ▻ Poprzez strumień zapisowy (`ostringstream`) można np. serializować nazwy pliku.
- ▻ Do strumienia zapisowego można dodawać tekst w miarę algorytmu, a decyzję o jego użyciu (wypis tekstu na ekran lub do pliku) podjąć później.
- ▻ W połączeniu z czytaniem z pliku: Gdy nie wiemy, ile słów ma linia w pliku, to wczytujemy ją do strumienia do odczytu (typu `istringstream`), a z niego wczytujemy do zmiennych.

⊙ W tym przykładzie tworzymy strumień do odczytu, od razu podając mu tekst rozdzielony spacjami.

Stosując `operator>>`, poszczególne słowa/liczby przypisujemy do zmiennych.

```
5  int main () {
6   string some_string;
7   int    some_int;
8   float  some_float;
9
10  istringstream myStr ("some_text 9 5.56") ;
11
12  myStr >> some_string >> some_int >> some_float;
13
14  if ( myStr.fail () )  {
15   cout << "<E> Error reading from stream.\n";
16   return -1 ;
17  }
18
19  cout << "I have read: \n"
20       << some_string << endl
21       << some_int    << endl
22       << some_float  << endl;
23 }
```

- **Strumień napisowy do zapisu**

Przykład: **serializacja** nazwy pliku, do zapisu danych z eksperymentu.

Idea: nasza aparatura wykonuje pomiary próbki, przy temperaturze zwiększającej się o pewien krok.

Ale każdy punkt temperaturowy mierzony jest 12 razy (dla minimalizacji niepewności).

W kroku pętli, dzięki strumieniowi zapisowemu, kod **formuje nazwę**. Nazwę sklejamy z prefiksu, temperatury, numeru pomiaru i z sufiksu.

- Metodą `ostringstream::str()` zwracamy gotowy `string`.

[Link]

```
6  int main () {
7    string prefix = "pomiary_" , suffix = ".dat";
8
9    cout << "Nazwy plikow do zapisu kolejnych danych: ";
10
11   for (double Temp = 44.5 ; Temp <= 44.6; Temp += 0.05)
12   {
13     for (int NrPomiaru = 1; NrPomiaru <= 12; NrPomiaru++)
14     {
15       ostringstream myStream;
16
17       myStream << prefix << fixed
18                << "Temp" << setprecision(2) << Temp
19                << "_nr"  << setw(2) << setfill ('0')
20                << NrPomiaru << suffix ;
21
22       string FileName = myStream.str() ;
23       cout << FileName << endl;
24     }
25   }
26 }
```

- **Strumień napisowy do odczytu**

Przykład: **rozczytywanie** linii do zmiennych,
gdy nie znamy ilości słów.

⊙ Użytkownik wpisuje linię o dowolnej ilości słów.
`getline` wstawia linię ze strumienia `cin` do `string`'u.
Teraz: tworzymy obiekt `istringstream` z tą linią.
– i czytamy z niego do `string`'u tyle słów, ile tam jest.

Gdy czytanie dojdzie na koniec, to warunek
w nagłówku `while` zamienia się na `false`.

[Link]

```
5  int main () {
6    string liniatekstu, slowo;
7    cout << "Napisz kilka slow:" << endl;
8
9    getline (cin, liniatekstu);
10   istringstream iss (liniatekstu);
11
12   while (iss >> slowo)
13     cout << '[' << slowo << "] ";
14 }
```

[Link]

```
6  int main () {
7    string cala_linia , slowo ;
8    ifstream myStr ("main.cpp");
9
10   while ( getline (myStr, cala_linia ) ) {
11     istringstream iss ( cala_linia );
12
13     while  (iss >> slowo)
14       cout << slowo << '.';
15
16     cout << endl;
17   }
18   myStr.close ();
19 }
```

⊙ Ten przykład demonstruje, jak czytać z pliku:

▷ linie, gdy nie wiemy a'priori, ile ich jest

▷ w każdej linii: jak rozdzielić na słowa,
gdy nie wiemy a'priori, ile linia ma słów.

- **Aktualna pozycja** w strumieniu plikowym i napisowym.

⊙ Każdy strumień przechowuje aktualną pozycję.
W strumieniu **zapisowym** metoda:
tellp poda pozycję, a **seekp** - ją ustawi.

W strumieniu **do odczytu** metoda:
tellg poda pozycję, a **seekg** - ją ustawi.

[Link]

```
10  int main() {
11   ofstream fout ("dane.txt");
12   fout << "|_____________|"; Pos (fout);
13   fout.seekp (2 , ios::beg); Pos (fout); fout << "Ala";
14   fout.seekp (1 , ios::cur); Pos (fout); fout << "ma";
15   fout.seekp (-6, ios::end); Pos (fout); fout << "kota";
16   fout.close();
17
18   ifstream fin ("dane.txt");
19   string linia;
20   getline (fin, linia);
21   cout << linia;
22   fin.close();
23  }
```

⊙ Przypuśćmy, że otwieramy 2 strumienie:
```
ofstream fout ("output.dat");
ifstream fin  ("input.dat" );
```

⊙ Aby ustawić pozycję, musimy wskazać punkt odniesienia. Mamy 3 możliwości:

ios::beg - bazą jest początek strumienia
ios::cur - bazą jest aktualna pozycja
ios::end - bazą jest koniec strumienia.

Np.:
`fout.seekp (3, ios::beg)` - skocz do bajtu nr. 3 od początku
`fin .seekg (2, ios::cur)` - skocz o 2 bajty do przodu
`fin .seekg (-5,ios::end)` - skocz do 5. bajtu przed końcem

⊙ *Uwaga*: próba skoku za koniec lub przed początek spowoduje, że strumień wejdzie w stan 'bad'.
Próby czytania lub zapisu będą ignorowane, aż do wyczyszczenia stanu: `fin.clear() ;`

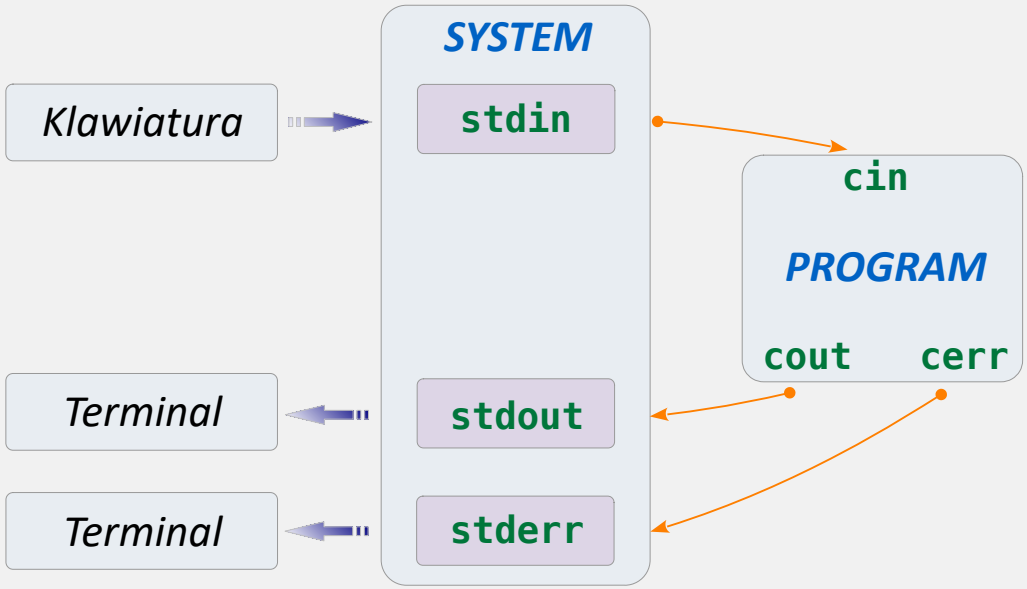
Strumienie standardowe w systemach Unixowych

- Dotychczas mówiliśmy, że:

  `cin` pobiera tekst z klawiatury, a
  `cout` i `cerr` wypisują tekst na ekran.

- Rzeczywistą sytuację ilustruje schemat.
  System posiada tzw. „**strumienie standardowe**” .
  Każdy posiada numer (tzw. „**file descriptor**”).

  Strumień wejścia, **`stdin`** ma nr 0 ,
  Strumień wyjścia, **`stdout`** ma nr 1 ,
  Strumień błędu , **`stderr`** ma nr 2 .



- Po stronie systemu (Linux, MacOs, …) ,
  gdy wywołujemy kod (aplikację) ,
  to można pod **`stdin`**, **`stdout`** i **`stderr`**
  podstawić coś innego, np. pliki lub tekst.

  Np. efekt jak na schemacie dostaniemy,
  gdy w terminalu napiszemy:

```
$ ./a.out <input.txt 1>output.txt
  2>errors.log
```