Języki programowania
Nowoczesne techniki programowania

Wykład 6

Witold Dyrka
witold.dyrka@pwr.wroc.pl

16/11/2012

Prawa autorskie

Slajdy do wykładu powstały
w oparciu o slajdy Bjarne Stroustrupa
do kursu Foundations of Engineering II (C++)
prowadzonego w Texas A&M University

http://www.stroustrup.com/Programming

Materiały

Literatura

- Bjarne Stroustrup. *Programowanie: Teoria i praktyka z wykorzystaniem C++*. Helion (2010)
- Jerzy Grębosz. *Symfonia C++ standard.* Edition 2000 (2008)
- Dowolny podręcznik języka C++ w standardzie ISO 98
- www.cplusplus.com - The C++ Reference Network (j.ang.)

Środowisko programistyczne

- Microsoft Visual C++ (rekomendowane)
- Dowolne środowisko korzystające z GCC

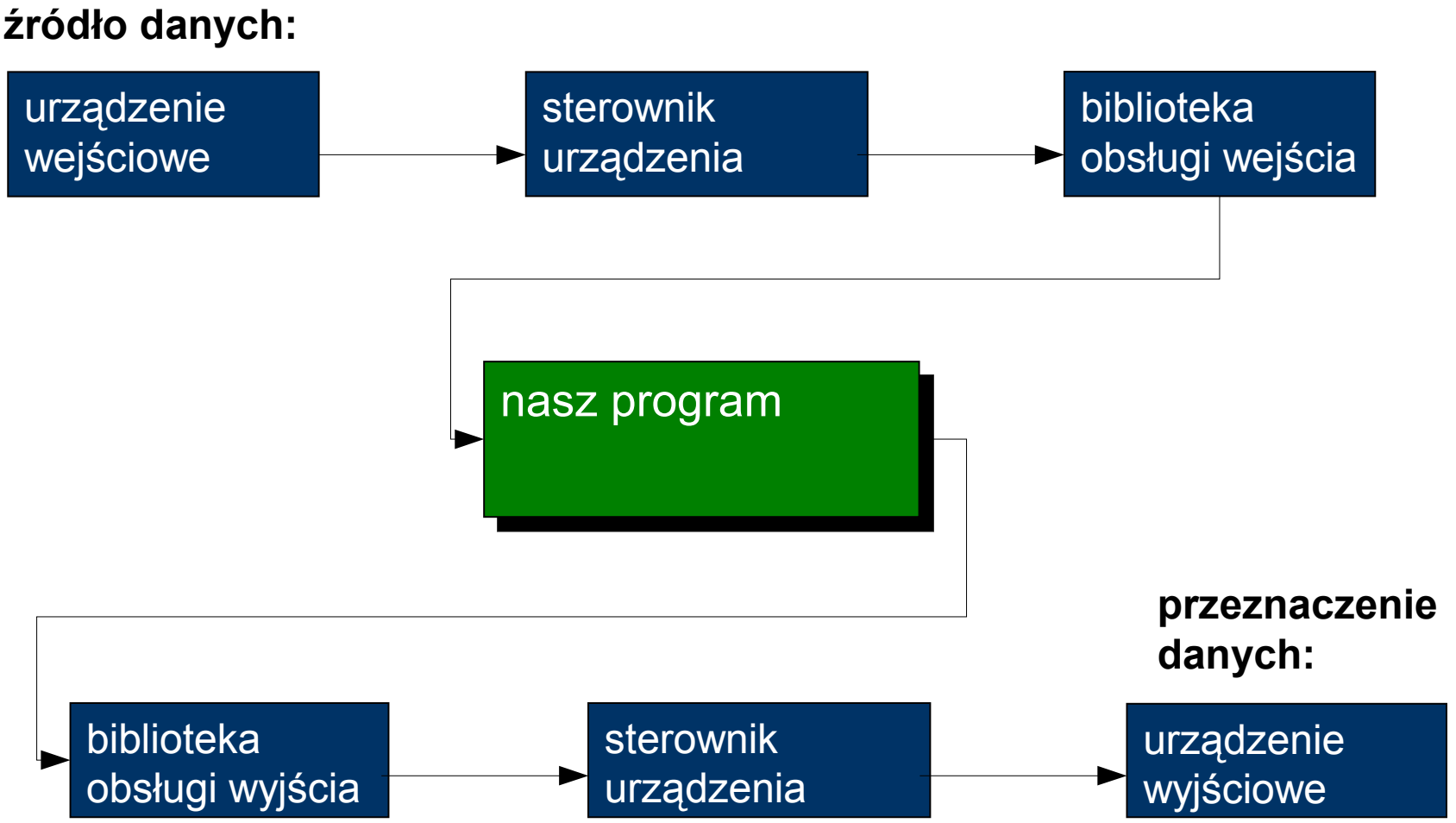
Program wykładów

1. Pierwszy program 5/10
2. Obiekty, typy i wartości. Wykonywanie obliczeń 12/10
3. Błędy. 19/10
4. Tworzenie oprogramowania 26/10
5. Techniki pisania funkcji i klas 9/11
6. **Strumienie wejścia/wyjścia** **16/11**
7. Wskaźniki i tablice 23/11

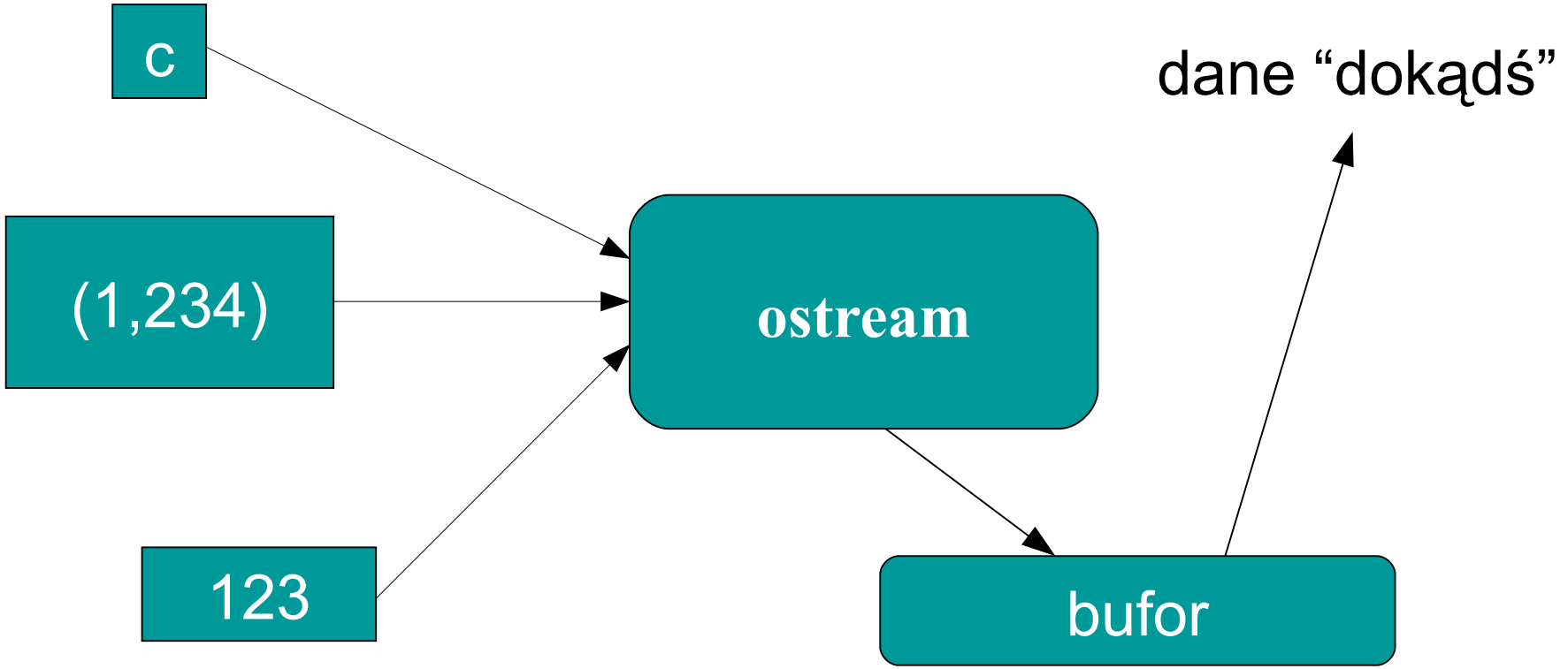
Zagadnienia

- Podstawowe koncepcje WE/WY
- Pliki jako strumienie
  - otwieranie
  - czytanie i zapisywanie strumieni
  - tryby dostępu do plików (tryb binarny, pozycjonowanie w pliku)
- Błędy WE/WY
- Formatowanie wyjscia
- Strumienie łańcuchowe, czytanie linii i znaków
- Operatory czytania i zapisu dla typu użytkownika

Wejście i wyjście

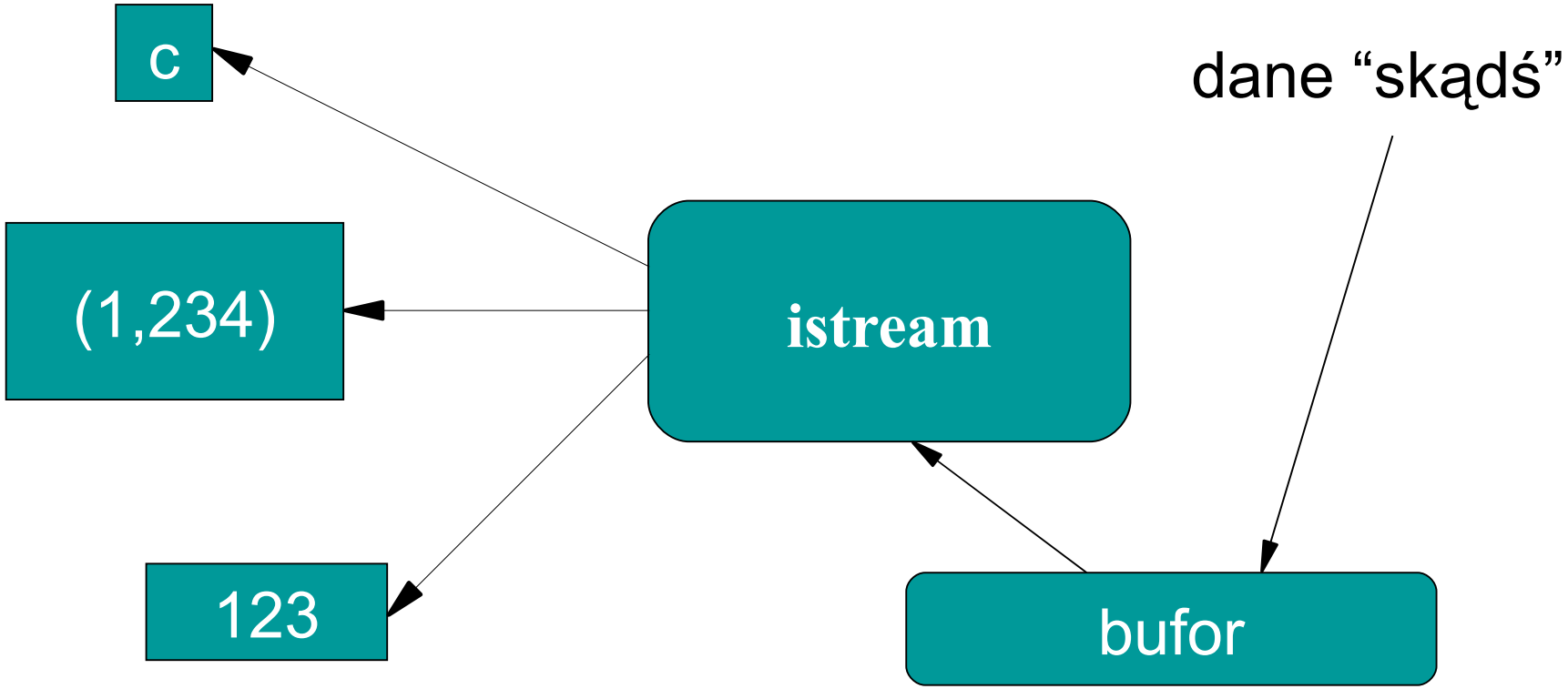


Model strumienia – wyjście



- Strumień wyjściowy, ang. *output stream* (**ostream**)
  - zamienia wartości różnych typów na sekwencje znaków
  - wysyła te znaki „dokądś”
    - np. na ekran, do pliku, do pamięci, do innego komputera

Model strumienia – wejście



- Strumień wejściowy, ang. *input stream* (**istream**)
  - pobiera „skądś” znaki (jednostki danych)
    - np. z klawiatury, z pliku, z pamięci, z innego komputera
  - zamienia sekwencje znaków na wartości różnych typów

Model strumienia – wejście i wyjście

- Strumienie zapewniają czytanie i zapis
  - elementów określonego typu
    - << (wypisanie) i >> (wprowadzenie) plus inne operacje
    - bezpieczeństwo typów
    - formatowanie
  - typowo są przechowywane jako tekst
    - ale niekoniecznie (np. strumienie binarne)
  - są rozszerzalne
    - można definiować własne operacje WE/WY dla własnych typów danych
  - można je podpiąć do każdego urządzenia WE/WY lub przechowywania danych

Pliki

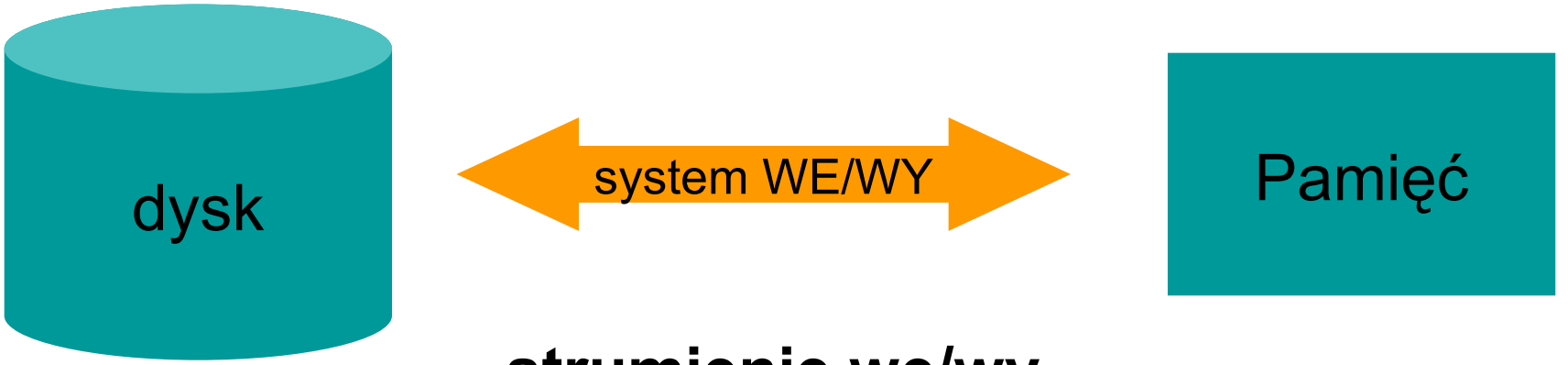
- Pamięć komputera jest ulotna
  - potrzebujemy przechowywać dane na stałych nośnikach, np. dysk
- Plik jest sekwencją bajtów przechowywaną na stałym nośniku, ponumerowanych od 0 w górę:

  - plik ma nazwę
  - dane w pliku mają format
    - np. sześć bajtów (znaków) "**123.45**" może być interpretowane jako liczba zmiennoprzecinkowa **123.45**
- Możemy czytać/zapisywać plik jeśli znamy jego nazwę i format

Zasady dostępu do pliku

- Czytanie z pliku
  - musimy znać jego **nazwę**
  - musimy go **otworzyć (do czytania)**
  - następnie możemy **czytać**
  - następnie musimy go **zamknąć**
    - typowo dzieje się to niejawnie
- Pisanie do pliku
  - musimy znać jego **nazwę**
  - musimy go **otworzyć (do pisania)**
    - lub stworzyć nowy plik o tej nazwie
  - następnie możemy **pisać**
  - następnie musimy go **zamknąć**
    - typowo dzieje się to niejawnie

Dostęp do pliku



strumienie we/wy

pliki
(sekwencje bajtów)

obiekty
(różnych typów)

Otwieranie pliku do czytania

```
// …
int main()
{
    cout << "Prosze wprowadzic nazwe pliku wejsciowego: ";
    string nazwa;
    cin >> nazwa;
    ifstream ist(nazwa.c_str());      // ifstream oznacza "input stream from a file"
                                      // czyli tworzymy zmienną ist, typu:
                                      // strumień wejściowy dla pliku o nazwie nazwa .
                                      // Funkcja c_str() zwraca niskopoziomowy
                                      // czyli "systemowy" łańcuch w stylu C
                                      // przechowywany przez obiekt C++ typu string

                                      // Zdefiniowanie zmiennej typu ifstream
                                      // dla nazwy pliku nazwa, otwiera plik
                                      // o tej nazwie do czytania

    if (!ist) throw runtime_error("Nie mozna otworzyc pliku wejsciowego!");
    // …
```

Otwieranie pliku do pisania

```
// …
int main()
{
    cout << "Prosze wprowadzic nazwe pliku wyjsciowego: ";
    string nazwa;
    cin >> nazwa;
    ofstream ost(nazwa.c_str());    // ofstream oznacza “output stream from a file”
                                    // czyli tworzymy zmienną ost, typu:
                                    // strumień wejściowy dla pliku o nazwie nazwa .
                                    // Funkcja c_str() zwraca niskopoziomowy
                                    // czyli “systemowy” łańcuch w stylu C
                                    // przechowywany przez obiekt C++ typu string

                                    // Zdefiniowanie zmiennej typu ofstream
                                    // dla nazwy pliku nazwa, otwiera plik
                                    // o tej nazwie do pisania

    if (!ost) throw runtime_error("Nie mozna otworzyc pliku wyjsciowego!“);
    // …
```

Czytanie z pliku - przykład

- Niech plik zawiera sekwencje znaków reprezentujące godziny i pomiary temperatury, np.

  0 -1.5
  1 -2.6
  2 -4.3
  3 -6.22

- Godziny są ponumerowane od **0..23**
- Nie ma więcej założeń co do formatu (przynajmniej na razie)
- Kończymy czytać, gdy
  - osiągniemy koniec pliku
  - pojawi się coś nieoczekiwanego, np. litera **q**

Czytanie z pliku temperatur

```
struct Reading {            // typ reprezentujący pomiar temperatury
        int hour;                       // godzina [0:23]
        double temperature;             // temperatura
        Reading(int h, double t) :hour(h), temperature(t) { }       // konstruktor
};

//...
cout << "Prosze wprowadzic nazwe pliku wejsciowego: ";
string name;
cin >> name;
ifstream ist(name.c_str());
if (!ist) throw runtime_error("Nie mozna otworzyc pliku wejsciowego!“);

vector<Reading> temps;              // tworzymy wektor do przechowywania pomiarów
int hour;
double temperature;
while (ist >> hour >> temperature) {                                        // czytaj
        if (hour < 0 || 23 <hour) throw runtime_error("godzina poza zakresem");   // sprawdź
                temps.push_back( Reading(hour,temperature) );                     // zapisz
}
```

Zapis temperatur do pliku

```
vector<Reading> temps;              // tworzymy wektor do przechowywania pomiarów
int hour;
double temperature;
cout << "Wprowadz dane w formacie: godzina temperatura:\n";
while (cin >> hour >> temperature) {                                    // czytaj
        if (hour < 0 || 23 <hour) throw runtime_error("godzina poza zakresem");   // sprawdź
                temps.push_back( Reading(hour,temperature) );          // zapisz do wektora
}

cout << "Prosze wprowadzic nazwe pliku wyjsciowego: ";
string name;
cin >> name;    // nie czeka na podanie nazwy!
ofstream ost(name.c_str());
if (!ost) throw runtime_error("Nie mozna otworzyc pliku wyjsciowego!");

for (int i=0;i<temps.size();i++) {
        ost << temps[i].hour << '\t' << temps[i].temperature << endl;
}
```

Przykłady błędów przy wczytywaniu **int**-ów

```
int a;
while (cin >> a);
```

- zakończenie przez podanie „znaku kończącego”: **1 2 3 4 5 ***
  - stan błędu: **fail()**
- zakończenie przez błąd formatu: **1 2 3 4 5.6**
  - stan błędu: **fail()**
- zakończenie przez znak końca pliku (EOF):

  1 2 3 4 5 EOF *// jeśli czytamy z pliku*
  1 2 3 4 5 Control-Z *// jeśli czytamy z klawiatury Windows*
  1 2 3 4 5 Control-D *// jeśli czytamy z klawiatury Linux*

  - stan błędu: **eof()**
- naprawdę poważny problem, np. błąd dysku
  - stan błędu: **bad()**

Obsługa błędów strumienia

```
vector<Reading> temps;              // tworzymy wektor do przechowywania pomiarów
int hour;
double temperature;
cout << "Wprowadz dane w formacie: godzina temperatura:\n";
while (cin >> hour >> temperature) {                                    // czytaj
        if (hour < 0 || 23 <hour) throw runtime_error("godzina poza zakresem");   // sprawdź
                temps.push_back( Reading(hour,temperature) );          // zapisz do wektora
}

if (cin.bad()) throw runtime_error("Poważny problem ze standardowym wejsciem!");
else if (cin.fail() || cin.eof()) {
        cin.clear();          // czyscimy stan strumienia, bez tego nie mozemy z niego korzystac
        cin.ignore(numeric_limits<streamsize>::max(),'\n');       // pomijamy wszystkie znaki
                                                                  // az do \n lub konca pliku (eof)
}
```

Zapis temperatur do pliku

```
// ...
while (cin >> hour >> temperature) {                                    // czytaj
        if (hour < 0 || 23 <hour) throw runtime_error("godzina poza zakresem");   // sprawdź
                temps.push_back( Reading(hour,temperature) );          // zapisz do wektora
}

if (cin.bad()) throw runtime_error("Poważny problem ze standardowym wejsciem!");
else if (cin.fail() || cin.eof()) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(),'\n');
}
cout << "Prosze wprowadzic nazwe pliku wyjsciowego: ";
string name;
cin >> name;
ofstream ost(name.c_str());
if (!ost) throw runtime_error("Nie mozna otworzyc pliku wyjsciowego!");

for (int i=0;i<temps.size();i++) {
        ost << temps[i].hour << '\t' << temps[i].temperature << endl;
}
```

Źródła błędów WE/WY

- Źródła błędów
  - pomyłki człowieka
  - pliki, które nie spełniają specyfikacji
  - specyfikacje, które nie odpowiadają życiu
  - błędy programisty
  - ...
- Dla biblioteki **iostream** błędy są opisane czterema stanami:
  - **good()** *// brak błędu, operacja się powiodła*
  - **eof()** *// dotarliśmy do końca wejścia (ang. end of file)*
  - **fail()** *// zdarzyło się coś nieoczekiwanego*
  - **bad()** *// zdarzyło się coś nieoczekiwanego i poważnego*

Wczytywanie z obsługą błędów

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
{       // czyta liczby całkowite z ist do v aż osiągnie stan eof() lub terminator
    int i = 0;
    while (ist >> i) v.push_back(i);     // czyta I zapisuje w v do wystąpienia błędu
    if (ist.eof()) return;               // jeśli koniec pliku, to ok
    if (ist.bad()) throw runtime_error("ze strumieniem wejsciowym jest zle");
                                         // jeśli coś poważnego, spadajmy stąd: wyjątek

    if (ist.fail()) { // jeśli to nic poważnego, to może po prostu znaleźliśmy terminator?
        ist.clear();         // czyścimy stan strumienia, abyśmy mogli z nim pracować
        char c;
        ist>>c;              // czytamy znak, aby sprawdzić czy to terminator
        if (c != terminator) {                          // jeśli nie terminator,
                                                        // to jednak coś jest nie tak:
            ist.unget();                                // odkładamy znak do strumienia
            ist.clear(ios_base::failbit);               // i z powrotem ustawiamy stan fail()
        }
    }
}
```

Wczytywanie z obsługą błędów
automatyczne rzucanie wyjątków stanu **bad()**

```
// Np. na początku funkcji main():
ist.exceptions(ist.exceptions()|ios_base::badbit);
    // Ustawia maskę wyjątków dla strumienia ist tak, by to strumień rzucał wyjątek,
    // gdy stanie się coś poważnego. Jaki typ wyjątku? Sprawdź w dokumentacji:-)
void fill_vector(istream& ist, vector<int>& v, char terminator) {
    int i = 0;
    while (ist >> i) v.push_back(i);     // czyta i zapisuje w v do wystąpienia błędu
    if (ist.eof()) return;               // jeśli koniec pliku, to ok
    if (ist.fail()) { // jeśli to nic poważnego, to może po prostu znaleźliśmy terminator?
        ist.clear();          // czyścimy stan strumienia, abyśmy mogli z nim pracować
        char c;
        ist>>c;               // czytamy znak, aby sprawdzić czy to terminator
        if (c != terminator) {                    // jeśli nie terminator,
            ist.unget();                          // odkładamy znak do strumienia
            ist.clear(ios_base::failbit);         // i z powrotem ustawiamy stan fail()
        }
    }
}
```

Czytanie pojedynczej wartości

```
// pierwsze podejście – dziurawe:
cout << "Podaj liczbe calkowita z zakresu 1 do 10 (wlaczenie):\n";
int n = 0;
while (cin>>n) {                          // czytaj
      if   (1<=n && n<=10) break;  // sprawdź zakres, jeśli okej to wyjdź z pętli
      cout << "Przykro mi, " << n <<" nie jest w [1:10] spróbuj ponownie:\n";
}
cout << "Podales: " << n << endl;
```

- Możliwe są trzy rodzaje problemów
  - użytkownik podał liczbę spoza zakresu (to działa)
  - nie otrzymaliśmy żadnej wartości (**EOF**, np. **Ctrl-Z**)
    - program powinien zapytać jeszcze raz?
  - użytkownik podaj coś złego typu (nie **int**)
    - jak program potraktuje znaki **'FF'**, a jak liczbę **10.2**?

Czytanie pojedynczej wartości – jak to ugryźć?

- Jak chcemy poradzić sobie z tymi trzema problemami?
  - zająć się problemem w kodzie czytającym zmienną?
  - rzucić wyjątek, aby ktoś inny zajął się problemem (ewentualnie kończąc program)?
  - zignorować problem?

- Decyzja jest poważna
  - pojedyncza wartość wczytujemy wiele razy
  - potrzebujemy czegoś co jest bardzo proste w użyciu

Opcja pierwsza, czyli zajmujemy się wszystkimi problemami naraz

Efekt? Działa, ale niezły bałagan!

```
cout << "Podaj liczbe calkowita z zakresu 1 do 10 (wlaczenie):\n";
int n = 0;
while (n==0) {          // Widzisz błąd, heh?
    cin >> n;
    if (cin) {          // strumień wejścia nie zgłosił błędu, mamy więc liczbę całkowitą
        if (1<=n && n<=10) break;
        cout << "Przykro mi, " << n <<" nie jest w zakresie [1:10] spróbuj ponownie:\n";
    }
    else if (cin.fail()) {      // jeśli jednak strumień zgłosił stan fail(), tzn. że mamy coś, ale nie int
        cin.clear();            // chcemy się temu przyjrzeć, czyścimy znacznik stanu
        cout << "Przykro mi, podałeś coś co nie jest liczbą, spróbuj ponownie!\n";
        char ch;
        while (cin>>ch && !isdigit(ch)) ;                   // w pętli usuwamy ze strumienia nie cyfry
        if (!cin) throw runtime_error("Nie znalazlem liczby");  // chyba, że pojawił się błąd cin
        cin.unget();                                        // odkładamy cyfrę spowrotem, aby móc wczytać liczbę
    }
    else
        throw runtime_error("Nie znalazlem liczby"); // stan eof() lub bad() - poddajemy się
}
cout << "Podales: " << n << endl;                   // jeśli dotarliśmy tu, to n jest w [1:10]:-)
```

Skąd ten bałagan?
Próbowaliśmy zrobić wszystko naraz!

- Zajmujemy się następującymi problemami:
  - czytanie wartości
  - proszenie użytkownika o podanie wejścia
  - wypisywanie komunikatów błędu
  - pomijanie „niewłaściwych” znaków
  - sprawdzanie zakresu
- A gdzie podział na logicznie odrębne części?

Czego potrzebujemy?

- Podział na logicznie odrębne części:
  - wczytanie liczby całkowitej w podanym zakresie ze std wejścia

    int get_int(int low, int high)
  - wymaga wczytania jakiejkolwiek liczby całkowitej ze std wejścia

    int get_int()
  - możemy wydzielić kod odpowiadający za pomijanie nie-**int**-ów (na standardowym wejściu)

    void skip_to_int()

Pomijanie nie-**int**-ów

```
void skip_to_int()
{
    // Warunki początkowe: wczytywano zmienną typu int, pojawił się stan błędu cin
    if (cin.good())  return; // warunek, że pojawił się błąd niespełniony
    if (cin.fail()) {   // jeśli strumień zgłosił stan fail(), tzn. że mamy coś, ale nie int
        cin.clear();       // chcemy się temu przyjrzeć, czyścimy znacznik stanu
        char ch;
        while (cin>>ch) {           // w pętli czytamy, czyli usuwamy znaki z cin
            if (isdigit(ch)) {   // aż natrafimy na cyfrę
                cin.unget();        // odkładamy ją spowrotem
                                    // aby móc wczytać liczbę
                return;
            }
        }
    }
    throw runtime_error("Nie znalazlem zadnej liczby");    // poddajemy się, jeśli
                                                           // stan eof() lub bad()
    // Warunki końcowe: na początku cin znajduje się cyfra, a więc i liczba typu int
    // lub zgłaszony wyjątek
}
```

Wczytanie jakiegokolwiek **int**-a

```
int get_int()
{
    int n = 0;
    while (true) {
        if (cin >> n) return n;
        cout << "Przykro mi, ale to nie liczba, sprobuj ponownie:\n";
        skip_to_int();
    }
}
```

Wczytanie **int**-a w zakresie

```
int get_int(int low, int high)
{
    cout << "Prosze podac liczbe calkowita w zakresie od "
         << low << " do " << high << " (wlacznie):\n";
    while (true) {
        int n = get_int();
        if (low<=n && n<=high) return n;
        cout << "Przykro mi, "
             << n << " nie jest w zakresie [" << low << ':' << high
             << "]; sprobuj ponownie:\n";
    }
}
```

Użycie naszego „cudeńka”

```
int n = get_int(1,10);
cout << "n: " << n << endl;

int m = get_int(2,300);
cout << "m: " << m << endl;
```

- Eleganckie?
- Tak, ale... wciąż jest pewien problem
  - dialogi z użytkownikiem są przemieszane z wczytywaniem

Czego tak naprawdę potrzebujemy?

- Często to naprawdę ważne pytanie
- Zadawaj je wielokrotnie podczas tworzenia oprogramowania
  - Poznając problem i jego rozwiązanie, ulepszasz swoją odpowiedź

- Ok, potrzebujemy parametryzacji zakresu i tekstu dialogów:

```
int strength = get_int(1, 10, "Podaj siłę", "Poza zakresem, probuj dalej");
cout << "Siła: " << strength << endl;


int altitude = get_int(0, 50000, "Please enter altitude in feet", "Not in range,
    please try again");
cout << "Altitude: " << altitude  << "ft. above sea level\n";
```

Parametryzujmy...

```
int get_int(int low, int high, const string& powitanie, const string& blad)
{
        cout << powitanie << ": [" << low << ':' << high << "]\n";
        while (true) {
                int n = get_int();
                if (low<=n && n<=high) return n;
                cout << blad  << ": [" << low << ':' << high << "]\n";
        }
}
```

- Rozwiązanie jest wciąż niekompletne
  - funkcje pomocnicze nie powinny generować własnych komunikatów błędów
  - poważne funkcje biblioteczne rzucają wyjątki, które mogą zawierać komunikat błędu

operator<< zdefiniowany przez użytkownika

- Przeważnie jest trywialny
  - dla klasy **Date z** poprzedniego wykładu:

```
ostream& operator<<(ostream& os, const Date& d)
{
        return os << '(' << d.year()
        << ',' << d.month()
        << ',' << d.day() << ')';
}
```

  - dzięki operatorowi wypisywania możemy dostosować sposób wyświetlania obiektu do gustu użytkownika
- Przykłady użycia:

```
void wydrukuj_cos(Date d1, Date d2)
 {
        cout << d1;             // to samo co:  operator<<(cout,d1) ;
        cout << d1 << d2;       // to samo co:  (cout << d1)  <<  d2;
                                // to samo co:  (operator<<(cout,d1)) <<  d2;
                                // to samo co:  operator<<((operator<<(cout,d1)), d2) ;
}
```

operator>> zdefiniowany przez użytkownika

```
istream& operator>>(istream& is, Date& dd)
    // Czyta datę w formacie: ( rok , miesiąc , dzień )
{
    int y, d, m;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;

    if (!is) return is;          // jeśli wystąpił błąd wejścia (bool operator!() const)

    if (ch1!='(' || ch2!=',' || ch3!=',' || ch4!=')') {        // jeśli błąd formatu daty
        is.clear(ios_base::failbit); // ustawia stan strumienia is na wartość zły format
        return is;
    }

    dd = Date(y,m,d);        // uaktualnia datę: dd ← nowa data
    return is;               // I wychodzi z funkcji zostawiając strumień is
                             // w dobrym stanie
}
```

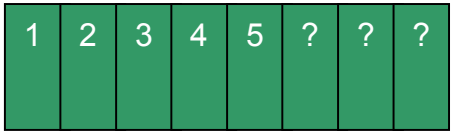
Tryby otwierania plików

- Domyślnie **ifstream** otwiera swój plik do czytania
- Domyślnie **ofstream** otwiera swój plik do zapisu
- Inne możliwości:
  - **ios_base::app** // **app**end – *dopisywanie na końcu pliku*
  - **ios_base::ate** // **at e**nd – *otwiera i ustawia na końcu pliku*
  - **ios_base::binary** // **binary** – *tryb binarny*
  - **ios_base::in** // **in**put – *do czytania*
  - **ios_base::out** // **out**put – *do zapisu*
  - **ios_base::trunc** // **trunc**ate – *obcina plik do rozmiaru zerowego*
- Tryb dostępu do pliku może być określony przy tworzeniu strumienia:
  - **ofstream of1(name1.c_str());** // *domyślnie tryb* ***ios_base::out***
  - **ifstream if1(name2.c_str());** // *domyślnie tryb* ***ios_base::in***
  - **ofstream ofs(name.c_str(), ios_base::app);**
    // *dopisywanie zamiast nadpisywania*
  - **fstream fs("myfile", ios_base::in|ios_base::out);** // *czytanie* ***lub*** *pisanie*

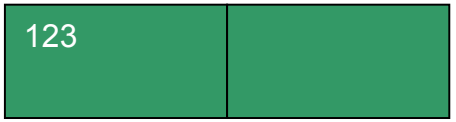
Plik tekstowy czy binarny?

123 jako znaki:

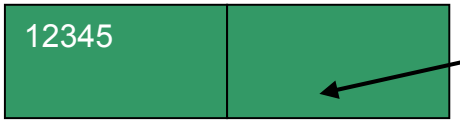
12345 jako znaki:



123 binarnie:

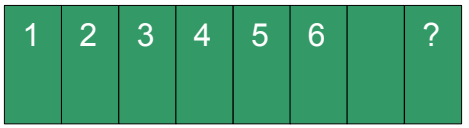


12345 binarnie:

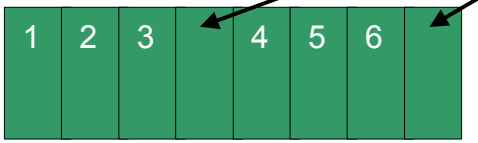


w pliku binarnym
wartości są ograniczone przez rozmiar pola

123456 jako znaki:



123 456 jako znaki:



w pliku tekstowym:
wartości są ograniczone znakami rozdzialającymi - często **spacja** lub **'\n'**

Plik tekstowy czy binarny?

- Tekstowy, kiedy się da
  - Możesz go czytać bez specjalnego programu
  - Łatwiej odrobaczysz swój program
  - Tekst jest przenośny pomiędzy różnymi systemami
  - Większość informacji można przedstawić jako tekst
- Binarny, kiedy musisz
  - pliki graficzne, muzyczne itp.
  - pliki bardzo duże

Otwieranie pliku binarnego

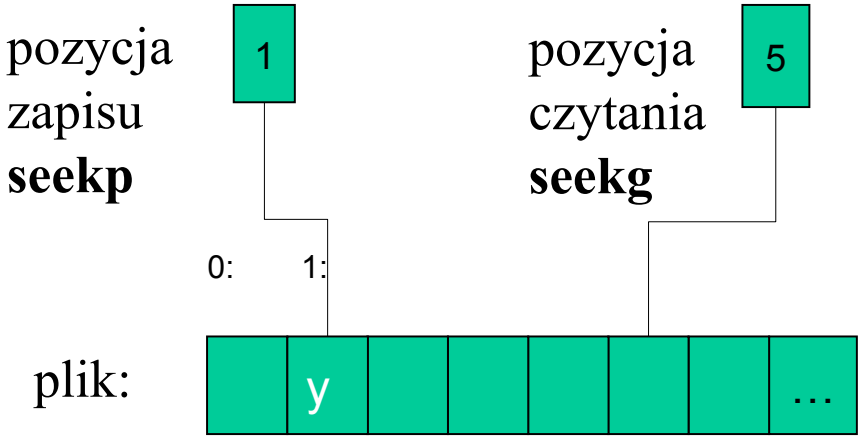
```
int main()
    // używa binarnego wejścia i wyjścia
    // ...

    cout << "Podaj nazwę pliku: ";
    string nazwa;
    cin >> nazwa;
    ifstream ifs(nazwa.c_str(),ios_base::binary);     // tryb binarny
    if (!ifs) throw runtime_error("Nie mogę otworzyć pliku!");

    cout << "Podaj nazwę pliku:";
    cin >> nazwa;
    ofstream ofs(nazwa.c_str(),ios_base::binary);    // tryb binarny
    if (!ofs) throw runtime_error("Nie mogę otworzyć pliku!");

    // Tryb “binary” mówi strumienowi, żeby nie próbował robić niczego mądrego z bajtami pliku
```

Plik binarny – odczyt i zapis

```
vector<int> v;

// odczyt z pliku binarnego:
char bufor[sizeof(int)];                    // tworzymy bufor odczytu
                                            // wielkości zmiennej int
while (ifs.read(bufor,sizeof(int)))         // czytanie bajtów danych do bufora
        v.push_back(int(bufor));            // zapisujemy odczytaną wartość
                                            // do wektora v, interpretując ją jako int

// ... rób coś z wektorem v ...

// zapis do pliku binarnego:
for(int i=0; i<v.size(); ++i) {
        *bufor = v[i];                      // do pamięci pod adresem bufor
                                            // wczytujemy i-tą wartość wektora v
        ofs.write(bufor,sizeof(int));       // zapis bajtów danych z bufora
}
return 0;
}
```

!! O tablicach i wskaźnikach – wykład 7 !!

Pozycja w strumieniu plikowym



fstream fs(name.c_str()); *// otwiera plik do odczytu lub zapisu*

// ...

fs.seekg(5); *// ustawia pozycję czytania (‘**g**’ jak ‘*get*’) na 5 (czyli 6-ty znak)*
char ch;
fs>>ch; *// czyta i inkrementuje pozycję czytania*
cout << "znak[6] to " << ch << '(' << int(ch) << ")\n";
fs.seekp(1); *// ustawia pozycję zapisu (‘**p**’ jak ‘*put*’) na 1 (czyli 2-gi znak)*
fs<<'y'; *// zapisuje i inkrementuje pozycję zapisu*

Pozycjonowanie

- Jest błędogenne
  - obsługa końca pliku zależy od systemu i nie jest kontrolowana
- Kiedy tylko możesz
  - używaj prostych operacji na strumieniach

Obserwacje

- Jako programiści preferujemy regularność i prostotę
  - hej, naszym zadaniem jest spełnienie oczekiwań użytkowników

- A ludzie potrafią być oryginalni, drobiazgowi i grymaśni co do tego jak ma wyglądać ich wyjście
  - często mają ku temu dobre powody
  - często wynika to z tradycji i konwencji,
    - np. co oznaczają zapisy (123) oraz 09/11/01 ?

Formaty zapisu liczb

- Liczby całkowite
  - 1234 (dziesiętny)
  - 2322 (ósemkowy)
  - 4d2 (szesnastkowy)
- Liczby zmiennoprzecinkowe
  - 1234.57 (ogólny)
  - 1.2345678e+03 (naukowy)
  - 1234.567890 (stałoprzecinkowy)
- Precyzja (dla liczb zmiennoprzecinkowych)
  - 1234.57 (precyzja 6)
  - 1234.6 (precyzja 5)
- Pole zapisu (justowanie)
  - |12| (domyślnie)
  - | 12| (12 w polu 4 znaków)

Jak to zrobić: zmiana podstawy

- Dostępne podstawy
  - **10** – zapis dziesiątkowy (**dec**); cyfry: **0 1 2 3 4 5 6 7 8 9**
  - **8** – zapis ósemkowy (**oct**); cyfry: **0 1 2 3 4 5 6 7**
  - **16** – zapis szesnastkowy (**hex**); cyfry: **0 1 2 3 4 5 6 7 8 9 a b c d e f**

// prosty test:
cout << dec << 1234 << "\t(dziesiętny)\n"
<< hex << 1234 << "\t(szesnastkowy)\n"
<< oct << 1234 << "\t(ósemkowy)\n";
// Znak '\t' to tabulator

// w efekcie dostaniemy:

1234 (dziesiętny)
4d2 (szesnastkowy)
2322 (ósemkowy)

Manipulator podstawy jest trwały

- Dostępne podstawy
  - **10** – zapis dziesiątkowy (**dec**); cyfry: **0 1 2 3 4 5 6 7 8 9**
  - **8** – zapis ósemkowy (**oct**); cyfry: **0 1 2 3 4 5 6 7**
  - **16** – zapis szesnastkowy (**hex**); cyfry: **0 1 2 3 4 5 6 7 8 9 a b c d e f**

// prosty test:

```
cout << dec << 1234 << "\t(dziesiętny)\n"
<< hex << 1234 << "\t(szesnastkowy)\n"
<< oct << 1234 << "\t(ósemkowy)\n";
cout << 1234 << '\n';
```

// w efekcie dostaniemy:

```
1234   (dziesiętny)
4d2    (szesnastkowy)
2322   (ósemkowy)
2322
```

Wyświetlanie podstawy

- Dostępne podstawy
  - **10** – zapis dziesiątkowy (**dec**); cyfry: **0 1 2 3 4 5 6 7 8 9**
  - **8** – zapis ósemkowy (**oct**); cyfry: **0 1 2 3 4 5 6 7**
  - **16** – zapis szesnastkowy (**hex**); cyfry: **0 1 2 3 4 5 6 7 8 9 a b c d e f**

// prosty test:

```
cout << dec << 1234 << "\t(dziesiętny)\n"
<< hex << 1234 << "\t(szesnastkowy)\n"
<< oct << 1234 << "\t(ósemkowy)\n";
cout << showbase;
cout << dec << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
```

// w efekcie dostaniemy:

```
1234   (dziesiętny)
4d2    (szesnastkowy)
2322   (ósemkowy)
1234   0x4d2    02322
```

Manipulatory zmiennoprzecinkowe

- Dostępne formaty:
  - domyślnie – **iostream** wybiera najlepszy format
  - **fixed** – format stałoprzecinkowy
  - **scientific** – format naukowy (mantysa + wykładnik)

```
// prosty test:

cout << 1234.56789 << "\t\t(domyślny)\n"          //  \t\t  żeby wyrównać kolumny
     << fixed << 1234.56789 << "\t(stałoprzecinkowy)\n"
     << scientific << 1234.56789 << "\t(naukowy)\n";

cout.unsetf(ios_base::fixed);          // w bibliotece standardowej nie ma eleganckiej
cout << 1234.56789 << endl;            // metody powrotu do formatu domyślnego

// efekt:

    1234.57             (domyślny)
    1234.567890         (stałoprzecinkowy)
    1.234568e+003       (naukowy)
    1234.57
```

Manipulator precyzji

- Precyzja (domyślnie: **6**) w różnych formatach zmiennoprzecinkowych:
  - domyślny: precyzja określa liczbę cyfr
  - **scientific:** precyzja określa liczbę cyfr po przecinku (kropce)
  - **fixed:** precyzja określa liczbę cyfr po przecinku (kropce)

// przykład:

```
cout << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';

cout.unsetf(ios_base::scientific);

cout << setprecision(5)  << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';

cout.unsetf(ios_base::scientific);

cout << setprecision(8) << 1234.56789 << '\t' << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
```

// efekty (zauważ zaokrąglanie):

```
1234.57     1234.567890     1.234568e+003
1234.6      1234.56789      1.23457e+003
1234.5679   1234.56789000   1.23456789e+003
```

Rozmiar pola wyjściowego

- Określenie rozmiary następnej operacji wyjścia
  - Uwaga! manipulator odnosi się tylko do następnego wyjścia
  - Uwaga! Wyjście nigdy nie jest przycinane, aby zmieściło się w polu
    - lepszy zły format niż zła wartość

// przykład:

```
cout << 123456 << '| '<< setw(4) << 123456 << '|'
     << setw(8) << 123456 << '|' << 123456 << "|\n";

cout << 1234.56 << '|' << setw(4) << 1234.56 << '|'
     << setw(8) << 1234.56 << '|' << 1234.56 << "|\n";

cout << "asdfgh" << '|' << setw(4) << "asdfgh" << '|'
     << setw(8) << "asdfgh" << '|' << "asdfgh" << "|\n";
```

// efekt:

```
123456|123456|  123456|123456|
1234.56|1234.56| 1234.56|1234.56|
asdfgh|asdfgh|  asdfgh|asdfgh|
```

Ważne spostrzeżenie

- To jest rodzaj szczegółów, których należy szukać w podręcznikach, manualach, helpach itp.
  - Nigdy nie spamiętasz wszystkich szczegółów!

http://www.cplusplus.com/reference/iostream/ios_base/fmtflags/

Czytanie linii i strumień łańcuchowy

- Czytanie łańcucha tekstowego

```
string name;
cin >> name;              // input: Dennis Ritchie
cout << name << '\n';     // output: Dennis
```

- Czytanie linii

```
string name;
getline(cin,name);        // input: Dennis Ritchie
cout << name << '\n';     // output: Dennis Ritchie
// chcemy rozbić na imię nazwisko - jak?
// może tak:
istringstream ss(name);
ss>>first_name;
ss>>second_name;

// getline(cin,name) działa tak jak: getline(cin,name,'\n')
// getline(cin,name,'\t') czyta ze strumienia aż do znaku tabulatora
```

Strumień łańcuchowy **stringstream**

- Obiekt klasy **stringstream** czyta i zapisuje do **string**-a,
  - tak jakby to był plik lub konsola (klawiatura/ekran)

```
double str2double(string s)
{
        // jeśli możliwe, konwertuje znaki w s na wartość zmiennoprzecinkową

        istringstream is(s);        // tworzy strumień is z łańcucha s
        double d;
        is >> d;
        if (!is) throw runtime_error("Błąd formatu zmiennoprzecinkowego");
        return d;
}

double d1 = str2double("12.4");                          // testujemy...
double d2 = str2double("1.34e-3");
double d3 = str2double("dwanascie przecinek trzy");      // błąd!
```

- **stringstream** jest bardzo użyteczny
  - do formatowania w ograniczonej przestrzeni (okno)
  - do wyodrębniania obiektów z łańcucha tekstowego

Czytanie znak po znaku

- Możesz też czytać znak po znaku
  - biblioteka **cctype** ma funkcje sprawdzające rodzaj znaku

```
char ch;
while (cin>>ch) {   // czyta do ch, omijając białe znaki
        if (isalpha(ch)) {    // sprawdza czy litera
                // robi coś
        }
}

while (cin.get(ch)) {       // czyta do ch, nie omijając białych znaków
        if (isspace(ch)) {    // sprawdza czy znak biały
                // robi coś
        }
        else if (isalpha(ch)) {     // sprawdza czy litera
                // robi coś
        }
}
```

Klasyfikacja znaków w **cctype**

- Niektóre z możliwości:
  - **isspace(c)** // *czy **c** jest znakiem białym? (' ', '\t', '\n' itp.)*
  - **isalpha(c)** // *czy **c** jest literą? ('a'..'z', 'A'..'Z')*
  - **isdigit(c)** // *czy **c** jest cyfrą? ('0'.. '9')*
  - **isupper(c)** // *czy **c** jest dużą literą?*
  - **islower(c)** // *czy **c** jest małą literą?*
  - **isalnum(c)** // *czy **c** jest znakiem alfanumerycznym?*

Czytanie zmiennych czy czytanie linii?

- Wskazówka Bjarne S.: raczej >> niż **getline()**
  - unikaj czytania linii kiedy możesz

- Ludzie używają **getline()**, bo nie widzą innej możliwości
  - ale często kod staje się wtedy pogmatwany

- Stosowanie **getline()**, często i tak kończy się
  - użyciem >> do parsowania linii ze **stringstream**-a
  - użyciem **get()** do czytania pojedynczych znaków

Dziś najważniejsze było to, że...

- Różne rodzaje wejścia i wyjścia systemu możemy przedstawić jako tzw. strumienie
  - konsola (klawiatura + ekran)
  - pliki dyskowe
  - łańcuchy tekstowe
  - tokeny...

- Szczegóły formatowania strumieni łatwo zapomnieć
  - warto wiedzieć jak je sprawdzić

A za tydzień...

- Wskaźniki i tablice