C++: IOStreams

IOStreams czyli obiektowe strumienie we/wy

Strumień we/wy to źródło lub cel danych (ciągu bajtów).

Strumień może dotyczyć: pliku, urządzenia we/wy, pamięci

Strumienie predefiniowane: cin, cout, cerr, clog

Na strumieniu wykonuje się opracje zapisu i odczytu:

- sformatowanego (tekstowego) - przy pomocy przeciążonych operatorów » i «

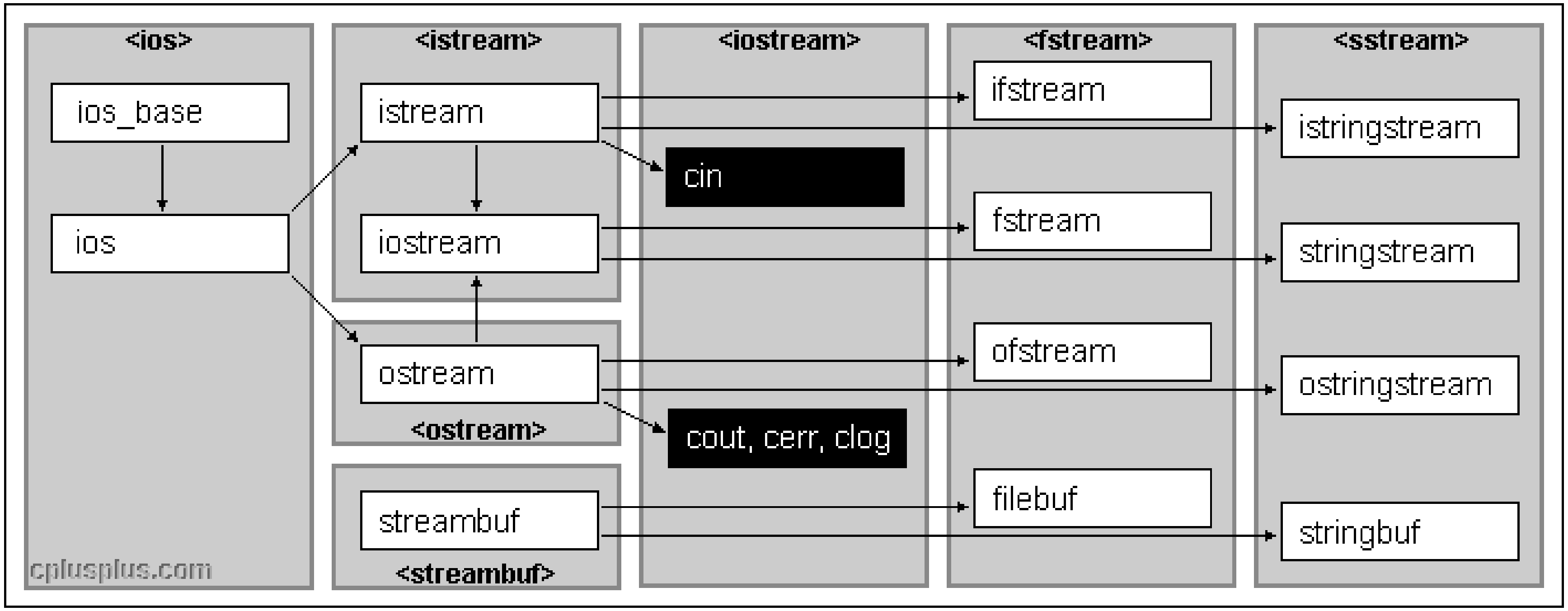
```
int i; cin>> i; // odczyt z cin
cout<< "i=" << i <<endl; // zapis do cout; endl - '\n'
```

- binarnego - przy pomocy funkcji składowych

```
int i=123; cout.write(&i, sizeof(i));
  // zapis binarnej reprezentacji do cout
```

C++: IOStreams

Hierarchia klas strumieni we/wy:



- *ios*
  najogólniesza klasa strumieniowa; dba o: stan strumienia oraz formatowanie przy zapisie (sposoby formatowania są wspólne dla wszystkich typów strumieni)
- *istream, ostream, iostream*
  klasy bazowe strumieni: "i" wejściowa, "o" wyjściowa,"io" dwukierunkowa

- *ifstream, ofstream, fstream*
  klasy strumieni plikowych: wejściowa, wyjściowa, dwukierunkowa
- *istringstream, ostringstream, stringstream,*
  klasy strumieni pamięciowych
- *istrstream, ostrstream, strstream,*
  klasy pprzestarzałych strumieni pamięciowych

Pliki nagłówkowe związane ze strumieniami:

```
#include <iostream> // główny plik
#include <iomanip> // potrzebne przy formatowaniu wyjścia przy pomocy manipulatorów
#include <fstream> // plikowe strumienie
#include <sstream> // pamięciowe strumienie
#include <strstream> // przestarzałe pamięciowe strumienie
```

C++: IOStreams

Operatory ≫ i ≪ są przeciążone dla IOStreams oraz podstawowych typów danych przy pomocy funkcji operatorowych

```
istream& operator>> (istream&, typ&) { /* odczyt z istream (podklasy) */ }
ostream& operator<< (ostream&, typ) { /* zapis do ostream (podklasy) */ }
```

Sposoby użycia operatorów ≫ i ≪ :

```
int i;  cin>> i; cout<< i <<endl;
float f;  cin>> f; cout<< f <<endl;
char t[100]; cin>> t; cout<< t <<endl;
  // wczytuje do tablicy "t" tylko 1 słowo (pomija biale znaki!)
std::string s; cin>> s; cout<< s <<endl;
  // podobnie jak wyżej
```

Większe wyrażenia złożone z operatorów ≪ :

```
cout<<123<<123.123<<endl;
  // to dziala tak samo jak:
((cout<<123)<<123.123)<<endl;
  // przeciążony operator zwraca ref na lewy argument !!!
```

Czytanie/pisanie po jednym znaku:

```
while(cin.get(c) && cout.put(c));
  // gdy cin "się skończy" to cin.get(c) zwraca wartość (void*)0
  // w przeciwnym wypadku zwraca wsk na strumień
      // pod DOS/Win, przy odczycie z konsoli gubi ostatnią linie
      // o ile ta linia nie ma na koncu '\n'
/* niech czym się różni: "char c; cin.get(c);" od "cin>>c;" ???
   odp: drugie pomija białe znaki!
*/

// inne sposóby czytania po 1 znaku:
while(1) {
  if(! cin.get(c) ) break;
  cout.put(c);
}
```

Sprawdzanie stanu strumienia przy pomocy fun. składowych:

```
while(1) {
  cin.get(c);
  //if( cin.eof() ) break;

  // alternatywnie:
  if( cin.fail() ) break;
    // uwaga: operator konw. na (void*) klasy ios jest zdefioniowany tak:
    //   ios::operator void*() { return fail() ? 0 : this; }
    // zatem "if(! cin ) break;" działa tak samo jak powyższa linia kodu
  cout.put(c);
}
```

C++: IOStreams

Formatowanie zapisu do strumienia przy pomocy:

- fun. składowych strumieni
- tzw "manipulatorów"

```
  cout<< 123.123456 <<endl;
  cout.precision(5); // to jest fun. składowa
  cout<< 123.123456 <<endl;
  cout<< 123.123456 <<endl; // tu tez dziala!
  cout<<setprecision(4)<<123.123456 <<endl; // to jest manipulator
  cout<< 123.123456 <<endl; // tu tez dziala!
  cout<<"--------------"<<endl;
  cout.width(7); // fun. składowa
  cout<< 123.123456 <<endl; // tu dziala
  cout<< 123.123456 <<endl; // tu juz nie!!!
/* wydruk
123.123
123.12
123.12
123.1
123.1
--------------
  123.1
123.1
*/
```

```
// inne manipulatory:
cout<<setw(9)<< 123.123456 <<endl; // setw(9) definiuje szerokość pola
cout<< 123.123456 <<endl; // tu juz setw(9) nie dziala!!!
// wydruk:
  123.123
123.123

cout<<hex<<"hex:"<< 123 <<endl;
cout<< 123 <<endl;
cout<<hex<<"hex:"<< 123 <<dec<< " dec:"<< 123 <<endl;
// wydruk:
hex:7b
7b // jak widac hex dziala w strumieniu tak dlugo az nie zostanie odwolane
hex:7b dec:123
```

C++: IOStreams

Własne manipulatory bezparametrowe tworzy się bardzo łatwo!
Takm manipulatorem jest dowolna funkcja pobierająca i zwracająca ref na ostream:

```
ostream& kropka(ostream& out) {
  out.put('.'); return out;
}

// sposób użycia:
cout<< 123<< kropka<<kropka<< 321<< endl;

// wydruk:
123..321
```

C++: IOStreams

Strumienie plikowe:

```
ifstream fstr; // deklarujemy strumień plikowy wejściowy
fstr.open("iostream.txt"); // otwieramy strumień
  // dwukrotnie odczytujemy plik i zapisujemy na cout
while(1) {
  if(! fstr.get(c) ) break;
  cout.put(c);
}
fstr.clear(); fstr.seekg(0);
  // to jest niezbędne (reset bitów stanu strumienia oraz bieżącej pozycji)
while(1) {
  if(! fstr.read(&c, 1) ) break;
  cout.put(c);
}
//fstr.close();

ofstream fstr2; // deklarujemy strumień plikowy wyjściowy
fstr2.open("iostream2.txt");
fstr.clear(); fstr.seekg(0);
while(fstr.get(c) && fstr2.put(c));

fstr.close();
fstr2.close();
```

C++: IOStreams

Strumienie pamięciowe (stringowe) starego typu:

```
char tab[1000];
strstream iostr(tab, 1000);
  // deklarujemy strumień pamięciowy o pojemności <=1000 znaków
iostr<<"111"<<endl;
iostr<<"222"<<endl;
iostr<<123.123<<endl;
  // zapis formatowany do strumienia

cout<<"odczyt ze strumienia pamieciowego na cout ..."<<endl;
iostr.clear(); iostr.seekg(0);
while(1) {
  if(! iostr.get(c) ) break;
  cout.put(c);
}
```

C++: IOStreams

Strumienie pamięciowe (stringowe) nowego typu:

```
stringstream iostr; // deklarujemy strumien pamięciowy
  // Uwaga: nie trzeba podawać jego max pojemności!!!

iostr<<111<<endl;
iostr<<222<<endl;
iostr<<333<<endl;

cout<<"odczyt 1 ..."<<endl;
while(iostr.get(c) && cout.put(c));

cout<<"odczyt 2 ..."<<endl;
iostr.clear(); iostr.seekg(0);
while(iostr.get(c) && cout.put(c));
```

C++: IOStreams

Bufor strumienia (**streambuf**) i jego zastosowania:

Każdy strumień posiada bufor; w przypadku strumieni pamięciowych bufory przechowują 100% danych.

Operatory « i » sa przeciążone dla klas ostream, istream, steambuf w nastepujący sposob: ostream«streambuf, istream»streambuf, dzieki czemu można w jednej instrukcji "przepisac" jeden strumień do drugiego; można też "przeadresować" strumień wymieniając mu bufor na inny ...

Dostęp do bufora strumienia uzyskujemy poprzez fun. składową rdbuf() i rdbuf(streambuf&)

```
/* przepisywanie strumieni:
*/
stringstream iostr; // deklarujemy strumien pamięciowy

cin>>iostr.rdbuf(); // to dziala ok!
//iostr<<cin.rdbuf(); // tez dziala ok!
//while(cin.get(c) && iostr.put(c)); // tez dziala ok!
     // czyta dane z cin aż do naciśnięcia Ctrl+D (unix) lub Ctrl+Z (dos/mswin)

iostr<<"dodatkowy tekst na koniec ..."<<endl;
iostr<<"dodatkowy tekst na koniec ..."<<endl;

cout<<"--- 1 ---"<<endl; // wypisanie iostr na cout
cout<<iostr.rdbuf();

iostr.clear(); iostr.seekg(0); // drugi odczyt iostr
cout<<"--- 2 ---"<<endl;
cout<<iostr.rdbuf();
```

```
/* inne zastosowanie streambuf - "przeadresowanie" strumienia:
*/
printf("--- aaa ---\n");
cout<<111<<endl;
cout<<222<<endl;
cout<<333<<endl;

stringstream iostr;
streambuf *cout_buf=cout.rdbuf();
cout.rdbuf(iostr.rdbuf()); // "przeadresowalem" cout na strumień pamięciowy

printf("--- bbb ---\n");
cout<<111<<endl; // te napisy idą do iostr!!!
cout<<222<<endl;
cout<<333<<endl;

cout.rdbuf(cout_buf); // przywracam cout stary bufor
printf("--- ccc ---\n");
cout<<111<<endl;
cout<<222<<endl;
cout<<333<<endl;
```

C++: IOStreams

Przeciążanie » i « dla własnych klas:

```
/* przeciążamy operator << dla klasy/struktury Macierz ...
*/
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

struct Macierz {
  int IleW, IleK; float *Elem;
  float &ElemMac(int W, int K) { return Elem[W*IleK + K]; }
  Macierz(int IleW, int IleK);
  ~Macierz() { free(Elem); }
};
Macierz::Macierz(int _IleW, int _IleK)
{
  Elem=(float*)calloc(IleW*IleK, sizeof(float));
  IleW=_IleW; IleK=_IleK;
}
```

```
ostream& operator<< (ostream& out, Macierz& M)
{
  long e=0;
  for(int w=0; w<M.IleW; w++) {
    out<<"wiersz nr "<<w<<": ";
    for(int k=0; k<M.IleK; k++) {
      //out<<setw(6)<<setprecision(4)<<M.Elem[e]<<" ";
      out<<setw(6)<<M.Elem[e]<<" ";
      e++;
    }
    out<<endl;
  }
  return out;
}
int main()
{
  Macierz M(10,6);

  int licz=1;
  for(int w=0; w<M.IleW; w++)
  for(int k=0; k<M.IleK; k++) M.ElemMac(w,k)=licz++;

                 // !!!!
  cout<<endl<<M; // !!!!
                 // !!!!
}
```

```
// wydruk:
wiersz nr 0:      1      2      3      4      5      6
wiersz nr 1:      7      8      9     10     11     12
wiersz nr 2:     13     14     15     16     17     18
wiersz nr 3:     19     20     21     22     23     24
wiersz nr 4:     25     26     27     28     29     30
wiersz nr 5:     31     32     33     34     35     36
wiersz nr 6:     37     38     39     40     41     42
wiersz nr 7:     43     44     45     46     47     48
wiersz nr 8:     49     50     51     52     53     54
wiersz nr 9:     55     56     57     58     59     60
```

C++: Wzorce (templates)

Wzorzec to sparametryzowany typ ...

Istnieją wzorce funkcji i wzorce struktur/klas.

Postać ogólna wzorca:

```
template < parametry_wzorca >
definicja_wzorca
```

Co może być parametrem wzorca?

```
class/typename T - typ
typ id= wartość - stała
```

Wzorce to pełnią częściowo podobną rolę jak OOP ... Po co je wprowadzono?

1. gdyż nie wszystkie klasy dziedziczą z jednej wspólnej klasy bazowej (np Element albo Object) i nie można umieszczać obiektów we wspólnym pojemniku (np liście)

2. wzorce powoduja mniejszy "narzut" niż klasyczne OOP

```
/* przykład wzorca funkcji:
*/
template <class T>
//template <typename T> // tak tez dziala!
T& fun(T& t) {
  return t;
}
// sposób użycia:
float f1, f2; f1=fun(f2); // ze wzorca zostaną wygenerowane 2 funcje
int i1, i2; i1=fun(i2);


/* przykład wzorca klasy:
*/
template <class T>
class Wektor2 {
public:
  Wektor2() {
    printf("Wektor2, konst. bezpar., %p\n", this);
  }
  Wektor2(T p1, T p2) {
    printf("Wektor2, konst. (T,T), %p\n", this);
    x1=p1; x2=p2;
  }
  Wektor2(Wektor2 const& w) {
```

```
    printf("Wektor2, konst. kop., %p %p\n", this, &w);
    x1=w.x1; x2=w.x2;
  }
  ~Wektor2() {
    printf("Wektor2, destructor, %p\n", this);
  }
  Wektor2& operator=(Wektor2 w) {
    printf("Wektor2, operator=, %p\n", this);
    x1=w.x1; x2=w.x2;
    return *this;
  }
  // --- obsluga IOStreams ---
  friend ostream& operator<<( ostream& o, Wektor2<T>& w2 );
private:
  T x1, x2;
};

// wszystkie sposoby użycia <<; nie da sie sprytniej???
ostream& operator<<( ostream& o, Wektor2<int>& w2 ) {
  cout<< w2.x1 <<" "<< w2.x2; return o;
}
ostream& operator<<( ostream& o, Wektor2<Wektor2<int> >& w2 ) {
  cout<< w2.x1 <<" "<< w2.x2; return o;
}
```

```
int main() {
  Wektor2<int> w_int1(1,2);
  Wektor2<int> w_int2(3,4);
  Wektor2<Wektor2<int> > w_wektor2(w_int1, w_int2);
    // mozna zagniezdzac typy sparametryzowane!!!
    // pamietac o spacji w "> >" !!!

  cout<< w_int1 <<endl;
  cout<< w_int2 <<endl;
  cout<< w_wektor2 <<endl;
}
// wydruk:
Wektor2, konst. (T,T), 0xbff09b60
Wektor2, konst. (T,T), 0xbff09b50
Wektor2, konst. kop., 0xbff09b30 0xbff09b50
Wektor2, konst. kop., 0xbff09b20 0xbff09b60
Wektor2, konst. bezpar., 0xbff09b40
Wektor2, konst. bezpar., 0xbff09b48
Wektor2, konst. (T,T), 0xbff09b40
Wektor2, konst. kop., 0xbff09ae0 0xbff09b20
Wektor2, operator=, 0xbff09b40
Wektor2, destructor, 0xbff09ae0
Wektor2, konst. kop., 0xbff09ae0 0xbff09b30
Wektor2, operator=, 0xbff09b48
Wektor2, destructor, 0xbff09ae0
```

```
Wektor2, destructor, 0xbff09b20
Wektor2, destructor, 0xbff09b30
1 2
3 4
1 2 3 4
Wektor2, destructor, 0xbff09b40
Wektor2, destructor, 0xbff09b48
Wektor2, destructor, 0xbff09b40
Wektor2, destructor, 0xbff09b50
Wektor2, destructor, 0xbff09b60
```

C++: Wzorce (templates)

Częściowa konkretyzacja wzorca ...

Wzorzec nakłada pewne wymagania na parametry wzorca (typy). Wymagania te są niejawne; w poniższym przykładzie T1 powinien mieć zdefiniowany operator ≪.
Problem ten można rozwiązać poprzez częściową konkretyzacje (specjalizację) wzorca i podanie kodu specyficznego dla danego typu parametru wzorca.

```
template <class T1, class T2>
struct CCC {
  T1 a; T2 b;
  void pisz();
};

template <class T1, class T2>  // def fun. skladowej poza klasa !!!
void CCC<T1,T2>::pisz() {
  cout<<"a="<<a<<endl;
  cout<<"b="<<b<<endl;
}
```

```
struct DDD { int p,q; };
  // nie ma: ostream& operator<< (ostream&, DDD&) { /**/ }

template <class T2>
struct CCC<DDD,T2> {
  DDD a; T2 b;
  void pisz() {
    cout<<"odzywa sie czesciowo skonkretyzowana klasa !!!"<<endl;
    cout<<"a.p="<<a.p<<endl;
    cout<<"a.q="<<a.q<<endl;
    cout<<"b="<<b<<endl;
  }
};

int main() {
  CCC<float,int> c1;
  c1.a=123.123F; c1.b=123;
  c1.pisz(); // ok

  int tab[]= {1,2,3};
  CCC<int*,float> c2;
  c2.a=tab; c2.b=321.321F;
  c2.pisz(); // ok
```

```
  DDD d1; d1.p=111; d1.q=222;
  CCC<DDD,int> c3;
  c3.a=d1; c3.b=333;
  c3.pisz();
   // BEZ czesciowo skonkretyzowanego wzorca CCC - blad kompilatora !!!!!!
     // gdyz DDD nie ma zdefiniowanego operatora <<

  // wydruk:
  a=123.123
  b=123
  a=0xbfedead0
  b=321.321
  odzywa sie czesciowo skonkretyzowana klasa !!!
  a.p=111
  a.q=222
  b=333
}
```

C++: Standard Template Library (STL)

Standardowa biblioteka C++ zawiera:

1. IOStreams

2. string
   - wreszcie wygodna obsluga stringów!!! (C tego nie ma)

3. STL/ **pojemniki** (**kontenery**)
   - klasy do przechowywania obiektów/danych tego samego typu (!)
   - przykłady kontenerów: **vector**, **map** (rodzaj słownika)
   - kontenery są zrealizowane jako wzorce klas
   - **iteratory**: uogólnienie wsk; służa do przebiegania po elementach kontenera
   - **algorytmy**: wykonuja typowe operacje na kontenerach, np copy() lub sort()
   - **adaptory**: przystosowują "niekompatybilny" kontener do współpracy z kontenerami STL

Biblioteka std. C++: std::string

```
#include <string>
#include <iostream>
using namespace std; // dzięki temu nie musimy pisac std::string

int main()
{ string s1="qwert";
    // konstruktor string(const char*)
  s1=s1+"12345";
    // string operator+(const string& s1, const string& s2 )
    // string& operator=( const string& s );
  cout<<"s1="<<s1<<endl;
    // ostream& operator<<( ostream& os, const string& s );
  cout<<"s1.length()="<<s1.length()<<endl;
    // metoda length() zwraca długość
  cout<<"s1.capacity()="<<s1.capacity()<<endl;
    // The capacity() function returns the number of elements
    // that the string can hold before it will need to allocate more space.

  printf("s1=%s\n", s1.c_str()); // "string C" ze "stringu C++"
  //printf("s1=%s\n", s1); // tak jest źle ...
```

```
  s1[3]='q'; // dostęp po jednym znaku
  cout<<s1[11]<<endl;
  try { cout<<s1.at(11)<<endl;
  } catch(const exception &e) { printf("catch exception, e.what()=%s\n", e.what());
     // tak jest bezpieczniej!
     // w razie błedu jest generowany wyjątek klasy exception...

  // ??? jak to rownie ogolnie zaprogramowac w C ???
  string s1;
  //cin>> s1; // czyta tylko do bialego znaku (!)
  getline(cin, s1); // czyta do znaku '\n'
  cout<<"odczytalem tekst "<<s1<<endl;
  cout<<"tekst w nawiasach "<<string("(")+s1+")"<<endl;
    // uwaga na konstruktor typu!
  cout<<"tekst w nawiasach "<<"("+s1+")"<<endl;
    // tak tez dziala !!! (konstruktor typu nie jest konieczny- dlaczego???)
}
```

Biblioteka std. C++: STL: kontener "vector", iteratory, algorytmy, ...

Naszym celem jest napisanie krótkiego programu, który: wczyta dowolny ciąg integerów, posortuje je, wypisze posortowane.

Głównym narzędziem jest kontener *vector* ...
- wektor to uogólniona tablica (tradycyjne tablice są przestarzałe !?!?!?)
- wektor może zawierać elementy dowolnego typu (wszystkie tego samego typu!)
- rozmiar wektora może wzrastać

```
  vector<float> tab(5); // tworzymy tablicę (początkowo) 5 floatów

  tab[0]=123.123; // modyfikujemy elementy tablicy
  tab[1]=321.321;
  tab[4]=321.321;

  vector<float>::iterator i1,i2; // iteratory
    // iteratory biegają po elementach wektora jak wsk
    // mają przeciążone operatory ++ i *
  i1=tab.begin();
  i2=tab.end(); // ZA ostatnim elementem wektora
  while(i1!=i2) { cout<<*i1<<endl; i1++; } cout<<endl;
    // wydruk:
123.123
321.321
0
0
321.321
```

```
  tab.insert(tab.begin(),111.222); // wstawiamy element przed wskazanym
  tab.insert(tab.end(),333.444);
  tab.push_back(444.555); // dopisujemy elementy na koncu
  tab.push_back(-123);

  i1=tab.begin(); i2=tab.end();
  while(i1!=i2) { cout<<*i1<<endl; i1++; } cout<<endl;
    // wydruk:
111.222
123.123
321.321
0
0
321.321
333.444
444.555
-123

  // różne obliczenia na wektorze ...
  i1=tab.begin(); i2=tab.end();
  cout<<"*min_element(i1,i2)="<<*min_element(i1,i2)<<endl;
  cout<<"*max_element(i1,i2)="<<*max_element(i1,i2)<<endl;

  tab.erase(tab.begin(), tab.end()); // usuwamy wszystkie elementy
  for(float f=0.0; f<100.0; f+=1.0) tab.push_back(f);
```

```
// rozwiązanie naszego problemu:
//
#include <iostream>
#include <vector>
#include <iterator>

int main ()
{
  vector<int> v; // (pusty) wektor integerów

  istream_iterator<int> start (cin); // początek kontenera
  istream_iterator<int> end; // za ostatnim elementem kontenera
    // istream_iterator<T> to "adaptor" przystosowujący istream do pracy jako kontener;
    // dokładnie, jest to iterator przebiegający po elementach kontenra
    // tak się ich używa: if(start!=end) { *start=123; start++; /*...*/ }

  back_insert_iterator<vector<int> > dest (v);
    // adaptor pozwalajacy wstawiac elementy na koniec wektora
    // przy pomocy iteratora; tak się go używa: *dest=123; dest++;

  copy (start, end, dest);
    // kopiuje liczby z cin na koniec wektora

  sort(v.begin(), v.end());
    // sortuje zawartość wektora
```

```
  copy (v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
    // ostream_iterator<int>(cout, "\n") to adaptor dla ostream;
    // seperatorem wyprowadzanych elementów będzie '\n'

}
```