STL – Standard Template Library

- **STL jest biblioteką kontenerów, algorytmów, iteratorów i wielu innych powszechnie używanych udogodnień programistycznych.**
- **STL jest wbudowany w standard języka C++, umożliwia więc tworzenie przenośnych programów**
- **STL zaprojektowano po to, by programiści nie wymyślali ciągle „prochu", „koła", itp.**
- **STL niemal w całości opiera się na szablonach.**
- **STL realizuje paradygmat programowania ogólnego (generic programming), w którym operuje się na ogólnych klasach typów.**
- **STL czyni z języka C++ jakościowo nowe narzędzie.**
- **STL nie ma nic wspólnego z programowaniem obiektowym.**

Podstawowe elementy STL:

- **KONTENERY służą do przechowywania obiektów. Posiadają standardowy interfejs poprzez który można manipulować obiektami. Przykłady kontenerów: wektor, lista, kolejka, mapa, zbiór.**
- **ALGORYTMY: określają standardowe operacje wykonywane na zawartości kontenerów lub obiektach. Niektóre algorytmy działają na każdym kontenerze, inne dostosowane są do konkretnych kontenerów. Posiadają dobrze określone właściwości (np. czas wykonania). Przykłady: sortowanie, usuwanie, zliczanie, porównywanie, wyszukiwanie obiektów w kontenerze; łączenie kontenerów. Permutacje, etc.**
- **ITERATORY: są uogólnieniem „zwykłych" wskaźników. Stanowią podstawowy sposób uzyskiwania dostępu do elementów kontenera. Powszechnie używane do definiowania zakresu („podkontenera"), na którym działają algorytmy. Każdy kontener można przeszukać stosując iterator (porządek liniowy). Istnieje kilka klas iteratorów: tylko-do-odczytu, tylko-do-przodu, tylko-do-tyłu, losowy-dostęp, etc. Iteratory wszystkich kontenerów mają bardzo podobny (lub identyczny) interfejs, np.:**
  - **Iterator wskazujący pierwszy element kontenera uzyskuje się poprzez metodę begin() kontenera, a metoda end() zwraca iterator wskazujący pierwszy element ZA ostatnim elementem kontenera.**
  - **Dostęp do elementu kontenera wskazywanego przez iterator uzyskujemy poprzez operator wyłuskania (*).**

Lista koktajli

```
#pragma warning( disable: 4786 )
// Bez powyższej dyrektywy program ten w VC++ 6.0 generuje
// multum ostrzeżeń nr 4786, które źle świadczą o kompilatorze

#include <iostream>
#include <string>
#include <list>

int main ()
{
    std::list<std::string> Koktaile;

    Koktaile.push_back("Czekoladowy");
    Koktaile.push_back("Truskawkowy");
    Koktaile.push_front("Cytrynowy");
    Koktaile.push_front("Waniliowy");

    Koktaile.push_front("Koktaile:");
    Koktaile.push_back("*** Koniec Menu ***");

    // dostep do elementow listy zapewnia iterator...
    // begin() zwraca iterator ustawiony na pierwszym elemencie
    // end() zwraca iterator ustawiony tuz za lista
    // operator++ przesuwa iterator do nastepnego elementu
    for (
         std::list<std::string>::iterator            // typ
           IteratorKoktaili = Koktaile.begin();      // obiekt
         IteratorKoktaili != Koktaile.end();         // warunek
         ++IteratorKoktaili                          // inkrementacja
        )
    {
        // wartość elementu listu wyłuskujemy (operator*) z iteratora
        std::cout << *IteratorKoktaili << std::endl;
    }
    return 0;
}

/* Output:
Koktaile:
Waniliowy
Cytrynowy
Czekoladowy
Truskawkowy
*** Koniec Menu ***
*/
```

Algorytm ogólny `for_each()`

```
#include <iostream>
#include <string>
#include <list>
#include <algorithm>   // Algorytmy ogolne (generic algorithms)

using namespace std;   // Żeby móc pominąć std::

int Drukuj (string& napis)
{
    cout << napis << endl;
    return 0;
}

int main (void)
{
    list<string>  Owoce_i_Warzywa;
    Owoce_i_Warzywa.push_back("marchew");
    Owoce_i_Warzywa.push_back("dynia");
    Owoce_i_Warzywa.push_back("pomidor");
    Owoce_i_Warzywa.push_front("jablko");
    Owoce_i_Warzywa.push_front("ananas");

    // piękna instrukcja:
    for_each ( Owoce_i_Warzywa.begin(), Owoce_i_Warzywa.end(), Drukuj );
    return 0;
}

/*
ananas
jablko
marchew
dynia
pomidor
*/
```

- **Dzięki zastosowaniu `for_each` kod jest „czystszy" i czytelniejszy.**
- **`for_each` wymaga podania zakresu: `K.begin()` ... `K.end()`.**

Dygresja: odczytywanie deklaracji szablonów funkcji i klas STL

Oryginalny prototyp funkcji `for_each`:

for_each

```
template<class InIt, class Fun>
    Fun for_each(InIt first, InIt last, Fun f);
```

„The template function evaluates `f(*(first + N))` once for each `N` in the range `[0, last - first)`. It then returns `f`. The call `f(*(first + N))` must not alter `*(first + N)`."

- **Widzimy, że `for_each` jest szablonem sparametryzowanym dwoma typami: `InIt` i `Fun`.**
  - **`InIt` oznacza „Input_Iterator" (iterator do odczytu)**
  - **`Fun` jest niemal dowolną funkcją: Z opisu dowiadujemy się bowiem, że `Fun` musi być funkcją przyjmującą dokładnie jeden argument typu iterowanego przez `InIt`, która ponadto nie modyfikuje tego argumentu. Typ zwracany przez `Fun` jest dowolny.**

Kategorie iteratorów:

- **Output (`Out`, `OutIt`): umożliwia zmianę zawartości elementu kontenera; najczęściej używany w strumieniach wyjścia.**
  Operacje: `*p=`, `++`
- **Input (`In`, `InIt`): umożliwia czytanie wartości elementu kontenera.**
  Operacje: `=*p`, `->`, `++`, `==`, `!=`.
- **Forward (`For`, `FwdIt`): Ma możliwości iteratorów Input i Output.**
  Przykład: lista wiązana pojedynczo.
  Operacje: `=*p`, `*p=`, `->`, `++`, `==`, `!=`.
- **Bidirectional (`Bi`, `BidIt`): Jak Forward, ale dwukierunkowy (`++` i `--`).**
  Przykład: lista podwójnie wiązana
  Operacje: `=*p`, `*p=`, `->`, `++`, `--`, `==`, `!=`.
- **Random Access (`Ran`, `RanIt`): Iterator o dostępie swobodnym.**
  Przykład: tablica.
  Operacje: `=*p`, `*p=`, `->`, `[]`, `++`, `--`, `+N`, `-N`, `+=N`. `-=N`, `==`, `!=`, `<`, `<=` `>`, `>=`.

Algorytm ogólny count

```
// Zliczamy ilość pewnych elementów na liście

#include <list>
#include <algorithm>
#include <iostream>

using namespace std;

int main ()
{
  list<int> Wyniki;

  Wyniki.push_back(100); Wyniki.push_back(80);
  Wyniki.push_back(45);  Wyniki.push_back(75);
  Wyniki.push_back(99);  Wyniki.push_back(100);

  int ileSetek =
    count (Wyniki.begin(), Wyniki.end(), 100);

  cout << "wynik 100 punktow osiagnelo "
       << ileSetek<< " zawodnikow" << endl;

  return 0;
}

/* OUTPUT:
wynik 100 punktow osiagnelo 2 zawodnikow
*/
```

- Algorytm `count` zwraca liczbę wystąpień danego elementu w kontenerze.

Algorytm `for_each` i obiekty funkcyjne („function objects”)

```
// Zliczamy elementy przy pomocy obiektu funkcyjnego

#include <string>
#include <list>
#include <algorithm>
#include <iostream.h>

using namespace std;

const string KodSzczoteczki("0003"); // globalna stała

// klasa obiektów funkcyjnych
class CzyToJestSzczoteczka
{
public:
  // koniecznie musimy zdefiniowac operator ()
  bool operator() ( string& opisTransakcji)
  {
     return
        opisTransakcji.substr(0,4) == KodSzczoteczki;
  }
};

// mniej trywialny obiekt funkcyjny – posiada wewnetrzną pamięć!
class CzyToJestSzczoteczka2
{
public:
  CzyToJestSzczoteczka2 (string& kod)
      : _kodPorownawczy(kod)
  {}
  bool operator() (string& opisTransakcji)
  {
    return
       opisTransakcji.substr(0,4) == _kodPorownawczy;
  }
private:
  string _kodPorownawczy;
};
```

```
int main (void)
{
    list<string> DziennikSprzedazy;

    DziennikSprzedazy.push_back("0001 Mydlo");
    DziennikSprzedazy.push_back("0002 Szampon");
    DziennikSprzedazy.push_back("0003 Szczoteczka");
    DziennikSprzedazy.push_back("0004 Pasta");
    DziennikSprzedazy.push_back("0003 Szczoteczka");

    int iloscSzczoteczek(0);
    iloscSzczoteczek =
      count_if (
                DziennikSprzedazy.begin(),
                DziennikSprzedazy.end(),
                CzyToJestSzczoteczka()  //<-- konstruktor!
               );

    cout << "metoda 1: Ilosc sprzedanych szczoteczek do zebow: "
         << iloscSzczoteczek << endl;

    string ZmiennyKod = "0003";
    iloscSzczoteczek =
      count_if (DziennikSprzedazy.begin(),
                DziennikSprzedazy.end(),
                CzyToJestSzczoteczka2(ZmiennyKod)
               );

    cout << "metoda 2: Ilosc sprzedanych szczoteczek do zebow: "
         << iloscSzczoteczek << endl;

    return 0;
}

/* OUTPUT:
metoda 1: Ilosc sprzedanych szczoteczek do zebow: 2
metoda 2: Ilosc sprzedanych szczoteczek do zebow: 2
*/
```


Algorytm ogólny `find`

```
#include <string>
#include <list>
#include <algorithm>
#include <iostream>

using namespace std;

int main (void)
{
    list<string> Owoce;
    list<string>::iterator IteratorOwocow;

    Owoce.push_back("Jablko");
    Owoce.push_back("Gruszka");
    Owoce.push_back("Banan");

    IteratorOwocow = find (Owoce.begin(),
                           Owoce.end(),
                           "Gruszka");

    if (IteratorOwocow == Owoce.end())
        cout << "Na liscie nie znaleziono poszukiwanego owocu\n";
    else
        cout << *IteratorOwocow << endl;

    return 0;
}


WYNIK:
Gruszka
```

- **Algorytm ogólny `find` poszukuje w danym zakresie kontenera pierwsze wystąpienie danego elementu. Wynikiem jego działania jest iterator do pierwszego poszukiwanego elementu lub iterator wskazujący na pierwszy element poza zakresem.**

Jak wykorzystać to, że algorytm `find` wymaga jedynie, by elementy kontenera można było porównywać operatorem==?

```
#include <string>
#include <list>
#include <algorithm>
#include <iostream>

class Owoc
{
public:
   Owoc(char const* s)  : _s(s) {  }
   bool operator== (Owoc const& rhs)
   {
      return rhs._s == _s || (jablko() && rhs.jablko() );
   }
   std::string const& Nazwa() const { return _s; }
private:
   bool jablko() const {return _s == "Papierowka" || _s == "Jablko";}
   const string  _s;
};

int main (void)
{
   list<Owoc> Owoce;
   list<Owoc>::iterator IteratorOwocow;

   Owoce.push_back("Gruszka");
   Owoce.push_back("Papierowka");
   Owoce.push_back("Jablko");

   IteratorOwocow = find (Owoce.begin(), Owoce.end(), "Jablko");

   if (IteratorOwocow == Owoce.end())
      cout << "Na liscie nie znaleziono poszukiwanego owocu\n";
   else
      cout << IteratorOwocow->Nazwa() << endl;
   return 0;
}
```

WYNIK:
Papierowka

- **W ten sam sposób można zastosować praktycznie dowolny algorytm ogólny do kontenerów obiektów klas zdefiniowanych przez użytkownika**

Algorytm ogólny `find_if`

```
#include <string, list, algorithm, iostream>
using namespace std;

struct ZdarzenieWRoku1997
{
   bool operator () (string const& zdarzenie)
   {
      // zapis roku rozpoczyna sie od znaku nr 12 i zajmuje 4 znaki
      return zdarzenie.substr(12,4)=="1997";
   }
};

int main ()
{
   list<string> Dziennik;

   Dziennik.push_back("07 stycznia 1995  Przygotowano szkic projektu domu");
   Dziennik.push_back("07 lutego   1996  Przygotowano szczegółowy projekt domu");
   Dziennik.push_back("13 stycznia 1997  Klient podpisuje umowe");
   Dziennik.push_back("18 stycznia 1997  Rozpoczecie prac budowlanych");
   Dziennik.push_back("30 kwietnia 1998  Zakonczenie prac");

   for (list<string>::iterator Iterator =
            find_if (Dziennik.begin(), Dziennik.end(), ZdarzenieWRoku1997() );
        Iterator != Dziennik.end();
        Iterator = find_if (Iterator, Dziennik.end(), ZdarzenieWRoku1997() ) )
   {
      cout << *Iterator++ << endl;
   }
   return 0;
}
```

WYNIK
13 stycznia 1997 Klient podpisuje umowe
18 stycznia 1997 Rozpoczecie prac budowlanych

- **`find_if` działa jak `find`, tyle, że jako 3 argument przyjmuje predykat (obiekt funkcyjny lub funkcję zwracającą wartość typu *bool*)**
- **Iterator w powyższym przykładzie zachowuje się jak zwykły wskaźnik! Można na nim wykonywać operacje `!=`, `++`, `*`**

Algorytm ogólny search

```
#include <string>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

int main ()
{
   list<char> PoszukiwanyCiag;
   list<char> DanyCiagZnakow;

   PoszukiwanyCiag.push_back('\n');
   PoszukiwanyCiag.push_back('\n');

   DanyCiagZnakow.push_back('a');
   DanyCiagZnakow.push_back('A');
   DanyCiagZnakow.push_back('\n');
   DanyCiagZnakow.push_back('\n');
   DanyCiagZnakow.push_back('\n');

   list<char>::iterator PozycjaPustegoWiersza =
      search(DanyCiagZnakow.begin(), DanyCiagZnakow.end(),
           PoszukiwanyCiag.begin(), PoszukiwanyCiag.end());

   if (PozycjaPustegoWiersza != DanyCiagZnakow.end())
      cout << "Znalazlem pusty wiersz!" << endl;
   return 0;
}
```

- `search` **poszukuje w określonym ciągu wystąpienia danego <u>podciągu</u> i zwraca iterator do pierwszego elementu odpowiedniego „podciągu w ciągu" lub do pierwszego elementu poza zakresem.**

Algorytm search z jawnym predykatem

- **Problem: znaleźć na liście pierwsze wystąpienie dwóch cyfr pod rząd**

```
#include <string, list, algorithm, iostream>
#include <cctype>
using namespace std;

bool porownaj(char c1, char c2)          // ← to jest ten „jawny predykat"
{
   return isdigit(c1) && isdigit(c2);
}

int main ()
{
   list<char> DanyCiagZnakow;
   list<char> Ciag2Cyfry;

   Ciag2Cyfry.push_back('0');
   Ciag2Cyfry.push_back('0');

   DanyCiagZnakow.push_back('a');
   DanyCiagZnakow.push_back('7');
   DanyCiagZnakow.push_back('1');
   DanyCiagZnakow.push_back('b');

   list<char>::iterator Pozycja2Cyfr =
      search(DanyCiagZnakow.begin(), DanyCiagZnakow.end(),
           Ciag2Cyfry.begin(), Ciag2Cyfry.end(), porownaj); // ← użycie jawnego predykatu

   if (Pozycja2Cyfr != DanyCiagZnakow.end())
      cout << "Znalazlem 2 cyfry pod rzad: pierwsza z nich to "
           << *Pozycja2Cyfr << endl;
   return 0;
}
```

WYNIK:
Znalazlem 2 cyfry pod rzad: pierwsza z nich to 7

Sortowanie listy

```
#include <string, list, algorithm, iostream>

void Drukuj (string& napis)
{
   cout << napis << ", ";
}

int main (void)
{
 list<string> Kompozytorzy;
 list<string>::iterator IteratorKompozytorow;

 Kompozytorzy.push_back("Szopen");
 Kompozytorzy.push_back("Wieniawski");
 Kompozytorzy.push_back("Penderecki");
 Kompozytorzy.push_back("Gorecki");
 Kompozytorzy.push_back("Kazik");

 cout << "Lista przed posortowaniem:\n";
 for_each( Kompozytorzy.begin(), Kompozytorzy.end(), Drukuj );

 Kompozytorzy.sort();

 cout << "\nLista po posortowaniu:\n";
 for_each(Kompozytorzy.begin(), Kompozytorzy.end(), Drukuj);
 cout << "\n";
 return 0;
}
```

```
WYNIK:
Lista przed posortowaniem:
Szopen, Wieniawski, Penderecki, Gorecki, Kazik,
Lista po posortowaniu:
Gorecki, Kazik, Penderecki, Szopen, Wieniawski,
```

- **Do sortowania list używamy metody sort.**
- **Istnieje też ogólny („globalny”) algorytm sort, jednak wymaga on zastosowania iteratorów o dostępie swobodnym, a lista takich nie dostarcza.**

Sortowanie tablic

```
#include <iostream>
#include <vector>
#include <algorithm>

bool porownaj(int x, int y)   // funkcja porównawcza
{
   return abs(x) < abs(y);
}

void drukuj (int tab[], int ile)
{
   for (int i = 0; i < ile; i++)
      cout << tab[i] << " ";
   cout << "\n";
}

void drukuj (vector<int> const&  v)
{
   for (int i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << "\n";
}

int main()
{
   int tab[] = {1, -5, 78, 3, -10};

   drukuj (tab, 5);
   sort (tab, tab+5);
   drukuj (tab, 5);
   sort (tab, tab+5, porownaj);
   drukuj (tab, 5);
   vector<int> v (tab, tab+5);  // kopiuje tablice tab na nowotworzony wektor v
   sort (v.begin(), v.begin() + 3);   // sortuje 3 pierwsze elementy vektora v
   drukuj(v);
   return 0;
}
```

```
WYNIK:
1 -5 78 3 -10
-10 -5 1 3 78
1 3 -5 -10 78
-5 1 3 -10 78
```

Sortowanie tablic w języku C

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

int porownaj(const void* px, const void* py)
{
    int x = *((int*)px);
    int y = *((int*)py);
    x = abs(x);
    y = abs(y);
    if ( x < y ) return -1;
    if ( x == y ) return 0;
    return 1;
}

void drukuj (int tab[], int ile)
{
    for (int i = 0; i < ile; i++)
        printf("%d ",tab[i]);
    printf("\n");
}

int main()
{
    int tab[] = {1, -5, 78, 3, -10};

    drukuj (tab, 5);
    qsort(tab, 5, sizeof(tab[0]), porownaj);
    drukuj (tab, 5);
    return 0;
}
```

- **W języku C jest dostępna jedna funkcja sortująca (`qsort`), która potrafi posortować tablicę obiektów dowolnego typu.**
- **Ponieważ jej twórcy nie wiedzieli, co będzie przy jej pomocy sortowane, musimy jej podać: adres początku tablicy, ilość jej elementów, rozmiar elementu w bajtach, funkcję sortującą działającą na adresach i „odartą" z wszelkiej kontroli typów.**
- **`void*` to „wskaźnik do czegokolwiek".**
- **W implementacji funkcji sortującej należy dokonywać jawnych konwersji typów jej argumentów.**
- **Funkcja sortująca zwraca -1 (l < p), 0 (l == p) lub 1 (l > p).**
- **`std::sort` jest z zasady łatwiejsze w użyciu i szybsze niż `qsort`.**
- **Inną ogólną standardową funkcją języka C jest `bsearch` (*binary serach*)**

Odczytywanie deklaracji kolekcji standardowych

- **Kolekcje standardowe w standardowy sposób przechowują informacje o używanych przez siebie typach:**

```
template<class T, class A = allocator<T> >
class vector
{
public:
  typedef A allocator_type;
  typedef A::size_type size_type;
  typedef A::difference_type difference_type;
  typedef A::reference reference;
  typedef A::const_reference const_reference;
  typedef A::value_type value_type;
  typedef T0 iterator;                          // typ T0 jest zależny od implementacji
  typedef T1 const_iterator;                    // typ T1 jest zależny od implementacji
  typedef reverse_iterator <iterator, value_type, reference, A::pointer,
                            difference_type> reverse_iterator;
  typedef reverse_iterator<const_iterator, value_type, const_reference,
                    A::const_pointer, difference_type> const_reverse_iterator;
...
```

- **Ostatnim argumentem (domyślnym) kontenerów standardowych jest alokator, czyli abstrakcyjny typ służący do zarządzania pamięcią.**
- **Używaj alokatora standardowego (domyślnego), czyli o nim zapomnij:**

```
std::vector<int> v;  // zostanie użyty alokator standardowy
```

- **„Pomyśl dwa razy, zanim zaczniesz pisać swój alokator" (B. Stroustrup).**
- **Oto fragment definicji standardowego alokatora:**

```
template<class T>
class allocator
{
  typedef size_t size_type;          // typ „indeksu tablicy"
  typedef ptrdiff_t difference_type;  // typ „różnicy wskaźników"
  typedef T *pointer;
  typedef const T *const_pointer;
  typedef T& reference;
  typedef const T& const_reference;
  typedef T value_type;
...
```

Do czego może się przydać informacja o typach?

- **Oto szablon funkcji, która oblicza sumę wszystkich elementów dowolnego standardowego kontenera:**

```
#include <vector>
#include <list>
#include <iostream>

template <class T>
typename T::value_type suma (const T& kontener)
{
   typename T::value_type wynik = 0;
   typename T::const_iterator p = kontener.begin();
   while (p != kontener.end())
   {
      wynik += *p;     // lepsze niż *p++, bo przyrostkowy++ może być nieefektywny
      ++p;
   }
   return wynik;
};


int main()
{
   int tmp [] = {1, 4, -1, 19, 11, 16};
   //„sztuczkę" z opeartorem sizeof warto zapamiętać:
   std::vector<int>  v( tmp, tmp + sizeof(tmp)/sizeof(tmp[0]) );
   std::list<int> lista (v.begin(), v.end());
   std::cout << suma(v) << "\n";              // 50
   std::cout << suma(lista) << "\n";          // 50
   return 0;
}
```

Uwagi:

- **Słowo kluczowe typename umożliwia kompilatorowi odróżnianie składowej, definiującej typ, od składowej, przechowującej dane.**
- **Operator sizeof zwraca ilość bajtów zajmowanych przez zmienną, obiekt, strukturę lub tablicę alokowaną statycznie.**
- **Unikaj używania przyrostkowych operatorów ++ i - -.**

Konstruktory kontenerów

```
template<class T, class A = allocator<T> >
class vector {
public:
   // pusty kontener
   explicit vector( const A& al = A() );

   // wektor n elementów o wartości v (dla typów wbudowanych domyślnie v = 0)
   explicit vector( size_type n, const T& v = T(), const A& al = A() );

   // konstruktor kopiujący
   vector( const vector& x );

   // konstruktor kontenera o elementych skopiowanych z innego kontenera
   vector( const_iterator first, const_iterator last, const A& al = A() );
...
};
```

- **Przykłady:**

```
std::vector<int>  v;             //pusty wektor
std::list<int>    lista (7,2);   // lista 7 dwójek
std::vector<int>  lz(100);       // wektor 100 zer
std::vector<int>  w(v);          // kopia wektora v
std::list<int>    q(w.begin(), w.end())  // „kopia" wektora w
```

Konstruktory jawne (explicit)

- **Naiwny wielomian:**

```
#include <iostream, ostream, vector>

class Wielomian : public std::vector<int>
{
public:
   Wielomian(std::vector<int> const& v):  std::vector<int>(v)    { }
   int operator() (int x) const
   {
     int stopien = size()-1;
     int wynik = operator[](stopien);
//   int wynik = (*this)[stopien];
     for (int i = stopien - 1; i >= 0; i--)
        wynik = wynik*x + at(i);
     return wynik;
   }
}

std::ostream& operator<< (std::ostream& F, Wielomian const& w)
{
   F << '(';
   for (int i = 0; i < w.size() - 1; i++)
      F << w[i] << ", ";
   if (!w.empty())
      F << w.back();
   F  << ')';
   return F;
}

int main()
{
   int tmp[3] = {1,2,3};
   std::vector<int> v (tmp, tmp+3);
   Wielomian w (v);

   std::cout << "wielomian w = " << w << "\n";
   std::cout << "wektor v =    " << v << "\n";
   std::cout << "w(1) = " << w(1) << "\n"  << "w(-1) = " << w(-1) << "\n";
   return 0;
}
```



```
//OUTPUT:
wielomian w = (1, 2, 3)      // ← OK., operator<< (ostream& F, Wielomian const& w)
wektor v =    (1, 2, 3)      // ← a skąd kompilator wie, jak wyswietlać wektory????
w(1) = 6
w(-1) = 2
```

- **Wniosek: Każdy konstruktor jednoargumentowy definiuje niejawny operator konwersji!!! To może prowadzić do nieoczekiwanych efektów.**
- **Opracowując wyrażenia, kompilator może (w ostateczności) zastosować jedną konwersję zdefiniowaną przez użytkownika.**
- **Aby wykluczyć tę nieprzyjemną możliwość, należy konstruktor zdefiniować jako „explicit”:**

```
class Wielomian : public std::vector<int>
{
public:
   explicit Wielomian(std::vector<int> const& v)
                        :  std::vector<int>(v)
                        {  }
   int operator() (int x) const;
};
```

Pozostałe metody kontenera std::vector

```
void reserve (size_type n);              // rezerwacja pamięci
size_type capacity () const;             // max. długość wektora bez realokacji
void resize (size_type n, T x = T());    // zmiana długości wektora + inicjacja
size_type size  () const;                // bieżąca długość wektora
size_type max_size () const;             // maksymalna (fizycznie) długość wektora
bool empty () const;                     // czy wektor jest pusty?

iterator insert (iterator it, const T& x = T());   // wstaw x przed it
void insert (iterator it, size_type n, const T& x); // wstaw n x-ów przed it
void insert (iterator it,
    const_iterator first, const_iterator last);  // wstaw [*first,..,*last) przed it
iterator erase (iterator it);            // usuń *it
iterator erase (iterator first, iterator last);       // usuń [*first,...,*last)
void clear ();                           // erase( begin(), end() )

iterator begin ();                       // iterator do piewszego elementu
const_iterator begin () const;           // stały iterator do pierwszego elementu
iterator end ();                         // iterator do elementu "1 za daleko"
iterator end() const;                    // stały iterator do elementu "1 za daleko"
reverse_iterator rbegin ();              // iterator odwrotny do "1-go" elementu
const_reverse_iterator rbegin () const; // stały -,,-
reverse_iterator rend ();                // iterator odwrotny do "końca"
const_reverse_iterator rend () const;    //stały -,,-

A get_allocator () const;                // Zwróć alokator

reference at (size_type pos);            // Kontrolowany dostęp do elementu
const_reference at  (size_type pos) const; // -,,- read-only
reference operator[] (size_type pos);    // niekontrolowany dostęp
const_reference operator[] (size_type pos); // -,,- read-only
reference front ();                      // pierwszy element
const_reference front () const;          // -,,- read-only
reference back ();                       // ostatni element
const_reference back () const;           // ostatni element read-only
void push_back (const T& x);             // resize() + wstaw na koniec wektora
void pop_back ();                        // usuń z końca wektora (+ resize())

void assign (const_iterator first, const_iterator last); // jak konstruktor
void assign (size_type n, const T& x = T());   // jak konstruktor
void swap (vector & x);                  // zamień *this i x
```


Wstawiacze

Obiekty klasy `back_insert_iterator` udają iteratory, a naprawdę służą do „wstawiania" wartości `val` przekazywanej w operacji
`*iterator++ = val`
na końcu kontenera operacją `push_back`. W ten sposób wirtualnie „wydłużają" istniejący kontener do „nieskończoności".
Przykład: niech `v` i `w` będą niepustymi wektorami typu `std::vector<int>`.
Wtedy

```
std::copy(v.begin(), v.end(), back_insert_iterator<std::vector<int> >(w) );
```

dołącza zawartość wektora `v` na końcu wektora `w` (powodując rozszerzenie `w`).

Jak to jest możliwe?

```
// przykładowa implementacja
#include <iostream>
#include <cstdlib>
#include <vector>

template<class KONTENER>
class back_insert_iterator
{
public:
   explicit back_insert_iterator (KONTENER & kontener)
   : _kontener(kontener)
   { }
   back_insert_iterator& operator*()          { return *this; }
   back_insert_iterator& operator++()         { return *this; }
   back_insert_iterator& operator++(int)      { return *this; }

   back_insert_iterator&
   operator= (const typename KONTENER::value_type & x)
   {
     _kontener.push_back(x);
     return *this;
   }
protected:
   KONTENER & _kontener;
};
```

Klasa `back_insert_iterator` ma fundamentalną wadę: nieludzko skomplikowaną składnię.

Na pomoc wzywamy szablony funkcji!!

```
template <class KONTENER>
inline back_insert_iterator<KONTENER>
back_inserter(KONTENER & kontener)
{
   return back_insert_iterator<KONTENER> (kontener);
}
```

Teraz zamiast

```
std::copy(v.begin(), v.end(), std::back_insert_iterator<std::vector<int> > (w) );
```

możemy napisać po prostu

```
std::copy(v.begin(), v.end(), back_inserter(w) );
```

Szablony funkcji są w STL często używane do upraszczania zapisu

W przeciwieństwie do szablonów klas nie wymagają bowiem od użytkownika jawnego podawania klas, parametyryzujących szablon. Uzyskanie takich informacji mogłoby być pracochłonne i zniechęcać do używania STL.

W funkcji `back_inserter` wykorzystuje się to, że kompilator może automatycznie odczytać typ argumentów szablonu funkcji i na tej podstawie samodzielnie wygenerować pożądany szablon klasy, parametrtyzując go typami odczytanymi z argumentów funkcji.

Takie funkcje nazywamy „wrapperami" (*„opakowaniami"*).

Oryginalne wstawiacze STL znajdują się w nagłówku <iterator>

Wstawiacze biblioteki STL

iterator	Wrapper	działanie instrukcji ???insert_iterator = value;
back_insert_iterator	back_inserter(KONTENER)	container.push_back(value);
front_insert_iterator	front_inserter(KONTENER)	container.push_front(value);
insert_iterator	inserter(KONTENER, KONTENER::iterator)	container.insert(container_iterator, value);

Przykład zastosowania wstawiacza `back_inserter`

```
template<typename T>
std::ostream& operator<< (std::ostream& F, const std::vector<T>& vec)
{
   F << "(";
   for (unsigned i = 0; i < vec.size(); ++i)
   {
      F << vec[i] << ", ";
   }
   F << "\b\b)"; // nieelegenckie
   return F;
}


int main(int argc, char *argv[])
{
  std::vector<int> v;                    // pusty wektor liczb całkowitych
  v.push_back(0);                        // wektor v został rozszerzony; v[0] == 0;
  std::back_insert_iterator<std::vector<int> > back_iter(v);   // definicja wstawiacza
  *back_iter++ = 1;                      // równoważne instrukcji v.push_back(1);
  *back_iter++ = 2;                      // równoważne instrukcji v.push_back(2);

  std::cout << "v = " << v << "\n";     // "v = (0, 1, 2)"

  int tab[] = {10,20,30};
  std::copy(tab, tab+3, back_iter);

  std::cout << "v = " << v << "\n";     // "v = (0, 1, 2, 10, 20, 30)"

  std::copy(tab, tab+3, back_inserter(v)); // wyjątkowo przedrostek std:: niepotrzebny!

  std::cout << "v = " << v << "\n";     // "v = (0, 1, 2, 10, 20, 30, 10, 20, 30)"

  system("PAUSE");
  return 0;
}
```

WYNIK:

```
v = (0, 1, 2)
v = (0, 1, 2, 10, 20, 30)
v = (0, 1, 2, 10, 20, 30, 10, 20, 30)
```

Iteratory strumieni wejścia i wyjścia

W bibliotece STL strumienie wejścia/wyjścia można traktować jak zwykłe kontenery.

W przypadku strumieni wejścia cel ten realizują iteratory „tylko do odczytu”, `istream_iterator`
a w przypadku strumieni wyjścia – iteratory „tylko do zapisu”, `ostream_iterator`

Dzięki tej właściwości instrukcje

```
vector<int> V;
copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(V));
```

powodują skopiowanie liczb całkowitych ze standardowego wejścia do początkowo pustego wektora `V`. Z kolei instrukcje

```
std::vector<int> V;
// ...
std::copy(V.begin(), V.end(), std::ostream_iterator<int>(std::cout, "\n") );
```

powodują skopiowanie zawartości wektora `V` na standardowe wyjście (po jednej liczbie w wierszu).

- **Aby użyć `istream_iterator` lub `ostream_iterator`, należy podać**
  a) z jakim strumieniem ma być związany dany iterator (np. cin)
  b) jaki jest typ elementów w strumieniu-kontenerze

  `istream_iterator<int>(cin)`

- **Dodatkowo iterator `ostream_iterator` umożliwia zdefiniowanie separatora danych:**

  `ostream_iterator<int>(std::cout, "\n")`

- **Bezargumentowy konstruktor iteratora `istream_iterator` tworzy iterator wskazujący „poza” kontener (= koniec pliku lub błąd).**

  `istream_iterator<int>()`

- **Neleży pamiętać, że operacje na iteratorach strumieni wejścia/wyjścia faktycznie realizowane są poprzez operatory `operator>>` lub `operator<<`. Dlatego np. stadardowo na wejściu pomijane są wszystkie białe znaki.**

Przykłady użycia iteratorów strumieni wejścia/wyjścia

Przykład 1

```
vector<int> v;
copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
copy(v.rbegin(), v.rend(), ostream_iterator<int>(cout, "\n"));
```

wejście (cin):
```
1 2 34   5    6 Antek
```
wyjście (cout):
```
653421
```

W powyższym kodzie wczytano 5 liczb typu `int` (aż do wystąpienia błędu przy próbie wczytania litery `'A'`), po czym przy pomocy iteratorów odwrotnych (`rbegin`, `rend`) wypisano je w odwrotnej kolejności, bez separatorów, w strumieniu `cout`.

Przykład 2

```
ifstream F("dane1.dat");
ifstream G("dane2.dat");
ofstream SUMA("suma.dat");
transform (istream_iterator<int>(F),              // początek źródła danych nr 1
        istream_iterator<int>(),                  // koniec źródła danych nr 1
        istream_iterator<int>(G),                 // początek źródła danych nr 2
        ostream_iterator<int>(SUMA, "\t"),        // gdzie zapisywać wyniki transformacji
        plus<int>() );              // co robić z parami obiektów pobieranych z wejścia?
```

wejście ("dane1.dat"):
```
1 2 3    4
```
wejście ("dane2.dat"):
```
1 4 9 16
```
wyjście ("suma.dat"):
```
2   6   12  20
```