Krótkie wprowadzenie do STL

XIV LO im. S. Staszica, K06_D

Arkadiusz Betkier Wojciech Sirko

30 marca 2010

Spis treści

1 Wstęp

STL (Standard Template Library) jest biblioteką języka C++ zawierającą implementacje pewnych przydatnych algorytmów i struktur danych. Niniejszy dokument stanowi jedynie, poddane pewnym uproszczeniom i przemilczeniom, wprowadzenie – dokumentacja STL znajduje się na stronie SGI (`http://www.sgi.com/tech/stl/`) i jest znacznie obszerniejsza.

STL zawdzięcza swoją użyteczność m.in. zastosowaniu szablonów. Są one uogólnieniem klas lub funkcji, których właściwy kod powstaje w czasie kompilacji, po podaniu przez programistę pewnych parametrów (parametry te mogą być typami danych, klasami, czy stałymi liczbowymi). Szablony pozwalają programiście zaoszczędzić czas, gdyż umożliwiają wykorzystanie pewnej ogólnej implementacji do konkretnego zastosowania bez konieczności kopiowania kodu, by np. zmienić użyty typ danych.

Mimo licznych zalet STL'a, nadużywanie go może prowadzić do niepożądanych skutków, takich jak niepotrzebnie skomplikowane czy niewydajne rozwiązania. Zaleca się więc umiar i rozwagę.

```
// Tak moga wygladac szablony.

// Szablon klasy AB.
template<class T> class AB
{
public:
    AB(T a, T b) : a(a), b(b) {}
    T sum() { return a+b; }
private:
    T a, b;
};

// Szablon funkcji abs.
template<class T> T abs(T a)
{
    if (x < 0)
        return -x;
    else
        return x;
}

// W przypadku szablonow klas nalezy wyraznie
// podac parametry szablonu w nawiasach ostrych.
AB<int> abi(1, 3);
abi.sum();  // zwroci 4
AB<float> abf(1.4f, 2.5f);
abf.sum();  // zwroci 3.9f

// W przypadku szablonow funkcji parametry sa
// rozpoznawane na podstawie podanych
// argumentow funkcji.
abs(-4);    // zwroci 4
abs(-3.9f)  // zwroci 3.9f
```

2 Podstawowe struktury danych

Poniższe opisy struktur składają się z tekstu, tabelek z kluczowymi (nie wszystkimi) operacjami i przykładowych kodów. Tabelki zawierają nazwy operacji, skrótowe opisy i złożoność obliczeniową. Typu wartości operacji nietrudno się domyślić, natomiast w kwestii argumentów przyjęta jest taka konwencja: „`v`" oznacza wartość typu takiego, jaki jest trzymany w strukturze, „`k`" oznacza indeks – można przyjąć, że jest to `int`, natomiast „`i`" oznacza *iterator* – więcej o tym przy listach.

Pomijając `pair` – przypadek zdegenerowany, wszystkie wymienione poniżej struktury udostępniają dodatkowo operację `clear()`, która działa w czasie liniowym i odpowiada za usunięcie wszystkich elementów ze struktury oraz operację `empty()`, która określa (w czasie stałym), czy struktura jest pusta. Oczywiście wszystkie złożoności obliczeniowe są podane względem liczby elementów w strukturze.

Ku przejrzystości przykładowego kodu (i aby nakłonić czytelnika do niestosowania metody copy-paste bez pełnego zrozumienia) nie będą się w nim pojawiać instrukcje wypisywania na wyjście. Zamiast tego pojawiać się będą wyrażenia zakończone średnikiem i opatrzone komentarzami określającymi ich wartości. Liczy się przy tym na domyślność czytelnika. Ponadto zakłada się, że przed miejscem użycia kodu wpisano `using namespace std`, w przeciwnym przypadku trzeba podopisywać `std::` w odpowiednie miejsca.

2.1 pair

Obiekt `pair` przechowuje dwa elementy – `first` i `second` (dostępne wprost). Funkcja `make_pair` tworzy odpowiednią parę. Oczywiście można też skorzystać ze standardowego konstruktora.

```
#include <utility>
// ...
pair<float, int> para(0.1f, 9);
para.first; // -> 0.1f
para.second; // -> 9
para.first = 0.5f;
para.second = 3;
// pow to procedura od potegowania
// trzeba dopisac #include <cmath>
pow(para.first, para.second); // -> 0.125f
para = make_pair(2.0f, 10);
pow(para.first, para.second); // -> 1024.0f
```

2.2 stack

Struktura `stack` to STL'owy stos. Stos, jaki jest, każdy widzi – można dokładać na wierzch, oglądać wierzch i z niego zdejmować, ale żeby nagle dostać się gdzieś do środka, trzeba się na ogół nieźle napracować.

`size()`	rozmiar stosu	$O(1)$
`push(v)`	dodawanie na wierzch	$O(1)$
`pop()`	zrzucanie z wierzchu	$O(1)$
`top()`	element na wierzchu	$O(1)$

```
#include <stack>
// ...
// typ int jest po to, aby przyklad uproscic
// moznaby zamiast tego np. utoworzyc stos
// stack< pair< stack<float>, int > >
// szablony mozna wiec bez problemu skladac
stack<int> stos;
stos.push(123);
stos.push(456);
stos.push(789);
stos.size(); // -> 3
while (!stos.empty())
{
    stos.top(); // -> { 789, 456, 123 }
    stos.pop();
}
```

2.3 queue

Struktura `queue` odpowiada kolejce (takiej typowej, jak w sklepiku po pączki). Można więc wstawić element na koniec, zrzucić z początku i obejrzeć ten na początku i ten na końcu. Jednak, tak jak poprzednio, nie tak łatwo dostać się do środka.

`size()`	rozmiar kolejki	$O(1)$
`push(v)`	dodawanie na koniec	$O(1)$
`pop()`	zrzucanie z początku	$O(1)$
`front()`	element na początku	$O(1)$
`back()`	element na końcu	$O(1)$

```
#include <queue>
// ...
queue<int> kolejka;
kolejka.push(123);
kolejka.push(456);
kolejka.push(789);
kolejka.size(); // -> 3
while (!kolejka.empty())
{
    kolejka.front(); // -> { 123, 456, 789 }
    kolejka.pop();
}
```

2.4 list

Więcej operacji udostępnia `list` – lista dwukierunkowa. Można wstawiać, oglądać i usuwać elementy z końców. Ponadto można listę przeglądać, zarówno od początku do końca jak i przeciwnie. Co więcej (i to jest kluczowa własność listy), można usunąć element z dowolnego miejsca, jak również wstawić element w dowolne miejsce, pod warunkiem posiadania pod ręką odpowiedniego wskaźnika. Taki wskaźnik w STL'u nazywa się *iteratorem* i jest dodatkowo wyposażony w instrukcję zwiększania (zmniejszania) odpowiednią dla konkretnej struktury danych. W przypadku listy zwiększenie oznacza przeskok do następnego elementu na liście. Iterator na pierwszy element uzyskuje się wywołując na liście `begin()`, a na za-ostani-element wywołując `end()` – będzie to widać w przykładzie.

`size()`	rozmiar listy	$O(?)$[a]
`push_back(v)`	dodawanie na koniec	$O(1)$
`pop_back()`	zrzucanie z końca	$O(1)$
`back()`	element na końcu	$O(1)$
`push_front(v)`	dodawanie na początek	$O(1)$
`pop_front()`	zrzucanie z początku	$O(1)$
`front()`	element na początku	$O(1)$
`erase(i)`	usuwanie wskazanego elementu	$O(1)$
`insert(i, v)`	wstawianie przed wskazany element	$O(1)$

[a]Nie ma gwarancji, że jest to $O(1)$ – równie dobrze może być $O(n)$. Zależy to od konkretnej implementacji STL, dlatego lepiej nie używać `size` w tym przypadku.

```
#include <list>
// ...
list<int> lista; // na poczatku pusta lista
list<int>::iterator it; // to sie jeszcze przyda
list<int>::iterator cit; // to tez
// wpierw zwykle dodawanie na konce
for (int x = 0; x < 5; x++)
{
    lista.push_back(x + 5);
    lista.push_front(-x * 2);
}
// po tej operacji lista wyglada tak:
// { -8, -6, -4, -2, 0, 5, 6, 7, 8, 9 }
lista.size(); // -> 10
for (it = lista.begin(); it != lista.end(); it++)
   *it; // -> { -8, -6, -4, -2, 0, 5, 6, 7, 8, 9 }
// usuwanie z brzegu...
lista.pop_front();
lista.pop_back();
lista.front(); // -> -6
lista.back(); // -> 8
// znajdzmy cos...
for (it = lista.begin(); it != lista.end(); it++)
    if (*it % 2 == 1)
        cit = it, break;
// teraz cit wskazuje na pierwszy nieparzysty
// element na liscie (5)
// (na szczescie taki istnieje,
// inaczej zrobilibysmy tu powazny blad)
// ponadto cit == it
lista.erase(++it);
// zobaczmy, co sie stalo z lista -- usunelismy
// nastepnik pierwszego nieparzystego elementu
for (it = lista.begin(); it != lista.end(); it++)
    *it; // -> { -6, -4, -2, 0, 5, 7, 8 }
// teraz wstawimy 6 z powrotem, ale tym razem
// przed 5 (cit nadal wskazuje na 5)
lista.insert(cit, 6);
for (it = lista.begin(); it != lista.end(); it++)
    *it; // -> { -6, -4, -2, 0, 6, 5, 7, 8 }
```

2.5 vector

Twór o nazwie `vector` łączy zalety stosu i tablicy. Odpowiada `stack` w sensie dodawania i usuwania elementów, ale pozwala na bezpośrednie odwoływanie się do każdego z nich – tak, jak w zwykłej tablicy. Idea kryjąca się w implementacji tego tworu jest dość prosta – stwórz tablicę, a gdy miejsce się w niej skończy, stwórz dwa razy większą tablicę i przekopiuj elementy ze starej do nowej, itd...

`size()`	rozmiar	$O(1)$
`push_back(v)`	dodawanie na koniec	$O(1)$
`pop_back()`	zrzucanie z końca	$O(1)$
`back()`	element na końcu	$O(1)$
`[k]`	referencja na k-ty element	$O(1)$

```
#include <vector>
// ...
vector<int> vec;
vec.push_back(123);
vec.push_back(456);
vec.push_back(789);
for (int i = 0; i < vec.size(); i++)
    vec[i]; // -> { 123, 456, 789 }
vec[1] = 666;
while (!vec.empty())
{
    vec.back(); // -> { 789, 666, 123 }
    vec.pop_back();
}
```

2.6 deque

Bardziej wyrafinowaną strukturą jest `deque`, która z zewnątrz zachowuje się podobnie jak `vector`, ale pozwala wstawiać i usuwać z obu stron – z końca i z początku. Innymi słowy, jest to kolejka dwustronna, która dodatkowo umożliwia bezpośrednie odwoływanie się do każdego elementu.

`size()`	rozmiar	$O(1)$
`push_back(v)`	dodawanie na koniec	$O(1)$
`pop_back()`	zrzucanie z końca	$O(1)$
`back()`	element na końcu	$O(1)$
`push_front(v)`	dodawanie na początek	$O(1)$
`pop_front()`	zrzucanie z początku	$O(1)$
`front()`	element na początku	$O(1)$
`[k]`	referencja na k-ty element	$O(1)$

```
#include <deque>
// ...
deque<int> deq;
deq.push_back(123);
deq.push_back(456);
deq.pop_front();
deq.push_front(789);
deq.push_back(111);
for (int i = 0; i < deq.size(); i++)
    deq[i]; // -> { 789, 456, 111 }
deq[1] = 666;
while (!deq.empty())
{
    deq.back(); // -> { 111, 666, 789 }
    deq.pop_back();
}
```

2.7 priority_queue

Czasem przydaje się też `priority_queue` – kolejka priorytetowa. Wstawia się do niej elementy, a ona zawsze daje szybki dostęp do elementu największego i umożliwia jego usunięcie. Niektórzy chcieliby, aby zamiast zwykłej „największości" można tu było umieścić bardziej wyrafinowane kryterium porównywania. Jest to możliwe, podobnie jak przy sortowaniu (patrz: 3.4), przy czym trzeba to jeszcze sprytnie zapakować w strukturę. Szczegóły w dokumentacji STL.

`size()`	rozmiar kolejki	$O(1)$
`push(v)`	dodawanie	$O(\log n)$
`pop()`	zrzucanie największego	$O(\log n)$
`top()`	element największy	$O(1)$

```
#include <queue>
// ...
priority_queue<int> pq;
pq.push(5);
pq.top(); // -> 5
pq.push(8);
pq.top(); // -> 8
pq.push(3);
pq.top(); // -> 8
pq.pop();
pq.top(); // -> 5
pq.pop();
pq.top(); // -> 3
```

2.8 set

Struktura `set` służy do reprezentacji zbioru (powtórzenia elementów redukują się). Elementy można dodawać, usuwać i wyszukiwać. Ponadto można wyszukać pierwszy element o wartości niemniejszej (większej) od podanej. Można też przeglądać `set` w kolejności od najmniejszej wartości do największej[1]. Dla bardziej dociekliwych – `set` jest zaimplementowany przy użyciu drzew czerwono-czarnych, a w komentarzach w kodzie pojawiają się odwołania do Cormena.

`size()`	rozmiar	$O(1)$
`insert(v)`	dodawanie	$O(\log n)$
`find(v)`	wyszukiwanie[a]	$O(\log n)$
`erase(v)`	usuwanie po wartości	$O(\log n)$
`erase(i)`	usuwanie po iteratorze[b]	$O(\log n)$
`lower_bound(v)`	pierwszy element niemniejszy	$O(\log n)$
`upper_bound(v)`	pierwszy element większy	$O(\log n)$

[a] Jeżeli element nie występuje w zbiorze, zwrócony zostanie iterator `s.end()`, gdzie „`s`" to nazwa tego zbioru. W przeciwnym przypadku będzie to iterator na znaleziony element.

[b] Usuwanie po iteratorze jest w rzeczywistości nieco szybsze niż po wartości, niemniej złożoność jest taka sama.

[1] W kwestii przeglądania i iteratorów, patrz: `list`.

```
#include <set>
// ...
set<int> s;
set<int>::iterator it;
s.insert(123);
s.insert(456);
s.insert(789);
s.insert(111);
s.insert(999);
s.erase(456);
for (it = s.begin(); it != s.end(); it++)
    *it; // -> { 111, 123, 789, 999 }
it = s.lower_bound(500);
*it; // -> 789
s.erase(it);
for (it = s.begin(); it != s.end(); it++)
    *it; // -> { 111, 123, 999 }
*s.begin(); // -> 111
// test na obecnosc -- wynik: nieobecny
it = s.find(1000);
(it != s.end()); // -> false
// a tu wynik pozytywny
it = s.find(111);
(it != s.end()); // -> true
```

2.9 map

Struktura `map` służy do reprezentacji przyporządkowania.
Działa prawie jak `set< pair<x,y> >` z tą różnicą, że elementy są porównywane tylko względem pierwszej wartości (klucza) z pary. Dodatkowo elementy można wstawiać i odwoływać się do nich przy użyciu operatora [ ]. Tu wyjątkowo „`k`” oznacza klucz, a „`v`” wartość przyporządkowaną.

`size()`	rozmiar	$O(1)$
`[k] = v`	dodawanie / zmiana wartości	$O(\log n)$
`[k]`	odwołanie do wartości pod kluczem[a]	$O(\log n)$
`find(k)`	wyszukiwanie po kluczu	$O(\log n)$
`erase(k)`	usuwanie po kluczu	$O(\log n)$
`erase(i)`	usuwanie po iteratorze	$O(\log n)$
`lower_bound(k)`	pierwszy element niemniejszy[b]	$O(\log n)$
`upper_bound(k)`	pierwszy element większy[c]	$O(\log n)$

[a]Odwołanie jest przez referencję – wartość można dowolnie modyfikować.
[b]Względem klucza, nie zaś wartości.
[c]Względem klucza.

```
#include<map>
// ...
map<int, float> m;
map<int, float>::iterator it;
m[0] = 1.0f;
m[256] = 3.14f;
m[42] = 4.2f;
m[1000*1000*1000] = 1.618f;
m[0] = 1.1f;
m.size(); // -> 4
m[256]; // -> 3.14f
// teraz iterator wskazuje na pare
// (klucz, wartosc)
// kolejnosc przegladania jest rosnaca po kluczu
for (it = m.begin(); it != m.end(); it++)
    (*it).first; // -> { 0, 42, 256, 1000000000 }
for (it = m.begin(); it != m.end(); it++)
    (*it).second;
    // -> { 1.1f, 4.2f, 3.14f, 1.618f }
m.erase(256);
m.size(); // -> 3
// sprawdzanie obecnosci klucza
it = m.find(256);
(it != m.end()); // -> false
it = m.find(42);
(it != m.end()); // -> true
m.erase(it);
m.size(); // -> 2
m[0] += 1.1f;
m[0]; // -> 2.2f
```

3 Podstawowe algorytmy

Przez ciąg będziemy tu rozumieć strukturę danych umożliwiającą bezpośrednie odwoływanie się do elementów (np.: tablica, `vector`). Litera n w złożoności odpowiada oczywiście długości danego ciągu.

3.1 swap

Złożoność obliczeniowa: $O(1)$

Funkcja `swap` powoduje, iż dwie zmienne wymieniają się wartościami.

```
#include <algorithm>
// ...
int a = 123, b = 321;
swap(a, b);
a; // -> 321
b; // -> 123
```

3.2 min

Złożoność obliczeniowa: $O(1)$

Funkcja `min` zwraca mniejszą (dokładniej niewiększą) z dwóch wartości. Konkretniej zwraca referencję do wartości, co bywa bardzo wygodne.

```
#include <algorithm>
// ...
int a = 26, b = 12;
min(a, b); // -> 12
min(a, b)++;
a; // -> 26
b; // -> 13
```

3.3 max

Złożoność obliczeniowa: $O(1)$

Funkcja `max` zwraca większą (dokładniej niemniejszą) z dwóch wartości. Reszta analogicznie do `min`.

3.4 sort

Złożoność obliczeniowa: $O(n \log n)$

Funkcja `sort` sortuje ciąg. Zamiast sortować ciąg standardowo, można podać własną funkcję porównującą, względem której odbędzie się sortowanie[2].

```
#include <algorithm>
#include <vector>
// ...
// Ta funkcja odpowiada porzadkowi, w ktorym
// wszystkie liczby nieparzyste sa mniejsze
// od kazdej parzystej,
// a liczby o jednakowej parzystosci
// sa uporzadkowane standardowo.
bool cmp_parz(const int& a, const int& b)
{
    if (a % 2 == b % 2)
        return a <= b;
    else
        return (a % 2 == 1);
}
// ...
int liczby[] = { 5, 8, 3, 6, 1, 2, 7 };
// sortujemy nasza funkcja porownujaca
sort(liczby, liczby + 7, cmp_parz);
for (int i = 0; i < 7; i++)
    liczby[i]; // -> { 1, 3, 5, 7, 2, 6, 8 }
// sortujemy standardowa funkcja porownujaca <=
sort(liczby, liczby + 7);
for (int i = 0; i < 7; i++)
    liczby[i]; // -> { 1, 2, 3, 5, 6, 7, 8 }
// ...
vector<int> vec;
for (int i = 0; i < 7; i++)
    vec.push_back(7 - i);
// vec : { 7, 6, 5, 4, 3, 2, 1 }
sort(vec.begin(), vec.end());
for (int i = 0; i < vec.size(); i++)
    vec[i]; // -> { 1, 2, 3, 4, 5, 6, 7 }
```

[2]Taka funkcja porównująca musi mieć dwa argumentu typu `const` *referencja do wyrazu ciągu* i zwracać `bool`. Funkcja ta nie może być zupełnie dowolna – musi odpowiadać pewnemu porządkowi liniowemu. Ciąg posortowany ma wówczas tę własność, że dla każdych dwóch kolejnych wyrazów ciągu wartość funkcji porównującej na tych wyrazach wynosi `true`.

3.5 stable_sort

Złożoność obliczeniowa: $O(n \log n)$

Do sortowania stabilnego przydaje się `stable_sort`. Używa się tak samo, jak `sort`, dlatego przykładu nie będzie.

3.6 nth_element

Złożoność obliczeniowa: $O(n)$

Funkcja `nth_element` przestawia wyrazy danego ciągu w taki sposób, że na n-tej pozycji stoi taki wyraz jaki stałby w ciągu posortowanym, a dodatkowo każdy wyraz stojący na prawo od n-tego jest niemniejszy od każdego stojącego na lewo od n-tego. Argumentem funkcji wyznaczającym n-tą pozycję nie jest jednak liczba n, a iterator (wskaźnik) wskazujący na n-tą pozycję.

```
#include <algorithm>
// ...
// Chcemy znalezc mediane takiego ciagu:
int liczby[] = { 10, 1, 2, 7, 2, 5, 9 };
int n = 7; // rozmiar naszego ciagu (nieparzysty!)
// po posortowaniu ciag wygladalby tak:
// { 1, 2, 2, 5, 7, 9, 10 }
// mediana jest na srodku -- na poz. (n - 1) / 2
// w tym wypadku mediana to 5 (na poz. 3)
nth_element( liczby,
             liczby + (n - 1) / 2,
             liczby + n );
liczby[(n - 1) / 2]; // -> 5
```

3.7 reverse

Złożoność obliczeniowa: $O(n)$

Funkcja `reverse` odwraca ciąg.

```
#include <algorithm>
#include <vector>
// ...
vector<int> vec;
for(int i = 0; i < 7; i++)
    vec.push_back(i);
// wyglada tak: { 0, 1, 2, 3, 4, 5, 6 }
reverse(vec.begin(), vec.end());
// a teraz tak: { 6, 5, 4, 3, 2, 1, 0 }
```

3.8 next_permutation

Złożoność obliczeniowa: $O(n)$

Funkcja `next_permutation` przekształca ciąg w jego kolejną leksykograficznie permutację[3]. Wartością funkcji jest `true`, jeżeli taka permutacja istnieje. W przeciwnym przypadku jest to `false`, co oznacza osiągnięcie największej leksykograficznie permutacji – ciągu posortowanego nierosnąco. Nasuwa się więc wniosek, że aby przejrzeć wszystkie permutacje musimy zacząć od ciągu posortowanego niemalejąco.

```
#include <algorithm>
// ...
vector<int> vec;
vec.push_back(3);
vec.push_back(5);
vec.push_back(7);
do
{
    for (int i = 0; i < vec.size(); i++)
        vec[i]; // -> (patrz nizej)
} while (next_permutation(vec.begin(),vec.end()));
// wyniki beda takie:
// -> { 3, 5, 7 }
// -> { 3, 7, 5 }
// -> { 5, 3, 7 }
// -> { 5, 7, 3 }
// -> { 7, 3, 5 }
// -> { 7, 5, 3 }
```

[3]Porządek leksykograficzny dla dwóch różnych ciągów tej samej długości wyznacza się w ten sposób, że porównuje się ich kolejne wyrazy aż do pierwszej napotkanej (ostrej) nierówności – ona zaś określa nierówność między ciągami.

3.9 random_shuffle

Złożoność obliczeniowa: $O(n)$

Funkcja `random_shuffle` ustawia wyrazy ciągu w losowej kolejności.

```
#include <algorithm>
// ...
int liczby[] = { 1, 2, 3, 4, 5, 6, 7 };
// tutaj, na koniec i jako, ze jest malo kodu:
// standardowy trik na okreslenie
// rozmiaru statycznie stworzonej tablicy
// uwaga: to dziala poprawnie tylko wtedy,
// gdy deklaracja tablicy jest widoczna
int n = sizeof(liczby)/sizeof(*liczby);
n; // -> 7
// a teraz do rzeczy
random_shuffle(array, array + n);
// trudno przewidzec efekt
// moze byc np. taki:
// { 5, 3, 2, 4, 6, 1, 7 }
```