

Dzisiejszy wykład

- # Standardowa biblioteka języka C++
 - STL (*Standard Template Library*)

STL – ogólne spojrzenie

STL – biblioteka komponentów, których można wielokrotnie używać

- Dostarcza programiście C++ kolekcje, algorytmy, iteratory itd.
- Łatwa w użyciu, o bardzo dużych możliwościach i efektywna
- Programowanie uogólnione, nie obiektowe

<http://www.sgi.com/tech/stl>

Kolekcje	Klasy które zawierają inne obiekty
Iteratory	„Wskaźniki” do elementów kolekcji, używane do indeksowania kolekcji
Adaptery	Klasy które dostosowują do siebie inne klasy
Alokatory	Obiekty alokujące pamięć

Ważniejsze kolekcje w STL

<code>vector<T></code>	Dostęp swobodny, zmienna długość, stały czas wstawiania/usuwania elementów na końcu
<code>deque<T></code>	Dostęp swobodny, zmienna długość, stały czas wstawiania/usuwania elementów na obu końcach
<code>list<T></code>	Liniowy czas dostępu, zmienna długość, stały czas wstawiania/usuwania w dowolnym miejscu listy
<code>stack<T></code>	Typowa implementacja stosu.
<code>set<Key></code>	Kolekcja unikatowych wartości typu Key
<code>map<Key,T></code>	Kolekcja wartości typu T indeksowana przez unikatowe wartości typu Key

Elementy wspólne dla wszystkich kolekcji

- # Niektóre funkcje składowe występują we wszystkich kolekcjach, np.:
 - *size()* zwraca liczbę elementów w kontenerze
 - *push_back()* dodaje obiekt „na końcu” kontenera
- # Dostęp bezpośredni do danych w kolekcjach
 - poprzez *operator[]* lub składową *at()*
- # Iteratory
 - metoda dostępu do elementów kolekcji, z użyciem pętli i „indeksu”
 - wiele różnych iteratorów: jednokierunkowy, odwrotny, stały itd.

STL vector

- Wzorzec *vector* zachowuje się jak dynamicznie alokowana tablica, ale dodatkowo umożliwia dynamiczną zmianę rozmiaru (przy dodawaniu elementów przez *insert()* lub *push_back()*)

vector deklaracje:	<pre>vector<int> iVector; vector<int> jVector(100); vector<int> kVector(Size); // Size is int var</pre>
vector dostęp do elementów:	<pre>jVector[23] = 71; // set member jVector[41]; // get member jVector.at(23); // get member jVector.front(); // get first member jVector.back(); // get last member</pre>
vector obserwatory:	<pre>jVector.size(); // num elements in container jVector.capacity(); // capacity of container jVector.max_capacity(); // max capacity of elements jVector.empty();</pre>

Konstruktory wzorca *vector*

- # Wzorzec *vector* posiada kilka konstruktorów

```
vector<T> V; //empty vector
```

```
vector<T> V(n,value);
```

```
//vector with n copies of value
```

```
vector<T> V(n);
```

```
//vector with n copies of default for T
```

- # Wzorzec *vector* posiada również przeciążony konstruktor kopiujący i operator przypisania wykonujące głębokie kopiowanie

Wzorzec *vector* - przykład

```
#include <iostream>
#include <vector> // for vector template definition
using namespace std;
```

Początkowy rozmiar wektora

```
int main() {
    int MaxCount = 100;
    vector<int> iVector(MaxCount);
    for (int Count = 0; Count < MaxCount; Count++) {
        iVector[Count] = Count;
    }
}
```

Dostęp, jak do tablicy

- ⚠ Uwaga: pojemność wektora nie zwiększa się automatycznie przy dostępie przy pomocy operatora indeksowania. Użycie metod *insert()* oraz *push_back()* do dodania elementów tablicy powiększy ją w miarę potrzeby.

Indeksowanie wzorca *vector*

- ✚ W najprostszym przypadku, obiekt typu *vector* może być używany jako prosta tablica z dynamiczną alokacją pamięci

```
int MaxCount = 100;  
vector<int> iVector(MaxCount);  
...  
for (int Count = 0; Count < 2*MaxCount; Count++) {  
    cout << iVector[Count];  
}
```

Wydajność

- Brak sprawdzania zakresu indeksu
- Brak dynamicznego rozszerzania. Błędny indeks spowoduje błąd dostępu (jeśli mamy szczęście)

```
int MaxCount = 100;  
vector<int> iVector(MaxCount);  
...  
for (int Count = 0; Count < 2*MaxCount; Count++) {  
    cout << iVector.at(Count);  
}
```

Bezpieczeństwo

- Użycie składowej `at()` spowoduje wyjątek *out_of_range* w tej sytuacji

Iteratory STL

Iterator

- obiekt, który przechowuje lokalizację wewnątrz odpowiadającej mu kolekcji STL, umożliwiającą przechodzenie po wszystkich elementach kolekcji (inkrementacja/dekrementacja), wyłuskanie oraz wykrywanie osiągnięcia krańców zakresu w kolekcji
- Iterator jest deklarowany dla konkretnego typu kolekcji i jego implementacja jest zależna od tego typu oraz nie ma szczególnego znaczenia dla użytkownika
- Iteratory są podstawą wielu algorytmów STL i są niezbędnym narzędziem do dobrego wykorzystania kolekcji STL
- Każda kolekcja STL zawiera składowe *begin()* i *end()*, które zwracają wartości iteratorów wskazujących odpowiednio na pierwszy element, oraz element "następny za końcowym"

Iterator wektora

Iterator wektora zachowuje się jak wskaźnik do dynamicznie zaalokowanej tablicy

deklaracja iteratora:	<code>vector<int>::iterator idx;</code> <code>vector<int> jVector;</code>
uzyskanie iteratora z wektora:	<code>jVector.begin(); // gets iterator</code> <code>jVector.end(); // gets sentinel (iterator)</code>
dostęp do elementów wektora poprzez iterator:	<code>idx[i]; // access ith element</code> <code>*idx; // access to element pointed by idx</code> <code>idx++; // moves pointer to next element</code> <code>idx--; // moves pointer to previous element</code>

```
vector<T> v;
```

```
vector<T>::iterator idx;
```

```
for (idx = v.begin(); idx != v.end(); ++idx)  
    do something with *idx
```

Typy iteratorów

▣ Różne kolekcje posiadają różne rodzaje iteratorów

- iteratory jednokierunkowe - definiują tylko ++
- iteratory dwukierunkowe - definiują ++ i --
- iteratory o dostępie swobodnym - definiują ++, -- oraz:
 - dodawanie i odejmowanie liczby całkowitej $r+n$, $r-n$
 - zmianę o liczbę całkowitą $r+=n$, $r-=n$
 - odejmowanie iteratorów $r-s$ zwraca liczbę całkowitą
 - kolekcja posiada operator indeksowania []

▣ Iteratory do stałej i zmiennej

- iteratory do stałej - `*p` nie pozwala na zmianę elementu kolekcji
- iteratory do zmiennej - umożliwiają zmiany w kolekcji

```
for (p = v.begin(); p != v.end(); ++p)
    *p = new value
```

▣ Iteratory odwrotne - pozwalają na przeglądanie elementów kolekcji od końca do początku

```
reverse_iterator rp;
for (rp = v.rbegin(); rp != v.rend(); ++rp)
    process *rp
```

Iteratory do stałej

- # Iteratory do stałej muszą być użyte dla stałych kontenerów - przeważnie przekazywanych jako parametry
- # Typ jest zdefiniowany w konenerze:
`vector<T>::const_iterator`

```
void ivecPrint(const vector<int>& V, ostream& Out) {  
    vector<int>::const_iterator It; // MUST be const  
  
    for (It = V.begin(); It != V.end(); ++It) {  
        cout << *It;  
    }  
    cout << endl;  
}
```

Przykład zastosowania iteratora

✚ Poniższy przykład tworzy kopię wektora *BigInt*

```
string DigitString = "45658228458720501289";  
vector<int> BigInt;  
  
for (int i = 0; i < DigitString.length(); i++) {  
    BigInt.push_back(DigitString.at(i) - '0');  
}  
vector<int> Copy;  
vector<int>::iterator It;  
for (It = BigInt.begin(); It != BigInt.end(); ++It) {  
    Copy.push_back(*It);  
}
```

Przesunięcie iteratora do następnego elementu

Inicjalizacja iteratora

Wartość pozwalająca na zakończenie operacji

- ✚ Wektor *Copy* ma początkowo zerowy rozmiar.
push_back() powiększy docelowy wektor do odpowiednich rozmiarów
- ✚ Stosujemy przedrostkowy, a nie przyrostkowy operator inkrementacji iteratora

Operacje na iteratorach

- Każdy iterator zapewnia pewne usługi za pośrednictwem standardowego interfejsu

```
string DigitString = "45658228458720501289";  
vector<int> BigInt;  
  
for (int i = 0; i < DigitString.length(); i++) {  
    BigInt.push_back(DigitString.at(i) - '0');  
}
```

```
vector<int>::iterator It;
```

Stwórz iterator do obiektu typu *vector<int>*

```
It = BigInt.begin();  
int FirstElement = *It;
```

Wskaż na pierwszy element *BigInt* i skopiuj go

```
It++;
```

Przesuń się do drugiego elementu *BigInt*

```
It = BigInt.end();
```

Teraz *It* nie wskazuje na żaden element *BigInt*.
Próba wyłuskania może spowodować błąd dostępu.

```
It--;  
int LastElement = *It;
```

Cofnij się do ostatniego elementu *BigInt*

Wstawianie elementów do wektorów

- ✚ Wstawianie elementów na końcu wektora (z użyciem *push_back()*) jest najbardziej efektywne.
 - Wstawianie w innym miejscu wymaga przesuwania danych w pamięci
- ✚ Wektor zachowuje się jak tablica o zmiennej długości
 - Wstawianie elementów do wektora powiększa zaalokowany obszar np. dwa razy
- ✚ Wstawienie elementu unieważnia wszystkie iteratory wskazujące na elementy po punkcie wstawiania
- ✚ Realokacja (powiększenie) wektora unieważnia wszystkie iteratory do elementów danego wektora
- ✚ Można ustawić minimalny rozmiar wektora V przy pomocy metody $V.reserve(n)$

Składowa *insert()*

- ✚ W dowolnym miejscu wektora można wstawić element używając iteratora i składowej *insert()*

```
vector<int> Y;  
for (int m = 0; m < 100; m++) {  
  
    Y.insert(Y.begin(), m);  
  
    cout << setw(3) << m  
         << setw(5) << Y.capacity()  
         << endl;  
}
```

Index	Cap
0	1
1	2
2	4
3	4
4	8
	.
8	16
	.
15	16
16	32
	.
31	32
33	64
63	64
	.
64	128

- ✚ Jest to najgorszy przypadek; element jest wstawiany zawsze na początku sekwencji, co maksymalnie zwiększa liczbę kopiowanych elementów
- ✚ Istnieją inne wersje metody *insert()*, wstawiające dowolną liczbę kopii danej wartości i wstawiające sekwencję elementów innego wektora

Usuwanie elementów wektorów

- ✚ Tak samo jak wstawianie, kasowanie elementów wektora wymaga przesunięć jego elementów (poza szczególnym przypadkiem ostatniego elementu)
- ✚ Usuwanie ostatniego elementu: *V.pop_back()*
- ✚ Usuwanie elementu wskazywanego przez iterator It: *V.erase(It)*
- ✚ Usuwanie unieważnia iteratory do elementów po punkcie usuwania, a więc

```
j = V.begin();  
while (j != V.end())  
    V.erase(j++);
```

nie działa
- ✚ Usuwanie grupy elementów:

```
V.erase(Iter1, Iter2)
```

Porównywanie kolekcji

- # Dwie kolekcje są równe, jeżeli:
 - mają ten sam rozmiar
 - elementy na wszystkich pozycjach są równe
- # Klasa elementu wektora musi posiadać operator porównania
- # Dla innych porównań klasa elementu musi posiadać odpowiedni operator ($<$, $>$, ...)

Kolekcja *deque*

deque

- double-ended queue

- # Umożliwia wydajne wstawianie na obu końcach
- # Umożliwia wstawianie/usuwanie dowolnych elementów posługując się iteratorami
- # Oprócz metod klasy *vector* posiada dodatkowo metody *push_front()* i *pop_front()*
- # Większość metod i konstruktorów identyczna jak dla wektora
- # Wymaga pliku nagłówkowego *<deque>*

Lista

- # Dwukierunkowa lista z dowiązaniem
- # Nie umożliwia dostępu swobodnego, ale umożliwia wstawianie i usuwanie elementów na dowolnej pozycji w stałym czasie
- # Pewne różnice w metodach w stosunku do wektora i deque (np. brak operatora indeksowania)
- # Wstawianie i usuwanie elementów nie unieważnia iteratorów

Kolekcje asocjacyjne

- # Standardowa tablica jest indeksowana wartościami numerycznymi
 - $A[0], \dots, A[\text{Size}-1]$
 - gęste indeksowanie
- # Tablica asocjacyjna może być indeksowana dowolnym typem
 - $A["alfred"], A["judy"]$
 - rzadkie indeksowanie
- # Asocjacyjne struktury danych umożliwiają bezpośrednie wyszukiwanie ("indeksowanie") za pomocą złożonych kluczy
- # STL zawiera wzorce kilku kolekcji asocjacyjnych

Posortowane kolekcje asocjacyjne

- Wartości (obiekty) w kolekcji są przechowywane w porządku posortowanym pod względem typu klucza

<code>set<Key></code>	kolekcja unikatowych wartości typu Key
<code>multiset<Key></code>	wartości Key mogą się powtarzać
<code>map<Key,T></code>	kolekcja wartości typu T indeksowanych unikatowymi wartościami Key
<code>multimap<Key,T></code>	wartości Key mogą się powtarzać

Zbiory i wielozbiory

- # Zarówno wzorzec *set* jak i *multiset* przechowuje wartości typu *key*, który musi mieć zdefiniowane uporządkowanie
- # *set* umożliwia przechowywanie jedynie pojedynczych obiektów, natomiast *multiset* umożliwia przechowywanie duplikatów

```
set<int> iSet;           // fine, built-in type has < operator
set<Employee> Payroll;  // class Employee did not
                        // implement a < operator
```

- # typ klucza musi definiować operator <

```
bool Employee::operator<(const Employee& Other) const {
    return (ID < Other.ID);
}
```

Przykład zbioru

```
#include <functional>
#include <set>
using namespace std;
#include "employee.h"

void EmpsetPrint(const set<Employee> S, ostream& Out);
void PrintEmployee(Employee toPrint, ostream& Out);

int main() {
    Employee Ben("Ben", "Keller", "000-00-0000");
    Employee Bill("Bill", "McQuain", "111-11-1111");
    Employee Dwight("Dwight", "Barnette", "888-88-8888");
    set<Employee> S;
    S.insert(Bill);
    S.insert(Dwight);
    S.insert(Ben);
    EmpsetPrint(S, cout);
}

void EmpsetPrint(const set<Employee> S, ostream& Out) {
    set<Employee>::const_iterator It;
    for (It = S.begin(); It != S.end(); ++It)
        Out<<*It<<endl;
}
```

```
000-00-0000 Ben Keller
111-11-1111 Bill McQuain
888-88-8888 Dwight Barnette
```


Wybór typu kolekcji

- # *vector* może być użyty zamiast dynamicznie alokowanej tablicy
- # *list* umożliwia dynamiczną zmianę rozmiaru przy dostępie sekwencyjnym
- # *set* może być użyty w przypadku potrzeby posortowania danych i braku konieczności dostępu swobodnego
- # *map* powinien być użyty, kiedy dane muszą być indeksowane przy pomocy unikatowego klucza
- # *multiset* i *multimap* powinny być użyte w sytuacjach analogicznych jak *set* i *map* oraz powtarzających się wartościach klucza

Rozważmy poniższy program...

```
#include <iostream>
#include <vector>
using namespace std;

int
main ()
{
    vector < int >v;
    vector < int >::iterator idx;
    int i, total;
    cout << "Enter numbers, end with ^D" << endl;
    cout << "% ";
    while (cin >> i)
    {
        v.push_back (i);
        cout << "% ";
    }
    cout << endl << endl;
    cout << "Numbers entered = " << v.size () << endl;
    for (idx = v.begin (); idx != v.end (); ++idx)
        cout << *idx << endl;
    total = 0;
    for (idx = v.begin (); idx != v.end (); ++idx)
        total = total + *idx;
    cout << "Sum = " << total << endl;
};
```

Powtórzona sekwencja dostępu
do wszystkich elementów kolekcji

Po uproszczeniu...

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

void print (int i) {
    cout << i << endl;
};

int main ()
{
    vector < int >v;
    vector < int >::iterator idx;
    int i, total;
    cout << "Enter numbers, end with ^D" << endl;
    cout << "% ";
    while (cin >> i)
    {
        v.push_back (i);
        cout << "% ";
    }
    cout << endl << endl;
    cout << "Numbers entered = " << v.size () << endl;
    for_each (v.begin (), v.end (), print);
    total = accumulate (v.begin (), v.end (), 0);
    cout << "Sum = " << total << endl;
}
```

Z użyciem STL

Programowanie uogólnione

- # Popularne algorytmy operujące na kolekcjach
- # Implementują sortowanie, wyszukiwanie i inne podstawowe operacje
- # Trzy typy algorytmów operujących na kolekcjach sekwencyjnych
 - modyfikujące algorytmy na ciągach
 - *fill()*, *fill_n()*, *partition()*,
random_shuffle(), *remove_if()*, ...
 - niemodyfikujące algorytmy na ciągach
 - *count()*, *count_if()*, *find()*, *for_each()*
 - algorytmy numeryczne (<numeric>)
 - *accumulate()*

Algorytmy modyfikujące

- # Funkcje modyfikujące kolekcję w różny sposób
- # Dostęp do kolekcji uzyskiwany za pośrednictwem iteratora
- # Założenie

vector<char> charV;

<code>void fill(iterator, iterator, T)</code>	<code>charV.fill(charV.begin(), charV.end(), 'x')</code> umieszcza 'x' we wszystkich elementach wektora
<code>iterator fill_n(iterator, int, T)</code>	<code>charV.fill_n(charV.begin(), 5, 'a')</code> umieszcza 'a' w pięciu pierwszych elementach wektora
<code>void generate(iterator, iterator, function)</code>	<code>char nextLetter() { static char letter = 'A'; return letter++; }</code> <code>charV.generate(charV.begin(), charV.end(), nextLetter);</code> wypełnia tablicę rezultatami wywołania funkcji 'nextLetter' kolejno dla każdego elementu

Algorytmy niemodyfikujące

Założenie

vector<int> v;

<code>T min_element(iterator, iterator)</code>	<code>min_element(v.begin(), v.end())</code> zwraca najmniejszy element w kolekcji
<code>function for_each(iterator, iterator, function)</code>	<code>void put(int val)</code> <code>{ cout << val << endl; }</code> <code>for_each(v.begin(), v.end(), put);</code> wywołuje funkcję put() dla każdego elementu tablicy; w tym przypadku wypisuje wartości wszystkich elementów
<code>int count(iterator, iterator, T)</code>	<code>v.count(v.begin(), v.end(), 5)</code> zwraca liczbę wystąpień liczby 5 w kolekcji
<code>int count_if(iterator, iterator, function)</code>	<code>bool GT10(int val)</code> <code>{ return val > 10; }</code> <code>v.count_if(v.begin(), v.end(), GT10);</code> zwraca liczbę elementów większych od 10 w kolekcji

Inne użyteczne algorytmy

Założenie

vector<int> v;

<code>iterator find(iterator, iterator, T)</code>	<pre>iterator r = find(v.begin(), v.end(), 25); if (r == v.end()) cout << "Not found" << endl; else cout << "Found at " << (r - v.begin());</pre>
<code>iterator find(iterator, iterator, function)</code>	Jak wyżej, ale do porównania używa funkcji
<code>bool binary_search(iterator, iterator, T)</code>	Binarne wyszukiwanie w kolekcji.
<code>iterator copy(iterator, iterator, iterator)</code>	Kopia z jednej kolekcji do drugiej. Użyteczne w połączeniu z <code>ostream_iterator</code> <pre>ostream_iterator<int> output(cout, " "); copy(v.begin(), v.end(), output);</pre>

I wiele, wiele innych

- # STL posiada wiele innych operacji oraz kilka innych kolekcji
- # Styl programowania zaprezentowany tutaj nazywany jest programowaniem uogólnionym (*generic programming*)
 - Pisanie funkcji zależnych od pewnych operacji zdefiniowanych na przetwarzanych typach
 - Na przykład, *find()* polega na operatorze `==` dostępnym dla typu danych
 - Dla poszczególnych funkcji, mówimy o zbiorze typów które mogą być używane z daną funkcją
 - np. *find()* może być użyta dla wszystkich typów, dla których jest zdefiniowany operator `==`
 - Brak związku z programowaniem obiektowym. Zbiór typów które definiują pewne operacje i mogą być zastosowane w konkretnej funkcji uogólnionej nie musi być spokrewniony przez dziedziczenie i polimorfizm nie jest używany.

Wskaźniki w STL

- STL jest bardzo elastyczna, umożliwia przechowywanie danych dowolnych typów w kolekcjach

```
vector< int > v;  
vector< int >::iterator vi;  
v.push_back( 45 );  
for (vi = v.begin(); vi != v.end(); vi++) {  
    int av = *vi;  
}  
  
vector< Foo * > v;  
vector< Foo * >::iterator vi;  
v.push_back( new Foo( value) );  
for (vi = v.begin(); vi != v.end(); vi++) {  
    Foo * av = *vi;  
}
```

- Kolekcja nie zwalnia pamięci zaalokowanej na obiekty, do których przechowuje wskaźniki

Obiekty funkcyjne w STL

- Obiekt funkcyjny to obiekt, który ma zdefiniowany operator wywołania funkcji (), tak że w poniższym przykładzie

```
FunctionObjectType fo;  
// ...  
fo(...);
```

wyrażenie *fo()* jest wywołaniem operatora () obiektu *fo*, a nie wywołaniem funkcji *fo*

Zamiast

```
void fo(void) {  
    // statements  
}
```

piszemy

```
class FunctionObjectType {  
public:  
    void operator() (void){  
        // statements  
    }  
};
```

- Obiektów funkcyjnych możemy użyć we wzorcach STL wszędzie tam, gdzie dopuszczalny jest wskaźnik do funkcji

Obiekty funkcyjne - po co ta komplikacja?

- # Obiekty funkcyjne mają następujące zalety w porównaniu ze wskaźnikami do funkcji
 - Obiekt funkcyjny może posiadać stan. Można mieć dwa egzemplarze obiektu funkcyjnego, które mogą być w różnych stanach. Nie jest to możliwe w przypadku funkcji
 - Obiekt funkcyjny jest przeważnie bardziej wydajny niż wskaźnik do funkcji

Przykład zastosowania obiektu funkcyjnego

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include <time.h>
using namespace std;

bool GTRM(long val)
{
    return val > (RAND_MAX >> 1);
}

int main ()
{
    srand(time(NULL));
    vector < long > v(10);
    generate(v.begin(), v.end(),
        random);
    cout << count_if(v.begin(),
        v.end(), GTRM);
    cout << endl;
};
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include <time.h>
using namespace std;

template <class T> class greater_than
{
    T reference;
public:
    greater_than (const T & v): reference (v)
    {}
    bool operator() (const T & w) {
        return w > reference;
    }
};

int main ()
{
    srand (time (NULL));
    vector < long > v (10);
    generate (v.begin (), v.end (), random);
    cout << count_if (v.begin (), v.end (),
        greater_than<long> (RAND_MAX >> 1));
    cout << endl;
};
```