Zawartość:

- kolekcje w C++

    * kolekcje w STL

    * iteratory w STL

- kolekcje w C#

-jak dobrać sobie kolekcje do zadania

STL – biblioteka kolekcji i algorytmów w C++

Kolekcja jest to zbiór danych konkretnego typu. **Kolekcje są podobne do tablic, jednak oferują znacznie bogatsze API i są samo rozszerzalne.**

Kolekcje w C++:

- tablice (to już poza STL)
- vector
- kolejka
- kolejka dwukierunkowa
- lista
- stos
- mapa
- zbiór
- std::array
- std::span

Czym się różnią kolekcje między sobą???

- rozmieszczeniem elementów w pamięci
- udostępnianym API
    * część kolekcji oferuje operator indeksowania [], a część nie
    * kolekcje różnią się sposobem dodawania i pobierania elementów oraz miejscem gdzie to następuje

std::vector

Wektor to najprostszy kontener w STL. Jest mu najbliżej do zwykłej tablicy:

- wektor sam się rozszerza

- wektor oferuje operator []

- wektor oferuje możliwość dodawania elementów na końcu i pobierania ich z końca (push_back(), pop_back())

- wektor podobnie jak tablica przechowuje elementy w pamięci w sposób **ciągły jeden za drugim**

```cpp
#include <cstdio>
#include <vector>

int main(){
    std::vector<int> wektor = {1,2,3};
    wektor.push_back(8);
    wektor.pop_back(); //ta metoda nie zwraca wartości
    int wartosc = wektor.back();

    return 0;
}
```

Jak widać metoda pop_back() tylko ściąga wartość z końca kolekcji. NIE ZWRACA JEJ.

Wektor jest najczęściej stosowanym kontenerem w STL, ponieważ zachowuje się jak tablice z języków wyższego poziomu. A fakt ten ułatwia i przyśpiesza pracę.

std::queue

Kolejka to kolejna liniowa struktura danych bardzo podobna do vectora:

- kolejka sama się rozszerza

- **nie** oferuje operatora []

- elementy można dodawać tylko na koniec kolejki (push()), a usuwać z początku (pop())

- do pozyskiwania samych wartości służą metody front(), back()

- kolejka podobnie jak tablica przechowuje elementy w pamięci w sposób **ciągły jeden za drugim**

```cpp
#include <cstdio>
#include <queue>

int main(){
    std::queue<int> kolejka;
    kolejka.push(8);
    kolejka.push(1);
    kolejka.pop(); //ta metoda nie zwraca wartości
    int wartosc = kolejka.back();

    return 0;
}
```

std::deque

Jest to kolejka dwu kierunkowa:

- pozwala dodawać i usuwać elementy z obu stron (push_front(), push_back(), pop_back(), pop_front())

- w przeciwieństwie do kolejki ma przeciążony operator []

- kolejka dwukierunkowa podobnie jak tablica przechowuje elementy w pamięci w sposób **ciągły jeden za drugim**

std::stack

Kolejna linowa struktura danych typu FILO (first In last out) lub LIFO (last In first out). Stos można wyobrazić sobie „w pionie”.

- dodawać (push()) można tylko na szczyt. Usuwać można tylko ze szczytu (pop()).

- odczyt elementu odbywa się za pomocą metody pop()

- brak tu operatora []

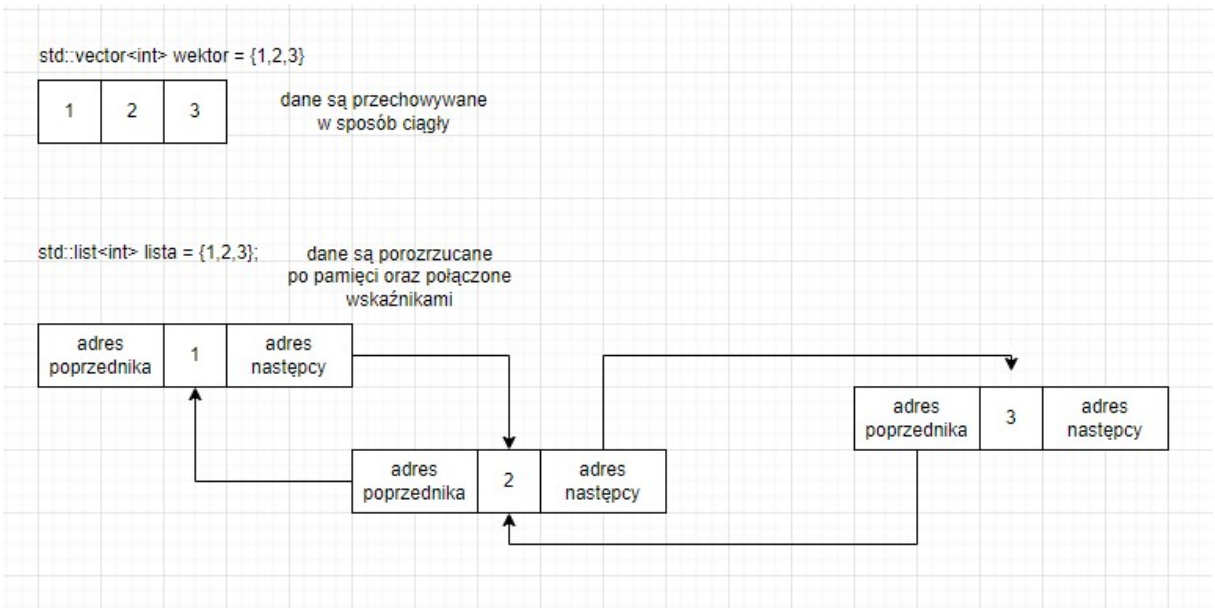
- stos podobnie jak tablica przechowuje elementy w pamięci w sposób **ciągły jeden za drugim**

```cpp
int main(){
    std::stack<int> stos;
    stos.push(8);
    stos.push(1);
    stos.pop(); //ta metoda nie zwraca wartości
    int wartosc = stos.top();

    return 0;
}
```

std::list

Lista jest bardzo wyjątkową kolekcją chociaż po samej nazwie można wnioskować inaczej. Jak ktoś pisze w innych językach też pewnie pomyśli, że wszystko jest tutaj standardowe.



Jak widać lista rozmieszcza elementy w sposób nie ciągły. Umieszcza jest w wolnych miejscach pamięci. Każdy element oprócz wartości ma wskaźnik na poprzednika i następcę dzięki czemu możliwa jest nawigacja po elementach (iterowanie).

Jakie są zalety tego podejścia?

- wstawienie elementu w środek listy jest szybsze bo nie trzeba przesuwać następnych elementów a jedynie zmodyfikować wskaźniki

Wady?

- wolniejszy odczyt ponieważ trzeba skakać po pamięci

```cpp
#include <cstdio>
#include <list>

int main(){
    std::list<int> lista = {1,2,3};
    lista.push_back(8);
    lista.push_front(1);
    lista.pop_back(); //ta metoda nie zwraca wartości
    int wartosc = lista.front();
    lista.pop_front();

    return 0;
}
```

std::map

Kolekcja, którą można było by określić mianem kolekcji asocjacyjnej czyli kolekcji klucz wartość.

- system klucz wartość

- dodajemy wartość za pomocą metody insert()

- pobieramy wartości za pomocą operatora []

```cpp
#include <cstdio>
#include <string>
#include <map>

int main(){
    std::map<std::string, int> dni = {{"PN", 1}, {"WT", 2}};
    dni.insert({"SR", 3});
    int sroda = dni["SR"];

    return 0;
}
```

std::set

Kolekcja, która przechowuje wartości w sposób uporządkowany rosnąco lub malejąco. Wartości w kolekcji nie mogą się powtarzać.

```cpp
#include <cstdio>
#include <set>

int main(){
    std::set<char> a;
    a.insert('G');
    a.insert('F');
    a.insert('G'); //nie zostanie dodane

    return 0;
}
```

W drugim parametrze <> można podać kolejność w jakiej ustawione będą elementy:

std::set<int, std::less> s;

std::set<int, std::greater> d;

std::array

Jest to zwykła struktura, która pełni role nakładki na podstawowe tablice.

- zwykłe tablice w C/C++ nie pamiętają swojego rozmiaru, std::array trzyma to w sobie

- kontener ten oferuje operator [] czyli cały czas posiadamy najważniejszy element tablic

- dodatkowo dodaje wiele metod w stylu empty(), max_size()

- std::array nie rozszerza się samoczynnie

```cpp
#include <cstdio>
#include <array>

int main(){

    std::array<int, 4> tablica = {1,2,3,4};
    int size = tablica.size();

    int tablica1[4] = {1,2,3,4};
    int size1 = tablica1.size(); //error, nie ma czegoś takiego
    //musimy rozmiar pamiętać

    return 0;
}
```

Jak dobrać kolekcję do zadania:

- po pierwsze należy pamiętać, że kolekcje są tylko ułatwieniem, które w środku i tak opierają się na tablicach

- każde zadanie da się zrobić na tablicach, każde

- należy zastanowić się czy potrzebujemy operatora [] lub innej drogi szybkiego dostania się do elementu pod indeksem x

- niektóre algorytmy z góry zakładają użycie konkretnego kontenera np. kolejki albo stosu (w opisie algorytmu może nie być jasno zaznaczone, że do poprawnego jego działania wymagane jest dodawanie wartości na końcu i pobieranie z frontu)

Iterator

Jest to bardzo proste pojęcie. Iterator to obiekt pewnej specjalnej klasy, która umożliwia przemieszczanie się po elementach kolekcji.

Iterator przyjmuje w kontruktorze kolekcję a w zamian oferuje metody next(), previous() itp.

```
#include <cstdio>
#include <vector>
#include <iterator>

int main(){
    std::vector<int> wektor = {1,2,3,4};
    std::vector<int>::iterator wektor_iter;

    for(wektor_iter = wektor.begin();wektor_iter<wektor.end();wektor_iter++){
        printf("%u\r\n", *wektor_iter);
    }

    return 0;
}
```

Podsumowując

Kolekcje to warstwa abstrakcji nałożona na tablice. Warstwa ta dodaje tablicą takie możliwości jak:

- sprawdzanie rozmiaru

- sprawdzanie czy tablica jest pusta

- auto rozszerzanie

- dodawanie/usuwanie z końca/początku/środka

- automatyczne przesuwanie elementów

Iteratory to warstwa abastracji nałożona na kolekcjie. Warstwa ta daje nam możliwość wygodniejszego przeszukiwania, przesuwania się po elementach kolekcji.

Kolekcje w C#

C# jest wysoko poziomowym językiem. Najbardziej podstawowa kolekcja w tym języku czyli tablica oferuje już całkiem sporo możliwości jak ta wymarzona funkcja/pole do pobierania rozmiaru.

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = new int[] { 1, 3, 5, 7, 9 };
```

Poza tym w języku tym znajdują się typy takie jak List<T> czy Dictionary<T,D>.

C# Copy

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Najważniejsze z punktu widzenia programisty c# są poniższe kolekcje:

Class	Description
Dictionary<TKey,TValue>	Represents a collection of key/value pairs that are organized based on the key.
List<T>	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
Queue<T>	Represents a first in, first out (FIFO) collection of objects.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

Wszystkie z nich są generyczne co oznacza, że typ zarówno klucza jak i wartości trzeba określić w momencie tworzenie kolekcji. Api powyższych typów kolekcji nie obiega znacząco od tych znanych z C++.

Istnieją jednak stare kolekcje:

Class	Description
ArrayList	Represents an array of objects whose size is dynamically increased as required.
Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.
Queue	Represents a first in, first out (FIFO) collection of objects.
Stack	Represents a last in, first out (LIFO) collection of objects.

Nie da się w ich przypadku określić typu przechowywanych wartości. Każda wartość wrzucona do takiej kolekcji jest zamieniana na typ object. Po ewentualnym pobraniu wartości należy sobie przeprowadzić kastowanie na typ oczekiwany.

Używanie powyższych kolekcji jest kompletnie bez sensu w obecnych czasach.