STL #4

OPERACJE NA PLIKACH, KONTENERY ASOCJACYJNE

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

ZADANIA

Repo GH `coders-school/stl`

https://github.com/coders-school/stl/tree/master/module4

Zadania wykonywane podczas zajęć online nie wymagają ściągania repo. Pliki będą tworzone od zera.

KRÓTKIE PRZYPOMNIENIE

CO JUŻ WIEMY

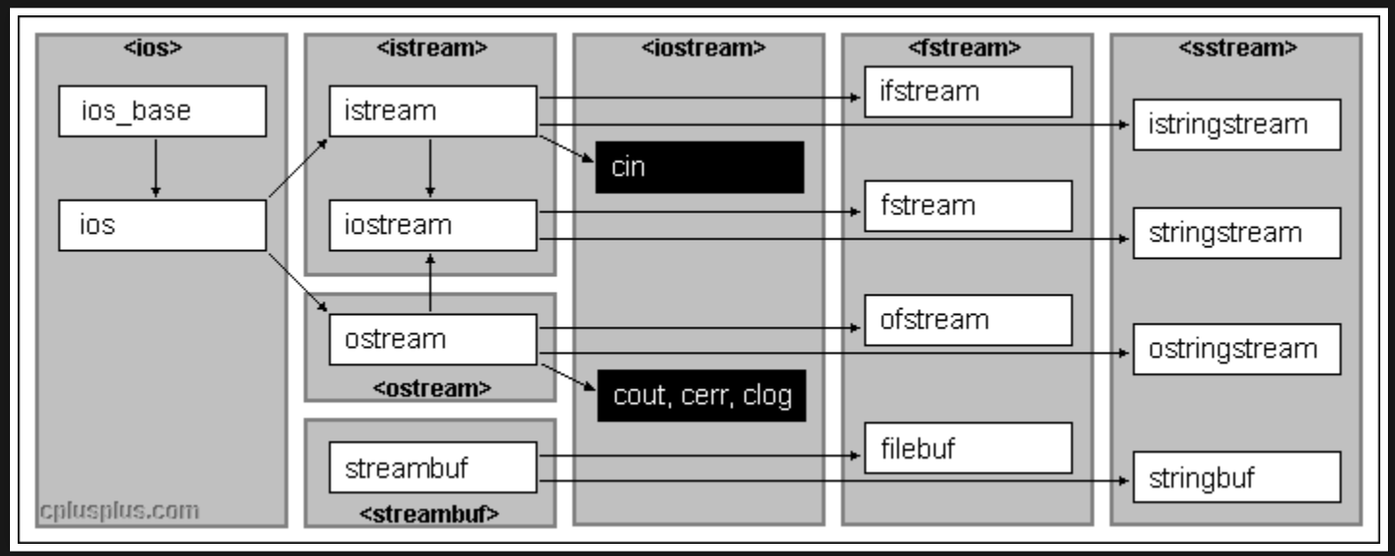
- co zapamiętaliście z poprzednich zajęć?
- co sprawiło największą trudność?
- co najłatwiej było wam zrozumieć?

AGENDA

- Operacje na plikach
- POD - Plain old data
- kontenery asocjacyjne

OPERACJE NA PLIKACH

KRÓTKA PASTA O ITERATORACH



ISTREAM, OSTREAM, IOSTREAM

O danych przychodzących i wychodzących będziemy mówić jak o strumieniach danych. Strumień może płynąć od programu do świata zewnętrzego poprzez `operator>>`, natomiast ze świata zewnętrzego do programu poprzez `operator<<`.

- istream (`std::cin`) -> od input stream, czyli dane przychodzące (np. wpisywane z klawiatury).
- ostream (`std::cout`, `std::cerr`, `std::clog`) -> od output stream, czyli dane wychodzące (np. wyświetlane na monitorze).
- iostream -> od input output stream, czyli dane mogą zarówno być przychodzące jak i wychodzące.

STRUMIEŃ DANYCH

Strumień danych może pochodzić z wielu źródeł, możemy je wczytywać od użytkownika, może on pochodzić z jakiegoś pliku zapisanego na dysku itp. Zapewnia nam to rozbudowana hierarchia klas, gdzie zaczynając od klasy bazowej `ios_base` docieramy do 3 klas pochodnych `istream`, `iostream` oraz `ostream`. W zależności od tego skąd będziemy chcieć wczytywać dane użyjemy:

- `fstream` do obsługi plików,
- `iostream` do obsługi poleceń wczytywanych z klawiatury,
- `sstream` służącą do wygodnego operowania na `std::string`, tak jakby to by były strumienie danych.

Każda z tych 3 klas dziedziczy albo po `istream`, gdy chcemy odczytywać dane, `ostream` gdy chcemy je zapisywać lub `iostream`, gdy chcemy robić obie czynności.

CZWARTY JEŹDZIEC, CZYLI `streambuf`

Z definicji streambuffer reprezentuje urządzenia wyjściowe i wejściowe (jak monitor, klawiatura, dysk itp.) oraz umożliwia nam dostęp do interfejsu niskiego poziomu. Raczej będzie rzadko przez nas stosowany ☺

Klasę tę użyjemy między innymi do tworzenia własnych strumieni. Popatrzmy na przykład jak utworzyć własny strumień wyjściowy `ostream`.

```
std::cout << 42 << '\n';
std::streambuf* buffer = std::cout.rdbuf();
std::ostream ost(buffer);
ost << 24 << '\n';
```

Output:

```
42
24
```

🤔 Ale po co nam własny strumień?

WŁASNY STYL WYŚWIETLANIA DANYCH

Aby nie ingerować w zwykły strumień `std::cout`, który jest używany globalnie.

```
std::cout << M_PI << '\n';
std::streambuf* buffer = std::cout.rdbuf();
std::ostream ost(buffer);
ost.precision(15);
ost << M_PI << '\n';
std::cout << M_PI << '\n';
```

Output:

```
3.14159
3.141592653589793
3.14159
```

Jako zadanie dla chętnych zachęcam do sprawdzenia co biblioteka `iomanip` nam umożliwia.

BIBLIOTEKA `fstream`

Jest to biblioteka umożliwiająca nam zapis i odczyt danych z pliku. Jest to bardzo rozbudowana biblioteka, jednak zwykle będziemy używać tylko kilka metod. Zgadnijcie proszę co mogą one robić?

- `is_open()` -> sprawdza czy plik jest otwarty (zwraca `bool`)
- `put()` -> zapisuje do pliku jeden znak
- `get()` -> pobiera jeden znak z pliku
- `peek()` -> odczytuje znak, ale nie przesuwa wskaźnika odczytu/zapisu
- `write()` -> zapisuje blok danych do pliku
- `read()` -> odczytuje blok danych z pliku
- `seekp()` -> ustawia nam pozycje wskaźnika odczytu/zapisu na danej pozycji w pliku
- `tellp()` -> informuje nas o pozycji wskaźnika odczytu zapisu
- `getline()` -> pobiera znaki z pliku tak długo, aż nie natrafi na podany znak (domyślnie znak nowej linii)

Wskaźnik w pliku możemy traktować tak jak migający kursor | w plikach tekstowych ☺ Czyli jak piszemy sobie jakiś tekst, zawsze patrzymy, gdzie miga nam ten znaczek i wiemy gdzie modyfikujemy aktualnie plik.

TROCHĘ Z LINUXA

Jakie możemy nadać uprawnienia danemu plikowi?

- Read
- Write
- Execute

Pliki możemy otworzyć w trybie do odczytu (read only), w trybie do modyfikacji (write only) lub w trybie (read-write). Dodatkowo mamy możliwość ustawienia kursora podczas otwierania pliku, wymazanie całej jego zawartości jeżeli istnieje, lub utworzenie pliku jeżeli nie istnieje. Zgadnijmy co oznaczają poszczególne tryby:

- `trunc` -> wymazuje wszystko w pliku co było do tej pory,
- `in` -> tryb do odczytu,
- `out` -> tryb do zapisu,
- `ate` -> ustawia `seek` na końcu pliku,
- `app` -> ustawia `seek` na końcu pliku przed zapisaniem do niego danych. Jednym słowem "doklejamy" nowe wartości na końcu pliku.
- `binary` -> zapis/odczyt w trybie binarnym.

PRZYKŁAD ZAPISU DZIENNIKA POKŁADOWEGO ZAŁOGI STATKU

```
std::fstream diary("Day1.txt", diary.out | diary.app);
// or longer -> std::ifstream::out | std::ifstream::app
if (diary.is_open()) {
    std::cout << "OPENED!\n";
    diary << "Today is my first day on ship, with my crew\n";
    diary << "I'm a little scared!\n";
    diary << "Hope it will be a marvelous adventure.\n";
    diary.close();
}
```

Plik do zapisu, dopisujący na końcu nowe dane. Ponieważ wszystko traktujemy jak strumienie, możemy również pisać do pliku używając `operatora<<`.

PRZYKŁAD ODCZYTU DZIENNIKA POKŁADOWEGO ZAŁOGI STATKU

```
diary.open("Day1.txt", diary.in);
if (diary.is_open()) {
    std::string str;
    while (diary >> str) {
        std::cout << str << ' ';
    }
    diary.close();
}
```

Output: Today is my first day on ship, with my crew I'm a little scared! Hope it will be a marvelous adventure.

getline()

Jeżeli chcemy odczytywać dane linijka po linijce, możemy użyć funkcji `getline()`.

```
diary.open("Day1.txt", diary.in);
if (diary.is_open()) {
    std::string str;
    while (!getline(diary, str, '\n').eof()) {
        std::cout << str << '\n';
    }
    diary.close();
}
```

Q&A

POD - PLAIN OLD DATA

POD - PLAIN OLD DATA

Czyli klasa/struktura bez konstruktora, destruktora i metod wirtualnych.

```
class Pod {
public:
    void DoSth() {}
    bool ReturnTrue() { return true; }
    bool ReturnFalse() { return false; }

private:
    size_t index_;
    const char name_[15];
    double average_;
};
```

POD - PLAIN OLD DATA #2

Ale to już nie jest POD.

```
class Pod {
public:
    void DoSth() {}
    bool ReturnTrue() { return true; }
    bool ReturnFalse() { return false; }

private:
    size_t index_;
    std::string name_; // std::string has constructor!
    double average_;
};
```

ZAPISYWANIE POD

```
class Pod {
public:
    void DoSth() {}
    bool ReturnTrue() { return true; }
    bool ReturnFalse() { return false; }

    void SetName(std::string name) { strncpy(name_, name.data(), 15); }
    void SetIndex(size_t index) { index_ = index; }
    void SetAverage(double average) { average_ = average; }

    std::string GetName() const { return name_; }
    size_t GetIndex() const { return index_; }
    double GetAverage() const { return average_; }

private:
    char name_[15];
    size_t index_;
    double average_;
};

int main() {
    Pod mateusz;
    mateusz.SetName("Mateusz");
```

ODCZYT POD

```
student.open("Student.txt", student.binary | student.in);
Pod mateusz_read;
if (student.is_open()) {
    student.read(reinterpret_cast<char*>(&mateusz_read), sizeof(Pod));
    std::cout << "Name: " << mateusz_read.GetName() << '\n';
    std::cout << "Index: " << mateusz_read.GetIndex() << '\n';
    std::cout << "Average: " << mateusz_read.GetAverage() << '\n';
    student.close();
}
```

Hexdump:

```
0000000 614d 6574 7375 007a 0000 0000 0000 0000
0000010 e240 0001 0000 0000 0000 0000 0000 4014
0000020
```

Output:

```
Name: Mateusz
Index: 123456
Average: 5
```

STRINGSTREAM - JAK WYGODNIE UŻYWAĆ `std::string` JAKO STRUMIEŃ

Podzielenie `std::string` na pojedyncze słowa. Niestety nie możemy tego w tak prosty sposób zrobić dla innch znaków niż spacja.

```
std::string str {"Ala ma kota, a kot ma ale, ale to nie to samo, co Sierotka ma rysia."
std::istringstream iss(str);
std::vector<std::string> vec {std::istream_iterator<std::string>(iss), {}};
std::copy(begin(vec), end(vec), std::ostream_iterator<std::string>(std::cout, "\t"));
```

Output:

```
Ala     ma      kota,   a       kot     ma      ale,    ale     to      nie     to
```

`std::ostream_iterator` pozwala nam wpisywać dane do `std::cout`, w wygodny sposób.

STRINGSTREAM -> JAK SKONWERTOWAĆ NA STRING

```
std::stringstream ss;
ss << "End of passion play, crumbling away\n";
ss << "I'm your source of self-destruction\n";
ss << "Veins that pump with fear, sucking darkest clear\n";
ss << "Leading on your deaths' construction\n";
std::string str = ss.str();
std::cout << "str: " << str;
```

Output:

```
str: End of passion play, crumbling away
I'm your source of self-destruction
Veins that pump with fear, sucking darkest clear
Leading on your deaths' construction
```

Q&A

KONTENERY ASOCJACYJNE

POSORTOWANE

- set
- multiset
- map
- multimap

NIEUPORZĄDKOWANE

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

QUIZ

Do czego możesz użyć:

- `set`?
- `multiset`?
- `map`?
- `multimap`?
- `unordered_set`?
- `unordered_multiset`?
- `unordered_map`?
- `unordered_multimap`?

NOTACJA DUŻEGO O

CECHY `std::map<K, T>` I `std::multimap<K, T>` #1

- Forma drzewa binarnego (red-black tree)
- Nie jest cache friendly
- Pozwala przechowywać parę klucz-wartość (key-value)
- Multimapa może mieć wiele takich samych kluczy
- Mapa ma unikatowe klucze
- Alternatywą dla multimapy jest `std::map<key, std::vector<value>>`

CECHY `std::map<K, T>` I `std::multimap<K, T>` #2

Ponieważ jest ona zaimplementowana przeważnie jako red-black tree (GNU standard C++ library) to czas wstawiania, usuwania i dodawania elementu to `O(log(n))`.

Zalety:

- Szybkie wyszukiwanie elementów (kontenery sekwencyjnie `O(n)`),
- Względnie szybki czas ich dodawania i usuwania np. `std::vector<T>` dodaje elementy w środku w czasie `O(n)`, ale `std::list<T>` (jeżeli mamy podany iterator) w czasie `O(1)`. Stąd mapa jest względnie szybka zarówno w dodawaniu jak i usuwaniu.
- Idealna, gdy często poszukujemy danych, a rzadziej je dodajemy lub usuwamy.

Jeżeli będziemy jej używać jako zwykłego kontenera, to stracimy na wydajności. Mapę należy stosować wtedy, kiedy faktycznie chcemy posiadać pary klucz-wartość i często je wyszukiwać. W innym przypadku możemy użyć `std::vector<pair<K, V>>` lub innego kontenera.

OPERACJE NA `std::map<K, T>` I `std::multimap<K, T>`

- dodawanie elementu: `insert()`, `emplace()`, `emplace_hint()`. Dodatkowo mapa posiada: `insert_or_assign()`, `try_emplace()` oraz `operator[]` (dodająco modyfikujący)
- modyfikowanie/dostęp do elementu: `at()`, `operator[]` (Multimapa nie posiada takich opcji)
- pierwszy/ostatni element: Brak
- rozmiar/czy kontener jest pusty: `size()`, `empty()`
- wyczyszczenie nieużywanej pamięci: Brak
- iterator początku/końca: `begin()`, `end()`

OPERACJE NA `std::map<K, T>` I `std::multimap<K, T>` #2

- odwrócony (ang. reverse) iterator: `rbegin()`, `rend()`
- stały iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- wyczyszczenie kontenera: `clear()`
- przygotowanie elementu do usunięcia: Brak
- wymazanie elementów z pamięci: `erase()`
- podmiana całego kontenera: `swap()`
- zliczenie elementów pasujących do danego klucza: `count()` (dla mapy to 0 albo 1, dla multimapy od 0 do n)
- odnalezienie elementu o podanym kluczu: `find()`

PRZYKŁAD UŻYCIA `emplace_hint`

```cpp
int main() {
    std::map<int, std::string> map;

    auto it = map.begin();
    map.emplace_hint(it, 10, "Ten");

    std::cout << map[10] << '\n';
}
```

Output: Ten

Podpowiadamy mapie miejsce, gdzie powinna wstawić element, dzięki temu taka operacja będzie miała złożoność `O(1)`. Jeżeli jednak źle podpowiemy, to czas wstawienia będzie `O(log(n))`. Raczej rzadko stosowane ☺

PRZYKŁAD UŻYCIA insert_or_assign

```cpp
int main() {
    std::map<int, std::string> map;

    auto it = map.begin();
    map.insert_or_assign(it, 10, "Ten");
    std::cout << map[10] << '\n';
    map.insert_or_assign(it, 10, "Dziesiec");
    std::cout << map[10] << '\n';
    map[10] = "Cent";
    std::cout << map[10] << '\n';
}
```

Output:

```
Ten
Dziesiec
Cent
```

PRZYKŁAD UŻYCIA count

```
int main() {
    std::multimap<int, std::string> map;

    map.insert({5, "Five"});
    map.insert({5, "Funf"});
    map.insert({5, "Piec"});
    map.insert({5, "Cinq"});
    std::cout << map.count(5) << '\n';
}
```

Output: 4

PRZYKŁAD UŻYCIA `find`

```
int main() {
    std::multimap<int, std::string> map;

    map.insert({5, "Five"});
    map.insert({5, "Funf"});
    map.insert({5, "Piec"});
    map.insert({5, "Cinq"});
    auto it = map.find(5);

    for (; it != map.end() ; ++it) {
        std::cout << it->first << " | " << it->second << '\n';
    }
}
```

Output:

```
5 | Five
5 | Funf
5 | Piec
5 | Cinq
```

ZADANIE 1

- Stwórz multimapę i wypełnij ją podanymi wartościami

```
map.insert({5, "Ala"});
map.insert({5, "Ma"});
map.insert({5, "Kota"});
map.insert({5, "A"});
map.insert({5, "Kot"});
map.insert({5, "Ma"});
map.insert({5, "Ale"});
```

- Napisz funkcję, która wyświetli słowa w mapie o liczbie znaków równej 3.

ROZWIĄZANIE

```
std::vector<std::pair<int, std::string>> result;
std::copy_if(it,
             end(map),
             std::back_inserter(result),
             [](const auto& pair) { return pair.second.size() == 3; });
std::for_each(begin(result),
              end(result),
              [](const auto& pair) { std::cout << pair.second << '\n'; });
```

CECHY `std::set<T>` I `std::multiset<T>` #1

- Forma drzewa binarnego (red-black tree)
- Nie jest cache friendly
- Pozwala przechowywać wartości w uporządkowanej kolejności
- Multiset może mieć wiele takich samych wartości
- Set ma unikatowe wartości
- Alternatywą dla multiset jest posortowany `std::vector<T>`

CECHY `std::set<T>` I `std::multiset<T>` #2

Ponieważ jest on zaimplementowany przeważnie jako red-black tree (GNU standard C++ library) to czas wstawiania, usuwania i dodawania elementu to `O(log(n))`.

Zalety:

- Szybkie wyszukiwanie elementów (kontenery sekwencyjnie `O(n)`),
- Względnie szybki czas ich dodawania i usuwania np. `std::vector<T>` dodaje elementy w środku w czasie `O(n)`, ale `std::list<T>` (jeżeli mamy podany iterator) w czasie `O(1)`. Stąd `set` jest względnie szybki zarówno w dodawaniu jak i usuwaniu.
- Idealny, gdy chcemy zawsze posiadać posortowane wartości.

Jeżeli nie zależy nam, aby kontener był zawsze posortowany, lecz jedynie w specyficznych momentach, to może lepiej nam użyć `std::vector<T>` i sortować go, gdy przyjdzie taka potrzeba. Jeżeli też zależy nam tylko czasami na unikatowych wartościach, to możemy wtedy użyć `std::unique()`.

OPERACJE NA `std::set<T>` I `std::multiset<T>`

- dodawanie elementu: `insert()`, `emplace()`, `emplace_hint()`
- modyfikowanie/dostęp do elementu: Brak
- pierwszy/ostatni element: Brak
- rozmiar/czy kontener jest pusty: `size()`, `empty()`
- wyczyszczenie nieużywanej pamięci: Brak
- iterator początku/końca: `begin()`, `end()`
- odwrócony (ang. reverse) iterator: `rbegin()`, `rend()`

OPERACJE NA `std::set<T>` I `std::multiset<T>` #2

- stały iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- wyczyszczenie kontenera: `clear()`
- przygotowanie elementu do usunięcia: Brak
- wymazanie elementów z pamięci: `erase()`
- podmiana całego kontenera: `swap()`
- zliczenie elementów pasujących do danego klucza: `count()` (dla `set` to 0 albo 1, dla `multiset` od 0 do n)
- odnalezienie elementu o podanym kluczu: `find()`

PRZYKŁAD UŻYCIA `std::set<T>`

```cpp
std::set<int> set {5, 4, 3, 2, 1, 0, 6, 8, 7};
for (const auto el : set) {
    std::cout << el << ' ';
}
std::cout << '\n';

std::set<int, std::greater<int>> set2 {5, 4, 3, 2, 1, 0, 6, 8, 7};
for (const auto el : set2) {
    std::cout << el << ' ';
}
std::cout << '\n';
```

Output:

```
0 1 2 3 4 5 6 7 8
8 7 6 5 4 3 2 1 0
```

PRZYKŁAD UŻYCIA `std::multiset<T>`

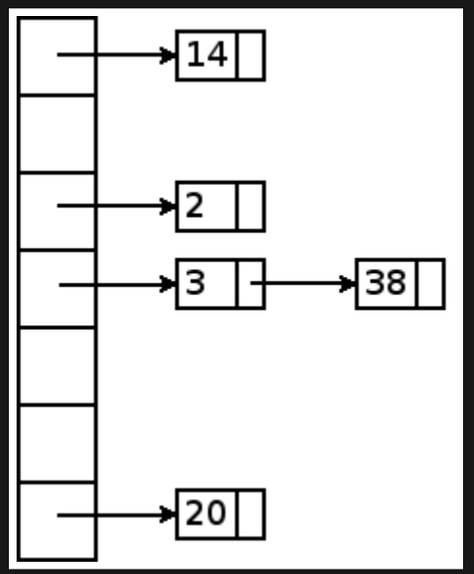
```cpp
std::multiset<int> set {5, 4, 3, 2, 1, 0, 6, 8, 7, 1, 2, 3, 4, 5, 6};
for (const auto el : set) {
    std::cout << el << ' ';
}
std::cout << '\n';

std::multiset<int, std::greater<int>> set2 {5, 4, 3, 2, 1, 0, 6, 8, 7, 1, 2, 3
for (const auto el : set2) {
    std::cout << el << ' ';
}
std::cout << '\n';
```

Output:

```
0 1 1 2 2 3 3 4 4 5 5 6 6 7 8
8 7 6 6 5 5 4 4 3 3 2 2 1 1 0
```

HASH TABLE



FUNKCJA MIESZAJĄCA

Jest to funkcja, która z dowolnego obiektu wygeneruje nam index w tablicy. Najważniejszą jej cechą jest to, że zawsze dla takich samych danych wejściowych musi wygenerować ten sam index. Kolejną ważną cechą jest takie generowanie indexu, aby tylko dla jednej kombinacji mógł on się powtórzyć np.:

```
size_t hash(const std::string& str) { return str.size(); }
```

```
size_t hash(const std::string& str) {
    size_t index = 0;
    for (size_t i = 0 ; i < str.size() ; ++i) {
        index += (int)str[i];
    }
    return index;
}
```

```
size_t hash(const std::string& str) {
    size_t index = 0;
    for (size_t i = 0 ; i < str.size() - 1 ; ++i) {
        index += ((int)str[i] * int(str[i + 1]) * (i + 5)) & (((int)str[i] + int(str[i + 1]) * i *
    }
    return index * str.size();
}
```

Oceń funkcje mieszające.

CECHY `std::unordered_set<T>` I `std::unordered_multiset<T>` #1

- Forma tablicy hash.
- Może, ale nie musi być cache friendly. Hash table, często są tworzone w formie hybrydy `std::vector<T>` i `std::list<T>`.
- Wartości nie są posortowane
- Multiset może mieć wiele takich samych wartości
- Set ma unikatowe wartości

CECHY `std::unordered_set<T>` I `std::unordered_multiset<T>` #2

Ponieważ jest on zaimplementowany jako hash table to średni czas dodawania, usuwania, dostępu oraz modyfikacji to `O(1)`. Najgorszy dla wszystkich operacji czas to `O(n)`.

Zalety:

- Błyskawiczne wyszukiwanie elementów
- Błyskawicznie szybki czas ich dodawania i usuwania
- Oczywiście przy założeniu, że mamy dobrą funkcję mieszającą.

Doskonały dla kontenerów read-only. Czas odczytu `O(1)`. Doskonały dla kontenerów mających dobrą funkcję mieszającą. Czas dodawania, dostępu i modyfikacji `O(1)`.

OPERACJE NA `std::unordered_set<T>` I `std::unordered_multiset<T>`

- dodawanie elementu: `insert()`, `emplace()`, `emplace_hint()`
- modyfikowanie/dostęp do elementu: Brak
- pierwszy/ostatni element: Brak
- rozmiar/czy kontener jest pusty: `size()`, `empty()`
- wyczyszczenie nieużywanej pamięci: Brak
- iterator początku/końca: `begin()`, `end()`
- odwrócony (ang. reverse) iterator: Brak

OPERACJE NA `std::unordered_set<T>` I `std::unordered_multiset<T>` #2

- stały iterator: `cbegin()`, `cend()`
- wyczyszczenie kontenera: `clear()`
- przygotowanie elementu do usunięcia: Brak
- wymazanie elementów z pamięci: `erase()`
- podmiana całego kontenera: `swap()`
- zliczenie elementów pasujących do danego klucza: `count()` (dla `set` to 0 albo 1, dla `multiset` od 0 do n)
- odnalezienie elementu o podanym kluczu: `find()`

PRZYKŁAD std::unordered_set<T> I std::unordered_multiset<T>

```
std::unordered_set<std::string> set{"Ala", "Ma", "Kota", "A", "Kot", "Ma", "AL
for (const auto el : set) {
    std::cout << el << ' ';
}
std::cout << '\n';
set.insert("Ala");
set.insert("Ala");
set.insert("Ala");
for (const auto el : set) {
    std::cout << el << ' ';
}
std::cout << '\n';
```

`unordered_map<K, T>` | `unordered_multimap<K, T>`

- Zgadnij cechy,
- Zgadnij implementację,
- Wyślij przykład wykorzystujący te 2 kontenery

ZADANIE 2

- Stwórz `std::unordered_map<int, std::string>` oraz `std::multiset<int>`
- Wypełnij je dowolnymi wartościami
- Usuń jeden z elementów,
- dodaj dodatkowy element
- Usuń elementy, według wymyślonego przez Ciebie predykatu.

Q&A

ZADANIE DOMOWE

ZADANIE 1

Napisz funkcję, która umożliwi zapisywanie nowych przepisów w pliku `recipes.txt`. Zwróć false, jeżeli nie uda się zapisać przepisu.

- Funkcja za 1 argument przyjmuje dane w `std::vector<std::string>`, który zawiera kolejne kroki przepisu.
- Funkcja za 2 argument przyjmuje `std::list<std::string>` zawierającą nazwę składników.
- Funkcja za 3 argument przyjmuje dane w `std::deque<std::pair<size_t, char>>` zawierające informacje o ilości składnika i jego reprezentacji `g` -> gramy `m` -> mililitry, `s` -> szklanki.

```
bool AppendNewRecipe(std::vector<std::string> steps,
                     const std::list<std::string>& ingredients,
                     const std::deque<std::pair<size_t, char>>& amount);
```

ZADANIE1 #2

Input:

```
std::vector<std::string> steps{"Wsypać do miski 20 gram cukru",
                               "Dorzucić 1 szklankę mąki",
                               "Dokładnie wymieszać",
                               "Nalać 40ml rumu do kieliszka",
                               "Wypić kieliszek",
                               "Wysypać zawartość miski"};
std::list<std::string> ingredients{"cukru", "mąki", "rumu"};
std::deque<std::pair<size_t, char>> amount{
    {20, 'g'},
    {1, 's'},
    {40, 'm'}};
```

ZADANIE1 #3

Oczekiwany format:

```
Składniki:
20 gram cukru,
1 szklanka(i) mąki,
40ml rumu,

Kroki:
1) Wsypać do miski 20 gram cukru.
2) Dorzucić 1 szklankę mąki.
3) Dokładnie wymieszać.
4) Nalać 40ml rumu do kieliszka.
5) Wypić kieliszek.
6) Wysypać zawartość miski.
```

ZADANIE1 #4

Pomocnicze funkcje:

```
std::vector<std::string> FormatIngredients(const std::list<std::string>& ingredients,
                                           const std::deque<std::pair<size_t, char>>& a
```

każdy rekord wektora powinien zawierać gotową sekcję jednego składnika np: `20 gram cukru` lub `1 szklanka(i) mąki`.

```
std::stringstream FormatRecipit(std::vector<std::string> steps,
                                const std::list<std::string>& ingredients,
                                const std::deque<std::pair<size_t, char>>& amount);
```

Oczekujemy gotowego formatu, który można od razu zapisać.

ZADANIE 2

Poczytaj i stwórz krótką notatkę oraz przykład wykorzystania następujących adapterów:

- stack
- queue
- priority_queue

Najciekawsze opracowanie nagrodzę 30xp. Drugie miejsce otrzyma 20xp a trzecie 10xp.

CODERS SCHOOL