Семинар #4: Контейнеры STL.

Часть 1: Контейнеры

Стандартная библиотека шаблонов (STL = Standard Template Library) включает в себя множество разных шаблонных контейнеров и алгоритмов для работы с ними.

контейнер	описание и основные свойства
`std::vector`	Динамический массив Все элементы лежат вплотную друг к другу, как в массиве Есть доступ по индексу за $O(1)$
`std::list`	Двусвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на элемент
`std::forward_list`	Односвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на предыдущий элемент
`std::set`	Реализация множества на основе сбалансированного дерева поиска Хранит элементы без дубликатов, в отсортированном виде Тип элементов должен реализовать `operator<` (или предоставить компаратор) Поиск/вставка/удаление элементов за $O(\log(N))$
`std::map`	Реализация словаря на основе сбалансированного дерева поиска Хранит пары ключ-значения без дубликатов ключей, в отсортированном виде Тип ключей должен реализовать `operator<` (или предоставить компаратор) Поиск/вставка/удаление элементов за $O(\log(N))$
`std::unordered_set`	Реализация множества на основе хеш-таблицы Хранит элементы без дубликатов, в произвольном порядке Поиск/вставка/удаление элементов за $O(1)$ в среднем
`std::unordered_map`	Реализация словаря на основе хеш-таблицы Хранит пары ключ-значения без дубликатов ключей,в произвольном порядке Поиск/вставка/удаление элементов за $O(1)$ в среднем
`std::multiset`	То же самое, что `std::set`, но может хранить дублированные значения
`std::deque`	Двухсторонняя очередь Добавление/удаление в начало и конец за $O(1)$

`std::stack`
`std::queue`
`std::priority_queue`
	Стек
	Очередь
	Очередь с приоритетом

`std::pair`	Пара элементов, могут быть объектами разных типов Элементы пары хранятся в публичных полях `first` и `second`
`std::tuple`	Фиксированное количество элементов, могут быть объектами разных типов
`std::array`	Массив фиксированного размера, все элементы имеют один тип

Часть 2: `std::set`

`std::set` – это реализация множества с помощью бинарного дерева поиска. Не хранит дупликатов. При попытке добавить в множество тот элемент, который в нём уже есть, ничего не произойдёт. Также все элементы в множестве всегда хранятся в отсортированном виде (так как это бинарное дерево поиска). Для типа элементов множество должен быть реализован `operator<`. В `std::set` нельзя менять элементы, так как это бинарное дерево поиска, но можно удалить элемент, а потом вставить новый.

Основные методы для работы с множеством:

метод	описание
`insert`	Вставляет элемент в множество
`erase`	Удаляет элемент. Можно удалять по значению элемента или по итератору. Также можно сразу удалить диапазон значений, если передать 2 указателя
`find(x)`	Принимает на вход значение `x` и ищет такой элемент в множестве. Возвращает итератор на этот элемент или итератор `end()`, если такого элемента нет
`count(x)`	Принимает значение и находит, сколько элементов равны этому значению (т.е. 0 или 1)
`lower_bound(x)`	Возвращает итератор на первый элемент, который больше или равен `x`
`upper_bound(x)`	Возвращает итератор на первый элемент, который больше `x`

- На вход подаётся n чисел. Напечатайте эти числа удалив все дупликаты.

вход	выход
10 8 2 1 2 2 1 8 7 1 2	1 2 7 8

`std::multiset`

То же самое, что и `std::set`, но может хранить дупликаты. Одна из неочевидных особенностей `multiset` это то, что при удалении элемента по значению `erase(x)`, удалятся все элементы, равные `x`. Для удаления одного элемента нужно передать в `erase` итератор на элемент.

- Считайте n чисел и отсортируйте их с помощью вставки в `multiset`. Распечатайте отсортированные числа.
- На прямой лежит верёвка длиной n метров. Затем её начинают последовательно разрезать. Все места разрезов – целые числа. Найти длину самого длинного куска после каждого разреза.

вход	выход
20 8 8 10 15 1 7 4 11 18	12 10 8 7 6 5 5 4

Часть 3: `std::map`

`std::map` – это реализация словаря с помощью бинарного дерева поиска. Не хранит ключе - дупликатов. При попытке добавить в этот словарь элемента с ключом, который в нём уже есть, ничего не произойдёт. Также все элементы в этом словаре всегда хранятся в отсортированном по ключам виде (так как это бинарное дерево поиска). Для типа ключей должен быть реализован `operator<`. В `std::map` можно менять значения, но нельзя менять ключи, так как это бинарное дерево поиска. Но можно удалить элемент каким-то ключом, а потом вставить новый с другим ключом.

Основные методы для работы с множеством:

метод	описание
`insert(k, v)`	Вставляет элемент с ключом `k` и значением `v` Если такой элемент уже есть, то ничего не делает
`operator[]` `m[k] = v`	Вставляет элемент с ключом `k` и значением `v` Если такой элемент уже есть, то меняет его значение
`erase(k)`	Удаляет элемент. Можно удалять по значению элемента или по итератору. Также можно сразу удалить диапазон значений, если передать 2 указателя
`find(x)`	Принимает на вход значение `x` и ищет элемент с таким ключом. Возвращает итератор на этот элемент или итератор `end()`, если такого ключа нет
`count(x)`	Принимает значение и находит, сколько ключей равны этому значению (т.е. 0 или 1)
`lower_bound(x)` `upper_bound(x)`	Возвращает итератор на первый элемент, который больше или равен `x` Возвращает итератор на первый элемент, который больше `x`

Пример программы, которая создаёт словарь из пар <название города, его население>. Строка выступает в качестве ключа, а целое число – в качестве значения.

```
#include <iostream>
#include <string>
#include <map>
using std::cout, std::endl;

int main ()
{
    std::map<string, int> m = {{"London", 8900000}, {"Moscow", 12500000}, {"Milan", 4300000}};

    std::string cityName;
    while (true)
    {
        std::cin >> cityName;
        if (cityName == "q" || cityName == "quit")
            break;

        std::map<std::string, int>::iterator it = m.find(cityName);
        if (it == m.end())
            cout << "No such city" << endl;
        else
            cout << "City " << cityName << " population = " << it->second << endl;
    }
}
```

- На вход подаётся n чисел и некоторое число `x`. Найдите пару элементов массива, такую что их сумма равна `x`. Напечатайте индексы этих элементов. При наличии нескольких таких пар, напечатайте любую. Решение должно работать за $O(n \log(n))$ или быстрее.

вход	выход
8 8 2 5 4 9 1 7 4 14	2 4

- Напишите программу, которая будет в бесконечном цикле считывать слова и после каждого считывания печатать все уникальные слова, считанные ранее и количество таких слов. Например, если пользователь ввёл слово `Cat` три раза, слово `Dog` 1 раз и слово `Elephant` 2 раза. То после очередного считывания программа должна напечатать:

```
Dictionary:
Cat: 3
Dog: 1
Elephant: 2
```

- Считайте все слова из файла и напечатайте все уникальные слова и то, как часто они встречались в файле. Сохраните результат в новом файле. Для работы с файлами можно использовать функции `C`.

входной файл	выходной файл
I'm having Spam, Spam, Spam, Spam, Spam, Spam, Spam, baked beans, Spam, Spam, Spam and Spam.	I'm 1 Spam 1 Spam, 9 Spam. 1 and 1 beans, 1 having 1

Часть 3: Ключевое слово `auto`

Ключевое слово `auto` используется для автоматического вывода типа.

```
#include <string>
int main()
{
    auto a = 123;   // a будет иметь тип int
    auto b = 4.1;   // b будет иметь тип double
    auto b = 4.1f;  // b будет иметь тип float

    auto s1 = "Hello";               // s1 будет иметь тип const char*
    auto s2 = std::string("Hello");  // s2 будет иметь тип std::string
}
```

Задачи:

- В примере ниже создан вектор строк и напечатано его содержимое. Тип итератора имеет очень длинное название (и название будет ещё больше если контейтер будет хранить не просто строки, а что-нибудь посложнее). Используйте `auto`, чтобы упростить код.

  ```
  #include <iostream>
  #include <vector>
  #include <string>

  int main()
  {
      std::vector<std::string> v {"Cat", "Dog", "Elephant"};
      for (std::vector<std::string>::iterator it = v.begin(); it != v.end(); ++it)
          std::cout << *it << std::endl;
  }
  ```

- Протестируйте, можно ли использовать `auto` вместо возвращаемого типа функции. Напишите функцию, которая принимает на вход вектор строк и возвращает строку, которая является результатом конкатенации всех строк. Вместо возвращаемого типа используйте `auto`.

- Протестируйте, можно ли создать функцию, которая будет принимать целое число и, в зависимости от этого числа, возвращать значения разных типов. (Если вместо возвращаемого типа используется `auto`).

- Протестируйте, можно ли использовать о указатель с помощью `auto`. Пусть есть такой участок кода:

  ```
  int a = 123;
  auto p = &a;
  auto* q = &a;
  ```

  Какой тип будет у `p` и `q`?

- Функция вычисления факториала, написанная ниже с использованием `auto` не работает.

  ```
  auto factorial(int n)
  {
      if (n > 0)
          return n * factorial(n - 1);
      return 1;
  }
  ```

  Почему? Исправьте эту функцию, не убирая `auto`.

Часть 4: Range-based циклы

Циклы, основанные на диапазоне, предоставляют более простой способ обхода контейнера:

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

int main()
{
    std::vector v {6, 1, 7, 4};
    for (int num : v)
        std::cout << num << std::endl;
}
```

Для изменения элементов контейнера при обходе нужно использовать ссылки:

```
for (int& num : v)
    num += 1;
```

Задачи:

- Проверьте, можно ли использовать ключевое слово `auto` внутри таких циклов.
- Пусть у нас есть вектор строк:

  ```
  vector<string> v {"Cat", "Axolotl", "Bear", "Elephant"};
  ```

  - Напишите range-based цикл, который будет печатать все элементы вектора
  - Напишите range-based цикл, который будет добавлять в конец каждой строки символ `s`.
  - Напишите range-based цикл, который будет обращать каждую строку. Используйте стандартную функцию `reverse`.
- Проверьте, можно ли использовать range-based циклы если контейнер является:
  - `std::list`
  - `std::set`
  - `std::map`
  - `std::pair`
  - Обычным массивом
  - `std::string`
  - Строкой в стиле `C`
- Для печати массива целых чисел была написана следующая функция:

  ```
  void print(int array[])
  {
      for (int num : array)
          std::cout << num << std::endl;
  }
  ```

  Оказывается, что она не работает. В чём заключается ошибка?

Часть 5: Structure binding (структурное связывание)

В стандарте C++17 был добавлен новый вид объявления и инициализации нескольких переменных. В коде ниже мы объявляем переменные a и b одной строкой с помощью структурного связывания.

```
#include <iostream>
#include <utility>

int main()
{
    std::pair p {5, 1};
    auto [a, b] = p;

    std::cout << a << " " << b << std::endl;
}
```

Структурное связывание работает только в том случае, если размер контейнера справа известен на стадии компиляции. Например, пары, кортежи(std::tuple), статические массивы, std::array, простые структуры.

Задачи:

- Пусть у нас есть пара:

  ```
  std::pair p {std::string{"Moscow"}, 1147};
  ```

  - Создайте две переменные name и age и присвойте их соответствующим элементам пары.
  - Создайте две ссылки name и age и инициализируйте их соответствующими элементами пары. Убедитесь, что при изменении переменной name меняется и пара p.

- Метод insert контейнера std::set пытается вставить элемент в множество. Если же такой элемент в множестве уже существует, то он ничего с множеством не делает. Но этот метод возвращает пару из итератора на соответствующий элемент и переменной типа bool, которая устанавливается в true если новый элемент был добавлен и в false, если такой элемент уже существовал. Вот пример программы, которая пытается вставить элемент в множество и печатает соответствующее сообщение. В любом случае программа печатает все элементы, меньшие вставляемого.

  ```
  #include <iostream>
  #include <utility>
  #include <set>
  using std::cout, std::endl;

  int main()
  {
      std::set<int> s {1, 2, 4, 5, 9};

      std::pair<std::set<int>::iterator, bool> result = s.insert(5);
      if (result.second == true)
          cout << "Element added successfully" << endl;
      else
          cout << "Element already existed" << endl;

      for (std::set<int>::iterator it = s.begin(); it != result.first; ++it)
          cout << *it << " ";
  }
  ```

  Упростите эту программу, используя ключевое слово auto и структурное связывание.

Структурное связывание можно использовать и в цикле.

```
#include <iostream>
#include <utility>
#include <vector>

int main()
{
    std::vector<std::pair<std::string, int>> v {{"Moscow", 1147}, {"Berlin", 1237},
                                                {"Rome", -753},  {"Bogota ", 1538}};
    for (auto [city, year] : v)
        std::cout << city << " " << year << std::endl;
}
```

Задачи:

- В файле `books.cpp` лежит заготовка кода. В ней содержится инициализированный массив из структур. Сделайте следующее:
  - Напечатайте массив `books`, используя range-based цикл. Нужно напечатать все поля через запятую.
  - Напечатайте массив `books`, используя range-based цикл со структурным связыванием.
  - Увеличьте поле `price` всех книг на одну величину, используя range-based цикл.
  - Увеличьте поле `price` всех книг на одну величину, используя range-based цикл со структурным связыванием.
- Ниже есть пример программы – решение задачи с предыдущего семинара. Она считывает слова и печатает количества всех введённых до этого слов. Упростите код этой программы, используя `auto` и структурное связывание.

```
#include <iostream>
#include <map>
#include <utility>
#include <string>
using std::cout, std::endl;

int main()
{
    std::map<std::string, int> wordCount;
    while (true)
    {
        std::string word;
        std::cin >> word;
        std::pair<std::string, int> wc {word, 1};
        std::pair<std::map<std::string, int>::iterator, bool> p = wordCount.insert(wc);
        if (p.second == false)
            wordCount[word] += 1;

        cout << "Dictionary:" << endl;
        for (std::map<std::string, int>::iterator it = wordCount.begin();
                                it != wordCount.end(); ++it)
        {
            cout << (*it).first << ": " << (*it).second << endl;
        }
        cout << endl;
    }
}
```