Задание 6

Реализация контейнеров

Поймём основные принципы построения самых популярных структур данных. А также научимся использовать шаблоны в своём коде.

Контейнеры очень полезны в повседневной практике кодирования. Такого рода объекты предназначены для хранения некоторого (обычно переменного, но иногда и константного) набора из других объектов. Дополнительным преимуществом является удобный интерфейс с относительно широким набором возможностей.

Перед контейнерами обычно не ставится задача хранения объектов разного типа – из-за сложности реализации (да и использования) такого рода хранения. Поэтому подавляющее большинство контейнеров представляют собой шаблонный класс с минимум одним параметром шаблона – типом хранимых объектов.

Контейнеры разделяют:

1. По принципу хранения объектов на:
    a. контейнеры динамического размера
    b. и контейнеры константного размера (размер определяется на этапе компиляции)

2. По дополнительным инвариантам на:
    a. последовательные контейнеры
    b. контейнеры спискового типа
    c. ассоциативные контейнеры
    d. хешированные ассоциативные
    e. и прочие типы…

Более подробно отличия разных типов контейнеров изучим на занятиях. А пока познакомимся с двумя целевыми с точки зрения задания типами – последовательными и контейнерами спискового типа.

Последовательные контейнеры предоставляют гарантию последовательного расположения элементов в памяти. Так, получив указатель на некий i-ый элемент контейнера, можно получить указатель на (i+1)-ый элемент обычной арифметикой указателей:

```
MyContainer values;
values.push_back(0);
values.push_back(1);

int *ptr = &values[0];
std::cout << *ptr << std::endl; // на экране 0
ptr += 1;
std::cout << *ptr << std::endl; // на экране 1
```

Реализация такого рода контейнеров подразумевает некоторую линейную область памяти в качестве мембера, а также расширение этой области памяти по необходимости. Так как язык C++ не предоставляет функциональности гарантированного расширения уже существующей области памяти, приходится это расширение эмулировать – через создание новой области памяти целевого (нового) размера и копирование в неё всех элементов из старой области памяти.

```
void MyContainter::push_back(T value) {
        T* new_region = new T[m_size + 1]; // новая область памяти
        for (size_t i = 0; i < m_size; ++i) {
                new_region[i] = m_region[i]; // копирование элементов
        }
        new_region[m_size] = value; // добавление нового элемента
        delete [] m_region; // удаление старой области
        m_region = new_region; // сохранение новой в мембер
        m_size += 1; // обновление информации о размере
}
```

Удаление элементов из контейнера происходит по тому же сценарию.

Контейнеры, построенные по списковому принципу, формируются несколько иначе. Вместо того, чтобы хранить все элементы в единой области памяти, каждый элемент сохраняется независимо. А для связи между элементами используются указатели на следующий и/или предыдущий элемент. Это достигается хранением непосредственно пользовательских данных во вспомогательных объектах – так называемых узлах. Дополнительно узлы хранят служебную информацию – те самые указатели на следующий и/или предыдущий элементы.

Такая схема реализации исключает возможность использования арифметики указателей для доступа к различным элементам. Но позволяет весьма экономно с точки зрения ресурсов осуществить операции вставки и удаления.

```
struct Node {
        Node* next; // указатель на следующий элемент Node
        Node* prev; // указатель на предыдущий элемент Node
        T data; // пользовательские данные (хранимый объект)
};

void MyContainer::push_back(T value) {
        Node* new_node = new Node{}; // создание нового узла
        new_node.prev = m_last; // предыдущим элементом станет последний
        new_node.next = nullptr; // следующего элемента пока нет
        new_node.data = value; // сохраняем пользовательские данные
        m_last = new_node; // обновляем указатель на последний
        m_size += 1; // обновляем размер
}
```

Формулировка задания

В задании требуется реализовать два контейнера динамического размера – последовательный и спискового типа – и пользовательский код с демонстрацией возможностей этих контейнеров.

1. Последовательный контейнер
   Отличительная черта такого контейнера – расположение элементов друг за другом подряд (как и у обычного массива в стиле Си).

2. Контейнер спискового типа – не предоставляет гарантий расположения элементов друг за другом в памяти. Связь между элементами осуществляется через указатели (указатель на предыдущий элемент и/или указатель на следующий элемент).

Пользовательский код (вызывается из функции main) должен содержать следующий набор действий с обоими контейнерами:

1. создание объекта контейнера для хранения объектов типа `int`
2. добавление в контейнер десяти элементов (0, 1 … 9)
3. вывод содержимого контейнера на экран
   ожидаемый результат: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
4. вывод размера контейнера на экран
   ожидаемый результат: 10
5. удаление третьего (по счёту), пятого и седьмого элементов
6. вывод содержимого контейнера на экран
   ожидаемый результат: 0, 1, 3, 5, 7, 8, 9
7. добавление элемента 10 в начало контейнера
8. вывод содержимого контейнера на экран
   ожидаемый результат: 10, 0, 1, 3, 5, 7, 8, 9
9. добавление элемента 20 в середину контейнера
10. вывод содержимого контейнера на экран
    ожидаемый результат: 10, 0, 1, 3, 20, 5, 7, 8, 9
11. добавление элемента 30 в конец контейнера
12. вывод содержимого контейнера на экран
    ожидаемый результат: 10, 0, 1, 3, 20, 5, 7, 8, 9, 30

Требования к минимальному интерфейсу:

- метод/ы (может быть несколько) добавления элементов в конец контейнера (**push_back**)

- метод/ы вставки элементов в произвольную позицию (**insert**)

- метод/ы удаления элементов из контейнера (**erase**)

- метод получения текущего размера контейнера (**size**)

- метод/ы получения доступа по индексу (**operator[]**)

Допустимо добавить и другие вспомогательные методы с пользовательским кодом, демонстрирующим их назначение.

Дополнительное задание 1. Реализовать последовательный контейнер с небольшим резервированием памяти заранее. Так при добавлении нового элемента контейнер сначала создаёт область памяти больше по размеру, чем нужно для размещения элементов (например, на 50%). Таким образом при последующих добавлениях новых элементов можно будет избежать повторного выделения памяти и копирования.

Примечание: выделять память со слишком большим запасом – плохое решение, так как приведёт к слишком значительному перерасходу памяти. Обычно используют коэффициенты 1.5, 1.6, 2.0.

Дополнительное задание 2. Реализовать два варианта спискового контейнера:

- двунаправленный список – каждый узел хранит указатель на следующий элемент и указатель на предыдущий элемент
- однонаправленный список – каждый узел хранит указатель только на следующий элемент

Дополнительное задание 3. Использовать в реализации семантику перемещения:

- реализовать перемещающие конструктор и оператор присваивания для контейнеров
- уметь принимать на вход r-value ссылку на пользовательский объект

Дополнительное задание 4. Реализовать дополнительную структуру – итератор, инкапсулирующую в себе логику обхода контейнера. Реализовать методы у контейнера:

- **begin()** – возвращает итератор на начало контейнера

- **end()** – возвращает итератор на конец контейнера

Реализовать методы у итератора:

- **operator*()** – унарный оператор разыменования

- и/или метод **get()** для получения значения.

Пример кода, который должен работать после реализации:

```
struct MyContainer {
	struct iterator {
		// …
	};
	// …

};

void test_func() {

	MyContainer values;
	// заполнение контейнера элементами
	for (auto iter = values.begin(); iter != values.end(); ++iter) {
		std::cout << *iter << std::endl;
		// или:
		// std::cout << iter.get() << std::endl;

	}
}
```

Итоговые требования

1. реализовать два контейнера и код их использования
2. для сборки использовать CMake
3. выгрузить результат на github.com в свой аккаунт

В Чат с преподавателем в Личном кабинете отправить:

1. ссылку на репозиторий с исходниками приложения