Лабораторная работа №4

по курсу «Высокоуровневое программирование» (2 семестр)

«Знакомство с контейнерами»

Оглавление

Цель: приобретение практических навыков по созданию и обработке контейнеров данных.

Задачи:

1. Изучить понятие контейнера;
2. Познакомиться с итераторами и научиться применять их;
3. Ознакомиться с классами-контейнерами в языке C++.

Содержание отчёта:

1. Титульный лист;
2. Цель, задачи работы;
3. Формулировка общего задания;
4. Листинги пользовательских классов и основной программы;
5. Результаты работы программы;
6. Выводы по работе в целом.
7. Ответы на контрольные вопросы.

Основные теоретические сведения

Контейнеры

Контейнер в языке C++ – это класс, предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных). Существует много разных контейнерных классов, каждый из которых имеет свои преимущества, недостатки или ограничения в использовании. Наиболее часто используемым контейнером в программировании является массив. Хотя в языке C++ определена стандартная реализация массива, большинство программистов используют контейнерные классы-массивы (std::array или std::vector) из-за их преимуществ: контейнерные классы-массивы имеют возможность динамического изменения своего размера. Это не только упрощает их использование, но делает их более безопасными.

Обычно, функционал классов-контейнеров языка C++ следующий:

- Создание пустого контейнера (через конструктор);
- Добавление нового объекта в контейнер;
- Удаление объекта из контейнера;
- Просмотр количества объектов, находящихся на данный момент в контейнере;
- Очистка контейнера от внутренних объектов;
- Доступ к сохраненным объектам;
- Сортировка объектов (не всегда).

Иногда функционал контейнерных классов может быть не столь обширным, как это указано выше. Например, пользовательские контейнерные классы-массивы часто не имеют функционала добавления/удаления объектов.

Типом отношений в классах-контейнерах является «член чего-либо». Например, элементы массива «являются членами» массива (принадлежат ему).

Типы контейнерных классов

Контейнерные классы обычно бывают двух типов:

- Контейнеры значения – это композиции, которые хранят **копии объектов** (и, следовательно, ответственные за создание/уничтожение этих копий).
- Контейнеры ссылки – это агрегации, которые хранят **указатели или ссылки** на другие объекты (не ответственные за создание/уничтожение этих объектов).

C++ не позволяет смешивать разные типы данных внутри одного контейнера. Например, если у вас целочисленный массив, то он может содержать только целочисленные значения. Если вам нужны контейнеры для хранения значений типов int и double, то вам придется написать два отдельных контейнера (или использовать шаблоны, о которых мы поговорим на соответствующем уроке). Несмотря на ограничения их использования, контейнеры чрезвычайно полезны, так как делают программирование проще, безопаснее и быстрее.

Итераторы

Итерация или перемещение по элементам является довольно распространенным действием в программировании. Способов прохождения по элементам контейнера множество, а именно: с использованием циклов и индексов (циклы for и while), с помощью указателей и адресной арифметики, а также с помощью циклов for с явным указанием диапазона.

Использование циклов с индексами для доступа к элементам, требует написания большего количества кода. При этом необходимо, чтобы контейнер, содержащий данные, давал возможность прямого доступа к своим элементам (что делают массивы, но не делают, например, списки).

Итератор – это объект, разработанный специально для перебора элементов контейнера и обеспечивающий во время перемещения по элементам доступ к каждому из них.

Использование интерфейса итераторов позволяет не беспокоиться о том, какой тип перебора элементов задействован в алгоритме и каким образом в контейнере хранятся данные. Итераторы в языке C++ обычно используют один и тот же интерфейс как для перемещения по элементам контейнера (оператор ++ для перехода к следующему элементу), так и для доступа к ним (оператор * для доступа к текущему элементу).

Контейнеры могут предоставлять различные виды итераторов. Например, контейнер на основе массива может предлагать прямой итератор (проходится по массиву в прямом порядке) и реверсивный (в обратном порядке).

Простейший пример итератора – это указатель, который (используя адресную арифметику) работает с последовательно расположенными элементами данных. Также, в качестве итератора мы можем использовать умный указатель, но, хорошей практикой считается написать отдельный итератор для взаимодействия с контейнерами. В стандартной библиотеке C++ есть уже реализованные итераторы и контейнеры, которые используют «шаблоны» для унификации кода.

Итераторы и циклы for с явным указанием диапазона

Все типы данных, которые имеют методы работы с итераторами **begin() и end()** или используют std::begin() и std::end(), могут быть задействованы в **циклах for с явным указанием диапазона**:

```
const int len = 5;
int arr[len]{ 1, 2, 3, 4, 5 };
for (int it : arr) {
	cout << it << endl;
}
```

Эта форма цикла for создана специально для перебора различных коллекций объектов и является аналогом подобной записи:

```
for (int i{}; i < len; ++i) {
	cout << arr[i] << endl;
}
```

Согласитесь, что первая форма намного удобнее. Но в таком виде, как она используется в первом примере, она не работает с присвоением значений массиву. Для такой возможности используют ссылку:

```
for (int &it : arr) {
	cin >> it;
}
```

Циклы for с явным указанием диапазона для осуществления итерации незаметно обращаются к вызовам функций begin() и end(). Тип данных std::array также имеет в своем арсенале методы begin() и end(), а значит и его мы можем использовать в циклах for с явным указанием диапазона. Массивы C-style с фиксированным размером также можно использовать с функциями std::begin() и std::end(). Однако с динамическими массивами данный способ не работает, так как в них не существует функции std::end() (отсутствует информация о длине массива).

Создадим итератор, который будет работать с классом-контейнером для строк.

```
class StrIterator {
	// делаем внешний класс дружественным к подклассу
	// чтобы он имел доступ к нашим закрытым свойствам
	friend class MyString;

public:
	// пишем конструктор копирования
	// который будет производить инициализацию членов класса
	// тело конструктора будет пустым
	StrIterator(const StrIterator& it) : m_symbol(it.m_symbol) {}

	// перегружаем оператор сравнения
	bool operator==(const StrIterator& it) const {
```

```
            return m_symbol == it.m_symbol;
        }

        // перегружаем оператор неравенства
        bool operator!=(const StrIterator& it) const {
            return m_symbol != it.m_symbol;
        }

        // перегружаем оператор инкремента
        StrIterator& operator++() {
            ++m_symbol;
            return *this;
        }

        // перегружаем оператор разыменования указателя
        char& operator*() const {
            return *m_symbol;
        }

    private:
        char* m_symbol{};
        // создаём закрытый конструктор инициализации членов класса
        StrIterator(char* p) : m_symbol(p) {}
};
```

Методы получения указателя на первый и следующий за последним элементы строки должны находиться в классе-контейнере и возвращать итераторы:

```
// возврат итератора на первый элемент
iterator begin() {
    return iterator(m_chars);
}

// возврат итератора на следующий за последним элемент
iterator end() {
    return iterator(m_end);
}

// возврат константного итератора на первый элемент
const_iterator begin() const {
    return const_iterator(m_chars);
}
```

```
// возврат константного итератора на следующий за последним элемент
const_iterator end() const {
	return const_iterator(m_end);
}
```

Обращение с циклом for будет выглядеть так:

```
MyString str("Hi!");

for (char ch : str) {
	cout << ch << endl;
}
```

Делегирующий конструктор – конструктор, который вызывает другой конструктор. Такой способ сокращает количество кода и делает его более аккуратным. Особенность написания заключается в том, что второй конструктор вызывается не в теле первого, а в заголовке после двоеточия. Только так можно вызывать другие конструкторы! Если мы попытаемся вызвать второй конструктор в теле, то получим создание анонимного объекта.

```
MyString(const char* str) : MyString() {}
```

Укороченная инициализация. Через двоеточие мы указываем имя поля и вызываем конструктор у этого поля передавая туда параметр, которым его необходимо инициализировать. Полей может быть сколько угодно, все они могут быть инициализированы таким способом, через запятую.

```
StrIterator(char* p) : m_symbol(p)
```

Таким же образом мы создавали делегирующий конструктор, который вызывал другой конструктор, здесь нужно сказать, что делегирующий конструктор, не может инициализировать поля класса, также, конструктор, который инициализирует поля класса не может быть делегирующем.

```
StrIterator(const StrIterator& it) : m_symbol(it.m_symbol) {}
```

Дружественные классы. Методы дружественного класса будут иметь доступ к сокрытым членам класса. `friend class MyString;`

Стандартные классы-контейнеры в языке C++

В программировании есть несколько общепринятых структур контейнеров. Все их образные представления возникли на основе массивов, но реализуют различный интерфейс доступа к элементам или структуру хранения. Так, например, очередь строго ограничивает порядок доступа к элементам, а списки помимо самих значений хранят дополнительное поле для ссылки на следующий элемент. Определив ограничения и дополнения к набору данных, можно создать собственную структуру хранения элементов, подходящую для конкретной задачи.

Рассмотрим два самых основных контейнера:

Вектор – класс-контейнер представляющий из себя *динамический массив произвольного доступа* с автоматическим *изменением размера* при добавлении/удалении элемента. Главное его дополнение к обычному массиву – при добавлении и удалении элементов контейнер сам корректирует длину.

Алгоритм добавления элемента: создается динамический массив большего размера, чем нынешний, в него *перемещаются элементы из старого массива* при помощи конструктора копирования и *добавляется новый элемент*, после чего старый массив удаляется. Также изменяется поле размера массива. Аналогичным образом работает удаление.

Можно, конечно, при каждом добавлении элемента совершать данный алгоритм, но создание массива, копирование элементов и удаление – очень затратные по времени операции. Для совершенствования операций было принято увеличивать размер реже, но больше, чем на 1 элемент (удлинять сразу в два раза). То есть увеличение длины происходит только с 4 до 8, с 8 до 16, с 16 до 32 и т. д.

Осуществляется это за счет хранения в контейнере двух длин массива: текущей и максимальной. Максимальная длина – это количество выделенных для хранения ячеек. Текущая – количество используемых. При добавлении две длины сравниваются между собой и если новый элемент выходит за границы

максимальной длины (len_cur > len_max) происходит расширение. В остальных случаях мы просто добавляем элемент после последнего и увеличиваем текущую длину на один.

```
#include <vector>
```

Методы: insert(вставка), emplace(замена), erase(удаление)

Строки – класс string предназначен для работы со строками типа char*, которые представляют собой строку с завершающим нулем. Строки можно рассматривать как `vector<char>`, однако есть небольшая разница в функциях для манипулирования строками и в политике работы с памятью.

```
#include <string>
using namespace std;
string s1 = "Hello";
```

Методы: append (соединение), assign (присвоение), compar e(сравнение), find(поиск).

Контейнеры vector и string самостоятельно управляют своей памятью. Занимаемая ими память расширяется по мере добавления новых элементов, а при уничтожении vector или string деструктор автоматически уничтожает элементы контейнера и освобождает память, в которой они находятся.

Как правило, string используется в том случае, если элемент является символьным типом, а vector — во всех остальных случаях.

Задание

1. Замените С-строки на использование встроенного класса-контейнера string.
2. Создайте собственный класс-контейнер для хранения набора персональных сущностей. По аналогии с приведенным примером (vector) он должен включать в себя динамический массив данных, размер массива и базовые функции по взаимодействию с ним: вывод всех элементов, получение, изменение, добавление, удаление элементов

по индексу, сортировку и фильтрацию по одному главному признаку. Последние две функции были разработаны в прошлой лабораторной работе.

3. В Вашем проекте замените имеющиеся массивы данных, подходящие по типу данных созданного вектора, на вектор.

Контрольные вопросы

1. Что такое контейнер?
2. Приведите примеры из реальной жизни.
3. Для чего используется контейнер в программировании?
4. Как реализуется класс-контейнер?
5. Какие существуют типы контейнерных классов?
6. Что представляет из себя элемент контейнера?
7. Что такое делегирующий конструктор?
8. Для чего он используется?
9. Что такое итераторы?
10. Как реализовать итератор?
11. Как работает диапазонный цикл for?
12. Какие операции можно с его помощью выполнять?
13. К каким данным можно применить этот вид цикла?

Список литературы

1. Программирование на языке высокого уровня C/C++ [Электронный ресурс]: конспект лекций / – Электрон. текстовые данные. – М.: Московский государственный строительный университет, Ай Пи Эр Медиа, ЭБС АСВ, 2016. – 140 с. – Режим доступа: http://www.iprbookshop.ru/48037.

Приложение 1

Пример реализации класса-контейнера для строк:

```
class MyString {
public:
	class StrIterator {
		// делаем внешний класс дружественным к подклассу
		// чтобы он имел доступ к нашим закрытым свойствам
		friend class MyString;
	public:
		// пишем конструктор копирования
		// который будет производить инициализацию членов класса
		// тело конструктора будет пустым
		StrIterator(const StrIterator& it) : m_symbol(it.m_symbol) {}

		// перегружаем оператор сравнения
		bool operator==(const StrIterator& it) const {
			return m_symbol == it.m_symbol;
		}

		// перегружаем оператор сравнения на не
		bool operator!=(const StrIterator& it) const {
			return m_symbol != it.m_symbol;
		}

		// перегружаем оператор икремента
		StrIterator& operator++() {
			++m_symbol;
			return *this;
		}

		// перегружаем оператор разыменования указателя
		char& operator*() const {
			return *m_symbol;
		}

	private:
		char* m_symbol{};
		// создаём закрытый конструктор инициализации членов класса
		StrIterator(char* p) : m_symbol(p) {}
	};
```

```
typedef StrIterator iterator;
typedef StrIterator const_iterator;

MyString() {
	/*
		Конструктор без параметров,
		вызывает функцию инициализации
		строки
	*/
	initStr();
}

MyString(const char* str) : MyString() {
	/*
		Делегирующий конструктор, с одним параметром
		В теле вызывает метод добавления строки в объект
	*/
	addStr(str);
}

MyString(const char* str, int count) : MyString() {
	/*
		Делегирующий конструктор с двумя параметрами
		В теле добавляет одну и туже строчку
		Заданное кол - во раз в объект
	*/
	for (int i{}; i < count; ++i) {
		addStr(str);
	}
}

MyString(const char* str_1, const char* str_2) : MyString(str_1) {
	/*
		Делегирующий конструктор с двумя параметрами
		Создаёт новую строку из двух предидущих
	*/
	addStr(str_2);
}

MyString(const char* str, char ch) : MyString(str) {
```

```
		/*
			Делегирующий конструктор с двумя параметрами
			Создаёт новую строку из одной строки и символа
		*/
		addSymbol(ch);
	}

	MyString(const MyString& str) : MyString(str.m_chars) {
		/*
			Пустой делегирующий конструктор для создания новой строки
			из строки
			Также, такой конструктор называется - конструктором копирования
		*/
	}

	MyString(const MyString& str_1, const MyString &str_2) :
MyString(str_1.m_chars, str_2.m_chars) {
		/*
			Пустой делегирующий конструктор для создания новой строки
			из двух строк
		*/
	}

	~MyString() {
		/*
			Деструктор объекта
		*/
		delete[] m_chars;
	}

	void addStr(const char* str) {
		/*
			Метод, добавляющий строку в объект
		*/
		for (const char* it = str; *it; ++it) {
			addSymbol(*it);
		}
	}

	void addSymbol(const char symbol) {
		/*
```

```
		Метод, добавляющий символ в строку
	*/

	char* m_tmp = m_chars; // сохранение старой строки
	m_chars = new char[++m_len]{}; // создание новой строки

	// копирование старой строки в новую
	for (size_t i{}; i < (m_len - 2); ++i) {
		m_chars[i] = m_tmp[i];
	}

	// запись нового символа в строку
	m_chars[m_len - 2] = symbol;

	// добавление нуль - терминатора
	m_chars[m_len - 1] = '\0';

	// запись указателя на элемент после последнего
	m_end = m_chars + m_len;

	// удаление старой строки
	delete[] m_tmp;
}

char& operator[](size_t index) {
	/*
		Перегруженный оператор индекса
		Возвращает "зацикленный" элемент строки
	*/
	return m_chars[index % m_len];
}

// возврат итератора на первый элемент
iterator begin() {
	return iterator(m_chars);
}

// возврат итератора на за-последний элемент
iterator end() {
	return iterator(m_end);
}
```

```
// возврат константного итератора на первый элемент
const_iterator begin() const {
	return const_iterator(m_chars);
}

// возврат константного итератора на за-последний элемент
const_iterator end() const {
	return const_iterator(m_end);
}

// возврат длины строки
size_t getLen() {
	return m_len;
}

// очистка строки
void erase() {
	delete[] m_chars;
	initStr();
}

// дублирование строки
MyString duplicate(int count) {
	return MyString(m_chars, count);
}

// перегруженный оператор умножения для дублирования строки
MyString operator*(int count) {
	return duplicate(count);
}

// перегруженный оператор сложение строки и С - строки
MyString operator+(const char* str) {
	return MyString(m_chars, str);
}

// перегруженный оператор сложение строки и строки
MyString operator+(const MyString &str) {
	return MyString(*this, str);
}
```

```
// перегруженный оператор сложение строки и символа
MyString operator+(const char ch) {
	return MyString(m_chars, ch);
}

// перегруженный оператор добавления к строке С - строки
MyString& operator+=(const char* str) {
	addStr(str);
	return *this;
}

// перегруженный оператор добавления к строке строки
MyString& operator+=(const MyString &str) {
	addStr(str.m_chars);
	return *this;
}

// перегруженный оператор для дублирования строки в строке
MyString& operator*=(int count) {

	MyString tmp(*this);
	for (int i{}; i < count - 1; ++i) {
		addStr(tmp.m_chars);
	}
	return *this;
}

// перегруженный оператор добавления к строке символа
MyString& operator+=(const char ch) {
	addSymbol(ch);
	return *this;
}

// перегруженный оператор присваивания к строке строки
MyString& operator=(const MyString& str) {

	if (&str == this) {
		return *this;
	}
	erase();
```

```
            addStr(str.m_chars);
            return *this;
        }

        // перегруженный оператор присваивания строке C - строки
        MyString& operator=(const char* str) {
            erase();
            addStr(str);
            return *this;
        }

        // перегруженный оператор присваивания строке символа
        MyString& operator=(const char ch) {
            erase();
            addSymbol(ch);
            return *this;
        }

        // перегруженный оператор вывода строки
        friend ostream& operator<<(ostream& out, const MyString& str);

        // перегруженный оператор ввода строки
        friend istream& operator>>(istream& in, MyString& str);

    private:
        size_t m_len{};
        char* m_chars{};
        char* m_end{};

        // скрытый метод инициализации строки
        void initStr() {
            m_len = 1;
            m_chars = new char[m_len]{ '\0' };
            m_end = m_chars + 1;
        }
};

ostream& operator<<(ostream& out, const MyString& str) {
    out << str.m_chars;
    return out;
}
```

```
istream& operator>>(istream& in, MyString& str) {
	char ch{};
	while (in.get(ch) && ch != '\n') {
		str.addSymbol(ch);
	}
	return in;
}
```