МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Новосибирский государственный технический университет»

НГТУ НЭТИ

Кафедра прикладной математики

Практическая работа №1

по дисциплине «Языки программирования и методы трансляции»

ФПМИ

Группа	ПМ-92
Вариант	7
Студенты	Кутузов Иван
	Иванов Владислав
Преподаватель	Еланцева И. Л.
Дата	03.03.2022

Новосибирск

Цель работы:

Получить представление о видах таблиц, используемых при трансляции программ. Изучить множество операций с таблицами и особенности реализации этих операций для таблиц, используемых на этапе лексического анализа. Реализовать классы таблиц, используемых сканером.

Задание:

Подмножество языка C++ включает:

- данные типа int, float, массивы из элементов указанных типов;
- инструкции описания переменных;
- операторы присваивания в любой последовательности;
- операции +, – , *, = =, != , <, > .

В соответствии с выбранным вариантом задания к лабораторным работам с использованием средств объектно-ориентированного программирования:

1. разработать структуру постоянных таблиц для хранения алфавита языка, зарезервированных слов, знаков операций, разделителей и пр.
2. разработать структуру переменных таблиц с вычисляемым входом для хранения идентификаторов и констант(вид хеш-функции и метод рехеширования задает разработчик)
3. реализовать для переменных таблиц алгоритмы поиска/добавления лексемы, поиска/добавления атрибутов лексемы.
4. Разработать программу для тестирования и демонстрации работы программ пп. 1-3

Классы:

Поскольку будет использоваться множество таблиц, то для таблиц будем использовать идентификаторы.

Опишем интерфейс для постоянных таблиц:

```
template<class TKey, class TValue>
class ReadOnlyTable {
public:
	virtual int id() = 0;

	virtual bool contains(TKey key) = 0;
	virtual TValue get(TKey key) = 0;
};
```

Т.к динамические таблицы по сути расширяют поведение постоянных таблиц, то используем наследование.

Интерфейс динамических таблиц:

```
template<class TKey, class TValue>
class Table : public ReadOnlyTable<TKey, TValue> {
public:
	virtual void add(TKey key, TValue value) = 0;
	virtual void remove(TKey key) = 0;
};
```

Достаточно описать один класс таблицы, который будет имплементировать интерфейс `Table`, им можно пользоваться как в качестве постоянной, так и в качестве временной таблицы (IoC).

В качестве реализации выберем хэш-таблицу, т.к предполагается уникальность ключей, а также сложность доступа к элементу таблицы - O(1).

Хэш-таблица:

```
template<class TKey, class TValue>
class HashTable : public Table<TKey, TValue> {
private:
	int _id;

	int _capacity = 10;
	list<pair<TKey, TValue>> *_pairs;

public:
	HashTable(int id);
	int id() override;

	TValue get(TKey key) override;
	bool contains(TKey key) override;

	void add(TKey key, TValue value) override;
	void remove(TKey key) override;

private:
	int getHashCode(TKey key);

	void resize();
};
```

Основная механика похожа на динамические массивы. Изначально выделяем некоторое количество памяти, при необходимости увеличиваем ее количество и перезаписываем данные, поскольку хэш ключей изменился.

Тесты:

Входные данные	Результат	Цель
`int id = 1;` `HashTable<string, Type> table(id);` `int expected = 1;` `int actual = table.id();` `return expected == actual;`	true	id
`table.add("key", Type::CONSTANT);` `table.add("another key", Type::IDENTIFER);` `bool expected = true;` `bool actual = table.contains("key");` `bool first = expected == actual;` `actual = table.contains("another key");` `bool second = expected == actual;` `expected = false;` `actual = table.contains("another_key");` `bool third = expected == actual;` `return first && second && third;`	true	add/contains
`try {` `  int id = 1;` `  HashTable<string, Type> table(id);` `  table.add("key", Type::CONSTANT);` `  Type expected = Type::CONSTANT;` `  Type actual = table.get("key");` `  return expected == actual;` `}` `catch (KeyDoesNotExistException e) {` `  return false;` `}`	true	get
`try {` `  int id = 1;` `  HashTable<string, Type> table(id);` `  table.add("key", Type::CONSTANT);` `  Type actual = table.get("another key");` `  return false;` `}` `catch (KeyDoesNotExistException e) {` `  return true;` `}`	true	get (несуществующий ключ)

`` try { int id = 1; HashTable<string, Type> table(id); table.add("key", Type::CONSTANT); table.add("key", Type::KEYWORD); return false; } catch (KeyAlreadyExistException e) { return true; } ```	true	add (существующий ключ)
`` try { int id = 1; HashTable<string, Type> table(1); table.add("key1", Type::CONSTANT); table.add("key2", Type::CONSTANT); table.add("key3", Type::CONSTANT); table.add("key4", Type::CONSTANT); table.add("key5", Type::CONSTANT); table.add("key6", Type::CONSTANT); table.add("key7", Type::CONSTANT); table.add("key8", Type::CONSTANT); table.add("key9", Type::CONSTANT); table.add("key10", Type::CONSTANT); table.add("key11", Type::CONSTANT); table.add("key12", Type::CONSTANT); bool actual = true; bool expected = table.contains("key1") && table.contains("key2") && table.contains("key3") && table.contains("key4") && table.contains("key5") && table.contains("key6") && table.contains("key7") && table.contains("key8") && table.contains("key9") && table.contains("key10") && table.contains("key11") && table.contains("key12"); return actual == expected; } catch (exception e) { return false; } ```	true	расширяемость таблицы (изначально можно хранить только 10 элементов)
`` int id = 1; HashTable<string, Type> table(id); table.add("key", Type::CONSTANT); table.remove("key"); bool expected = false; bool actual = table.contains("key"); return actual == expected; ```	true	remove
`` try { int id = 1; HashTable<string, Type> table(id); table.remove("key"); return false; } catch (KeyDoesNotExistException e) { return true; } ```	true	remove (несуществующий ключ)

Текст программы?:

```
class KeyDoesNotExistException : public exception {
public:
	const char* what() const throw() {
		return "there is no such key";
	}
};

class KeyAlreadyExistException : public exception {
public:
	const char* what() const throw() {
		return "the key already exist";
	}
};

template<class TKey, class TValue>
class ReadOnlyTable {
public:
	virtual int id() = 0;

	virtual bool contains(TKey key) = 0;
	virtual TValue get(TKey key) = 0;
};

template<class TKey, class TValue>
class Table : public ReadOnlyTable<TKey, TValue> {
public:
	virtual void add(TKey key, TValue value) = 0;
	virtual void remove(TKey key) = 0;
};

template<class TKey, class TValue>
class HashTable : public Table<TKey, TValue> {
private:
	int _id;

	int _capacity = 10;
	list<pair<TKey, TValue>> *_pairs;

public:
	HashTable(int id);
	int id() override;

	TValue get(TKey key) override;
	bool contains(TKey key) override;

	void add(TKey key, TValue value) override;
	void remove(TKey key) override;

private:
	int getHashCode(TKey key);

	void resize();
};

template<class TKey, class TValue>
HashTable<TKey, TValue>::HashTable(int id) {
	_pairs = new list<pair<TKey, TValue>>[_capacity];
	_id = id;
}
```

```
template<class TKey, class TValue>
int HashTable<TKey, TValue>::id() {
	return _id;
}

template<class TKey, class TValue>
TValue HashTable<TKey, TValue>::get(TKey key) {
	int hashCode = getHashCode(key);
	auto& cell = _pairs[hashCode];

	for (auto pair = begin(cell); pair != end(cell); pair++) {
		if (pair->first == key) {
			return pair->second;
		}
	}

	throw KeyDoesNotExistException();
}

template<class TKey, class TValue>
void HashTable<TKey, TValue>::add(TKey key, TValue value) {
	if (_pairs->size() == _capacity) {
		resize();
	}

	int hashCode = getHashCode(key);

	auto& cell = _pairs[hashCode];

	for (auto pair = begin(cell); pair != end(cell); pair++) {
		if (pair->first == key) {
			throw KeyAlreadyExistException();
		}
	}

	cell.emplace_back(key, value);
}

template<class TKey, class TValue>
void HashTable<TKey, TValue>::remove(TKey key) {
	int hashCode = getHashCode(key);

	auto& cell = _pairs[hashCode];

	if (cell.size() == 0) {
		throw KeyDoesNotExistException();
	}

	for (auto pair = begin(cell); pair != end(cell); pair++) {
		if (pair->first == key) {
			pair = cell.erase(pair);
			break;
		}
	}
}
```

```
template<class TKey, class TValue>
bool HashTable<TKey, TValue>::contains(TKey key) {
	int hashCode = getHashCode(key);

	auto& cell = _pairs[hashCode];

	for (auto pair = begin(cell); pair != end(cell); pair++) {
		if (pair->first == key) {
			return true;
		}
	}

	return false;
}

template<class TKey, class TValue>
int HashTable<TKey, TValue>::getHashCode(TKey key) {
	return hash<TKey>()(key) % _capacity;
}

template<class TKey, class TValue>
void HashTable<TKey, TValue>::resize() {
	_capacity *= 2;

	list<pair<TKey, TValue>> *newPairs = new list<pair<TKey,
TValue>>[_capacity];
	swap(_pairs, newPairs);

	for (int i = 0; i < _capacity / 2; i++) {
		auto& cell = newPairs[i];

		if (cell.size() != 0) {
			for (auto pair = begin(cell); pair != end(cell); pair++) {
				add(pair->first, pair->second);
			}
		}
	}
}
```