Лабораторная работа № 5

Объектно-ориентированное программирование

Цель работы

1. Изучить устройство связанного списка как структуры данных, а также идеи и особенности алгоритмов операций над ним.
2. Применить языковые средства C++ поддержки объектно-ориентированного программирования (ООП) для реализации типа данных-связанного списка.
3. Овладеть концепцией итераторов и её применением в сочетании со стандартной библиотекой языка C++.
4. Освоить создание проектов и отладку программ в среде Code::Blocks.

Задание на лабораторную работу

1. Определить псевдоним для типа данных, значения которого хранятся в узлах списка:

```
typedef double Data;
```

Пояснение. При необходимости изменить хранимый тип данных, следует заменить **double** на нужный тип и перекомпилировать программу.

2. Определить класс `LinkedList` связанного списка со следующим интерфейсом.

```
class LinkedList
{
public:
```

2.1. Обеспечить корректное создание пустого списка (по умолчанию) и возможность инициализации элементов списка при создании в фигурных скобках (`{1,2,3}`):

```
LinkedList();
LinkedList(std::initializer_list<Data> values);
```

2.2. Обеспечить корректное копирование, перемещение и присваивание списков:

```
LinkedList(const LinkedList& source);
LinkedList(LinkedList&& source);
LinkedList& operator=(LinkedList rhs);
```

Указание. При выполнении может быть полезно реализовать внешнюю функцию (не метод) обмена состояний связанных списков:

```
void swap(LinkedList& a, LinkedList& b);
```

Для доступа к полям объектов её следует объявить дружественной в классе `LinkedList`.

2.3. Предусмотреть очистку связанного списка при уничтожении объекта:

```
virtual ~LinkedList();
```

2.4. Предоставить методы для определения текущего количества элементов —

```
size_t get_size() const;
```

и для доступа к элементам по индексу:

```
Data& operator[](size_t index) const;
```

2.5. Реализовать методы для добавления и удаления элементов из начала (front) и с конца (back) связанного списка, аналогичные методам `std::vector`:

```
void push_front(const Data& value);
void push_back(const Data& value);
void pop_front();
void pop_back();
```

2.6. Реализовать метод очистки связанного списка:

```
void clear();
};
```

3. Реализовать операторы ввода и вывода элементов объекта `LinkedList` в поток STL:

```
ostream& operator<<(ostream& output, const LinkedList& xs);
istream& operator>>(istream& input, LinkedList& xs);
```

4. Обеспечить совместимость класса `LinkedList` с STL, добавив поддержку итераторов.

4.1. Определить класс `Iterator` со следующим интерфейсом.

```
class Iterator
{
public:
```

4.1.1. Обеспечить сравнение итераторов —

```
bool operator!=(const Iterator& other) const;
bool operator==(const Iterator& other) const;
```

и перемещение итератора к следующему элементу связанного списка:

```
Iterator& operator++();
```

4.1.2. Реализовать операторы для доступа к значению элемента связанного списка, соответствующего текущему положению итератора —

```
Data& operator* () const;
```

и для доступа к адресу этого значения:

```
Data* operator->() const;
```

Пояснение. Это позволяет использовать итератор подобно указателю на значение.

```
};
```

Указание. Удобно и типично выполнить класс итератора как вложенный в `LinkedList`.

4.2. Добавить к класс `LinkedList` методы для получения итераторов, находящихся на начальном элементе и за конечным:

```
Iterator begin() const;
Iterator end() const;
```

4.3. Реализовать методы класса `LinkedList`, использующие итераторы.

4.3.1. Метод для поиска в списке возвращает итератор, находящийся на найденном элементе, или `end()`, если искомый элемент не найден:

```
Iterator find(const Data& what) const;
```

4.3.2. Метод для удаления элемента из списка принимает итератор, находящийся на удаляемом элементе, и возвращает итератор на элемент, расположенный следом за удаленным (как у `std::vector`):

```
Iterator erase(const Iterator& which);
```

4.4. Добавить псевдонимы типов, необходимые алгоритмам STL.

4.4.1. В классе `LinkedList`:

```
typedef Data     value_type;
typedef Iterator iterator;
```

4.4.2. В классе `Iterator`:

```
typedef Data                  value_type;
typedef Data&                 reference;
typedef Data*                 pointer;
typedef size_t                difference_type;
typedef forward_iterator_tag iterator_category;
```

Пояснение. Алгоритмам STL неизвестно, какой тип (класс) служит итератором для связанного списка, какой — данными в узлах и т. п. Стандартом C++ ведено правило, что для любого контейнера `C` тип-итератор называется `C::iterator`, тип элемента — `C::value_type` и т. д. Особенным является `C::iterator_category`: в зависимости от того, для какого типа это псевдоним, алгоритмы STL «делают вывод» о том, какие операции (`*`, `->`, `!=`, `==`) доступны (о категории итераторов).

Указания к выполнению лабораторной работы

Порядок решения задач

Не обязательно выполнять пункты задания по порядку и сразу целиком. Например, для реализации конструктора со списком инициализации или конструктора копирования (п. п. 2.1 и 2.2) целесообразно использовать метод `push_back()` из п. 2.5.

В задании описаны только интерфейсы классов. Детали реализации: поля, структура узла списка, служебные закрытые методы — не упомянуты, и их следует добавлять по мере надобности. Например, целесообразен закрытый метод

```
void erase(const Node* node);  // Node — узел списка.
```

который мог бы удалять любой узел, чтобы на его основе легко реализовать метод `pop_front()`, `pop_back()`, а затем `erase(const Iterator&)`. Однако, необходимо соблюдать инкапсуляцию: интерфейс класса не должен позволять внешнему коду нарушать согласованность внутреннего состояния объекта, а детали реализации — например, тип узла списка — должны быть скрыты от пользователя (при помощи **`private`**).

Автоматическое тестирование

Учебная программа, помимо класса связанного списка и итератора, должна содержать простые автоматические тесты для *всей* реализованной функциональности. Состав и содержание тестов остается на усмотрение автора кода. Чем более подробно будет проверяться работа класса, тем вероятнее обнаружить ошибки и не допустить новых. Рекомендуется выделять тесты в небольшие функции, проверяющие один метод на отдельном новом объекте-списке при помощи `assert()`.

Не сто́ит требовать ввода данных в тестах — удобнее запустить программу и сразу провести все тесты. Проверить работу операторов ввода и вывода можно при помощи класса `stringstream`. Совместимость с STL целесообразно производить, применяя к `LinkedList` алгоритмы STL: `find()`, `remove()` и т. п.

Настройка проекта

Из лекционного курса известно, что объявление и определение методов класса целесообразно выполнять в различных файлах, в заголовочном и в файле реализации. Сборка программы, состоящей из нескольких файлов, осуществляется в одном проекте.

Создать проект в Code::Blocks можно из пункта меню *File* → *New* → *Project…*; шаблон проекта — *Console Application*. После создания проекта его структура (файлы) будут отображаться на панели слева. Если панель не видна, вызвать её можно пунктом меню *View* → *Manager* или нажатием *Shift + F2*.

При добавлении новых файлов необходимо включать их в конфигурации для отладочной (Debug) и выпускной (Release) сборки проекта, отмечая флажок *«Add file to active project»* и все флажки в списке под ним (кнопка *All*). Шаблон заголовочного файла в диалоге создания называется *«C/C++ header file»*.

Отладка программ

Отладка программ, знакомая из предшествующих курсов, позволяет остановить работу программы в выбранной точке (точке останова, breakpoint), чтобы просмотреть значения переменных и проследить ход выполнения.

Внимание! Отладчик GDB, используемый в лаборатории, требует, чтобы путь к коду и исполняемому файлу программы не содержал пробелов и русских букв.

Code::Blocks предоставляет возможность отладки только в проектах. Создать точку останова можно, щелкнув мышью слева от строки кода (справа от её номера) или нажав *F5,* — появится красный кружок. Переход к следующей строке при отладке выполняется нажатием *F7*, переход в функцию при её вызове — *Shift + F7*. Прочие команды отладки см. в меню *Debug*.

Начало или продолжение отладки (до следующей точки останова) производится нажатием клавиши *F8*. В силу чужеродности GDB для Windows, иногда приходится запускать отладку как *Shift + F7* и клавишей *F7* перемещаться к нужной точке.

Во время отладки при наведении указателя мыши на переменную появляется значение этой переменной. Увеличить кегль всплывающего текста можно в диалоге *Settings → Debugger…*, поле *Value Tooltip Font*.

Использование системы контроля версий

Поощряется, но не навязывается, использование системы контроля версий (СКВ). Побуждающие меры: при необходимости проконсультироваться через интернет решения можно будет демонстрировать только в хранилище СКВ (например, на GitHub), — код, высланный почтой, в файловом хранилище, зачитанный вслух и т. п., рассмотрен не будет.

Контрольные вопросы и задания

1. Опишите устройство: а) односвязного или б) двусвязного списка, его преимущества и недостатки по сравнению с другими известными вам структурами данных.
2. Что такое алгоритмическая сложность и её асимптотическая оценка? Каковы оценки алгоритмической сложности операций над связанным списком, реализованных в ЛР?
3. Какой алгоритм поиска стоит выбрать для связанного списка при требовании наибольшего быстродействия и почему?
4. Какой алгоритм сортировки стоит выбрать для связанного списка при требовании наибольшего быстродействия и почему?

5. Предложите и реализуйте алгоритм разворота: а) односвязного и б) двусвязного списка, если список представлен: 1) указателем на начальный элемент или 2) указателями на начальный и конечный элементы.
6. В чем состоит идея и каковы основные принципы ООП? Что такое класс и что такое объект с точки зрения языка программирования?
7. Что такое инкапсуляция и как она обеспечивается для классов в C++? В чем состоит отличие класса от структуры в C++? Что такое друзья класса?
8. Что такое метод объекта (функция-член)? Зачем нужны и когда вызываются (и должны явно вызываться) специальные методы: конструктор и деструктор?
9. Какие вам известны виды отношений владения между объектами? Опишите их. В каких отношениях находятся: а) список и узел списка (начальный, конечный, иной); б) узел списка и полезные данные; в) соседние узлы списка?
10. Чем отличается копирование объекта от перемещения? Зачем нужны и когда вызываются конструкторы копирования и перемещения объекта (на примере списка)?
11. Как классифицируются выражения в C++? Чем отличается `ссылка&` от `ссылки&&`?
12. Объясните концепцию итератора и её применение с сочетании со стандартной библиотекой языка C++, а также реализацию итераторов в ходе ЛР.
13. Что такое псевдоним типа и чем он бывает полезен? Что такое и зачем нужны неизменяющие методы? Обоснуйте их применение в ходе ЛР.
14. Что такое (в C++) лямбда-функция? Объясните понятие захвата контекста лямбда-функцией, захват по значению и по ссылке.