Семинар 6. Двусвязный список

С. А. Шершаков

15 октября 2018 г.

Ред. 1.3 от 15.10.2018 г.

В документе представлен 6-й цикл заданий для самостоятельного выполнения студентами курса «Алгоритмы и структуры данных» и методические рекомендации по их выполнению. Задание посвящено изучению связнных списков с прямыми и инвертированными связями. Реализация структуры данных «двусвязный список» в виде шаблона кода на языке C++ позволяет в большей степени изучить концепции обобщенного программирования. Реализация задачи включает взаимодействие с инструментом автодокументирования кода Doxygen и unit-тестированием на основе библиотеки gtest.

Рис. 1: Рисунок из глубин интернетов, призванный навевать правильные ассоциации

1 *Требования, цели и ожидаемые результаты*

1.1 *Требования*

Студенты должны владеть следующими знаниями, умениями и навыками для выполнения задания.

- Разработка модульных приложений на языке C++.
- Концепция *шаблонов*, *обобщенного программирования*, параметризованные функции и классы C++.
- Основы стандартной библиотеки C/C++.
- Тип данных «связный список», его основные операции.

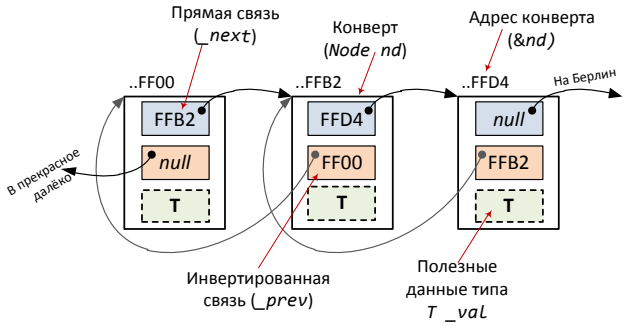

Рис. 2: Двусвязный список и его основные компоненты

1.2 *Цели и задачи*

Основная цель работы — практическое изучение реализации двусвязного списка и основных операций над ним.

Связанные задачи: unit-тестирование с использованием библиотеки gtest[1].

Задачи для выполнения в рамках самостоятельной работы:

- Изучить способы имплементации двусвязного списка (рис. 2).

[1] https://github.com/google/googletest

- Изучить интерфейс шаблонного класса `BidiLinkedList<T>` (заголовочный файл `BidiLinkedList.h`) по предложенному коду и/или предложенной документации.
- Изучить примеры псевдо-реализации[2] методов класса `BidiLinkedList<T>` (заголовочный файл `BidiLinkedList.hpp`).
- Реализовать недостающие методы класса `BidiLinkedList<T>`, ориентируясь на спецификацию метода и настоящее руководство. Реализацию необходимо разместить в файле `BidiLinkedList.hpp`[3].
- [*Опционально*] протестировать полученную реализацию на множестве предложенных unit-тестов.

[2] Здесь в действительности корректнее говорить о «продолжении интерфейса класса», так как у шаблонного класса реализация, как таковая, отсутствует до тех пор, пока шаблон не будет конкретизирован. Однако с целью концептуального разделения описания методов и заготовок их тел-реализаций, мы производим разделение шаблона на условно-декларационную часть (файл с расширением `.h`) и условно-имплементационную (одноименный файл с расширением `.hpp`, включаемый из файла `.h`).

[3] Следует обратить внимание, что в файле могут присутствовать заготовки одних методов и отсутствовать заготовки других.

1.3 *Ожидаемые результаты*

Студенты, успешно[4] выполнившие задание, получат навыки разработки шаблонных классов с вынесением методов за пределы описания классов[5], тестирования их на предложенном наборе unit-тестов.

[4] И самостоятельно.

[5] А также связанную с этим специфику, например, использование ключевого слова `typename`.

2 *Предпосылки*

Двусвязный список (*Dual Linked List*) — структура данных, представляющая элементы полезных данных (payload) в виде последовательной цепочки *узлов* (конвертов, *nodes*), содержащих эти элементы. Каждый узел содержит прямую связь[6] (*link*), указывающую на *следующий* узел в последовательности, и еще одну связь, указывающую на предыдущий узел в последовательности. Наличием этой последней (*обратной*) связи двусвязный список отличается от простого (одно)связного списка.

Также, как и односвязный, двусвязный список обладает *стабильностью по перемещению* элементов, то есть при изменении структуры списка адреса его узлов, хранящих соответствующие элементы, *не меняются*, что позволяет сохранять их действительными отдельно от списка[7].

Основным очевидным преимуществом двусвязного списка перед односвязным является возможность осуществлять просмотр (traverse) списка не только в прямом направлении относительного текущего элемента, но и в обратном. *Побочным (и крайне положительным) эффектом* является возможность осуществления операций с подцепочками списка путем указания непосредственно элемента(ов) (адреса(ов) *узла(ов)*), участвующего(их) в операциях, вместо указания «предыдущего элемента», как то требовалось в односвязном списке.

Ценой за удобство является бóльший размер памяти, требуемой для хранения одного узла. Прибавка в размере равна размеру одного указателя, то есть для 32-разрядной платформы прибавка составит 4 байта. Так, общий размер узла при хранении целочисленных 32-разрядных данных составит $4 + 4 + 4 = 12$ байт[8], то есть появление инвертированной связи

[6] Мы намеренно избегаем использования привычного термина «ссылка», чтобы избежать путаницы с ссылками языка C++. Реализация такой связи возможна с использованием *указателей* и невозможна с использованием *C++-ссылок*, так как последние всегда должны быть инициализированы, что в свою очередь невозможно для первого и последнего элемента последовательности элементов списка.

[7] Такая особенность является важным требованием, например, для хранения дополнительных атрибутов элементов списка отдельно от самих элементов, — с помощью *ассоциативных контейнеров*, — что позволяет достигать компактности представления в памяти и расширяемости карты атрибутов без изменения структуры элементов.

[8] Без учета оптимизационного выравнивания структур в памяти, если таковое выполняется оптимизирующим компилятором.

увеличивает размер на 50 % по сравнению с первоначальным размером, но занимает всего 33 % от конечного.

3 Описание задания

В рамках самостоятельной работы студентам предлагается разработать имплементацию двусвязного списка, представленного шаблонным классом `BidiLinkedList<T>`. Класс описывается в модуле, состоящем из пары файлов `BidiLinkedList.h`/`.hpp`.

Далее представлена краткая информация по основным модулям/типам программного кода.

3.1 Файл `BidiLinkedList.h`

Файл содержит единственный класс верхнего уровня `BidiLinkedList<T>`, представляющий параметризуемый двусвязный список элементов типа `T`. Требования, предъявляемые к семантике типа `T`:

- должен иметь конструктор по умолчанию (т.е. быть default constructable) — для создания узла без инициализации данными;
- должен иметь поддержку операции копирования (copyable).

Внутри класса `BidiLinkedList<T>` содержится описание класса `Node` узла соответствующего списка. Это позволяет избежать загромождения пространства имен верхнего уровня[9] и избежать необходимости дополнительно параметризировать класс `Node`, так как он имеет доступ к шаблонному паметру `T` объемлющего его класса.

[9] И тем самым снизить вероятность конфликта имен, для предотвращения которого приходится прибегать к более громоздкому именованию.

Файл содержит стандартную конструкцию `#ifndef .. #define`, предотвращающую конфликты имен при повторном включении заголовочного файла в модуль трансляции.

3.2 Файл `BidiLinkedList.hpp`

Файл содержит внешние шаблоны методов, описанных в классах `BidiLinkedList<T>` и `BidiLinkedList<T>::Node`. Данный файл включается непосредственно из файла `BidiLinkedList.h` и не предусматривает возможность быть включенным извне самостоятельно[10].

[10] По этой причине он не содержит конструкцию `#ifndef .. #define`.

3.3 Класс `BidiLinkedList<T>::Node`

Класс представляет узел связного списка. Структура узла соответствует изображенному на рис. 2 и содержит:

- указатель `_next` на следующий узел в цепочке или `nullptr`, если узел последний в цепочке;
- указатель `_prev` на предыдущий узел в цепочке или `nullptr`, если узел первый в цепочке;
- поле `_val` данных полезной нагрузки (payload).

Необходимо обратить особое внимание, что описание вложенных типов в шаблонных классах накладывает дополнительные требование на появление имени этого типа *вне* объемлющего класса. Чтобы сообщить компилятору о том, что при обращении `BidiLinkedList<T>::Node` мы имеем в виду тип, необходимо явно использовать ключевое слово `typename`; в противном случае компилятор сообщит об ошибке. Так, если необходимо определить объект `aNode` – указатель на элемент списка, необходимо использовать следующую запись:

```
1 template <typename T>
2 typename BidiLinkedList<T>::Node* aNode = nullptr;    //
```

NB: `typename`

3.4 Класс `BidiLinkedList<T>`

Содержит всего два поля:

- `_head` — указатель на первый элемент в списке, или `nullptr`, если список пуст;
- `_tail` — указатель на последний элемент в списке, или `nullptr`, если список пуст;
- `_size` — «кешированное» значение размера списка.

Вычисление размера требует некоторых дополнительных комментариев. Так, возможны разные подходы для вычисления размера[11] связного списка. Например, этот размер может вычисляться каждый раз при обращении к соответствующему методу получения размера, либо вычисляться каждый раз, когда по отношению к списку производится некоторая модифицирующая его операция. Мы предлагаем смешанный вариант, при котором размер вычисляется только при первом обращении к соответствующему методу и сохраняется для дальнейшего использования. Как только к списку применена любая модифицирующая операция, размер отмечается как «недействительный», и при следующем обращении к методу получения размера он будет пересчитан заново[12].

[11] Операция имеет стоимость $O(n)$ от размера списка.

[12] Таким образом реализуется паттерн «кеширование».

Метод, возвращающий размер списка `getSize()`, имеет возвращаемое значение типа `size_t`. Это стандартный тип, использующий наиболее подходящий с точки зрения библиотеки для текущей платформы интегральный беззнаковый тип данных, позволяющий представлять «достаточно большой» размер. Чтобы не заводить дополнительную переменную, указывающую на признак «недействительного размера», можно выделить специальное значение типа `size_t`, представляющее такой размер. Мы используем для этого выражение `(size_t)-1`, которое для беззнакового целого превращается в максимальное положительное значение, которое может быть представлено этим типом. Чтобы не использовать в коде т.н. «magic number», мы вводим специальную константу `NO_SIZE`, представляющую это значение.

Важно: при реалиации любой модифицирующей операции необходимо (не забыть) вызывать метод `invalidateSize()`, помечающий текущее значение размера как недействительное.

3.4.1 Конструкторы и деструктор

Класс содержит конструктор по умолчанию и деструктор, освобождающий память из-под элементов списка. Для облегчения этой операции и для предоставления возможности пользователю освобождать память, вводится метод `clear()`.

3.4.2 Немодифицирующие методы

Методы, предоставляющие информацию о списке:

- `getHeadNode()` — возвращает головной элемент списка;
- `getLastNode()` — возвращает конечный элемент списка путем прохода по всей цепочки узлов, начиная с головного;
- `getSize()` — возвращает размер списка;
- `findFirst()` — ищет и возвращает первый узел в последовательности, значение элемента которого совпадает с искомым;
- `findAll()` — ищет и возвращается все узлы в последовательности, значения элементов которого совпадают с искомым.

3.4.3 Модифицирующие методы

Методы, изменяющие структуру списка, включают методы по добавлению *элементов* и добавлению и вырезанию *узлов*. Следует обратить внимание, что большинство операций выполняется посредством указания адресов узлов, что позволяет представлять диапазоны элементов (цепочки) в памяти естественным и компактным образом.

- `appendEl()` — добавляет элемент в конец списка; под элемент создается новый узел, адрес которого возвращается по значению;
- `insertNodeAfter()` — вставляет один новый узел после заданного узла;
- `insertNodesAfter)` — вставляет цепочку новых узлов, представленную первым и последним узлом, после заданного узла;
- `insertNodeBefore()` — вставляет новый узел перед заданным узлом[13];
- `insertNodesBefore()` — вставляет цепочку новых узлов, представленную первым и последним узлом, перед заданным узлом[13];
- `cutNode()` — вырезает переданный узел и делает его свободным[14];
- `cutNodes()` — вырезает цепочку, определяемую начальным и конечным узлами, и делает ее свободной;
- `cutFirst()` — вырезает первый узел, значение которого совпадает с указанным, и возвращает указатель на него (или `nullptr`, если таких узлов нет);

[13] Задача «со звездочкой». См. раздел. 3.5

[14] *Свободным* будем называть узел или цепочку узлов, не относящуюся посредством косвенных связей ни к одному списку. Такие узлы могут свободно быть вставлены в любой список. Ответственность за освобождение памяти из-под свободных узлов лежит на пользователе.

- `cutAll()` — вырезает все узлы со значениями, совпадающими с переданным; возвращает С-массив указателей на такие узлы и размер массива (по ссылке), либо `nullptr`, если нет ни одного такого элемента[13].

3.5 Задача «со звездочкой»

Некоторые методы, описанные в классе `BidiLinkedList<T>`, предлагается студенту реализовать «по желанию»[15]. К ним относятся методы `insertNodeBefore()`, `insertNodesBefore()` и `cutAll()`.

Описание опциональных методов, их заготовки и unit-тесты для них обрамлены директивами препроцессорного блока `#ifdef ... #endif`. По умолчанию эти методы будут являться неактивными. Чтобы активизировать эти методы в своем коде, необходимо определить[16] директиву препроцессора `IWANNAGET10POINTS` **до** первого включения заголовочного файла `BidiLinkedList.h`.

Сделав это, студент берет на себя ответственность за правильную реализацию указанных методов, так как противное может повлиять на оценку в меньшую сторону. Не сделав это, студент ограничивает себя в возможности получить наибольшую оценку за задание.

3.6 Итераторы

Помимо основного функционала связного списка необходимо научить его «быть» контейнером в смысле библиотеки STL: класс должен реализовывать ряд методов, возвращающих объекты-*итераторы*, позволяющие перечислять коллекцию стандартным образом.

Зарезервированные STL названия типов итераторов, которые будут проверяться в тестах: `iterator`, `const_iterator`, `reverse_iterator`, `const_reverse_iterator`. Каждый итератор должен относиться к категории *bidirectional_iterator*[17].

В классе списка должны быть реализованы следующие методы с зарезервированными названиями:

- `iterator begin();`
- `iterator end();`
- `const_iterator cbegin();`
- `const_iterator cend();`
- `reverse_iterator rbegin();`
- `reverse_iterator rend();`
- `const_reverse_iterator crbegin();`
- `const_reverse_iterator crend();`

Здесь `end()` соответствует несуществующему элементу в конце обхода итератора. `begin()` — первый элемент итератора, равный `end()` для пустой коллекции. `const`-версия не должна позволять менять данные объектов шаблонного типа `T`, которые хранят вершины класса `Node`.

[15] В общем случае, только реализация всех методов, включая и эти опциональные, позволяет студенту претендовать на максимальную оценку в 10 баллов, однако конкретная система требований и оценок устанавливается каждым преподавателем индивидуально.

[16] Независимый от IDE вариант объявить директиву препроцессора заключается в описании строки
`#define IWANNAGET10POINTS`
перед первым включением заголовочного файла, например в `main.cpp` и перед определением unit-тестов в файле `BidiLinkedList_Test.cpp`. Чтобы объявить директиву на уровне IDE: а) в Visual Studio необходимо открыть свойства проекта, *C/C++* → *Preprocessor* → *Preprocessor Definitions* и добавить туда директиву; б) в случае с CMake/CLion можно воспользоваться советом, предложенным по ссылке: http://stackoverflow.com/questions/9639449/cmake-how-to-pass-preprocessor-macros.

[17] http://www.cplusplus.com/reference/iterator/

4 *Структура проекта*

Студенты получают для работы проект со следующей структурой:

- `/docs` — каталог с документацией, включающий настоящее руководство;
- `/src` — исходные коды двусвязного списка и элементарного демометода (`demos.hpp`) для проверки корректности сборки;
- `/tests` — тесты в формате gtest;
- `/readme.md` — самая общая и самая полезная дополнительная информация по проекту.

5 *Указания по выполнению задания*

Выполнение задачи сводится к следующей последовательности[18].

- Ознакомиться с настоящим руководством-заданием.
- Получить исходные файлы для работы[19].
- Ознакомиться с предлагаемыми для работы исходными файлами и автодоксидокументацией к ним.
- Определиться с объемом выполняемой работы. Задекларировать макрос `IWANNAGET10POINTS` в файле `/src/demos.hpp`, если планируется выполнять задачу «со звездочкой».
- Реализовать недостающие методы, изменив файл `/src/BidiLinkedList.hpp`.
  Важный момент: необходимо внимательно просмотреть код модуля `/src/BidiLinkedList.hpp` на предмет комментариев, чтобы понять, какие методы в каком объеме надо реализовывать. Некоторые методы представлены только заголовком и пустым телом;в других — тело содержит некоторый код, призванный подсказать, в каком направлении надо двигаться[20]; некоторые методы реализованы полностью;наконец, есть методы, для которых нет даже заголовка[21].
- Протестировать (по желанию) реализацию на предложенном наборе тестов (файл `/tests/gtest/src/BidiLinkedList_Test.cpp`).
- При необходимости расширения множества тестов, необходимо поместить их в новый файл `/tests/gtest/src/BidiLinkedList_Student_Test.cpp`.
- Загрузить результаты работы в виде единого архива в ассоциированный проект системы LMS до крайнего срока, указанного в сводке проекта.

При выполнении задачи не подразумевается изменение каких-либо файлов, не перечисленных выше. В случае, если такая необходимость возникает, при сдаче такого файла необходимо сопроводить его комментариями в виде файла `student_notes.md`, который следует поместить в тот же каталог, что и «нормальнонесдаваемый» файл. В `student_notes.md` необходимо аргументировать потребность к изменению файла[22].

[18] Рекомендованная последовательность.

[19] С использованием системы LMS: `http://lms.hse.ru`

[20] Такие методы можно переписать как полностью, так и «дописать» нехватающий код.

[21] Необходимо найти соответствующий комментарий и на его место написать метод целиком.

[22] Примером такой необходимости является выявленная ошибка в исходном задании/файлах, не позволяющая решить задачу. В случае, если будет признано, что такой необходимости в действительности не было, это может повлечь за собой снижение оценки.

5.1 Сдаваемые файлы

- `/src/BidiLinkedList.hpp` — реализация основной задачи;
- `/src/demos.hpp` — любое необходимое для проверки работоспособности расширение кода;
- `/tests/gtest/src/BidiLinkedList_Student_Test.cpp` — дополнительные тесты, если разрабатывались;
- *при необходимости*, другие файлы с учетом замечаний, изложенных выше.