Глава 3

Абстрактные типы данных: определения, контейнеры и прикладные данные

3.1. Введение

В главе 2 мы рассмотрели пример абстрактного типа данных, который помог вам составить общее представление об этом вопросе. В данной главе мы продолжим обсуждение этой темы. Сначала будут приведены некоторые дополнительные определения и аналогии из других областей, а затем мы рассмотрим два вида абстрактных типов данных: контейнеры и прикладные данные.

3.2. Определения и аналогии

Чтобы лучше изучить абстрактные типы данных, полезно чередовать определения с примерами и аналогиями из других сфер деятельности. Зная основное определение абстрактного типа данных и рассмотрев в качестве примера тип `Board_T`, продолжим обсуждение этого вопроса.

3.2.1. Определения

Напомню вам определение абстрактного типа данных, приведенное в главе 2.

> *Абстрактный тип данных* — это тип, для которого описание значений и операций, выполняемых над ними, отделен от представления значений и реализации операций.

Для объекта `Board_T` мы определили его значение как набор значений `Position_T`. Затем мы описали простейшие операции, такие как `Q_OnBoard` и `GetStatus`. После этого была рассмотрена реализация с использованием двухмерного массива размером 8×8 элементов и упомянуты две альтернативные реализации. Некоторые реализации предпочтительнее других; в качестве критериев для сравнения реализаций можно выбрать такие характеристики, как простота разработки и эффективность выполнения. Несмотря на различие в деталях, *семантика* разных реализаций должна совпадать, т.е. все реализации должны соответствовать одной и той же *спецификации*.

Имея набор примитивов для объекта `Board_T`, мы можем разработать на их базе более сложные операции. Однако, чтобы реализовать эти операции, нам нужна готовая реализация `Board_T`. После рассмотрения `Board_T` целесообразно изучить новые определения.

Абстрагирование данных — это отделение описания набора значений некоторого типа данных от реализации этих значений. Абстрагирование данных имеет отношение к разработке приложения и позволяет говорить, например, о позиции на доске, о состоянии клетки и вводить другие подобные понятия еще до того, как был выбран язык для реализации типа.

Абстрагирование процедур — это отделение описания множества процедур над множеством значений типа данных от реализации процедур. Абстрагирование процедур осуществляется с помощью спецификаций, определяющих синтаксис и семантику. Использование предусловий и постусловий позволяет организовать взаимодействие программиста, занимающегося реализацией абстрактного типа данных `Board_T`, и разработчика приложения. Если предусловие для операции равно `TRUE`, то разработчик, реализующий абстрактный тип данных, обеспечивает, чтобы после выполнения операции постусловие также было равно `TRUE`. Если разработчик приложения использует операцию, то он должен добиться того, чтобы перед ее выполнением предусловие было равно `TRUE`. При возникновении проблем следует сравнить предусловие и постусловие. Если предусловие равно `TRUE`, виноват программист, реализующий абстрактный тип данных. В противном случае — это проблема разработчика приложения.

Сокрытие информации — это меры, препятствующие доступу пользователя к несущественным для него деталям реализации значений или набора операций конкретного типа данных. Особенности выполнения этих мер на практике диктуются средствами, предоставляемыми конкретным языком программирования. Так, например, мы можем использовать неявный тип данных, применяя для этого указатель `void` и помещая объявления и коды функций соответственно в файлы `.h` и `.cpp`.

3.2.2. Аналогии из других сфер деятельности

Пусть термин “разработка программного обеспечения” означает все виды работ, которые мы выполняем или должны выполнять при создании программы. При написании программ разработчики имеют дело с символами и синтаксическими правилами, подобно тому как разработчики аппаратуры оперируют с микросхемами, разъемами и соединениями.

Хороший инженер, разрабатывая устройство, примет меры, чтобы ограничить доступ пользователя к элементам конструкции необходимым минимумом. Так, например, проектируя мост, инженер позаботится о том, чтобы автомобили могли въехать на мост лишь там, где действительно предусмотрен въезд. Разрабатывая электрическую плиту, конструктор построит ее так, чтобы пользователь, изменяя температуру, вынужден был пользоваться регулятором и не мог делать это, перепаивая резисторы в схеме, даже если он того захочет.

Участвуя в рабочей группе (в которую могут входить специалисты разных компаний, распложенных, возможно, на разных континентах), инженеры разрабатывают компоненты конструкции, которые должны соединяться с другими компонентами, образуя сложный механизм. При этом необходимы специфика-

ции, регламентирующие действия, выполняемые компонентами, и интерфейсы, определяющие порядок соединения их с другими компонентами. Предположим, что корпорация Acme приобрела одну деталь у компании Ajax, а вторую — у компании Perfect Products. Что делать, если эти детали не соединяются? Возможно, инженеры Perfect Products посоветуют: "Откройте крышку, поменяйте местами два зубчатых колеса и увеличьте диаметр втулки, тогда, может быть, все заработает". Или руководство Axme скажет представителям Ajax и Perfect Products: "Пусть ваши инженеры соберутся и оперативно выработают комплекс мер, позволяющих соединить детали". Если такие подходы будут применяться при построении самолета, катастрофа неизбежна.

Используя абстрактные типы данных, мы работаем как инженеры, создающие сложные механизмы. В нашем случае аналогами таких механизмов являются броузеры, текстовые процессоры и другие программы, работающие в соответствии со спецификацией. Правда, некоторые программисты утверждают, что если программы будут работать так, как им положено, пользователи будут обделены, так как лишатся удовольствия самостоятельно найти ошибку и сообщить о ней[1].

3.3. Абстрактный тип данных `Sequence_T`

Примером структуры данных, состояние которой изменяется очень часто, является последовательность. Последовательность можно описать как множество элементов, на котором определен порядок следования. Формальное определение последовательности элементов выглядит следующим образом.

- В последовательности определены первый и последний элементы.
- После каждого элемента, за исключением последнего, находится последующий элемент.
- Перед каждым элементом, за исключением первого, находится предшествующий элемент.
- Если x является последующим элементом по отношению к y, то y является предшествующим элементом по отношению к x.
- Если последовательность не пуста, то один из ее элементов считается текущим элементом.

Например, в последовательности (3, 4, 7, 2) первый элемент — 3, а последний — 2. Элемент 4 является последующим по отношению к 3, 7 — последующим по отношению к 4, а 2 — последующим по отношению к 7. Элемент 7 является предшествующим по отношению к 2, 4 — предшествующим по отношению к 7, а 3 — предшествующим по отношению к 4. Элемент 4 выбран в качестве текущего. Понятие текущего элемента необходимо потому, что в каждый момент времени мы можем оперировать лишь с одним элементом, подобно тому, как в каждый момент времени мы читаем лишь одну страницу книги и обрабатываем лишь одну запись базы данных.

---

[1] *По словам профессора Тони Хоара, разработка программ — это единственная сфера деятельности, где специалистам платят за то, что они устраняют собственные ошибки.* — Прим. авт.

Тип элементов может быть произвольным, однако в составе конкретной последовательности могут присутствовать лишь элементы одного типа. Так, мы можем говорить о последовательности целых чисел, о последовательности символов, последовательности учетных записей или о последовательности товаров, выбранных в интерактивном магазине.

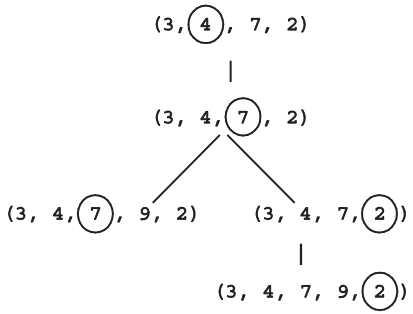
Предположим, что при разработке приложения у вас возникла необходимость создать последовательность данных. (Если этот пример кажется вам надуманным, спешу разуверить вас: при работе как над исследовательскими, так и над коммерческими программами последовательности используются очень часто.) В частности, спецификацию последовательности, которая рассматривается в данной главе, я разработал исходя из моего опыта работы в рамках исследовательского проекта по созданию интеллектуальной системы для телефонии.

3.3.1. Определение абстрактного типа данных `Sequence_T`

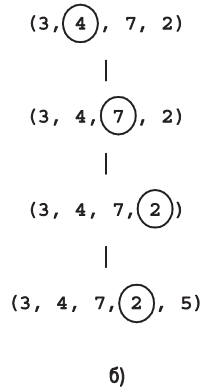
Обсудим в общих чертах структуру последовательности и действия, выполняемые над ней, а потом формализуем сказанное. Работая с последовательностью, мы должны иметь возможность обратиться к первому элементу. Нам также нужна операция получения элемента в текущей позиции. При необходимости мы должны иметь возможность включать в состав последовательности новые элементы и удалять существующие. Кроме того, в некоторых случаях работа начинается с последовательностью, не содержащей ни одного элемента, т.е. с пустой последовательностью. Заметьте, что добавляя элемент к последовательности, мы можем включить его после текущего элемента либо перед ним. Предположим, что мы хотим включить элемент 9 между элементами 7 и 2; эта ситуация показана на рис. 3.1,а. Один из способов сделать это — переместить кружок, отмечающий текущий элемент, на одну позицию вперед так, чтобы текущим элементом стал 7, а затем добавить 9 *после* текущего элемента. Мы можем поступить и по-другому: сдвинуть кружок еще на одну позицию, сделать текущим элемент 2 и включить 9 *перед* текущим элементом.

Если же мы хотим добавить элемент 5 в конце последовательности, после элемента 2, мы можем сделать это единственным способом: переместить кружок, выполняющий роль маркера, вперед на элемент 2 и добавить новый элемент 5 *после* текущей позиции. Описанные действия условно показаны на рис. 3.1,б. Аналогично, единственный способ добавить элемент в начало последовательности — сделать текущим первый элемент и вставить новый элемент *перед* ним.

Таким образом, оказывается, что нам необходима дополнительная функция, которая сообщала бы о том, что текущий элемент является последним элементом последовательности. При отсутствии такой функции мы рано или поздно попытаемся перейти за пределы последовательности и создадим проблему. Наконец, нам нужна функция, сообщающая о том, что последовательность пуста, и еще одна функция, которая позволяет определить, заполнена ли последовательность, т.е. содержит ли она максимально возможное число элементов.



а)



б)

Рис. 3.1. Действия с последовательностью

Теперь формализуем синтаксис и семантику операций над последовательностью `Sequence_T`. Последовательность содержит элементы типа `Element_T`. В данном случае `Element_T` является синонимом `KAPRec_T`; в поле адреса находится ссылка на запись `DataRec_T`. Эта запись может содержать любые данные, в состав которых входит ключевое значение. (Предполагается, что в приведенном выше рассмотрении мы оперировали именно с ключевыми значениями.) Обратите внимание, что в данном случае явно описаны операция `Create_Sequence` и противоположная ей операция `Free_Sequence`, которая будет рассмотрена в разделе 3.7.

```
// PRE    TRUE
// POST   RETURNS инициализированную
//                последовательность Sequence_T
Sequence_T
Create_Sequence();

// PRE    TRUE
// POST   IF последовательность(Sequence) заполнена
```

```
//          THEN RETURNS TRUE
//          ELSE RETURNS FALSE
BOOLEAN
Q_Full(Sequence_T Sequence);

// PRE    TRUE
// POST   IF последовательность (Sequence) пуста
//          THEN RETURNS TRUE
//          ELSE RETURNS FALSE
BOOLEAN
Q_Empty(Sequence_T Sequence);

// PRE    NOT Q_Empty(Sequence)
// POST   IF текущий элемент является последним
//           элементом последовательности(Sequence)
//          THEN RETURNS TRUE
//          ELSE RETURNS FALSE
BOOLEAN
Q_At_End(Sequence_T Sequence);

// PRE    NOT Q_Empty(Sequence)
// POST   первый элемент последовательности Sequence'
//        становится текущим
void
Go_To_Head(Sequence_T Sequence);

// PRE    NOT Q_Empty(Sequence) AND
//        NOT Q_At_End(Sequence)
// POST   текущим элементом последовательности Sequence'
//        становится элемент, следующий за текущим
//        элементом последовательности Sequence
void
Go_To_Next(Sequence_T Sequence);

// PRE    NOT Q_Empty(Sequence)
// POST   RETURNS возвращает текущий элемент
Element_T
Get_Current(Sequence_T Sequence);

// PRE    TRUE
// POST   Sequence' = Sequence с добавленным
//        элементом NewRec
//        AND IF Q_Empty(Sequence) THEN
//               текущим элементом последовательности
//               Sequence' становится ее первый элемент
//            ELSE
//               NewRec становится последующим по отношению
//               к текущему элементу Sequence'
void
Append_After_Current(Element_T NewRec, Sequence_T Sequence);

// PRE    NOT Q_Empty(Sequence)
// POST   Sequence' = Sequence с добавленным
```

```
//         элементом NewRec
//         AND NewRec в последовательности Sequence'
//             становится предшествующим элементом
//             по отношению к текущему
void
Insert_Before_Current(Element_T NewRec, Sequence_T Sequence);

// PRE   NOT Q_Empty(Sequence)
// POST  Sequence' = Sequence с удаленным текущим элементом
//         AND IF NOT(Q_Empty(Sequence'))
//               текущим элементом последовательности Sequence'
//                   становится ее первый элемент
void
Delete_Current(Sequence_T Sequence);

// PRE    Q_Empty(Sequence)
// POST   Память, выделенная для последовательности
//        Sequence, освобождается
// NOTES  Перед последующим использованием
//        последовательность Sequence должна быть повторно
//        инициализирована с помощью операции
//        Create_Sequence()
void
Free_Sequence(Sequence_T Sequence);
```

Приведенные выше описания представлены в форме прототипов функций C++, которые входят в состав файла `sequence.h`. До определенного времени мы не будем рассматривать вопросы представления данных и реализации функций. Ознакомьтесь с синтаксисом и семантикой функций и представьте себе, что они реально выполняют все действия, которые обсуждались выше. Заметьте, что для включения элемента в пустую последовательность может быть использована лишь функция `Append_After_Current`. Мы могли бы разрешить применять для этой цели также функцию `Insert_Before_Current`, но не будем делать этого. Если мы хотим работать с последовательностью, не содержащей ни одного элемента, то хотя бы одна из указанных выше функций должна допускать включение элемента в пустую последовательность.

3.3.2. Тип контейнера и чистый полиморфизм

Сейчас самое время обратить внимание на две характеристики `Sequence_T`, отличающие данный тип от `Board_T`. Во-первых, `Sequence_T` является *контейнером*. Данный термин позаимствован из объектно-ориентированных программ, где используются классы-контейнеры. Это означает, что `Sequence_T` содержит другие типы (`Element_T`) — содержимое анкет, учетные записи, целые числа и любые другие данные. Вы уже знакомы с двумя типами контейнеров, поддерживаемыми в C++, — массивами и записями (структурами). Во-вторых, `Sequence_T` является *чисто полиморфным типом*. Этот термин скорее относится к функциональному программированию, чем к объектно-ориентированному. Рассмотрим данное понятие подробнее. Тип `Sequence_T` может содержать любые элементы `Element_T`, необходимо лишь,

чтобы эти элементы были одного типа. Так, я могу в одной программе использовать Sequence_T для хранения анкет, в другой программе помещать в Sequence_T учетные записи, в третьей — включать в состав последовательности целые числа и т.д. При этом описать, разработать, реализовать и отладить Sequence_T нужно лишь однажды, после чего данную последовательность можно использовать в любом приложении. Некоторые языки накладывают ограничение, согласно которому в программе можно использовать лишь последовательность, содержащую один конкретный тип Element_T. Это означает, что если в одной и той же программе нужны две последовательности: одна — для сведений о студентах, а другая — для учетных записей пользователей, придется создать два аналогичных фрагмента кода: один — для поддержки последовательности StudSequence_T, а другой — для поддержки AccSequence_T. (Пример подобного подхода будет приведен в главе 11.) В последних версиях C++ это ограничение снято за счет использования шаблонов классов, а в таких языках, как Gofer или ML, можно реализовать Sequence_T один раз и применять ее для различных типов записей в одной и той же программе.

Обычно Element_T является абстрактным типом данных. В нашем примере это тип KAPRec_T, который большей частью представляет собой структурный тип, так как он предназначен для организации хранения и упорядочения DataRec_T. В составе KAPRec_T содержится ссылка на DataRec_T, который является *прикладным типом*, поскольку эти данные отражают ситуацию в реальной предметной области. При работе с содержимым последовательности решаются вопросы организации ввода-вывода и выбора компонентов, например ключевых значений. Соответствующий код содержится в файлах data.h/cpp и store.h/cpp.

3.4. Тестирование Sequence_T

На данном этапе разработки мы можем приступить к написанию плана тестирования. Вам это может показаться несколько преждевременным, поскольку мы еще не реализовали функции. Однако в процессе проектирования операций мы учитываем ситуации, возникающие при их работе, и можем планировать процесс тестирования. Более того, реализация кода может повредить составлению плана тестирования. Предположим, что в процессе написания функции у вас возникла проблема, которую вам с трудом, использовав весь имеющийся опыт, удалось преодолеть. Наверняка при тестировании вы уделите основное внимание фрагментам кода, связанным с этой проблемой, в ущерб остальным компонентам. В идеале план тестирования должен составлять на основе спецификаций сотрудник, не участвовавший в проектировании и реализации функций. Таким образом, план тестирования желательно составить еще до того, как вы вплотную займетесь написанием кода.

Часто спецификация позволяет начать тестирование, не дожидаясь, пока все процедуры будут реализованы, однако большинство процедур можно тестировать только в сочетании с другими. Так, например, вам придется совместно тестировать Append_After_Current, Get_Current и Go_To_Next.

Рассмотрим пример написания плана тестирования для Append_After_Current. В данном случае нам надо учесть следующие ситуации.

1. Последовательность пуста.

2. Последовательность содержит только один элемент.
3. Последовательность содержит несколько элементов, а в качестве текущего выбран первый элемент.
4. Последовательность содержит несколько элементов, а в качестве текущего выбран последний элемент.
5. Последовательность содержит несколько элементов, а текущий элемент не является ни первым ни последним.

Для выполнения простых операций программа тестирования должна предоставлять минимальную "оболочку", в которой перед вызовом примитивов выполнялась бы проверка предусловий. Мы предполагаем, что при тестировании мы сможем добавлять элементы к последовательности, переходить к следующему элементу, выводить информацию о текущем элементе и т.д.

Для каждой из перечисленных выше ситуаций предусмотрен один тест. Перед выполнением нового теста программа перезапускается. Пример плана тестирования приведен в табл. 3.1. Предполагается, что у нас уже есть четыре записи, предназначенные для включения в состав последовательности. Содержимое этих записей в данный момент не имеет значения.

Таблица 3.1. План тестирования для `Append_After_Current`

Номер теста	*Операция*	*Данные*	*Ожидаемый результат*	*Реальный результат*
1	Добавление элемента	Rec1	Вывод Rec1	
	Вывод текущего элемента			
2	Добавление элемента	Rec1		
	Добавление элемента	Rec2		
	Переход к следующему элементу			
	Вывод текущего элемента		Вывод Rec2	
3	Добавление элемента	Rec1		
	Добавление элемента	Rec2		
	Добавление элемента	Rec3		
	Добавление элемента	Rec4		
	Переход к следующему элементу			
	Вывод текущего элемента		Вывод Rec4	

Окончание табл. 3.1

Номер теста	*Операция*	*Данные*	*Ожидаемый результат*	*Реальный результат*
4	Добавление элемента	Rec1		
	Добавление элемента	Rec2		
	Переход к следующему элементу			
	Добавление элемента	Rec3		
	Переход к следующему элементу			
	Добавление элемента	Rec4		
	Вывод текущего элемента		Вывод Rec4	
5	Добавление элемента	Rec1		
	Добавление элемента	Rec2		
	Переход к следующему элементу			
	Добавление элемента	Rec3		
	Добавление элемента	Rec4		
	Переход к следующему элементу			
	Вывод текущего элемента		Вывод Rec4	

Заметьте, что если предусловие не равно `TRUE`, мы не должны тестировать функцию. Например, проверка выполнения `Go_To_Head` проводится только в том случае, когда последовательность не пуста, поскольку это явно указано в предусловии. Для сравнения можно привести процедуру проверки телевизионного приемника. Оценка качества изображения не проводится, если устройство не включено в сеть[2].

Результаты тестирования необходимо записать и сохранить вместе с другими документами, относящимися к данному проекту. Если при работе возникнет проблема, результаты тестирования помогут вам разрешить ее. Даже когда опытная эксплуатация проходит идеально, результаты тестирования все равно надо сохра-

[2] *На первый взгляд может показаться, что при реализации операции следует предусмотреть проверку предусловия. Однако в реальной работе такой подход практически неприемлем. Выполнение предусловия должен обеспечить не разработчик операции, а ее пользователь.* — Прим. авт.

нить. Если впоследствии возникнет ошибка; они помогут вам или сотруднику, занимающемуся сопровождением продукта, локализовать ее.

Применение абстрактных типов данных не гарантирует отсутствие ошибок, однако структурный подход позволяет быстро выявить фрагмент кода, ответственный за их появление. При отсутствии структуры можно лишь констатировать наличие ошибки. Меры по ее поиску неочевидны.

3.5. Проектирование и реализация Sequence_T

Коды, частично реализующие Sequence_T, содержатся файлах sequence.cpp/h, которые находятся в каталоге ch3\exercises. Основная программа содержится в файле stud_db.cpp. Одна из задач, которые вам предстоит решить, — закончить разработку и реализацию этого типа данных. В этой главе мы остановимся лишь на некоторых деталях.

3.5.1. Типы и структуры данных

Как и в предыдущем примере, мы будем применять неявный тип, используя для этого указатель void. В этом разделе приводятся некоторые фрагменты кода, содержащиеся в файле sequence.h, а прототипы функций будут приведены в разделе 3.3.2.

```
typedef void *Sequence_T;
```

Поскольку тип Element_T является синонимом KAPRec_T, нам надо включить в состав sequence.h файл описаний для KAPRec_T.

```
#include "my_const.h"
#include "store.h"
```

Синоним определяется с помощью следующего выражения:

```
typedef KAPRec_T Element_T;
```

Какая информация потребуется нам для описания последовательности? Конечно же, нам надо хранить сами элементы последовательности и отражать их порядок следования. Необходимо также идентифицировать текущий элемент. Кроме того, мы должны иметь возможность определить, сколько элементов содержится в последовательности. Зная один элемент, мы сможем найти все остальные.

Ниже представлена структура данных, реализующая Sequence_T.

```
typedef struct {
     Element_T       Recs[MaxElements];
     int CurrentElement,
         NumberElements;
} SeqRec_T, *SeqRec_Ptr_T;
```

В ней используется константа MaxElements.

```
#define MaxElements 8
```

Последовательность, состоящая максимум из восьми элементов, вряд ли может найти широкое применение на практике, но для тестирования этого вполне достаточно.

Итак, мы определили массив для хранения записей `Element_T`, поле курсора, задающее текущий элемент последовательности, и поле, указывающее, сколько элементов хранится в составе последовательности. Данная структура имеет имя `SeqRec_T`, а для работы с ней мы будем применять указатель `Seq_Ptr_T`. Именно с этим указателем мы будем иметь дело, работая с `Sequence_T`. Как вы помните из предыдущей главы, `Board_T` на самом деле представлял собой указатель `BoardStruct_Ptr_T`. Структура, используемая в данном примере, условно показана на рис. 3.2.

Рассмотренная структура может показаться сложной, особенно для программистов, которые только начинают работать с указателями. Однако, читая данную книгу, вы учитесь правильно работать со сложными структурами, а не избегать их использования. Фактически особенности работы с `Element_T` и связанными с ними записями `DataRec_T` были рассмотрены в главе 1. Новой в данном случае является только последовательность `Sequence_T`. Упрощенная диаграмма, на которой показаны только ключевые значения `KAPRec_T`, представлена на рис. 3.3.

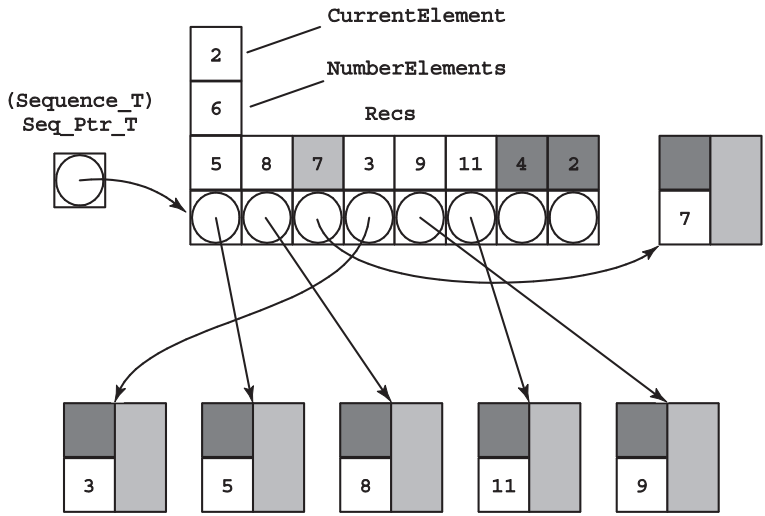


Рис. 3.2. Указатель `Seq_Ptr_T` указывает на структуру, содержащую массив `Recs` элементов `Element_T`. Поскольку `Element_T` является синонимом `KAPRec_T`, каждый из элементов содержит ключ и указатель на запись `DataRec_T`, которая в свою очередь содержит ключевое значение и другие данные. Поле `NumberElements` (в данном случае его значение равно 6) представляет число элементов, содержащихся в `Recs`; последние два элемента с ключами 4 и 2 представляют собой случайные значения, и на рисунке соответствующие клетки закрашены темно-серым цветом. Поле `CurrentElement` может принимать значения в интервале от 0 до `NumberElements-1`. В данном случае его значение равно 2, т.е. текущим является элемент с ключом 7; соответствующая клетка отмечена светло-серым цветом

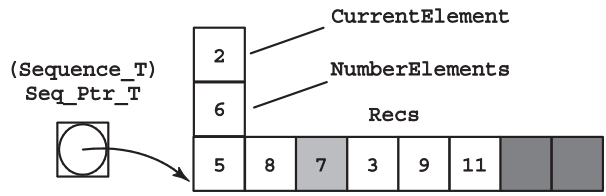


Рис. 3.3. Упрощенное представление `Sequence_T/Seq_Ptr_T`

3.5.2. Процедуры

Ниже рассмотрены коды трех процедур, выбранных в качестве примера.

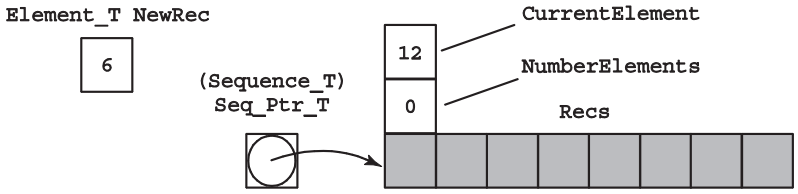
В теле функции `Q_Empty` указатель `Sequence_T`, переданный в качестве параметра, отображается в тип `Seq_Ptr_T`. Поле `NumberElements` используется для того, чтобы определить, является ли `Sequence` пустой последовательностью.

```
BOOLEAN
Q_Empty(Sequence_T Sequence) {
     SeqRec_Ptr_T Seq;

     Seq = (SeqRec_Ptr_T) Sequence;
     return(Seq->NumberElements == 0);
}
```

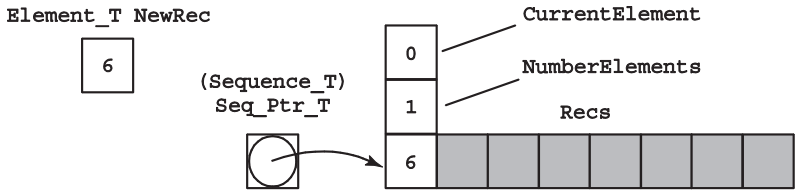
Единственное, что надо сделать в теле `Go_To_Next`, — инкрементировать значение поля `CurrentElement`.

```
void
Go_To_Next(Sequence_T Sequence) {
     SeqRec_Ptr_T Seq;

     Seq = (SeqRec_Ptr_T) Sequence;
     (Seq->CurrentElement)++;
}
```

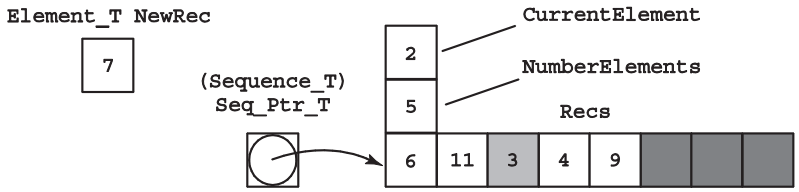
Реализуя функцию `Append_After_Current`, необходимо рассмотреть те же случаи, которые были описаны при составлении плана тестирования. Первый случай соответствует пустой последовательности. Значение `NumberElements` равно 0, а значение `CurrentElement` в данном случае не важно.



В этом случае элемент включается в состав последовательности очень просто. После включения элемента значение `CurrentElement` приобретает смысл.



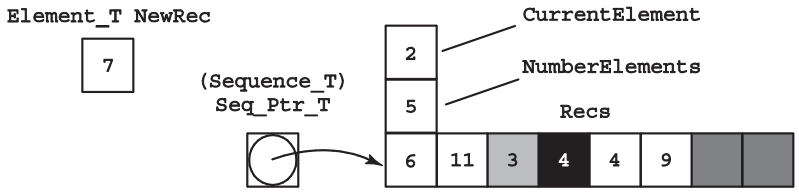
Остальные случаи можно рассматривать совместно. Выполняемые действия различаются лишь в зависимости от того, является ли текущий элемент последним.



Если текущий элемент не является последним, нам необходимо сдвинуть все элементы, следующие за текущим, на одну позицию.

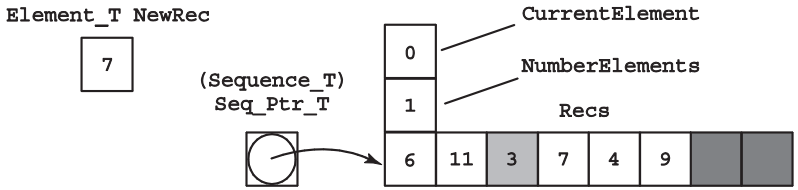
```
if Seq->NumberElements > Seq->CurrentElement + 1 ; {
  for (Index = Seq->NumberElements - 1;
       Index > Seq->CurrentElement;
       Index--) {
    Seq->Recs[Index + 1] = Seq->Recs[Index];
}
// Index = Seq->CurrentElement
```

Обратите внимание на комментарии. В них указывается состояние переменной `Index`. После завершения цикла приведенное выражение должно быть равно TRUE.



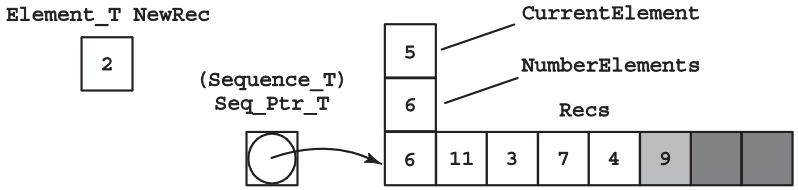
После сдвига элементов мы помещаем новый элемент на освободившееся место, а затем увеличиваем значение `NumberElements`.

```
    Seq->Recs[Seq->CurrentElement + 1] = NewRec;
    (Seq->NumberElements)++;
}
```



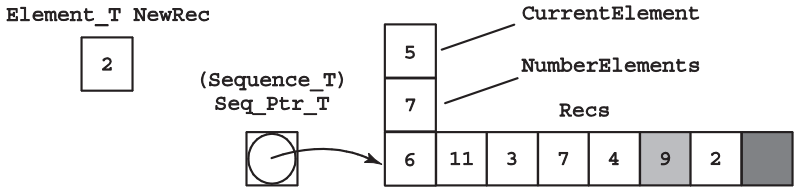
Если текущий элемент является последним, выполнять сдвиг не надо.

```
else  {
//  NumberElements == Seq->CurrentElement + 1
```



В этом случае мы лишь включаем `NewRec` после текущего элемента и инкрементируем `NumberElements`.

```
Seq->Recs[Seq->CurrentElement + 1] = NewRec;
    (Seq->NumberElements)++;
}
```



Если значение выражения `Seq->NumberElements == Seq->CurrentElement + 1` равно `TRUE`, тело цикла не обрабатывается, поскольку начальное значение переменной `Index` выглядит следующим образом:

```
Seq->NumberElements - 1 == Seq->CurrentElement + 1 - 1 == Seq->CurrentElement
```

Поэтому условие продолжения цикла не выполняется.

```
Index > Seq->CurrentElement;
```

В этом случае управление непосредственно передается следующим выражениям:

```
Seq->Recs[Seq->CurrentElement + 1] = NewRec;
(Seq->NumberElements)++;
```

В результате код, включающий элемент в непустую последовательность, можно записать более компактно. Полностью процедура `Append_After_Current` выглядит так, как показано ниже.

```
void
Append_After_Current(Element_T NewRec,
                     Sequence_T Sequence) {
     SeqRec_Ptr_T Seq;
     int Index;

     Seq = (SeqRec_Ptr_T) Sequence;
     if (Q_Empty(Seq)) {
          Seq->CurrentElement = 0;
          Seq->NumberElements = 1;
          Seq->Recs[0] = NewRec;
     }
     else {
          for ( Index = Seq->NumberElements - 1;
                Index > Seq->CurrentElement;
                Index--) {
            Seq->Recs[Index + 1] = Seq->Recs[Index];
          }
          // Index = Seq->CurrentElement
          Seq->Recs[Seq->CurrentElement + 1] = NewRec;
          (Seq->NumberElements)++;
    }
}
```

Исходные тексты других процедур находятся в файле `sequence.cpp`. Процедуры, отсутствующие в данном примере, вам предлагается реализовать самостоятельно.

3.6. Высокоуровневые операции на `Sequence_T`

В файле `stud_db.cpp` содержатся процедуры, предназначенные для тестирования простых операций, определенных на `Sequence_T`. Каждая из этих процедур использует примитивы очевидным образом. Кроме того, в данном файле содержатся коды функций более высокого уровня, в которых для получения требуемого результата вызываются примитивы. Код одной из этих функций приведен ниже.

```
// PRE   NOT Q_Empty_Sequence(DataBase)
// POST  RETURNS число элементов в базе данных
//       AND Q_At_End(DataBase')
// NOTES это функция с побочными эффектами
int
Find_Size(StudentDB_T DB) {
    int Size;
    Go_To_Head(DB);
    for (Size = 1; !Q_At_End(DB); Size++) {
```

```
        Go_To_Next(DB);
    }
    return(Size);
}
```

Заметьте, что размер `Sequence_T` определяется с помощью простых операций; при этом не используются знания о внутренней структуре последовательности. Возможно, вы захотите ради повышения производительности реализовать `Find_Size` как примитив, однако семантика функции останется неизменной.

3.7. Освобождение памяти

На протяжении всей книги мы будем время от времени возвращаться к вопросу об управлении памятью. Сейчас самое подходящее время рассмотреть небольшой пример. Функция `Create_Sequence` выделяет память, достаточную для хранения структуры `SeqRec_T`. После завершения работы функции возвращается указатель на начало выделенной области. По окончании работы с последовательностью память остается занятой. Чтобы она могла быть использована для хранения других данных, ее надо освободить. Конечно же, это можно сделать с помощью оператора `delete`, предусмотренного в языке C++; оператору `delete` передается указатель, а в результате его выполнения память, занимаемая структурой, освобождается. Однако на уровне абстрактных типов данных мы не знаем, какая именно структура используется для хранения информации. В `sequence.h` эти сведения "скрыты" за указателем `void`. Таким образом, для освобождения памяти нам нужна специальная операция на уровне абстрактного типа данных. Описание и реализация этой операции приведены ниже.

```
// PRE   Q_Empty(Sequence)
// POST  Освобождается вся память,
//       выделенная для последовательности
// NOTES Перед последующим использованием
//       последовательность Sequence должна быть повторно
//       инициализирована с помощью операции
//       Create_Sequence()
void
Free_Sequence(Sequence_T Sequence) {
       SeqRec_Ptr_T Seq;

       Seq = (SeqRec_Ptr_T) Sequence;
       delete(Seq);
}
```

В предусловии требуется, чтобы последовательность была пуста. До тех пор, пока это требование не будет выполняться, `Free_Sequence` не может быть вызвана. В состав спецификации я включил раздел `NOTES`. Комментарии, содержащиеся в нем, могут быть полезны при работе. Создавая комментарии, следите, чтобы они были не слишком объемны, в противном случае они могут не прояснить спецификацию, а затруднить ее понимание.

Ниже представлен текущий вариант `Finalise`, содержащийся в `stud_db.cpp`.

```
// PRE   TRUE
// POST  Очистка
void
Finalise(StudentDB_T DB) {
//   for(;!Q_Empty(DB);) {
//      Delete_Current(DB);
//   }
//   Free_Sequence(DB);
     cout << "That is it !\n";
}
```

Большая часть кода представлена как комментарии. Дело в том, что функция `Delete_Current` отсутствует. Реализовать ее вам предстоит в качестве упражнения.

3.8. Структура программы и абстрактные типы данных

В файлах `stud_db.cpp` и `ord_db.cpp` содержатся два примера программ, использующих `Sequence_T`, `DataRec_T` и `KAPRec_T`. Структура этих программ представлена на рис. 3.4.

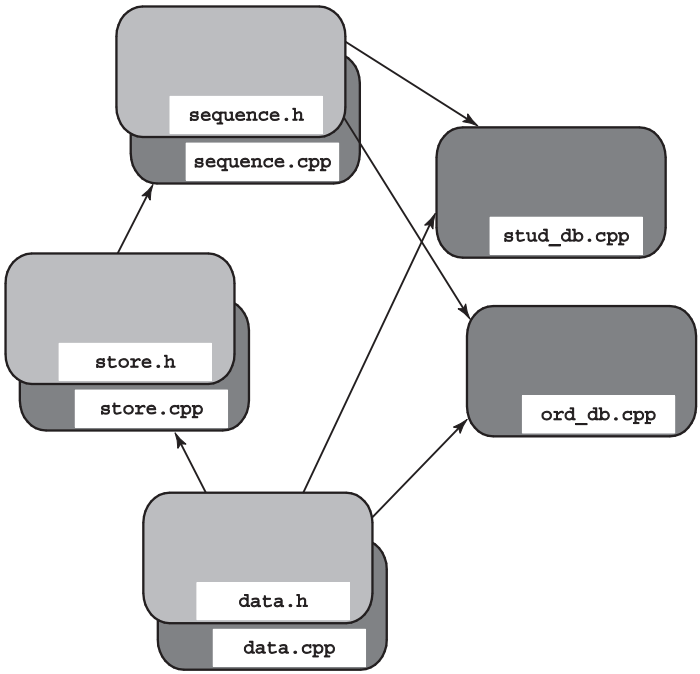


Рис. 3.4. Структура программ `stud_db.cpp` и `ord_db.cpp`

Если мы реализуем Sequence_T по-другому (это будет сделано в главе 4), нам надо лишь удалить старые файлы sequence.h/cpp, поместить вместо них в каталог новые файлы sequence.h/cpp и перекомпилировать stud_db.cpp и ord_db.cpp. Если тип данных изменится, соответственно изменятся файлы data.h и data.cpp. В этом случае придется перекомпилировать все файлы, но вносить изменения в их исходные коды не потребуется.

Структура программ зависит от структуры используемых абстрактных типов данных. Желательно, чтобы спецификации абстрактных типов и реализующие их коды содержались в отдельных файлах .h/.cpp.

Упражнения

1. Чем средства удаленного управления телевизионным приемником напоминают абстрактный тип данных? Перечислите типичные операции, имеющие отношение к вводу и выводу. Как реализован этот абстрактный тип данных?
2. Что общего между субподрядчиком в строительстве и абстрактным типом данных?
3. Функции Q_At_End, Go_To_Head, Insert_Before_Current и Delete_Current, объявленные в файле sequence.cpp, требуют модернизации.
   а) В какой последовательности должны модифицироваться эти функции, чтобы вы могли по ходу работы выполнять тестирование?
   б) Для каждой функции разработайте план реализации и тестирования. При необходимости применяйте диаграммы. Напишите коды программ и протестируйте их.
4. Используя в качестве примера Find_Sise, разработайте и реализуйте функцию Print_All, спецификация которой приведена ниже. Считайте, что сообщение, которое должно отображаться, уже существует.

```
// PRE   TRUE
// POST  IF   база данных не пуста, все значения
//            выводятся на экран
//       ELSE отображается сообщение о том, что
//            база данных пуста
void
Finalise(StudentDB_T DB)
```

5. В файле ord_db.cpp содержится программа, реализующая простую базу данных, в которой записи должны располагаться в порядке, определяемом ключевыми значениями. Напишите код функции Add_Stud, реализующий указанное поведение базы. Руководствуясь спецификацией, составьте план тестирования. (Подсказка: обратите внимание на функцию Q_Present.)
6. В каталоге ch3\examples\numbers содержится приложение, которое использует целые числа (int) в качестве элементов Element_T, хранящихся в составе последовательности Sequence_T. Изучите код и определите, какие изменения надо внести в sequence.h, чтобы в Sequence_T вместо KAPRec_T использовался тип int.

3.9. Резюме

В данной главе вы узнали следующее.

1. В дополнение к определению абстрактного типа данных в этой главе были даны определения принципов абстрагирования данных, абстрагирования процедур и сокрытия информации. Для пояснения определений приведены аналогии из других областей.
2. Тип `Sequence_T` сначала был описан неформально, а затем определен как чисто полиморфный, абстрактный тип данных, являющийся контейнером.
3. Вы узнали о назначении плана тестирования и научились составлять его.
4. Процесс проектирования и реализации `Sequence_T` иллюстрировался диаграммами, связывающими структуру данных с процедурами.
5. При рассмотрении типа `Sequence_T` были затронуты вопросы управления памятью.
6. Была рассмотрена зависимость структуры программы от выбранных абстрактных типов данных.