Содержание

Над книгой работали

Авторы

Прасид Пай работает в индустрии программного обеспечения на протяжении 25 лет. Начинал с системного программирования в среде MS DOS на языке ANSI C. Принимал активное участие в разработке крупных кроссплатформенных систем на языке C++ для систем Windows, GNU Linux и macOS. Обладает опытом применения технологий COM+ и CORBA на языке C++. В последнее десятилетие работает с языком Java и платформой .Net.

Выступил основным разработчиком компилятора с языка SLANG4.net, первоначально написанного на языке C#, а затем портированного на язык C++ с использованием системы LLVM. Соавтор книги «.NET Design Patterns» («Шаблоны проектирования для платформы .NET»), вышедшей в издательстве Packt Publishing.

Питер Абрахам стал приверженцем языка программирования C++ и пылким борцом за производительность кода со времён своей учёбы в колледже, где он достиг высот в программировании для операционных систем Windows и GNU Linux. Он обогатил свой опыт программированием для архитектуры CUDA, обработкой изображений, компьютерной графикой, работая в таких компаниях, как Quest Global, Siemens и Tektronics.

В своей профессиональной деятельности Питер постоянно использует последние нововведения языка C++ и библиотеку RxCpp. Он обладает большим опытом работы с инструментами разработки кроссплатформенных графических приложений, такими как Qt, WxWidgets и FOX toolkit.

Рецензент

Сумант Тамбе – разработчик программного обеспечения, исследователь, участник разработки с открытым кодом, блогер, докладчик на различных конференциях, автор многочисленных статей и любитель компьютерных игр. Опытен в применении современного языка C++, брокера сообщений Kafka, различных служб распространения данных, методологии реактивного программирования и потоков данных для решения новых задач, возникающих в области больших данных и интернета вещей.

Автор блога «C++ Truths» («Истины о языке C++») и вики-книги «More C++ Idioms» («Ещё идиомы программирования на языке C++»). Делится своими знаниями в блоге, на конференциях, семинарах и встречах профессионального сообщества. Удостаивался звания Microsoft MVP в области технологий разработки программ на протяжении пяти лет. Имеет докторскую степень по компьютерным наукам в университете Вандербильта.

Предисловие

Эта книга поможет читателю овладеть парадигмой реактивного программирования на языке C++ и создавать асинхронные и многопоточные приложения. Книга включает в себя задачи, взятые из реальной практики, которые читателю предстоит решать с помощью реактивной модели программирования. Здесь освещен долгий путь становления средств обработки событий. Читатель узнает о поддержке параллельного программирования в языке C++ и о функциональном реактивном программировании. Описанные в книге конструкции на базе объектно-ориентированного и функционального программирования позволят читателю создавать эффективные программы. Наконец, читатель узнает о программировании микросервисов на языке C++ и научится создавать собственные операции для библиотеки `RxCpp`.

Для кого предназначена эта книга

Разработчик, программирующий на языке C++ и интересующийся применением реактивного программирования для создания асинхронных и параллельных приложений, найдёт эту книгу чрезвычайно интересной. От читателя не требуется наличие каких-либо знаний реактивного программирования.

Что охватывает эта книга

В главе 1 «Модель реактивного программирования – обзор и история» вводятся некоторые структуры данных, ключевые для реактивной модели программирования (для краткости – Rx). В ней речь идёт также об обработке событий в графических интерфейсах пользователя, дан общий обзор реактивного программирования, рассказано о реализации графических версий различных видов интерфейса на основе библиотеки классов MFC.

В главе 2 «Современный язык C++ и его ключевые идиомы» рассказано о тонкостях языка C++, о правилах вывода типов, шаблонах с переменным числом аргументов, ссылках rvalue и семантике перемещения, лямбда-функциях, основах функционального программирования, соединении операций в конвейер, а также о том, как реализовать итератор и наблюдателя.

Глава 3 «Параллельное и многопоточное программирование на языке C++» содержит сведения о средствах многопоточного программирования, включённых в стандарт языка C++. Читатель узнает, как запустить поток и управлять им, а также о различных тонкостях стандартной библиотеки, связанных с этим. Эта глава представляет собой хорошее введение в средства поддержки многопоточности, появившиеся в новом стандарте языка C++.

В главе 4 «Асинхронное программирование и неблокирующая синхронизация в языке C++» рассказано о средствах, предоставляемых стандартной библиотекой для организации параллельных вычислений на основе задач. Также в ней говорится о появившейся в современном языке C++ модели памяти для многопоточного программирования.

В главе 5 «Знакомство с наблюдаемыми источниками» говорится об одном из шаблонов проектирования «Банды четырёх» – шаблоне «Наблюдатель» и о его недостатках. Читатель узнает о шаблонах проектирования «Компоновщик» и «Посетитель» в контексте моделирования дерева синтаксического разбора выражения.

В главе 6 «Введение в программирование потоков событий на языке C++» внимание сосредоточено на программировании потоков событий. Также рассматривается библиотека Streamulus, в которой реализован подход к обработке потоков событий на основе встраиваемых предметно-ориентированных языков (англ. Domain-Specific Embedded Language, DSEL). Изложение сопровождается рядом примеров программ.

Глава 7 «Знакомство с моделью потоков данных и библиотекой RxCpp» открывается общим обзором вычислительной парадигмы на основе потоков данных, затем показаны основы создания программ с помощью библиотеки `RxCpp`. Читатель изучит набор операций, предоставляемых этой библиотекой.

Глава 8 «Ключевые элементы библиотеки RxCpp» даёт представление о том, как средства библиотеки `RxCpp` взаимодействуют между собой. Глава открывается разбором объектов наблюдения (Observable), далее описаны механизм подписки и реализация расписания.

Глава 9 «Реактивное программирование графических интерфейсов с помощью библиотеки Qt» посвящена применению парадигмы реактивного программирования для создания графических интерфейсов пользователя на основе библиотеки Qt. Читатель узнает о концептуальной основе библиотеки Qt, об иерархии её классов, системе метаобъектов, а также о механизме сигналов и слотов. В качестве примера создаётся приложение, обрабатывающее события от мыши и фильтрующее их. Затем читателю предстоит знакомство с более сложной темой – как создавать собственные реактивные операции средствами библиотеки `RxCpp`, если имеющихся в ней операций не хватает для той или иной задачи. Знание этих аспектов также поможет понять, как создавать композитные операции, состыковывая уже имеющиеся. Последняя тема в данной книге не освещается, подробную информацию можно найти по адресу https://www.packtpub.com/sites/default/files/downloads/Creating_Custom_Operators_in_RxCpp.pdf.

Глава 10 «Шаблоны и идиомы для реактивного программирования на языке C++» погружает читателя в чудесный мир шаблонов проектирования и идиом языка программирования. Изложение начинается с шаблонов «Банды четырёх», затем разобраны шаблоны, специфические для реактивного программирования.

В главе 11 «Реактивные микросервисы на языке C++» показано, как реактивная модель программирования может использоваться для создания реактив-

ных микросервисов на языке C++. Читатель узнает о наборе средств разработки Microsoft C++ REST SDK и связанной с ним модели программирования.

Глава 12 «Особые возможности потоков и обработка ошибок» посвящена средствам обработки ошибок из библиотеки RxCpp, а также некоторым сложным конструкциям и операциям для обработки потоков событий. Рассказано, как продолжать поток событий после возникновения ошибки, как ожидать, пока источник потока исправляет ошибочное состояние, и возобновлять обработку и как работают обобщённые операции, способные обрабатывать как ошибочные, так и нормальные ситуации.

Как извлечь из этой книги максимум пользы

Для понимания большинства вопросов, рассматриваемых в этой книге, читателю необходимо владеть программированием на языке C++.

Загрузка исходного кода примеров

Файлы с исходным кодом примеров программ можно найти на сайте www.packtpub.com. Покупатель этой книги может посетить страницу www.packtpub.com/support, зарегистрироваться и получить файлы по электронной почте на свой адрес.

Для загрузки файлов нужно выполнить следующие шаги.

1. Войти или зарегистрироваться на сайте www.packtpub.com.
1. Перейти по ссылке **Support**.
2. Щёлкнуть по ссылке **Code Downloads & Errata**.
3. Ввести английское название книги в поле **Search** и выполнить дальнейшие инструкции на сайте.

Когда файлы получены, нужно распаковать их с помощью последних версий архиваторов:

- WinRAR или 7-Zip для ОС Windows;
- Zipeg, iZip или UnRarX для системы Mac;
- 7-Zip или PeaZip для системы Linux.

Исходный код всех примеров к этой книге также можно найти в системе GitHub по адресу https://github.com/PacktPublishing/CPP-Reactive-Programming. Возможные обновления этих примеров будут публиковаться в этом же репозитории.

По адресу https://github.com/PacktPublishing/ можно также найти обширные репозитории с примерами и к другим книгам и видеолекциям. Рекомендуем читателю ознакомиться с ними.

Как загрузить цветные иллюстрации

Издательство также предоставляет файл в формате PDF, в котором собраны цветные рисунки, снимки экрана, диаграммы и другие иллюстрации к этой

книге. Их можно загрузить по адресу https://www.packtpub.com/sites/default/files/downloads/CPPReactiveProgramming_ColorImages.pdf.

Принятые в тексте соглашения

На протяжении всей книги текст оформлен с использованием следующих соглашений.

`Код_в_тексте`: таким шрифтом набраны размещённые в основном тексте элементы исходного кода, имена таблиц баз данных, имена директорий и файлов, расширения имён файлов, пути к файлам, адреса сетевых ресурсов, пользовательский ввод, идентификаторы пользователей в сети Twitter и т. д. Пример использования: «Приведённый выше фрагмент кода инициализирует объект структурного типа `WNDCLASS` (или `WNDCLASSEX` на новых системах) нужным шаблоном окна».

Самостоятельные фрагменты кода, вынесенные в отдельный абзац, оформлены следующим образом:

```
/* закрыть соединение с сервером */
XCloseDisplay(display);

return 0;
}
```

Если нужно обратить внимание читателя на определённую часть кода, она набирается полужирным шрифтом.

```
/* закрыть соединение с сервером */
XCloseDisplay(display);

return 0;
}
```

Всё, что вводится или выводится в консоли, передано в книге следующим образом:

```
$ mkdir css
$ cd css
```

Полужирным начертанием отмечены новые термины, важные слова и текст, отображаемый программой на экране. Например, таким способом показаны пункты меню и сообщения в диалоговых окнах. Пример использования: «На жаргоне оконного программирования это называется циклом обработки **сообщений**».

Так будут оформляться предупреждения и важные примечания.

Так будут оформляться советы или рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу www.dmkpress.com.

Список опечаток

Хотя мы приняли все возможные меры, для того чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу www.dmkpress.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты www.dmkpress.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Модель реактивного программирования – обзор и история

Появление систем X Window System, Microsoft Windows и IBM OS/2 Presentation Manager сделало популярным программирование графических интерфейсов для платформы PC. Это стало большим шагом вперёд по сравнению с преобладавшим ранее интерфейсом командной строки и подходом к программированию, ориентированным на пакетную обработку. Реагирование на событие оказалось в центре внимания программистов по всему миру, тогда как разработчики платформ принялись за создание прикладных программных интерфейсов, с помощью которых программисты могли бы обрабатывать события, – низкоуровневых, в духе языка C, использующих указатели на функции обратного вызова. Модели программирования при этом основывались в основном на модели кооперирующихся потоков, а по мере появления более совершенных микропроцессоров большинство платформ стало поддерживать также и вытесняющую многопоточность. Обработка событий (как и другие асинхронные задачи) становилась всё сложнее, и традиционные подходы к реагированию программ на события всё менее поддавались масштабированию. Несмотря на то что появлялись превосходные инструменты для создания графических интерфейсов, основанные на языке C++, для обработки событий по-прежнему использовались числовые коды сообщений, диспетчеризация через указатели на функции и другие низкоуровневые технологии. Ведущие разработчики компиляторов даже пытались добавлять свои расширения в язык C++, чтобы облегчить программирование в среде Windows. Обработка событий, асинхронные вычисления и другие подобные задачи требовали новых подходов. К счастью, современный стандарт языка C++ поддерживает функциональную парадигму программирования, содержит средства управления потоками вместе с подходящей моделью памяти и улучшенные средства управления памятью, что позволяет программистам работать с асинхронными потоками данных, в частности трактовать события как потоки. Всё это достигается благодаря модели

программирования, называемой реактивным программированием. Чтобы показать общую картину явления, осветим в этой главе следующие вопросы:

- событийно-ориентированная модель программирования и её реализации на различных платформах;
- что собой представляет реактивное программирование;
- различные модели реактивного программирования;
- разбор нескольких простых программ для закрепления понимания основных понятий;
- методология, принятая в данной книге.

Событийно-ориентированная модель программирования

Событийно-ориентированное программирование – это такая модель программирования, в которой ход выполнения программы определяется событиями. Примерами событий могут быть нажатия на кнопку мыши, нажатия клавиш на клавиатуре, жесты на сенсорном экране, сигналы от датчиков, сообщения от других программ и т. д. Событийно-ориентированное приложение основано на механизмах, позволяющих обнаружить события в реальном масштабе времени (или близко к тому) и отвечать, реагировать на них, вызывая подходящие процедуры – обработчики событий. Поскольку большинство ранних программ, обрабатывающих события, было написано на языках C и C++, для организации обработчиков в них применялись низкоуровневые технологии наподобие указателей на функции обратного вызова. Более поздние системы, такие как Visual Basic, Delphi и другие среды быстрой разработки приложений, содержали уже встроенные средства событийно-ориентированного программирования. Чтобы отчётливее показать предмет, сделаем краткий обзор механизмов обработки событий в нескольких различных платформах. Это поможет читателю лучше понять круг проблем, для решения которых предназначены реактивные модели программирования (в контексте разработки графических интерфейсов пользователя).

В реактивном программировании данные рассматриваются как потоки, а события в системах оконного интерфейса могут рассматриваться как потоки, разнородные элементы которых должны обрабатываться единообразно. Реактивная модель программирования предоставляет средства для сбора событий из разных источников в поток, фильтрации потоков, различных преобразований над потоками, выполнения тех или иных действий над элементами потоков и т. д. Эта модель программирования содержит в своей основе средства асинхронной обработки и управление расписанием асинхронных действий. В этой главе в основном рассматриваются ключевые структуры данных, характерные для реактивного программирования, и то, как создавать простейшие реактивные программы. Реальным реактивным программам внутренне присущ асинхронный принцип работы, тогда как примеры из этой главы работают синхронно. Прежде чем переходить к асинхронному порядку выполнения и управлению расписанием, предстоит сперва объяснить ряд теоретических принципов и соответствующих языковых конструкций, это будет сделано в следующих главах. Примеры из данной главы служат только для первоначального знакомства с предметом и представляют лишь учебный интерес.

Событийно-ориентированное программирование в системе X Window

Система X Window представляет собой кроссплатформенный интерфейс прикладного программирования, поддерживается главным образом в системах, отвечающих стандарту POSIX, а также перенесена в систему Microsoft Windows. Фактически программный интерфейс X представляет собой сетевой протокол оконного графического ввода-вывода, которому требуется оконный менеджер, управляющий совокупностью окон. Клиентское приложение формирует графические образы, а X-сервер отвечает за их отображение на экране конкретной машины. В персональных настольных системах, как правило, графический клиент и сервер работают локально, на одной и той же машине. Следующая программа поможет читателю понять дух модели программирования, лежащей в основе библиотеки XLib, и присущий ей способ обработки событий.

```
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    Display *display;
    Window window;
    XEvent event;
    char *msg = "Hello, World!";
    int s;
```

В этом фрагменте кода подключены необходимые заголовочные файлы, в которых содержатся прототипы функций, предоставляемых библиотекой XLib для языка C. Создавая программы на основе одной лишь библиотеки XLib, программисту нужно иметь дело с некоторыми специфическими структурами данных. В наше время, впрочем, для создания коммерческих программных продуктов чаще всего используют такие высокоуровневые библиотеки-обёртки, как Qt, WxWidgets, Gtk+ или Fox.

```
    /* open connection with the server */
    display = XOpenDisplay(NULL);
    if (display == NULL){
        fprintf(stderr, "Cannot open display\n");
        exit(1);
    }
    s = DefaultScreen(display);
    /* create window */
    window = XCreateSimpleWindow(display,
        RootWindow(display, s), 10, 10, 200, 200, 1,
        BlackPixel(display, s), WhitePixel(display, s));

    /* select kind of events we are interested in */
```

```
    XSelectInput(display, window, ExposureMask | KeyPressMask);

    /* map (show) the window */
    XMapWindow(display, window);
```

В этом фрагменте кода инициализируется соединение с графическим сервером, затем создаётся окно с заданными параметрами. Как правило, программы в среде X Window работают под управлением оконного менеджера, который управляет взаимным расположением окон. Мы выбрали интересующие нашу программу типы сообщений, вызвав функцию `XSelectInput`, перед тем как отобразить окно.

```
    /* event loop */
    for (;;)
    {
        XNextEvent(display, &event);

        /* draw or redraw the window */
        if (event.type == Expose)
        {
            XFillRectangle(display, window,
                DefaultGC(display, s), 20, 20, 10, 10);
            XDrawString(display, window,
                DefaultGC(display, s), 50, 50, msg, strlen(msg));
        }
        /* exit on key press */
        if (event.type == KeyPress)
            break;
    }
```

Затем программа входит в бесконечный цикл, в котором запрашивает очередное событие, а соответствующая функция из библиотеки XLib отображает в окне текстовую строку. На жаргоне оконного программирования это называется циклом обработки **сообщений**. Для получения очередного события используется функция `XNextEvent`.

```
    /* close connection to server */

    XCloseDisplay(display);
    return 0;
}
```

Покинув «бесконечный» цикл обработки сообщений, программа закрывает соединение с графическим сервером.

Событийно-ориентированное программирование в среде Microsoft Windows

Корпорация Microsoft разработала модель программирования графических интерфейсов пользователя, которую можно считать наиболее успешной оконной системой в мире. Третья версия системы Windows имела ошеломительный

успех в 1990 г., и фирма Microsoft продолжила его развивать в версиях Windows NT, Windows 95, 98, ME. Рассмотрим в общих чертах модель событийно-ориентированного программирования в системе Microsoft Windows (за подробной информацией о том, как работает эта модель, можно обратиться к документации фирмы Microsoft). Следующий пример поможет понять суть программирования в среде Windows на языках C и C++.

```
#include <windows.h>
//----- Prtotype for the Event Handler Function
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
    WPARAM wParam, LPARAM lParam);
//--------------- Entry point for a Idiomatic Windows API function
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{

    MSG msg = { 0 };
    WNDCLASS wc = { 0 };
    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
    wc.hbrBackground = (HBRUSH)(COLOR_BACKGROUND);
    wc.lpszClassName = "minwindowsapp";
    if (!RegisterClass(&wc))
        return 1;
```

Приведённый выше фрагмент кода инициализирует объект структурного типа `WNDCLASS` (или `WNDCLASSEX` на новых системах) нужным шаблоном окна. Самое важное в этой структуре – это поле `lpfnWndProc`, в нём находится адрес функции, посредством которой экземпляр окна отвечает на события.

```
    if (!CreateWindow(wc.lpszClassName,
        "Minimal Windows Application",
        WS_OVERLAPPEDWINDOW | WS_VISIBLE,
        0, 0, 640, 480, 0, 0, hInstance, NULL))
        return 2;
```

Вызов функции `CreateWindow` (или `CreateWindowEx` на новых версиях ОС Windows) создаёт окно на основе оконного класса с именем, взятым из параметра `WNDCLASS.lpszClassname`.

```
    while (GetMessage(&msg, NULL, 0, 0) > 0)
        DispatchMessage(&msg);
    return 0;
}
```

Этот блок кода запускает цикл, в который берёт из очереди новые сообщения до тех пор, пока не будет получено сообщение `WM_QUIT`. Сообщение `WM_QUIT` завершает цикл. В некоторых случаях их необходимо также подвергнуть некоторой предварительной обработке, перед тем как передавать их функции `DispatchMessage`. Наконец, системная функция `DispatchMessage` вызывает оконную функцию обратного вызова, адрес которой был ранее передан через поле `lpfnWndProc`.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
    WPARAM wParam, LPARAM lParam) {
    switch (message) {
    case WM_CLOSE:
        PostQuitMessage(0); break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Показанный выше фрагмент кода представляет собой минимальную функцию обратного вызова (в англоязычной литературе – callback). Обратившись к документации фирмы Microsoft, можно узнать больше о прикладном интерфейсе программирования в среде Windows и о том, как события обрабатываются в программах.

Событийно-ориентированное программирование в каркасе Qt

Каркас Qt представляет собой кроссплатформенный и многоплатформенный инструментарий профессиональной разработки программ, включая разработку графических интерфейсов пользователя, который работает в средах Windows, GNU Linux, macOS X и в других системах семейства Mac. Этот инструментарий поддерживает также встроенные системы и мобильные устройства. Модель программирования на языке C++ в этом каркасе основана на использовании специального инструмента, называемого **метаобъектным компилятором** (англ. Meta Object Compiler, MOC); он просматривает исходный код в поисках директив (особых макросов и расширений языка), помещённых в исходный код, и определённым образом генерирует вспомогательный исходный код, отвечающий за обработку событий. Таким образом, перед тем как компилятор языка C++ получит исходный код, этот код должен пройти сквозь инструмент MOC, который сгенерирует код, отвечающий стандарту ANSI C++ и не содержащий языковых конструкций, специфических для системы Qt. Более подробные сведения можно почерпнуть из документации по каркасу Qt. Следующая простая программа демонстрирует ключевые моменты программирования в каркасе Qt и присущую ему логику обработки событий.

```
#include <qapplication.h>
#include <qdialog.h>
#include <qmessagebox.h>
#include <qobject.h>
#include <qpushbutton.h>

class MyApp : public QDialog {
    Q_OBJECT
public:
    MyApp(QObject* /*parent*/ = 0):
        button(this)
    {
```

```
        button.setText("Hello world!");
        button.resize(100, 30);
        // When the button is clicked, run button_clicked
        connect(&button,
            &QPushButton::clicked, this, &MyApp::button_clicked);
    }
```

Макрос Q_OBJECT указывает метаобъектному компилятору сгенерировать таблицу диспетчеризации события (`Event Dispatch`). Когда источник событий подключается к приёмнику, в эту таблицу вставляется новая строка. Сгенерированный код обрабатывается компилятором наряду с исходным кодом, написанным программистом, в результате чего строится исполняемая программа.

```
public slots:
    void button_clicked() {
        QMessageBox box;
        box.setWindowTitle("Howdy");
        box.setText("You clicked the button");
        box.show();
        box.exec();
    }
    protected:
        QPushButton button;
};
```

Слово `slots` как расширение языка особым образом обрабатывается метаобъектным компилятором при генерации вспомогательного кода, но для компилятора языка C++ прозрачно, так как представляет собой обычный макрос.

```
int main(int argc, char** argv) {
    QApplication app(argc, argv);
    MyApp myapp;
    myapp.show();
    return app.exec();
}
```

Этот последний фрагмент кода инициализирует объект, играющий роль обёртки над приложением, и отображает главное окно. В целом Qt можно назвать наилучшим из всех каркасов для разработки приложений, разработанных для языка C++. Кроме того, в нём имеются хорошие средства интеграции с популярным языком программирования Python.

Событийно-ориентированное программирование средствами библиотеки MFC

Библиотека классов MFC (Microsoft Foundation Classes) по сей день остаётся довольно популярным средством для создания приложений в среде Microsoft Windows. Если прибавить к ней ещё и библиотеку **ATL** (**ActiveX Template Library**), можно получить также некоторую поддержку веб-программирования. Для обработки событий в библиотеке MFC используется механизм, называемый схемой сообщений, или таблицей сообщений (message map). Таблица

сообщений, оформленная с помощью специальных макросов, как показано в следующем примере, присутствует в каждой программе, созданной на основе библиотеки MFC.

```
BEGIN_MESSAGE_MAP(CClockFrame,CFrameWnd)
    ON_WM_CREATE()
    ON_WM_PAINT()
    ON_WM_TIMER()
END_MESSAGE_MAP()
```

Эта таблица определяет реакцию на стандартные сообщения системы Windows: сообщения `WM_CREATE`, `WM_PAINT` и `WM_TIMER` должны обрабатываться, соответственно, функциями `OnCreate`, `OnPaint` и `OnTimer`. На уровне внутренних механизмов реализации, глубоко скрытых от пользователя, эти таблицы представляют собой массивы, в которых целочисленный код сообщения используется для поиска нужной строки, содержащей указатель на функцию-обработчик. Таким образом, отличие данного подхода от модели обработки сообщений на основе системных вызовов Windows оказывается при внимательном рассмотрении не слишком значительным.

Мы не приводим здесь пример исходного кода, поскольку среди доступных для скачивания примеров программ к этой книге имеется полная реализация одного характерного графического приложения в духе модели реактивного программирования на основе библиотеки MFC. Читатель может изучить исходный код и комментарии к нему, чтобы разобраться в неочевидных аспектах обработки событий в библиотеке MFC.

Прочие модели событийно-управляемого программирования

Системы распределённой обработки объектов, такие как COM+ или CORBA, обладают собственными подсистемами обработки событий. Модель событий, принятая в технологии COM+, основана на понятии «точки соединения» (англ. connection point), представленном интерфейсами `IConnectionPointContainer` и `IConnectionPoint`, а в технологии CORBA реализована модель на основе так называемого сервиса событий[1] (event service). Спецификация CORBA поддерживает оба механизма оповещения о событиях: «втягивание» и «вталкивание». Разбор технологий COM+ и CORBA выходит за рамки этой книги, читатель может обратиться к соответствующей документации.

Ограничения классических моделей обработки событий

Цель этого краткого обзора моделей обработки событий, присущих различным платформам, состоит в том, чтобы показать предмет с нужной точки зрения.

[1] Следует отметить, что сервис событий в технологии CORBA подвергается критике из-за ряда серьёзных недостатков (схема доставки сообщений, при которой каждый клиент получает сообщения обо всех событиях от сервера, перегружает сеть; отсутствует фильтрация сообщений по каким-либо критериям, синхронный режим отправки сообщений) – см, например, http://www.k-press.ru/cs/2000/3/corba/corba_callback.asp. – *Прим. перев.*

Логика реагирования на события в этих моделях обычно тесно связана с платформой, для которой создан код. Хотя с появлением многоядерных процессоров создание многопоточного кода на основе низкоуровневых средств стало чересчур сложным, в стандартной библиотеке языка C++ стала доступна высокоуровневая модель параллельного программирования на основе задач. Но ведь источники событий чаще всего написаны отнюдь не на основе стандартной библиотеки! Так, в стандарт языка C++ не включена библиотека для создания графических интерфейсов пользователя, единый интерфейс для доступа к внешним устройствам и т. д. Как преодолеть это противоречие? К счастью, события и данные от внешних источников можно организовывать в потоки или последовательности, которые затем можно весьма эффективно обрабатывать, используя средства функционального программирования, такие как лямбда-функции. При этом нетрудно получить и дополнительную выгоду: если наложить некоторые ограничения на изменение значений объектов, параллельная обработка окажется неотъемлемой частью модели обработки потоков.

Реактивная модель программирования

Говоря упрощённо, реактивное программирование – это не что иное, как программирование с асинхронными потоками данных. Применяя к потокам те или иные операции, можно решать различные вычислительные задачи. Первая задача реактивной программы – превратить свои данные в поток, из какого бы источника эти данные ни были получены. Так, создавая современное приложение с графическим интерфейсом, нужно обрабатывать события перемещения и нажатия кнопок мыши. Сейчас в большинстве приложений функция обратного вызова (callback) вызывается при наступлении такого события и сразу обрабатывает его. При этом большую часть времени обработчик события занят фильтрацией событий по тем или иным критериям, а затем вызывает функцию для обработки конкретной разновидности события. В контексте этой задачи применить реактивную модель программирования – значит собрать события от мыши (типа перемещения и нажатия кнопки) в коллекцию, затем установить на эту коллекцию фильтр, наконец оповещать обработчики. В таком случае логика взаимодействия приложения с обработчиком не будет вызываться без необходимости.

Модель, основанная на обработке потоков, широко известна и довольно проста в реализации. Почти всё, что угодно, можно превратить в поток. Примерами потоков могут служить сообщения, логи, каналы в сети Twitter, блоги, ленты новостей и т. д. Методы функционального программирования очень хорошо подходят для обработки потоков. Такой язык, как C++ современного стандарта, включающий превосходную поддержку как объектно-ориентированного, так и функционального стиля программирования, становится очевидным выбором для написания реактивных программ. Главная идея, лежащая в основе реактивного программирования, состоит в том, что некоторые типы данных

могут представлять значения, протяжённые во времени. Такие типы данных (конкретно, последовательности) в этой парадигме программирования играют роль наблюдаемых источников (observable). Результаты вычислений, зависящих от изменяющихся со временем значений, в свою очередь, представляют собой значения, изменяющиеся со временем. Изменяемое значение должно получать асинхронные оповещения всякий раз, когда изменяется другое значение, от которого оно зависит.

Несмотря на то что главный предмет этой книги составляет реактивное программирование, в этой главе мы будем заниматься в основном объектно-ориентированным подходом. Это необходимо для того, чтобы определить ряд ключевых интерфейсов (для чего в языке C++ используется аппарат виртуальных функций), которые в дальнейшем понадобятся для собственно реактивного программирования. Позднее, когда будут изучены конструкции языка C++ для поддержки функционального программирования, читатель сможет самостоятельно построить отображение объектно-ориентированных конструкций на функциональные. Кроме того, в этой главе мы намеренно устраняемся от обсуждения вопросов параллельной обработки, чтобы сконцентрировать внимание на программных интерфейсах. В главах 2 «Современный язык C++ и его ключевые идиомы», 3 «Параллельные вычисления и потоки в языке C++» и 4 «Асинхронное программирование и неблокирующая синхронизация в языке C++» разъясняются основы реактивного программирования с использованием средств функционального программирования.

Ключевые интерфейсы реактивной программы

Чтобы прояснить, что на самом деле происходит внутри реактивной программы, создадим несколько простых программ, демонстрирующих основные моменты. С точки зрения проектирования, если отвлечься от вопросов параллельной обработки и сфокусировать внимание лишь на программных интерфейсах, реактивная программа должна состоять из следующих частей:

- источник событий, реализующий интерфейс `IObservable<T>`;
- приёмник событий (также называемый наблюдателем или подписчиком), реализующий интерфейс `IObserver<T>`;
- механизм подписки наблюдателя на события от некоторого подписчика;
- механизм оповещения подписчиков о данных, поступающих из источника.

В этой главе, и только в ней, примеры написаны с использованием классической версии языка C++, так как необходимые элементы новейшего стандарта языка будут описаны лишь в последующих главах. В частности, в приведённых ниже примерах используются «сырые» указатели, чего практически всегда можно избежать, если создавать код на новой версии языка C++. Код примеров в этой главе написан так, чтобы в целом соответствовать требованиям, описанным в документации к системе ReactiveX. При создании кода на языке C++ мы не будем пользоваться подходами на основе наследования, которые обычно применяются при программировании на языках Java и C#.

Для начала определим интерфейсы для наблюдателя (IObserver) и наблюдаемого источника (IObservable), а также класс исключения CustomException.

```
#pragma once
//Common2.h

struct CustomException /*: public std::exception */ {
    const char * what() const throw () {
        return "C++ Exception";
    }
};
```

Класс CustomException – лишь заглушка, нужная для полноты интерфейса. Поскольку мы договорились придерживаться в этой главе стандартных средств языка C++, сохраним совместимость с классом std::exception.

```
template<class T> class  IEnumerator
{
public:
    virtual bool HasMore() = 0;
    virtual T next() = 0;
};

template <class T> class IEnumerable
{
public:
    virtual IEnumerator<T> *GetEnumerator() = 0;
};
```

Интерфейс IEnumerable составляет основу для источников данных; он предоставляет клиенту возможность перебирать свои элементы посредством интерфейса IEnuerator<T>.

Обращаем внимание читателя, что принцип, положенный в основу пары интерфейсов «перечисляемое-перечислитель» (IEnumerable<T> и IEnuerator<T>), зеркально отражает идею, на которой основана пара интерфейсов «наблюдаемое-наблюдатель» (IObserver<T> и IObservable<T>). Эту последнюю пару определим следующим образом:

```
template<class T> class  IObserver
{
public:
    virtual void OnCompleted() = 0;
    virtual void OnError(CustomException *exception) = 0;
    virtual void OnNext(T value) = 0;
};

template<typename T>
class  IObservable
{
public:
    virtual bool Subscribe(IObserver<T>& observer) = 0;
};
```

Интерфейс IObserver<T> – это интерфейс для приёмника данных, через который он получает оповещения от источника данных. Источник же, в свою очередь, должен реализовывать интерфейс IObservable<T>.

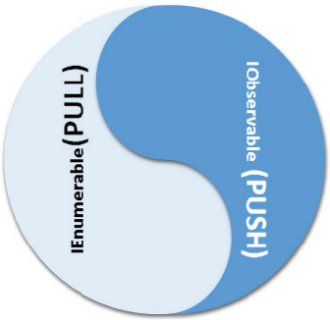
Мы определили интерфейс наблюдателя IObserver<T> с тремя методами. Это методы OnNext, с помощью которого наблюдатель получает оповещение об очередном элементе данных, OnCompleted, которым источник извещает, что у него более нет данных, и метод OnError, которым источник извещает об исключении. Интерфейс IObservable<T> воплощается источником данных, а приёмники регистрируют в нём свои объекты, обладающие интерфейсом IObserver<T>, чтобы получать оповещения.

Методы вталкивания и втягивания данных

В реактивном программировании выделяются два подхода: на основе вталкивания и на основе втягивания. В системе, основанной на принципе втягивания, источник данных ждёт запроса от приёмника (в нашей терминологии – подписчика), чтобы отправить ему очередные элементы потока данных. Таков классический подход, при котором источник данных играет пассивную роль, а приёмник активно запрашивает у него информацию. Для этого удобно применять шаблон итератора; интерфейсы IEnumerable<T> и IEnumerator<T> предназначены именно для этой модели, синхронной по своей природе (поток-приёмник данных блокируется, пока источник выполняет его запрос). В системе, основанной на вталкивании, напротив, источник данных рассылает события по сети приёмников, инициируя их обработку. В этом случае, в отличие от систем с втягиванием, новые элементы данных передаются подписчикам самим источником, который тем самым моделирует последовательность элементов. Асинхронная природа этой модели обеспечивается тем, что подписчики не блокируются в ожидании данных, а вместо этого реагируют на поступившие изменения. Легко видеть, что использование этого подхода предпочтительнее при создании сложных интерфейсов пользователя, в которых нежелательно блокировать главный поток графического интерфейса в ожидании событий. Вталкивание даёт отличный механизм для обеспечения отзывчивости приложения.

Дуальность интерфейсов IEnumerable и IObservable

Если присмотреться внимательно, можно обнаружить, что различие между двумя описанными выше подходами не столь значительно. Интерфейс IEnumerable<T> можно считать втягивающим эквивалентом вталкивающего интерфейса IObservable<T>. Эти интерфейсы фактически дуальны. Когда две сущности обмениваются информацией, втягивание данных со стороны первой из них соответствует вталкиванию со стороны второй. Эта дуальность иллюстрируется следующей диаграммой:



Чтобы лучше понять дуализм втягивания и вталкивания, рассмотрим пример кода.

В этой главе при написании кода мы придерживаемся классического языка C++, так как новые средства языка и стандартной библиотеки для поддержки потоков, неблокирующего программирования и другие темы, существенные для реализации реактивной модели программирования на современном языке C++, обсуждаются в других главах.

```
#include <iostream>
#include <vector>
#include <iterator>
#include <memory>
#include "Common2.h"
using namespace std;

class ConcreteEnumberable : public IEnumerable<int>
{
    int  *numberlist;
    int _count;
    friend class Enumerator;
public:
    ConcreteEnumberable(int numbers[], int count):
        numberlist(numbers), _count(count) {}
    ~ConcreteEnumberable() {}
    class Enumerator : public IEnumerator<int> {
        int *inumbers, icount, index;
    public:
        Enumerator(int *numbers, int count):
            inumbers(numbers), icount(count), index(0) {}
        bool HasMore() { return index < icount; }
        // строго говоря, следующий метод должен выбрасывать
        // исключение при выходе индекса за допустимые границы;
        // пускай он пока возвращает -1
        int next() { return (index < icount) ?
            inumbers[index++] : -1; }
        ~Enumerator() {}
    };
```

```
    IEnumerator<int> *GetEnumerator()
        { return new Enumerator(numberlist, _count); }
};
```

Показанный выше класс хранит указатель на массив целых чисел вместе с его размером и позволяет перебирать один за другим его элементы, воплощая интерфейс `IEnumerable<T>`. Логика перебора элементов реализована вложенным классом, который воплощает интерфейс `IEnumerator<T>`.

```
int main()
{
    int x[] = { 1,2,3,4,5 };
    // здесь используются "сырые" указатели, так как умные
    // указатели unique_ptr и shared_ptr излагаются позднее
    ConcreteEnumberable *t = new ConcreteEnumberable(x, 5);
    IEnumerator<int>  * numbers = t->GetEnumerator();
    while (numbers->HasMore())
        cout << numbers->next() << endl;
    delete numbers;
    delete t;
    return 0;
}
```

В главной функции программы создаётся экземпляр класса `ConcreteEnumberable`, играющий роль обёртки над обычным массивом, затем происходит перебор элементов через посредство этой обёртки.

Теперь напишем генератор чётных чисел, чтобы показать, как видоизменяется взаимодействие типов данных при преобразовании втягивающей программы во вталкивающую. При этом надёжность программы принесём в жертву краткости листинга.

```
#include <iostream>
#include <vector>
#include <iterator>
#include <memory>
#include "Common2.h"
using namespace std;

class EvenNumberObservable : IObservable<int>
{
    int *_numbers;
    int _count;
public:
    EvenNumberObservable(int numbers[], int count):
        _numbers(numbers),_count(count){}
    bool Subscribe(IObserver<int>& observer) {
         for (int i = 0; i < _count; ++i)
             if (_numbers[i] % 2 == 0)
                observer.OnNext(_numbers[i]);
        observer.OnCompleted();
        return true;
    }
};
```

Этот класс принимает массив целых чисел, отбрасывает нечётные значения, а о каждом найденном в массиве чётном значении информирует наблюдателя, т. е. объект, воплощающий интерфейс IObservable<T>. Иными словами, источник данных вталкивает свои данные наблюдателю. Реализация наблюдателя показана ниже:

```
class SimpleObserver :  public IObserver<int>
{
public:
     void OnNext(int value) { cout << value << endl; }
     void OnCompleted() { cout << "hello completed" << endl; }
     void OnError( CustomException * ex) {}
};
```

Тем самым класс SimpleObserver реализует интерфейс IObserver<int> и, следовательно, способен получать оповещения и реагировать на них.

```
int main()
{
    int x[] = { 1,2,3,4,5 };

    EvenNumberObservable * t = new EvenNumberObservable(x, 5);
    IObserver<int> *xy = new SimpleObserver();

    t->Subscribe(*xy);

    delete xy;
    delete t;

    return 0;
}
```

Из этого примера видно, как это просто – подписаться на получение от наблюдаемого источника одних лишь чётных чисел, имея последовательность натуральных чисел. Построенная нами система автоматически проталкивает (иначе говоря, публикует) сообщение для наблюдателя всякий раз, когда обнаруживает в исходных данных чётное число. В этом коде представлены такие образцы реализации ключевых интерфейсов, чтобы из них стало очевидно, что на самом деле происходит «под капотом» системы.

Превращение событий в наблюдаемый источник

Выше мы разобрали, как преобразовать втягивающую программу, основанную на интерфейсе IEnumerable<T>, в программу вталкивающую, построенную на паре интерфейсов IObservable<T>-IObserver<T>. В реальных приложениях, однако, источник данных не столь прост, как показанный выше поток целых чисел. Рассмотрим, как преобразовать события MouseMove в поток, для чего создадим небольшую программу на основе библиотеки MFC.

Библиотека MFC выбрана для этого примера потому, что реактивному программированию на основе более современной библиотеки Qt посвящена отдельная глава. В ней мы рассмотрим создание реактивных программ на основе идиоматичных асинхронных потоков, работающих по принципу вталкивания. В этой же программе, основанной на библиотеке MFC, мы всего лишь будем фильтровать события перемещения мыши, выделяя из них те, что попадают в заданный прямоугольник, и оповещая о таких событиях наблюдателя. В этом примере события обрабатываются синхронно.

```
#include "stdafx.h"
#include <afxwin.h>
#include <afxext.h>
#include <math.h>
#include <vector>
#include "../Common2.h"

using namespace std;

class CMouseFrame :public CFrameWnd, IObservable<CPoint>
{
private:
    RECT _rect;
    POINT _curr_pos;
    vector<IObserver<CPoint> *> _event_src;
public:
    CMouseFrame()
    {
        HBRUSH brush = (HBRUSH)::CreateSolidBrush(
            RGB(175, 238, 238));
        CString mywindow = AfxRegisterWndClass(
            CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS, 0, brush, 0);
        Create(mywindow, _T("MFC Clock By Praseed Pai"));
    }
```

В данной части кода объявлен класс для окна приложения. Этот класс порождён от класса `CFrameWnd` из библиотеки MFC и, помимо того, воплощает интерфейс `IObservable<T>`, тем самым вынуждая программиста реализовать в этом классе метод `Subscribe`. Вектор указателей на объекты, обладающие интерфейсом `IObserver<T>`, используется для хранения всех подписчиков (иначе говоря, наблюдателей), подключенных к этому источнику событий. Хотя в этом примере будет только один наблюдатель, код допускает наличие любого числа наблюдателей.

```
virtual bool Subscribe(IObserver<CPoint>& observer) {
    _event_src.push_back(&observer);
    return true;
}
```

Метод `Subscribe` просто сохраняет указатель на наблюдателя в векторе и возвращает значение `true`, означающее успех. Всякий раз, когда мышь изменит своё положение, окно получит оповещение через внутренние механизмы библиотеки MFC, затем, если координаты мыши попадают в заданную прямо-

угольную область, наблюдатели будут об этом извещены. За это отвечает представленный ниже код.

```
bool FireEvent(const CPoint& pt) {
    vector<IObserver<CPoint> *>::iterator it  =
         _event_src.begin();
    while (it != _event_src.end()){
        IObserver<CPoint> *observer = *it;
        observer->OnNext(pt);
        // в настоящих реактивных программах установлен
        // строгий порядок вызова методов: метод OnCompleted
        // должен вызываться только после того, как все
        // данные обработаны; этот код лишь демонстрирует
        // общую идею.
        observer->OnCompleted();
        it++;
    }
    return true;
}
```

Метод `FireEvent` проходит по всем наблюдателям и в каждом из них вызывает метод `OnNext`, также вызывает он и метод `OnCompleted`. Механизмы обработки событий в реактивном программировании предполагают ряд правил вызова методов у наблюдателей. В частности, если вызван метод `OnCompleted`, то метод `OnNext` для данного наблюдателя более вызываться не должен. Подобным же образом, если у наблюдателя вызван метод `OnError`, дальнейшие сообщения ему не отправляются. Если бы мы стремились в точности соблюдать все соглашения, принятые в модели реактивного программирования, текст программы вышел бы слишком сложным. Предназначение показанного здесь кода состоит в том, чтобы схематически показать реактивную модель программирования в действии.

```
int OnCreate(LPCREATESTRUCT l){
    return CFrameWnd::OnCreate(l);
}

void SetCurrentPoint(CPoint pt) {
    this->_curr_pos = pt;
    Invalidate(0);
}
```

Метод `SetCurrentPoint` вызывается наблюдателем и устанавливает координаты для последующего отображения текста. Вызов метода `Invalidate` приводит к генерации сообщения `WM_PAINT`, по которому механизмы библиотеки MFC, согласно таблице обработчиков сообщений, вызовут метод `OnPaint`.

```
void OnPaint()
{
    CPaintDC d(this);
    CBrush b(RGB(100, 149, 237));
    int x1 = -200, y1 = -220, x2 = 210, y2 = 200;
```

```
    Transform(&x1, &y1); Transform(&x2, &y2);
    CRect rect(x1, y1, x2, y2);
    d.FillRect(&rect, &b);
    CPen p2(PS_SOLID, 2, RGB(153, 0, 0));
    d.SelectObject(&p2);

    char *str = "Hello Reactive C++";
    CFont f;
    f.CreatePointFont(240, _T("Times New Roman"));
    d.SelectObject(&f);
    d.SetTextColor(RGB(204, 0, 0));
    d.SetBkMode(TRANSPARENT);
    CRgn crgn;
    crgn.CreateRectRgn(
        rect.left, rect.top,
        rect.right, rect.bottom);
    d.SelectClipRgn(&crgn);
    d.TextOut(
        _curr_pos.x, _curr_pos.y,
        CString(str), strlen(str));
}
```

Метод `OnPaint` вызывается самой библиотекой MFC в ответ на вызов метода `Invalidate`. Этот метод отображает строку текста в том месте, которое ранее было установлено методом `SetCurrentPoint`.

```
void Transform(int *px, int *py)
{
    ::GetClientRect(m_hWnd, &_rect);
    int width = (_rect.right - _rect.left) / 2;
    int height = (_rect.bottom - _rect.top) / 2;
    *px = *px + width;
    *py = height - *py;
}
```

Метод `Transform` вычисляет границы внутренней области окна и переводит абстрактные декартовы координаты точки в координаты относительно данного окна. Это вычисление можно было бы ещё лучше выполнить с помощью преобразователей координат, поддерживаемых системой Windows.

```
void OnMouseMove(UINT nFlags, CPoint point)
{
    int x1 = -200,y1= -220, x2 = 210,y2 = 200;
    Transform(&x1, &y1);Transform(&x2, &y2);
    CRect rect(x1, y1, x2, y2);
    POINT pts;
    pts.x = point.x;
    pts.y = point.y;
    rect.NormalizeRect();
    // В реальной программе эти точки накапливались бы
    // в списке
    if (rect.PtInRect(point)) {
```

```
            // лучше отправлять оповещения, не блокируя поток
            FireEvent(point);
        }
    }
```

Метод OnMouseMove проверяет, находится ли текущее положение мыши в прямоугольнике, центрированном относительно экрана, и если это так, извещает об этом наблюдателей.

```
    DECLARE_MESSAGE_MAP();
};

BEGIN_MESSAGE_MAP(CMouseFrame, CFrameWnd)
    ON_WM_CREATE()
    ON_WM_PAINT()
    ON_WM_MOUSEMOVE()
END_MESSAGE_MAP()

class WindowHandler : public IObserver<CPoint>
{
private:
    CMouseFrame *window;
public:
    WindowHandler(CMouseFrame *win) : window(win) { }
    virtual ~WindowHandler() { window = 0; }
    virtual void OnCompleted() {}
    virtual void OnError(CustomException *exception) {}
    virtual void OnNext(CPoint value) {
        if (window) window->SetCurrentPoint(value);
    }
};
```

Показанный выше класс WindowHandler реализует интерфейс IObserver<CPoint> и обрабатывает события, полученные от объекта CMouseFrame, который воплощает интерфейс IObservable<CPoint>. В этом небольшом примере наблюдатель передаёт окну координаты для отображения текста.

```
class CMouseApp :public CWinApp
{
    WindowHandler *reactive_handler;
public:
    int InitInstance()
    {
        CMouseFrame *p = new CMouseFrame();
        p->ShowWindow(1);
        reactive_handler = new WindowHandler(p);
        p->Subscribe(*reactive_handler);
        m_pMainWnd = p;
        return 1;
    }

    virtual ~CMouseApp() {
```

```
        if (reactive_handler) {
            delete reactive_handler;
            reactive_handler = 0;
        }
    }

};

CMouseApp a;
```

Последний фрагмент кода отвечает за инициализацию и запуск приложения. Сначала создаётся и отображается главное окно, затем создаётся объект-наблюдатель, далее наблюдатель подключается к источнику данных. Деструктор отвечает за удаление объекта-наблюдателя. В последней строке создаётся объект, инкапсулирующий приложение в целом, что приводит к запуску всей системы.

Методологические замечания

Цель данной главы состоит в том, чтобы познакомить читателя с ключевыми интерфейсами, лежащими в основе реактивного подхода к программированию, а именно с интерфейсами `IObservable<T>` и `IObserver<T>`. Эти два интерфейса, по существу, дуальны интерфейсам `IEnumerable<T>` и `IEnumerator<T>`. Мы разобрали, как смоделировать эти пары интерфейсов на классической версии языка C++, и написали их упрощённые реализации. Наконец, мы создали программу с графическим интерфейсом, которая перехватывает события от мыши и извещает о них множество наблюдателей. Эти игрушечные реализации позволили немного прикоснуться к идеям и принципам реактивной модели программирования. Показанные здесь реализации можно считать примером объектно-ориентированного реактивного программирования.

Чтобы стать профессионалом в реактивном программировании на языке C++, программисту нужно уверенно овладеть следующими темами:

- новые языковые конструкции в новом стандарте языка C++;
- средства поддержки функционального программирования, поддерживаемые в новом стандарте языка C++;
- модель асинхронного программирования (библиотека RxCpp берёт эту заботу на себя!);
- обработка потоков событий;
- мощные, пригодные для реальных задач библиотеки для реактивного программирования, такие как библиотека RxCpp;
- применение парадигмы реактивного программирования для разработки графических интерфейсов и веб-программирования;
- усложнённые конструкции реактивного программирования;
- обработка ошибок и исключений.

В этой главе речь шла главным образом о ключевых идиомах и о том, зачем вообще нужна стройная модель асинхронной обработки данных. В следующих трёх главах будут разобраны новые средства, появившиеся в языке C++, в особенности средства многопоточного и параллельного программирования, а также средства неблокирующей синхронизации, ставшие возможными благодаря появлению моделей памяти с гарантиями. Всё это заложит прочный фундамент для овладения функциональным реактивным программированием.

В главе 5 «Знакомство с наблюдаемыми источниками» мы вернёмся к теме наблюдателей и воплотим знакомые интерфейсы в духе функционального программирования, по-новому осветив некоторые понятия. В главе 6 «Введение в программирование потоков событий на языке C++» мы займёмся более сложными вопросами обработки потоков событий с помощью двух промышленных библиотек, использующих подход на основе встроенных предметно-ориентированных языков (англ. Domain-Specific Embedded Language, DSEL).

Тем самым будет подготовлена основа для знакомства с промышленной библиотекой RxCpp и её тонкостями, позволяющими создавать программные продукты современного уровня и профессионального качества. Этой замечательной библиотеке посвящены глава 7 «Введение в модель потоков данных и библиотеку RxCpp» и глава 8 «Ключевые элементы библиотеки RxCpp». В последующих главах рассматривается реактивное программирование графических интерфейсов с использованием библиотеки Qt и некоторых изощрённых операций библиотеки RxCpp.

Последние три главы посвящены вопросам повышенной сложности: шаблонам проектирования реактивных программ, созданию микросервисов на языке C++, обработке ошибок и исключений. Читатель, приступивший к чтению книги, владея лишь классической версией языка C++, к концу книги приобретёт значительный опыт не только в создании реактивных программ, но и в использовании современной версии языка. В силу самой темы этой книги разобрана будет большая часть нововведений из стандарта C++ 17 (доступных на момент написания книги).

Итоги

Из этой главы читатель узнал о некоторых структурах данных, ключевых для реактивной модели программирования. Мы построили их упрощённые реализации, что позволило проиллюстрировать лежащие в их основе понятия. Для начала мы разобрали, как обрабатывать события от графического интерфейса пользователя с помощью функций API системы Windows, с помощью функций библиотеки XLib, а также с использованием библиотек MFC и Qt. Кроме того, мы вкратце рассмотрели, как обрабатываются события в технологиях COM+ и CORBA. Затем был представлен краткий обзор реактивной модели програм-

мирования в целом. Мы определили несколько интерфейсов и сделали набросок их реализации. Наконец, для полноты изложения было показано, как встроить реализации этих интерфейсов в программу с графическим интерфейсом, основанную на библиотеке MFC. Также были изложены основные методологические аспекты данной книги.

В следующей главе будет сделан беглый обзор основных новшеств современного языка C++ (под этим словом будем понимать стандарты C++ 11, 14 и 17) с упором на семантику перемещения, лямбда-выражения, вывод типов, циклы по диапазонам, сочленение функций и умные указатели. Всё это нужно для написания даже простейшего кода в реактивной парадигме.