УДК 004.4
ББК 32.973.202
П49

Антон Полухин

П49 Разработка приложений на C++ с использованием Boost. Рецепты, упрощающие разработку вашего приложения / пер. с анг. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 346 с.: ил.

ISBN 978-5-97060-868-5

Это руководство знакомит читателя с библиотеками Boost, которые помогают разрабатывать качественные, быстрые и портативные приложения. Удобная структура книги, включающая ряд стандартных разделов, упрощает изучение материала. От простых тем (повседневное использование библиотек, управление ресурсами) автор последовательно переходит к сложным (метапрограммирование, многопоточность, межпроцессное взаимодействие, асинхронное взаимодействие, работа с большими библиотеками Boost).

Издание предназначено для разработчиков, желающих улучшить свои знания в области Boost и упростить процессы разработки приложений. Для освоения изложенных в книге приемов необходимы знакомство с C++ и базовые знания стандартной библиотеки. Также понадобятся современный компилятор C++, библиотеки Boost (подойдет любая версия, но рекомендуется 1.65 или более новая), среда разработки QtCreator, утилита qmake. Есть возможность модифицировать и запускать примеры онлайн: http://apolukhin.github.io/Boost-Cookbook/.

УДК 004.4
ББК 32.973.202

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 978-1-78728-224-7
ISBN (рус.) 978-5-97060-868-5

© 2017 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2020

Оглавление

Об авторе

Если вам интересно, кто такой Антон Полухин и можно ли доверять ему в вопросах обучения C++ и библиотекам Boost, то вот несколько фактов:

- Антон Полухин в настоящее время представляет Россию в международном комитете по стандартизации C++;
- он является автором нескольких библиотек Boost и поддерживает ряд старых библиотек Boost;
- он перфекционист: все исходные коды из книги проходят автоматическое тестирование на нескольких платформах с использованием различных стандартов C++.

Но давайте начнем с самого начала.

Антон Полухин родился в России. В детстве он мог говорить на русском и венгерском языках и изучал английский в школе. Со школьных лет участвовал в различных соревнованиях по математике, физике и химии и побеждал в них.

Дважды был принят в университет: один раз за участие в городской олимпиаде по математике и второй раз за то, что получил высокий балл по вступительным олимпиадам в вуз. В его университетской жизни был год, когда он вообще не участвовал в экзаменах: он получил «зачет автоматом» во всех дисциплинах, написав программы повышенной сложности по каждому предмету. Свою будущую жену он встретил в университете, который закончил с отличием.

Более трех лет работал в VoIP-компании, разрабатывая бизнес-логику для коммерческой альтернативы Asterisc. В то время он начал вносить свой вклад в Boost и стал сопровождающим библиотеки `Boost.LexicalCast`. Он также начал делать переводы на русский язык для Ubuntu Linux в то время.

Сегодня работает в компании Yandex.Taxi, помогает русскоговорящим людям с предложениями по стандартизации C++, продолжает вносить вклад в opensource-проекты и язык C++ в целом.

Его код можно найти в библиотеках Boost, таких как Any, Conversion, DLL, LexicalCast, Stacktrace, TypeTraits, Variant и др.

Счастлив в браке уже более семи лет.

Я хотел бы поблагодарить свою семью, особенно мою жену Ирину Полухину, за то, что она рисовала эскизы рисунков и диаграмм для этой книги.

Огромное спасибо Полу Энтони Бристоу за обзор первого издания данной книги и за то, что он прошел через безумное количество запятых, которые я использовал в первых черновиках.

Отдельное спасибо Глену Джозефу Фернандесу за то, что он предоставил много полезной информации и комментариев по второму изданию.

Что касается русского издания книги – неоценимую помощь оказал Кирилл Марков. За что ему отдельное спасибо!

Я также хотел бы поблагодарить всех членов сообщества Boost за написание этих замечательных библиотек и за то, что они открыли для меня удивительный мир C++.

О рецензентах

Глен Джозеф Фернандес работал инженером-программистом в компаниях Intel и Microsoft. Он является автором библиотеки *Boost.Align*, основным участником поддержки библиотек *Boost.SmartPointers* и *Boost Core*, а также внес вклад в ряд других библиотек Boost. Участвует в поддержке стандарта C++, создавая документы по предложениям и отчеты о дефектах, и у него даже есть по крайней мере одна функция, принятая для будущего стандарта C++20 (P0674r1: расширение make_shared для поддержки массивов). Глен живет со своей женой Кэролайн и дочерью Айрин в США. Он окончил Университет Сиднея в Австралии, до этого жил в Новой Зеландии.

Марков Кирилл увлёкся программированием ещё в школе. Начал заниматься коммерческой разработкой ПО с ранних курсов университета. С тех пор освоил множество платформ, технологий и языков программирования, является full stack разработчиком, но предпочитает backend разработку. На данный момент живёт и трудится в Москве ведущим программистом в одной из крупнейших компаний. Проповедует педантичный формализованный подход к процессам разработки. Неисправимый любитель чая и интересной беседы.

Вступительное слово автора

Более 10 лет назад, когда я только начал осваивать C++, с хорошей литературой было очень тяжко. В итоге навыки C++ приходилось оттачивать, изучая исходные коды библиотеки Boost. Дело двигалось очень медленно, все было непонятно, документация и комментарии на английском не сильно помогали. С тех пор прошло уже много лет, библиотеки Boost отчасти стали стандартом C++ и продолжают развиваться, опережая по своим возможностям стандартную библиотеку C++ на десятки лет. Функционал внутри Boost огромен… и все так же непонятен для начинающих.
Эта книга содержит ответы на типичные вопросы:

- Как мне решить вот эту проблему?
- Как это работает?
- Как это устроено под капотом?
- Как бы поэкспериментировать, не заморачиваясь с настройкой окружения?
- А разве подобного нет в стандартной библиотеке?
- А какие есть хитрости при работе с этим инструментом?
- А есть ли способ решить это получше?
- А что еще почитать на эту тему?

Другими словами, эта книга – тот помощник, которого мне не хватало в свое время и которого не хватает многим разработчикам поныне.

Надеюсь, вам понравится!

Антон Полухин,
представитель России в Международном комитете по стандартизации C++,
разработчик и автор многих библиотек Boost,
руководитель группы Общих Компонент в Яндекс.Такси,
сопредседатель РГ21 C++ и модератор https://stdcpp.ru ,
спикер на конференциях РГ21, Corehard, C++ Russia,
корпоративный консультант по вопросам C++ https://apolukhin.github.io/ ,
автор этой книги :)

Вступительное слово от сообщества C++

История собрания библиотек Boost насчитывает почти столько же лет, сколько и стандарт языка C++. Это означает, что сообщество программистов с первых лет остро ощущало нехватку в стандарте необходимых инструментов и брало инициативу в свои руки. Ничего удивительного, что уже через несколько лет Boost прочно занял место неофициального стандарта и заодно испытательного полигона для смелых идей – кандидатов на включение в будущие версии стандарта.

Кроме того, Boost можно считать своеобразным эталоном качества библиотечного кода. Реализации структур данных и алгоритмов имеют под собой солидное математическое обоснование (скажем, где были бы контейнеры Boost.MultiIndex без строгого понятия точной нижней или верхней грани, а алгоритмы – без строгой меры временной сложности, знаменитого O-большого). С другой стороны, основываются на филигранной технике использования всех средств языка (чего стоит одно метапрограммирование на шаблонах). И все это умножается на необходимость поддерживать различные платформы, разные стандарты языка C++ и требование сохранять концептуальную согласованность между разными библиотеками. Круг задач, для которых в Boost можно найти готовое решение, поражает воображение.

Однако широчайший охват задач вместе с высочайшим качеством достается не бесплатно. Чтобы отыскать в Boost наиболее подходящий инструмент и гибко настроить на свою конкретную задачу, от программиста требуется определенный уровень профессиональной культуры и широта кругозора. Поэтому если в отсутствие Boost программист-профессионал страдает от нехватки необходимых средств, то при наличии Boost новичок может страдать от необъятной груды функций и классов без надежды в ней разобраться. Если первой эмоцией от знакомства с собранием библиотек Boost обычно бывает восхищение от ее мощи, стройности и продуманности всех мелочей, то второй – испуг: как все это постичь и не запутаться и где добыть книгу-путеводитель от простого к сложному. Наконец, когда благодаря Антону Полухину такая книга появилась, остается чистое восхищение.

Boost, равно как и STL, в силу необходимости быть кросс-платформенным, поддерживать множество опций компилятора и огромное количество взаимосвязей внутри самой библиотеки, выглядит весьма громоздким и сложным для понимания. В своей книге Антон Полухин рассматривает не только применение элементов библиотеки, но и указывает на особенности их реализации, что невозможно переоценить: понимание подкапотных процессов позволяет осознанно использовать элементы библиотеки с точки зрения корректности и производительности кода, а при необходимости и заменить их на аналогичные собственного производства меньшей кровью. Из вышеперечисленного естественным образом следует и просветительская функция данной книги, поскольку внимательный читатель может почерпнуть из нее как новые идеи, так и освежить особенности различных стандартов языка.

C++ сообщество, CoreHard 2020

Предисловие

Если вы хотите воспользоваться преимуществами Boost и C++ и не путаться, какую библиотеку в какой ситуации использовать, тогда эта книга для вас. Начиная с основ, вы перейдете к изучению того, как библиотеки Boost упрощают разработку приложений. Вы научитесь преобразовывать данные: строки в числа, числа в строки, числа в числа и многое другое. Управление ресурсами станет проще некуда. Вы увидите, какую работу можно выполнить во время компиляции и на что способны контейнеры Boost. Вы узнаете все, что нужно, для разработки качественных, быстрых и портативных приложений. Напишите программу один раз, и вы сможете использовать ее в операционных системах Linux, Windows, macOS и Android. От манипулирования изображениями до графов, каталогов, таймеров, файлов и работы в сети – каждый найдет для себя интересную тему.

Обратите внимание, что знания, полученные в ходе прочтения этой книги, не устареют, поскольку все больше и больше библиотек Boost становятся частью стандарта C++.

О чем эта книга

Глава 1 «*Приступаем к написанию приложения*» рассказывает о библиотеках для повседневного использования. Мы увидим, как получить параметры конфигурации из разных источников и как упростить себе жизнь, используя некоторые из типов данных, представленных авторами библиотеки Boost.

Глава 2 «*Управление ресурсами*» посвящена типам данных, представленных библиотеками Boost, по большей части фокусируясь на работе с указателями. Мы увидим, как с легкостью управлять ресурсами и использовать тип данных, способный хранить любые функциональные объекты, функции и лямбда-выражения. После прочтения этой главы ваш код станет более надежным, а утечки памяти уйдут в прошлое.

Глава 3 «*Преобразование и приведение*» описывает, как преобразовывать строки, числа и пользовательские типы друг в друга, как безопасно приводить полиморфные типы и как писать маленькие и большие парсеры прямо в исходных файлах C++. Рассматривается несколько способов преобразования данных как для повседневного использования, так и для редких случаев.

Глава 4 «*Уловки времени компиляции*» описывает базовые приемы библиотек Boost, которые можно использовать при проверках во время компиляции, для настройки алгоритмов и в других задачах метапрограммирования. Понимание исходных файлов Boost и других схожих с Boost библиотек без этого невозможно.

Глава 5 «*Многопоточность*» посвящена основам многопоточного программирования и синхронизации доступа к данным.

Глава 6 «*Манипулирование задачами*» показывает, как работать с функциональными объектами – задачами. Основная идея этой главы заключается

в том, что мы можем разделить всю обработку, вычисления и взаимодействия на функторы (задачи) и обрабатывать каждую из этих задач практически независимо. Более того, мы можем не блокировать поток выполнения на некоторых медленных операциях (таких как получение данных из сокета или ожидание тайм-аута), а вместо этого предоставить задачу обратного вызова (callback) и продолжить работу с другими задачами. Как только ОС закончит медленную операцию, будет выполнен наш обратный вызов.

Глава 7 «*Манипулирование строками*» показывает различные аспекты изменения, поиска и представления строк. Мы увидим, как можно с легкостью выполнять некоторые распространенные задачи, связанные со строками, с помощью библиотек Boost.

Глава 8 «*Метапрограммирование*» раскрывает классные и трудные для понимания методы программирования на этапе компиляции. В этой главе мы посмотрим, как можно упаковать несколько типов в один тип, подобный кортежу. Мы создадим функции для управления коллекциями типов, посмотрим, как можно изменять типы коллекций во время компиляции и как трюки во время компиляции можно смешивать с вычислениями времени выполнения (runtime).

Глава 9 «*Контейнеры*» рассказывает о boost-контейнерах и вещах, непосредственно связанных с ними. В этой главе содержится информация о классах Boost, которые можно использовать в повседневном программировании, что сделает ваш код намного быстрее и облегчит разработку новых приложений.

Глава 10 «*Сбор информации о платформе и компиляторе*» описывает различные вспомогательные макросы, используемые для обнаружения возможностей компилятора, платформы и функциональности Boost. Вы познакомитесь с макросами, которые широко используются в библиотеках Boost и которые необходимы для написания переносимого кода, способного работать с любыми флагами компилятора.

Глава 11 «*Работа с системой*» подробно рассматривает файловую систему и способы создания и удаления файлов. Мы увидим, как данные могут передаваться между различными системными процессами, как читать файлы с максимальной скоростью и как решать другие системные задачи.

Глава 12 «В*ерхушка айсберга*» посвящена большим библиотекам Boost и знакомит вас с необходимыми для их использования основами.

Что нужно для этой книги

Вам нужен современный компилятор C++, библиотеки Boost (подойдет любая версия, но рекомендуется 1.65 или более новая) и среда разработки QtCreator и утилита qmake. Или просто перейдите по адресу http://apolukhin.GitHub.io/Boost-Cookbook, чтобы запускать примеры и экспериментировать с ними в режиме онлайн.

Для кого эта книга

Эта книга предназначена для разработчиков, желающих улучшить свои знания в области Boost и упростить процессы разработки приложений. Предполагается, что вы уже знакомы с C++ и имеете базовые знания стандартной библиотеки.

Разделы

В этой книге вы найдете несколько заголовков, которые часто появляются в тексте («Подготовка», «Как это делается…», «Как это работает…», «Дополнительно…» и «См. также»). Чтобы предоставить четкие инструкции относительно того, как завершить рецепт, мы будем использовать эти разделы следующим образом:

Подготовка

В этом разделе рассказывается, чего ожидать в рецепте, и описывается, как настроить какое-либо программное обеспечение или какие-либо предварительные параметры, необходимые для рецепта.

Как это делается…

Этот раздел содержит шаги, необходимые для того, чтобы следовать рецепту.

Как это работает…

Данный раздел обычно состоит из подробного объяснения того, что произошло в предыдущем разделе.

Дополнительно…

Этот раздел состоит из дополнительной информации о рецепте, чтобы читатель был более осведомлен.

См. также

Данный раздел содержит полезные ссылки на другую полезную информацию для рецепта.

Обозначения

В этой книге вы найдете ряд текстовых стилей, используемых для того, чтобы различать разные виды информации. Вот несколько примеров этих стилей и объяснение их значения.

Код в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-адреса, пользовательский ввод и учетные записи в Twitter отображаются следующим образом:

«Помните, что эта библиотека состоит не только из заголовочных файлов, ваша программа должна линковаться с `libboost_program_options`.».

Блок кода выглядит так:

```
#include <boost/program_options.hpp>
#include <iostream>
namespace opt = boost::program_options;
int main(int argc, char *argv[])
{
```

Когда мы хотим обратить ваше внимание на определенную часть блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
#include <boost/program_options.hpp>
#include <iostream>
namespace opt = boost::program_options;
int main(int argc, char *argv[])
```

Любой ввод или вывод командной строки записывается следующим образом:

```
  $ ./our_program.exe --apples=10 --oranges=20
Fruits count: 30
```

Новые термины и **важные слова** выделены жирным шрифтом.

Предупреждения или важные заметки сопровождаются таким знаком.

Советы и хитрости выглядят так.

Загрузка примеров кода

Файлы исходных кодов примеров, представленных в этой книге рецептов, есть в репозитории автора на сайте GitHub. Для получения последней версии кода вы можете зайти на страницу https://GitHub.com/apolukhin/Boost-Cookbook.

Глава 1

Приступаем к написанию приложения

Темы, которые мы рассмотрим в этой главе:

- получение параметров конфигурации;
- хранение любого значения в контейнере или переменной;
- хранение одного из нескольких выбранных типов в контейнере или переменной;
- использование более безопасного способа работы с контейнером, в котором хранится один из нескольких выбранных типов;
- возвращение значения или флага «значения нет»;
- возвращение массива из функции;
- объединение нескольких значений в одно;
- привязка и переупорядочение параметров функции;
- получение удобочитаемого имени типа;
- использование эмуляции перемещения C++11;
- создание некопируемого класса;
- создание некопируемого, но перемещаемого класса;
- использование алгоритмов C++14 и C++11.

Вступление

Boost – это коллекция библиотек для языка C++. Каждая из этих библиотек была проверена множеством профессиональных программистов, прежде чем была принята Boost. Библиотеки тестируются на многих платформах с использованием множества компиляторов и реализаций стандартной библиотеки C++. Используя Boost, вы можете быть уверены, что в ваших руках находится одно из самых портативных, быстрых и надежных решений, которое распространяется по лицензии, подходящей для коммерческих проектов и проектов с открытым исходным кодом.

Многие части Boost были включены в стандарты C++11, C++14, C++17 и C++20. Кроме того, некоторые библиотеки Boost попадут и в следующие стандарты C++. В каждом рецепте в этой книге вы найдете примечания, касающиеся стандарта C++.

Без долгого вступления, давайте начнем!

В этой главе мы увидим несколько рецептов для повседневного использования. Мы увидим, как получить параметры конфигурации из разных источников и что можно сделать, используя популярные типы данных, представленные авторами библиотек Boost.

Получение параметров конфигурации

Взгляните на консольные программы, такие как `cp` в Linux. Все они имеют красивую «справку», их входные параметры не зависят от какой-либо позиции и имеют читабельный синтаксис. Например:

```
$ cp --help
Usage: cp [OPTION]... [-T] SOURCE DEST
  -a, --archive           same as -dR --preserve=all
  -b                      like --backup but does not accept an argument
```

Вы можете реализовать ту же функциональность для своей программы за 10 минут. Все, что вам нужно, – это библиотека `Boost.ProgramOptions`.

Подготовка

Все, что требуется для этого рецепта, – базовые знания C++. Помните, что библиотека `Boost.ProgramOptions` состоит не только из заголовочных файлов, поэтому ваша программа должна линковаться с библиотекой `libboost_program_options`.

Как это делается...

Давайте начнем с простой программы, которая принимает количество яблок (`apples`) и апельсинов (`oranges`) в качестве входных данных и подсчитывает общее количество фруктов. Вот результат, который мы хотим получить:

```
$ ./our_program.exe --apples=10 --oranges=20
Fruits count: 30
```

Выполните следующие шаги.

1. Включите в код заголовочный файл `boost/program_options.hpp` и создайте псевдоним для пространства имен для `boost::program_options` (его слишком долго набирать на клавиатуре!). Нам также понадобится заголовочный файл `<iostream>`:

```
#include <boost/program_options.hpp>
#include <iostream>
namespace opt = boost::program_options;
```

2. Теперь мы готовы описать наши опции в функции `main()`:

```
int main(int argc, char *argv[])
{
  // Создаем опции, описывающие переменную, и даем ей текстовое описание
  // «All options».
  opt::options_description desc("All options");

  // Когда мы добавляем опции, первый параметр – это имя, которое будет использоваться
  // в командной строке. Второй параметр – это тип данных опции, заключенный в класс
  // value <>. Третий параметр должен быть кратким описанием этой опции.
```

```
    desc.add_options()
       ("apples", opt::value<int>(), "how many apples do you have")
       ("oranges", opt::value<int>(), "how many oranges do you have")
       ("help", "produce help message")
    ;
```

3. Давайте выполним парсинг командной строки:

```
    // Переменная для хранения аргументов нашей командной строки
    opt::variables_map vm;

    // Парсинг и сохранение аргументов
    opt::store(opt::parse_command_line(argc, argv, desc), vm);

    // Эта функция должна вызываться после парсинга и сохранения.
    opt::notify(vm);
```

4. Добавим немного кода для обработки опции `help`:

```
    if (vm.count("help")) {
        std::cout << desc << "\n";
        return 1;
    }
```

5. Заключительный этап. Подсчет фруктов можно реализовать следующим образом:

```
      std::cout << "Fruits count: "
           << vm["apples"].as<int>() + vm["oranges"].as<int>()
           << std::endl;
} // Конец функции `main`
```

Теперь если мы вызовем нашу программу, используя параметр `help`, то получим следующий вывод:

```
All options:
    --apples arg      how many apples do you have
    --oranges arg     how many oranges do you have
    --help            produce help message
```

Как видите, в коде мы не предоставляем тип данных для значения параметра `help`, потому что не ждем, что ему будут переданы какие-либо значения.

Как это работает...

Этот пример довольно просто понять, исходя из кода и комментариев. Его запуск дает ожидаемый результат:

```
  $ ./our_program.exe --apples=100 --oranges=20
Fruits count: 120
```

Дополнительно...

Стандарт C++ принял множество библиотек Boost; однако вы не найдете `Boost.ProgramOptions` даже в C++20. В настоящее время ее не планируют включать и в C++23.

Библиотека `ProgramOptions` очень мощная и имеет множество возможностей. Она может:

- записать значения параметров конфигурации непосредственно в переменную и сделать этот параметр обязательным:

```
int oranges_var = 0;
desc.add_options()
    // ProgramOptions сохраняет значение параметра в переменную, которая передается
    // по указателю. Здесь значение параметра «--oranges» будет сохранено в «oranges_var».
    ("oranges,o", opt::value<int>(&oranges_var)->required(),
                                          "oranges you have")
```

- получить необязательный строковый параметр :

```
    // Опция 'name' не помечена как 'required ()', поэтому пользователь может
    // не предоставлять ее.
    ("name", opt::value<std::string>(), "your name")
```

- добавить сокращенное обозначение и установить 10 в качестве значения по умолчанию для apples:

```
    // «a» - сокращенное обозначение для яблок. Используйте его как в «-a 10».
    // Если значение не указано, используется значение по умолчанию.
    ("apples,a", opt::value<int>()->default_value(10),
                            "apples that you have");
```

- получить недостающие параметры из файла конфигурации:

```
opt::variables_map vm;

// Парсинг параметров командной строки и сохранение значений в 'vm'.
opt::store(opt::parse_command_line(argc, argv, desc), vm);

// Мы также можем выполнить парсинг переменных окружения. Просто используйте функцию
// 'opt::store' с 'opt::parse_environment'.
// Добавляем недостающие параметры из конфигурационного файла "apples_oranges.cfg".
try {
    opt::store(
        opt::parse_config_file<char>("apples_oranges.cfg", desc),
        vm
    );
} catch (const opt::reading_file& e) {
    std::cout << "Error: " << e.what() << std::endl;
}
```

Синтаксис конфигурационного файла отличается от синтаксиса командной строки. Нам не нужно ставить знаки «минус» перед опциями. Поэтому наш файл apple_oranges.cfg должен выглядеть так:

```
oranges=20
```

- проверить, что были установлены все необходимые параметры:

```
    try {
        // Если один из обязательных параметров не был задан, выбрасывается исключение
        // `opt::required_option`
        opt::notify(vm);
    } catch (const opt::required_option& e) {
```

```
    std::cout << "Error: " << e.what() << std::endl;
    return 2;
}
```

Если мы объединим все вышеупомянутые советы в один исполняемый файл, то команда `help` выдаст такой вывод:

```
$ ./our_program.exe --help
 All options:
   -o [ --oranges ] arg          oranges that you have
   --name arg                    your name
   -a [ --apples ] arg (=10)     apples that you have
   --help                        produce help message
```

Если запустить его без конфигурационного файла, это приведет к следующему выводу:

```
$ ./our_program.exe
 Error: can not read options configuration file 'apples_oranges.cfg'
 Error: the option '--oranges' is required but missing
```

Запуск программы с `oranges=20` в конфигурационном файле сгенерирует 30, поскольку для яблок по умолчанию установлено значение 10:

```
$ ./our_program.exe
 Fruits count: 30
```

См. также

- Официальная документация по Boost содержит еще много примеров. В ней рассказывается о более продвинутых функциях `Boost.ProgramOptions`, таких как параметры, зависящие от позиции, нетрадиционный синтаксис и т. д.; посетите страницу http://boost.org/libs/program_options;
- вы можете изменять и запускать все примеры из этой книги в режиме онлайн на странице http://apolukhin.github.io/Boost-Cookbook/.

Сохранение любого значения в контейнере или переменной

Если вы программировали на Java, C # или Delphi, то в C++ вам определенно не хватает возможности создания контейнеров с типом значения `ObjectC++`. Класс `Object` в этих языках является базовым классом почти для всех типов, поэтому вы можете в любое время присвоить ему практически любое значение. Только представьте, как было бы здорово иметь такую возможность в C++:

```
typedef std::unique_ptr<Object> object_ptr;

std::vector<object_ptr> some_values;
some_values.push_back(new Object(10));
some_values.push_back(new Object("Hello there"));
some_values.push_back(new Object(std::string("Wow!")));

std::string* p = dynamic_cast<std::string*>(some_values.back().get());
```

```
assert(p);
(*p) += " That is great!\n";
std::cout << *p;
```

Подготовка

Мы будем работать с библиотекой header-only (состоящей только из заголовочных файлов). Все, что требуется для этого рецепта, – базовые знания C++.

Как это делается...

Boost предлагает решение, библиотеку `Boost.Any`, которая имеет даже более выразительный синтаксис:

```
#include <iostream>
#include <vector>
#include <string>

int main() {
    std::vector<boost::any> some_values;
    some_values.push_back(10);
    some_values.push_back("Hello there!");
    some_values.push_back(std::string("Wow!"));

    std::string& s = boost::any_cast<std::string&>(some_values.back());
    s += " That is great!";
    std::cout << s;
}
```

Здорово, правда? Кстати, у `boost::any` есть пустое состояние, которое можно проверить с помощью функции-члена `empty()` (как в контейнерах стандартных библиотек).

Можно получить значение из `boost::any`, используя два подхода:

```
void example() {
    boost::any variable(std::string("Hello world!"));

    // При использовании приведенного ниже метода может выбрасываться исключение
    // boost::bad_any_cast, если фактическое значение в переменной
    // не является std::string.
    std::string s1 = boost::any_cast<std::string>(variable);

    // Исключение не будет выброшено. Если фактическое значение в переменной не является
    // std::string, будет возвращен указатель NULL.
    std::string* s2 = boost::any_cast<std::string>(&variable);
}
```

Как это работает...

Класс `boost::any` хранит в себе любое значение. Чтобы добиться этого, он использует метод **стирания типов** (близкий к тому, что Java или C # делает со всеми типами). Чтобы использовать эту библиотеку, вам не нужно подробно знать ее внутреннюю реализацию, но ниже приводится краткий обзор вышеупомянутого метода для любопытных.

При присвоении некоторой переменной типа T библиотека Boost.Any создает экземпляр holder<T>, который может хранить значение указанного типа T и который является производным от некого базового типа placeholder:

```
template<typename ValueType>
struct holder : public placeholder {
    virtual const std::type_info& type() const {
         return typeid(ValueType);
    }
    ValueType held;
};
```

Тип placeholder имеет виртуальные функции для получения std::type_info хранимого типа T и клонирования:

```
struct placeholder {
    virtual ~placeholder() {}
    virtual const std::type_info& type() const = 0;
};
```

Класс boost::any хранит указатель на placeholder. При использовании any_cast<T>() boost::any проверяет, что вызов ptr->type() дает std::type_info, эквивалентный typeid(T), и возвращает static_cast<holder<T>*>(ptr)->held.

Дополнительно...

Такая гибкость не обходится без затрат. Конструирование копий, конструирование значений, копирование присваивания и присваивание значений экземплярам boost::any выполняют динамическое выделение памяти; все приведения типов производят проверки **динамической идентификации типа данных (RTTI)**; класс boost::any часто использует виртуальные функции. Если вас интересует производительность, следующий рецепт даст вам представление о том, как достичь почти таких же результатов без динамических аллокаций и применения RTTI.

Класс boost::any использует **rvalue-ссылки**, но не может использоваться в **constexpr**-функциях.

Библиотека Boost.Any была принята в стандарт C++17. Если ваш компилятор совместим с C++17 и вы хотите избежать использования Boost для any, просто замените пространство имен boost на пространство имен std и подключите заголовочный файл <any> вместо <boost/any.hpp>. Ваша стандартная реализация библиотеки может работать немного быстрее, если вы храните крошечные объекты в std::any.

У std::any *есть функция* reset() *вместо функции* clear() *и* has_value() *вместо* empty(). *Почти все исключения в Boost наследуются от класса* std::exception *или из его производных, например* boost::bad_any_cast *является производным от* std::bad_cast. *Это означает, что вы можете перехватывать почти все исключения Boost с помощью* catch (const std::exception& e).

См. также

- В официальной документации по Boost можно найти еще несколько примеров; посетите страницу http://boost.org/libs/any.
- Рецепт «*Использование более безопасного способа работы с контейнером, в котором хранится несколько выбранных типов*» приводится для получения дополнительной информации по теме.

Хранение одного из нескольких выбранных типов в контейнере или переменной

Объединения (union) C++03 могут содержать только очень простые типы под названием **простая структура данных (POD)**. Например, в C++03 нельзя хранить `std::string` или `std::vector` в объединении.

Вы знаете о концепции **неограниченных объединений (unrestricted unions)** в C++11? Позвольте мне кратко рассказать вам о них. C++11 ослабляет требования для объединений, но вы должны сами управлять созданием и уничтожением не-POD-типов. Вы должны вызывать конструирование или уничтожение по месту (in-place contrcution/destruction) и запомнить, какой тип хранится в объединении. Огромный объем работы, не так ли?

Можно ли в C++03 получить переменную, которая ведет себя как неограниченное объединение C++ и которая управляет временем жизни объекта, запоминает его тип?

Подготовка...

Мы будем работать с библиотекой header-only, которая проста в использовании. Все, что требуется для этого рецепта, – базовые знания C++.

Как это делается...

Позвольте представить вам библиотеку `Boost.Variant`.

1. Библиотека `Boost.Variant` может хранить любые типы, указанные во время компиляции. Она также управляет созданием или уничтожением по месту, и ей даже не требуется стандарт C++11:

```
#include <boost/variant.hpp>
#include <iostream>
#include <vector>
#include <string>

int main() {
    typedef boost::variant<int, const char*, std::string> my_var_t;
    std::vector<my_var_t> some_values;
    some_values.push_back(10);
    some_values.push_back("Hello there!");
    some_values.push_back(std::string("Wow!"));

    std::string& s = boost::get<std::string>(some_values.back());
    s += " That is great!\n";
```

```
        std::cout << s;
    }
```

Здорово, правда?

2. `Boost.Variant` не имеет пустого состояния, но у нее есть функция `empty()`, которая бесполезна и всегда возвращает значение `false`. Если вам нужно представить пустое состояние, просто добавьте простой тип первым шаблонным параметром `boost::variant`. Если `Boost.Variant` содержит этот тип, интерпретируйте его как пустое состояние. Вот пример, в котором мы будем использовать тип `boost::blank` для представления пустого состояния:

```
void example1() {
  // Конструктор по умолчанию создает экземпляр boost::blank.
  boost::variant<
      boost::blank, int, const char*, std::string
  > var;
  // Метод which() возвращает индекс типа, который в настоящее время содержится
  // в  variant
  assert(var.which() == 0); // boost::blank

  var = "Hello, dear reader";
  assert(var.which() != 0);
}
```

3. Можно получить значение из `boost::variant`, используя два подхода:

```
void example2() {
  boost::variant<int, std::string> variable(0);

  // При использовании приведенного ниже метода может выбрасываться исключение
  // boost::bad_get, если фактическое значение в  variable не является int.
  int s1 = boost::get<int>(variable);

  // Если фактическое значение в переменной не является int, будет возвращено NULL.
  int* s2 = boost::get<int>(&variable);
}
```

Как это работает...

Класс `boost::variant` содержит массив байтов и хранит значения в этом массиве. Размер массива определяется во время компиляции путем применения функции `sizeof()` и функций для определения выравнивания (alignment) каждого из типов шаблонов. При присваивании или создании класса `boost::variable` предыдущее значение уничтожается по месту, а новое значение создается поверх массива байтов с использованием оператора placement new.

Дополнительно...

`Boost.Variant` обычно не выделяет память динамически и не требует RTTI. Это чрезвычайно быстрая библиотека, и она широко используется другими библиотеками Boost. Для достижения максимальной производительности убедитесь, что в шаблонном списке типов в первой позиции указан простой

тип (POD). `boost::variant` использует rvalue-ссылки C++11, если они доступны в вашем компиляторе.

Библиотека `Boost.Variant` является частью стандарта C++17. `std::variant` немного отличается от `boost::variant`:

- `std::variant` объявляется в файле заголовка `<variant>`, а не в `<boost/variant.hpp>`;
- `std::variant` никогда динамически не выделяет память;
- `std::variant` можно использовать в constexpr-функциях;
- вместо того чтобы писать `boost::get<int>(& variable)`, вы должны писать `std::get_if<int> (&variable)`;
- `std::variant` не может рекурсивно содержать сам себя и лишен некоторых других передовых методик;
- `std::variant` имеет конструкторы in-place;
- `std::variant` имеет функцию `index()` вместо `which()`.

См. также

- Рецепт «*Использование безопасного способа работы с контейнером, в котором хранится несколько выбранных типов*»;
- в официальной документации Boost содержатся дополнительные примеры и описания некоторых других функций библиотеки `Boost.Variant`. Их можно найти по адресу: http://boost.org/libs/variant;
- поэкспериментируйте с кодом в режиме онлайн на странице http://apolukhin.github.io/Boost-Cookbook.

Использование более безопасного способа работы с контейнером, в котором хранится один из нескольких выбранных типов

Представьте, что вы создаете C++-обертку для некоторого низкоуровневого интерфейса базы данных SQL. Вы решили, что класс `boost::any` будет идеально соответствовать требованиям одной ячейки таблицы базы данных.

Какой-то другой программист будет использовать ваши классы, и его/ее задачей будет получить строку из базы данных и посчитать сумму арифметических типов в строке.

Вот как будет выглядеть такой код:

```
#include <boost/any.hpp>
#include <vector>
#include <string>
#include <typeinfo>
#include <algorithm>

#include <iostream>
// typedefы и методы будут в нашем заголовке, который оборачивает
// нативный интерфейс SQL.
typedef boost::any cell_t;
```

```
typedef std::vector<cell_t> db_row_t;
// Это всего лишь пример, реальной работы с базой данных нет.
db_row_t get_row(const char* /*query*/) {
    // В реальном приложении параметр 'query' должен иметь тип 'const char *'
    // или 'const std::string &'? См. рецепт "Использование типа «ссылка на строку»",
    // чтобы найти ответ.
    db_row_t row;
    row.push_back(10);
    row.push_back(10.1f);
    row.push_back(std::string("hello again"));
    return row;
}

// Так пользователь будет использовать ваши классы обертки
struct db_sum {
private:
    double& sum_;
public:
    explicit db_sum(double& sum)
        : sum_(sum)
    {}

     void operator()(const cell_t& value) {
         const std::type_info& ti = value.type();
         if (ti == typeid(int)) {
             sum_ += boost::any_cast<int>(value);
         } else if (ti == typeid(float)) {
              sum_ += boost::any_cast<float>(value);
         }
       }
};

int main() {
    db_row_t row = get_row("Query: Give me some row, please.");
    double res = 0.0;
    std::for_each(row.begin(), row.end(), db_sum(res));
    std::cout << "Sum of arithmetic types in database row is: "
              << res << std::endl;
}
```

Если вы скомпилируете и запустите этот пример, он выдаст правильный ответ:

```
Sum of arithmetic types in database row is: 20.1
```

Вы помните, о чем думали, когда читали реализацию `operator()`? Полагаю, вы думали так: «*А как насчет* `double`, `long double`, `short`, `unsigned` *и других типов*?» Те же мысли придут в голову программисту, который будет использовать ваш интерфейс. Поэтому вам нужно тщательно документировать значения, хранящиеся в вашем типе `cell_t`, или использовать более элегантное решение, которое описано в следующих разделах.

Подготовка

Если вы еще незнакомы с библиотеками Boost.Variant и Boost.Any, настоятельно рекомендую вам прочитать два предыдущих рецепта.

Как это делается...

Библиотека Boost.Variant реализует идиому проектирования «Посетитель» (visitor) для доступа к хранимым данным. Это намного безопаснее, чем получать значения с помощью boost::get<>. Данная идиома заставляет программиста заботиться о каждом типе в variant, иначе код нельзя будет скомпилировать. Вы можете воспользоваться этой идиомой с помощью функции boost::apply_visitor, которая принимает функциональный объект visitor в качестве первого параметра и variant в качестве второго параметра. Если вы используете компилятор стандарта, предшествующего C++14, то функциональные объекты visitor должны наследоваться от класса boost::static_visitor<T>, где T – это тип, возвращаемый visitor. У объекта visitor должны быть перегрузки operator() для каждого типа, который может содержать variant.

Давайте изменим тип cell_t на boost::variable<int, float, string> и изменим наш пример:

```
#include <boost/variant.hpp>
#include <vector>
#include <string>
#include <iostream>

// typedefы и методы будут в нашем заголовке, который оборачивает интерфейс SQL.
typedef boost::variant<int, float, std::string> cell_t;
typedef std::vector<cell_t> db_row_t;

// Это всего лишь пример, реальной работы с базой данных нет.
db_row_t get_row(const char* /*query*/) {
    // См. рецепт "Использование типа «ссылка на строку»",
    // где приводится более подходящий тип для параметра "query".
    db_row_t row;
    row.push_back(10);
    row.push_back(10.1f);
    row.push_back("hello again");
    return row;
}

// Это код, необходимый для суммирования значений.
// Мы могли бы просто использовать boost::static_visitor<>,
// если бы наш объект visitor ничего не возвращал.
struct db_sum_visitor: public boost::static_visitor<double> {
    double operator()(int value) const {
        return value;
    }
    double operator()(float value) const {
        return value;
    }
```

```
    double operator()(const std::string& /*value*/) const {
       return 0.0;
    }
};

int main() {
    db_row_t row = get_row("Query: Give me some row, please.");
    double res = 0.0;
    for (auto it = row.begin(), end = row.end(); it != end; ++it) {
        res += boost::apply_visitor(db_sum_visitor(), *it);
    }
    std::cout << "Sum of arithmetic types in database row is: "
              << res << std::endl;
}
```

Как это работает...

Во время компиляции библиотека `Boost.Variant` генерирует большой оператор `switch`, каждый `case` которого вызывает объект `visitor` для одного типа из списка типов варианта. Во время выполнения индекс сохраненного типа извлекается с помощью функции `which()`, и происходит переход на нужный case в switch. Нечто подобное будет сгенерировано для `boost::variant<int, float, std::string>`:

```
switch (which())
{
case 0 /*int*/:
    return visitor(*reinterpret_cast<int*>(address()));
case 1 /*float*/:
    return visitor(*reinterpret_cast<float*>(address()));
case 2 /*std::string*/:
    return visitor(*reinterpret_cast<std::string*>(address()));
default: assert(false);
}
```

Здесь функция `address()` возвращает указатель на внутреннюю память `boost::variant<int, float, std::string>`.

Дополнительно...

Если мы сравним этот пример с первым примером из данного рецепта, то увидим следующие преимущества `boost::variant`:

- мы знаем, какие типы может хранить переменная;
- если разработчик библиотеки интерфейса SQL добавляет или изменяет тип, содержащийся в `variant`, вместо неправильного поведения мы получим ошибку времени компиляции.

`std::variant` из стандарта C++17 также поддерживает эту идиому проектирования. Просто напишите `std::visit` вместо `boost::apply_visitor` – и готово.

Вы можете скачать примеры файлов с кодами для всех книг Packt, которые вы приобрели, со своего аккаунта на сайте http://www.PacktPub.com. Если вы купили эту книгу в другом месте, можете зайти на страницу http://www.PacktPub.com/support и зарегистрироваться, чтобы получить файлы по электронной почте.

См. также

- Прочитав рецепты из главы 4 «Уловки времени компиляции», вы сможете создавать универсальные объекты visitor, которые будут работать правильно, даже при изменении базовых типов;
- официальная документация Boost содержит дополнительные примеры и описание некоторых других функций библиотеки Boost.Variant: http://boost.org/libs/variant.

Возврат значения или флага «значения нет»

Представьте, что у нас есть функция, которая не выбрасывает исключение, а возвращает значение или как-то говорит о том, что произошла ошибка. В языках программирования Java или C# такие случаи обрабатываются путем сравнения возвращаемого значения из функции с нулевым указателем. Если функция вернула ноль, то произошла ошибка. В языке C++ возврат указателя из функции сбивает с толку пользователей библиотеки и обычно требует медленного динамического выделения памяти.

Подготовка

Все, что вам нужно для этого рецепта, – базовые знания C++.

Как это делается...

Дамы и господа, позвольте мне представить вам библиотеку `Boost.Optional`, используя приведенный ниже пример.

Есть некая функция `try_lock_device()`. Она пытается захватить уникальный доступ к устройству и вернуть объект, владеющий устройством. Операция может успешно завершиться или провалиться, в зависимости от различных условий (в нашем примере – от вызова функции `try_lock_device_impl()`):

```
#include <boost/optional.hpp>
#include <iostream>

class locked_device {
    explicit locked_device(const char* /*param*/) {
       // У нас есть уникальный доступ к устройству.
       std::cout << "Device is locked\n";
    }
    static bool try_lock_device_impl();
public:
    void use() {
        std::cout << "Success!\n";
    }
```

```
    static boost::optional<locked_device> try_lock_device() {
      if (!try_lock_device_impl()) {
        // Не удалось получить уникальный доступ.
        return boost::none;
      }

      // Успешно!
      return locked_device("device name");
    }

    ~locked_device(); // Снимает блокировку с устройства.
};
```

В примере функция возвращает переменную `boost::optional`, которую можно преобразовать в тип `bool`. Так, в нашем примере если возвращаемое значение преобразуется в `true`, тогда мы захватили устройство, и экземпляр класса для работы с устройством можно получить путем разыменования возвращаемой `boost::optional` переменной:

```
int main() {
    for (unsigned i = 0; i < 10; ++i) {
        boost::optional<locked_device> t
             = locked_device::try_lock_device();

        // Можно преобразовать в bool.
        if (t) {
           t->use();
           return 0;
        } else {
            std::cout << "...trying again\n";
        }
    }

    std::cout << "Failure!\n";
    return -1;
}
```

Эта программа выведет следующее:

```
...trying again
...trying again
Device is locked
Success!
```

Созданную по умолчанию переменную `optional` можно преобразовать в `false`, и она не должна быть разыменована, потому что у нее нет базового типа.

Как это работает...

`boost::optional<T>` «под капотом» имеет правильно выровненный массив байтов, в котором объект типа `T` может быть создан по месту, а также переменную `bool` для запоминания состояния объекта `T` (сконструирован он или нет?).

Дополнительно...

Класс `Boost.Optional` не использует динамическую аллокацию памяти, и ему не требуется конструктор по умолчанию для базового типа. Текущая реализация `boost::optional` может работать с rvalue-ссылками в C++11, но ее нельзя использовать в constexpr-функциях.

Если у вас есть класс T, у которого нет пустого состояния, но ваша программная логика требует пустого состояния или неинициализированного T, нужно как-то выкручиваться. Зачастую пользователи применяют умный указатель для класса T, используют нулевой указатель для обозначения пустого состояния и динамически выделяют T, если пустое состояние не требуется. Не делайте этого! Вместо этого используйте boost::optional<T>. Это гораздо более быстрое и надежное решение.

Стандарт C++17 включает в себя класс `std::optional`. Просто замените `<boost/optional.hpp>` на `<optional>` и `boost::` на `std::`, чтобы использовать стандартную версию этого класса. Класс `std::optional` подходит для использования в constexpr-функциях.

См. также

В официальной документации по Boost содержатся дополнительные примеры и описываются расширенные функции `Boost.Optional` (например, конструкторы in-place). Документация доступна по адресу: http://boost.org/libs/optional.

Возвращение массива из функции

Давайте поиграем в угадайку! Что можно сказать об этой функции?

```
char* vector_advance(char* val);
```

Должно возвращаемое значение быть освобождено (деаллоцировано) программистом или нет? Функция пытается освободить входной параметр? Должен ли входной параметр заканчиваться нулем или функция должна предполагать, что этот параметр имеет какую-то определенную длину?

Теперь усложним задачу! Взгляните на следующую строку:

```
char ( &vector_advance( char (&val)[4] ) )[4];
```

Не волнуйтесь. Я также полчаса чесал голову, прежде чем понял, что здесь происходит. `vector_advance` – это функция, которая принимает и возвращает массив из четырех элементов. Можно ли написать эту функцию так, чтобы было понятно?

Подготовка

Все, что вам нужно для этого рецепта, – базовые знания C++.

Как это делается...

Мы можем переписать функцию следующим образом:

```
#include <boost/array.hpp>

typedef boost::array<char, 4> array4_t;
array4_t& vector_advance(array4_t& val);
```

Здесь `boost::array<char, 4>` – это простая обертка вокруг массива из четырех элементов `char`.

Этот код отвечает на все вопросы из нашего первого примера и является гораздо более читабельным, чем код из второго примера.

Как это работает...

`boost::array` – это массив фиксированного размера. Первый шаблонный параметр `boost::array` является типом элемента, а второй – это размер массива. Если вам нужно изменить размер массива во время выполнения, то класс `boost::array` вам не подходит, используйте `std::vector`, `boost::container::small_vector`, `boost::container::stack_vector` или `boost::container::vector`.

Класс `boost::array<>` не имеет рукописных конструкторов, и все его члены являются открытыми, поэтому компилятор будет рассматривать его как простую структуру данных.

Дополнительно...

Давайте посмотрим еще несколько примеров использования `boost::array`:

```
#include <boost/array.hpp>
#include <algorithm>
typedef boost::array<char, 4> array4_t;

array4_t& vector_advance(array4_t& val) {
    // Лямбда-функция C++11
    const auto inc = [](char& c){ ++c; };

    // У массива boost::array есть begin(), cbegin(), end(), cend(),
    // rbegin(), size(), empty() и другие функции, которые являются общими
    // для стандартных контейнеров
    std::for_each(val.begin(), val.end(), inc);
    return val;
}

int main() {
    // Можно инициализировать boost::array так же, как массив в C++11:
    // array4_t val = {0, 1, 2, 3};
    // но в C++03 требуется дополнительная пара фигурных скобок.
    array4_t val = {{0, 1, 2, 3}};

    array4_t val_res; // У boost::array есть конструктор по умолчанию
    val_res = vector_advance(val); // и оператор присваивания
    assert(val.size() == 4);
    assert(val[0] == 1);
    /*val[4];*/ // Сработает внутренний assert, т.к. максимальный индекс равен 3

    // Мы можем заставить отработать этот assert во время компиляции.
```

```
    // Интересно? См. рецепт «Проверка размеров во время компиляции»
    assert(sizeof(val) == sizeof(char) * array4_t::static_size);
}
```

Одним из самых больших преимуществ `boost::array` является то, что он не выделяет динамически память и обеспечивает точно такую же производительность, что и обычный массив C. Людям из комитета по стандартизации C++ это тоже понравилось, поэтому он был принят в стандарт C++11. Попробуйте подключить заголовочный файл `<array>` и проверьте наличие `std::array`. `std::array` имеет полную поддержку constexpr начиная с C++17.

См. также

- В официальной документации по Boost приводится полный список методов `Boost.Array` с описанием сложности метода. Он доступен по ссылке: http://boost.org/libs/array.
- Тип `boost::array` широко используется в разных рецептах, например *см.* рецепт «*Привязка значения в качестве параметра функции*».

Объединение нескольких значений в одно

Хороший подарок для тех, кому нравится `std::pair`. В Boost есть библиотека под названием `Boost.Tuple`. Она похожа на `std::pair`, но также может работать с тройками типов, четверками и даже более крупными коллекциями типов.

Подготовка

Все, что вам нужно для этого рецепта, – базовые знания C++.

Как это делается...

Выполните следующие шаги, чтобы объединить несколько значений в один кортеж (tuple).

1. Чтобы начать работать с кортежами, вам нужно подключить правильный заголовочный файл и объявить переменную типа `boost::tuple`:

```
#include <boost/tuple/tuple.hpp>
#include <string>

boost::tuple<int, std::string> almost_a_pair(10, "Hello");
boost::tuple<int, float, double, int> quad(10, 1.0f, 10.0, 1);
```

2. Получение определенного значения осуществляется с помощью функции `boost::get<N>()`, где `N` – это индекс необходимого значения:

```
#include <boost/tuple/tuple.hpp>

void sample1() {
    const int i = boost::get<0>(almost_a_pair);
    const std::string& str = boost::get<1>(almost_a_pair);
    const double d = boost::get<2>(quad);
}
```

Функция boost::get<> имеет множество перегрузок и широко используется в Boost. Мы уже видели, что ее можно использовать с другими библиотеками, в рецепте «*Хранение нескольких выбранных типов в контейнере или переменной*».

3. Вы можете создавать кортежи, используя функцию boost::make_tuple(), которую быстрее писать, потому что вам не нужно явно перечислять шаблонные параметры кортежа:

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_comparison.hpp>
#include <set>

void sample2() {
    // Операторы сравнения кортежей определены в заголовке
    // «boost/tuple/tuple_comparison.hpp»
    // Не забудьте включить его!
    std::set<boost::tuple<int, double, int> > s;
    s.insert(boost::make_tuple(1, 1.0, 2));
    s.insert(boost::make_tuple(2, 10.0, 2));
    s.insert(boost::make_tuple(3, 100.0, 2));

    // Требует C++11
    const auto t = boost::make_tuple(0, -1.0, 2);
    assert(2 == boost::get<2>(t));
    // Мы можем выполнить эту проверку на этапе компиляции. Интересно?
    // См. главу «Трюки времени компиляции»
}
```

4. Еще одна функция, облегчающая жизнь, – это boost::tie(). Она работает почти как make_tuple, но добавляет неконстантную ссылку для каждого из передаваемых типов. Такой кортеж можно использовать для получения значений в переменные из другого кортежа. Это можно лучше понять из приведенного ниже примера:

```
#include <boost/tuple/tuple.hpp>
#include <cassert>

void sample3() {
    boost::tuple<int, float, double, int> quad(10, 1.0f, 10.0,
1);
    int i;
    float f;
    double d;
    int i2;

    // Передача значений из 'quad' в переменные 'i', 'f', 'd', 'i2'.
    boost::tie(i, f, d, i2) = quad;
    assert(i == 10);
    assert(i2 == 1);
}
```

Как это работает...

Некоторые читатели могут задаться вопросом, зачем нам нужен кортеж, когда мы всегда можем написать собственные структуры с более подходящими именами, например вместо того, чтобы писать `boost::tuple<int, std::string>`, мы можем создать структуру:

```
struct id_name_pair {
    int id;
    std::string name;
};
```

Да, эта структура определенно понятнее, чем `boost::tuple<int, std::string>`. Основное применение библиотеки кортежей – упрощение шаблонного программирования.

Дополнительно...

Кортеж работает так же быстро, как шаблон структуры `std::pair` (он не выделяет память в кучи и не имеет виртуальных функций). Комитет по C++ счел этот класс очень полезным, и тот был включен в стандартную библиотеку. Вы можете найти его в C++11-совместимой реализации в файле заголовка `<tuple>` (не забудьте заменить все пространства имен `boost::` на `std::`).

Стандартная версия кортежа имеет несколько микрооптимизаций и, как правило, обеспечивает немного большую производительность. Тем не менее `std::tuple` не гарантирует порядок конструирования элементов. Поэтому если вам нужен кортеж, который создает свои элементы, начиная с первого, вы должны использовать `boost::tuple`:

```
#include <boost/tuple/tuple.hpp>
#include <iostream>

template <int I>
struct printer {
    printer() { std::cout << I; }
};

int main() {
    // На выходе дает 012
    boost::tuple<printer<0>, printer<1>, printer<2> > t;
}
```

Текущая реализация кортежа Boost не использует variadic templates, не поддерживает rvalue-ссылки и декомпозицию C++17 (structured bindings), и ее нельзя использовать в constexpr-функциях.

См. также

- В официальной документации по Boost содержится больше примеров и информации о производительности и возможностях библиотеки `Boost.Tuple`. Она доступна по ссылке http://www.boost.org/libs/tuple;
- в рецепте «*Преобразование всех элементов кортежа в строку*» из главы 8 «*Метапрограммирование*» показаны продвинутые способы использования кортежей.

Привязка и переупорядочение параметров функции

Если вы много работаете со стандартной библиотекой и используете заголовочный файл <algorithm>, вы определенно пишете много функциональных объектов. Начиная с C++11 вы можете использовать лямбды с алгоритамами стандартной библиотеки. В более ранних версиях стандарта C++ вы можете создавать функциональные объекты, используя такие адаптеры, как bind1st, bind2nd, ptr_fun, mem_fun, mem_fun_ref, или можете писать функциональные объекты вручную (потому что адаптеры выглядят пугающе). Хорошая новость: библиотеку Boost.Bind можно использовать вместо уродливых адаптеров, и она обеспечивает более понятный синтаксис.

Подготовка

Знание стандартных библиотечных функций и алгоритмов будет полезно.

Как это делается...

Давайте посмотрим на примеры использования библиотеки Boost.Bind и аналогичные примеры с лямбдами C++11:

1. Все примеры в этом рецепте требуют подключения следующих заголовочных файлов:

```
// Содержит функцию boost::bind и _1, _2, _3....
#include <boost/bind.hpp>

// Полезные вещи, которые нужны образцам.
#include <boost/array.hpp>
#include <algorithm>
#include <functional>
#include <string>
#include <cassert>
```

2. Подсчитаем значения больше 5, как показано в приведенном ниже коде:

```
void sample1() {
    const boost::array<int, 12> v = {{
        1, 2, 3, 4, 5, 6, 7, 100, 99, 98, 97, 96
    }};

    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [](int x) { return 5 < x; }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(std::less<int>(), 5, _1)
    );
    assert(count0 == count1);
}
```

3. Вот как можно сосчитать пустые строки:

```
void sample2() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};
```

```
    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [](const std::string& s) { return s.empty(); }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(&std::string::empty, _1)
    );
    assert(count0 == count1);
}
```

4. Теперь давайте посчитаем строки длиной менее 5:

```
void sample3() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};

    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [](const std::string& s) { return s.size() < 5; }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(
            std::less<std::size_t>(),
            boost::bind(&std::string::size, _1),
            5
        )
    );
    assert(count0 == count1);
}
```

5. Сравним строки:

```
void sample4() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};
    std::string s(
        "Expensive copy constructor is called when binding"

    );
    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [&s](const std::string& x) { return x < s; }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(std::less<std::string>(), _1, s)
    );
    assert(count0 == count1);
}
```

Как это работает...

Функция `boost::bind` возвращает функциональный объект, в котором хранятся копии всех аргументов функции. Когда выполняется фактический вызов функции `operator()`, сохраненные параметры передаются в исходный функциональный объект вместе с параметрами, переданными во время вызова.

Дополнительно...

Взгляните на предыдущие примеры. Когда мы вызываем `boost::bind`, то копируем значения в функциональный объект. Для некоторых классов это затратная операция. Есть ли способ обойти копирование?

Да, есть! Здесь нам поможет библиотека `Boost.Ref`! Она содержит две функции, `boost::ref()` и `boost::cref()`, первая из которых позволяет сохранять параметр в качестве ссылки, а вторая сохраняет параметр в качестве константной ссылки. Функции `ref()` и `cref()` просто создают объект типа `reference_wrapper<T>` или `reference_wrapper<const T>`, который умеет неявно преобразовываться в ссылочный тип. Давайте внесем изменения в наши последние примеры:

```
#include <boost/ref.hpp>

void sample5() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};
    std::string s(
        "Expensive copy constructor is NOT called when binding"
    );

    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(std::less<std::string>(), _1, boost::cref(s))
    );
    // ...
}
```

Вы также можете переупорядочивать, игнорировать и дублировать параметры функции, используя функцию `bind`:

```
void sample6() {
    const auto twice = boost::bind(std::plus<int>(), _1, _1);
    assert(twice(2) == 4);

    const auto minus_from_second = boost::bind(std::minus<int>(), _2, _1);
    assert(minus_from_second(2, 4) == 2);

    const auto sum_second_and_third = boost::bind(
        std::plus<int>(), _2, _3
    );
    assert(sum_second_and_third(10, 20, 30) == 50);
}
```

Функции `ref`, `cref` и `bind` приняты в стандарт C++11 и определены в заголовке `<functional>` в пространстве имен `std::`. Все эти функции не выделяют память динамически и не используют виртуальных функций. Возвращаемые ими объекты легко оптимизируются толковыми компиляторами.

Реализации этих функций в стандартной библиотеке могут иметь дополнительные оптимизации для сокращения времени компиляции. Вы можете использовать версии функций `bind`, `ref`, `cref` стандартных библиотек с любой библиотекой Boost или даже смешивать версии из Boost и стандартных библиотек.

Если вы применяете компилятор C++14, то используйте универсальные лямбда-выражения вместо `std::bind` и `boost::bind`, поскольку их проще понять. В отличие от `boost::bind`, лямбды с C++17 можно использовать со спецификатором типа constexpr.

См. также

В официальной документации содержится еще больше примеров и описание расширенных функций: http://boost.org/libs/bind.

Получение удобочитаемого имени типа

Часто возникает необходимость получить читабельное имя типа во время выполнения:

```
#include <iostream>
#include <typeinfo>

template <class T>
void do_something(const T& x) {
    if (x == 0) {
        std::cout << "Error: x == 0. T is " << typeid(T).name()
        << std::endl;
    }
    // ...
}
```

Тем не менее приведенный ранее пример не является особо переносимым. Он не работает, если динамическая идентификация типа данных (RTTI) отключена, и не всегда выдает красивое удобочитаемое имя. На некоторых платформах такой код будет давать на выходе только `i` или `d`.

Ситуация ухудшается, если нам нужна полная спецификация типа без удаления квалификаторов `const`, `volatile` и ссылок:

```
void sample1() {
    auto&& x = 42;
    std::cout << "x is "
              << typeid(decltype(x)).name()
              << std::endl;
}
```

К сожалению, предыдущий код в лучшем случае выведет `int`, а это не то, что мы ожидали увидеть.

Подготовка

Для этого рецепта требуется базовое знание C++.

Как это делается...

В первом примере нам нужно удобочитаемое имя типа без квалификаторов. Нам поможет библиотека `Boost.TypeIndex`:

```
#include <iostream>
#include <boost/type_index.hpp>

template <class T>
void do_something_again(const T& x) {
    if (x == 0) {
        std::cout << "x == 0. T is " << boost::typeindex::type_id<T>()
                  << std::endl;
    }
    // ...
}
```

Во втором примере нам нужно оставить квалификаторы, поэтому нужно вызвать немного другую функцию из той же библиотеки:

```
#include <boost/type_index.hpp>

void sample2() {
    auto&& x = 42;
    std::cout << "x is "
              << boost::typeindex::type_id_with_cvr<decltype(x)>()
              << std::endl;
}
```

Как это работает...

Библиотека `Boost.TypeIndex` знает о внутреннем устройстве разных компиляторов и обладает знаниями о наиболее эффективном способе создания удобочитаемого имени для типа. Если вы предоставляете тип в качестве параметра шаблона, библиотека гарантирует, что все возможные вычисления, связанные с типами, будут выполняться во время компиляции, и код будет работать, даже если динамическая идентификация типа данных (RTTI) отключена.

`cvr` в `boost::typeindex::type_id_with_cvr` расшифровывается как `const`, `volatile` и `reference`. Эта функция гарантирует, что информация о квалификаторах и ссылках окажется в результате функции.

Дополнительно...

Все функции `boost::typeindex::type_id*` возвращают экземпляры `boost::typeindex::type_index`. Этот тип очень похож на `std::type_index`; однако у него есть метод `raw_name()` для получения необработанного имени типа и метод `pretty_name()` для получения человекочитаемого имени типа.

Даже в самых новых стандартах C++ `std::type_index` и `std::type_info` возвращают специфичные для платформы представления имен типов, которые довольно сложно декодировать или использовать переносимо.

В отличие от метода стандартной библиотеки `typeid()`, некоторые классы из библиотеки `Boost.TypeIndex` можно использовать со спецификатором типа constexpr. Это означает, что вы можете получить текстовое представление вашего типа во время компиляции, если используете специфический класс `boost::typeindex::ctti_type_index`.

Пользователи могут создавать собственные реализации динамической идентификации типов данных, используя библиотеку `Boost.TypeIndex`. Это мо-

жет быть полезно для разработчиков встраиваемых систем и для приложений, которым требуется чрезвычайно эффективная динамическая идентификация типа данных, настроенная на определенные типы.

См. также

Документация по расширенным функциям и дополнительные примеры доступны на странице http://www.boost.org/libs/type_index.

Использование эмуляции перемещения C++11

Одна из главных особенностей стандарта C++11 – это rvalue-ссылки, та же move-семантика. Эта особенность позволяет нам изменять временные объекты, «крадя» у них ресурсы. Как вы можете догадаться, стандарт C++03 не имеет rvalue-ссылок, но, используя библиотеку `Boost.Move`, можно написать переносимый код, который эмулирует их.

Подготовка

Настоятельно рекомендуется, чтобы вы хотя бы были знакомы с основами rvalue-ссылок C++11.

Как это делается...

1. Представьте, что у вас есть класс с несколькими полями, некоторые из которых являются контейнерами стандартной библиотеки:

```
namespace other {
    class characteristics{};
}
struct person_info {
    std::string name_;
    std::string second_name_;
    other::characteristics characteristic_;
    // ...
};
```

2. Настало время добавить перемещающее присвоение (move assignment) и перемещающий конструктор (move constructor)! Помните, что в C++03 стандартные контейнеры не имеют перемещающего конструктора и перемещающего присваивания.
3. Правильная реализация перемещающего присваивания аналогична перемещающему конструированию временной переменной от входного аргумента и обмену значениями с `this`. Правильная реализация конструктора перемещения близка к конструированию объекта по умолчанию и обмену значениями с входным параметром. Итак, начнем с функции-члена для обмена значениями `swap`:

```
#include <boost/swap.hpp>

void person_info::swap(person_info& rhs) {
    name_.swap(rhs.name_);
```

```
    second_name_.swap(rhs.second_name_);
    boost::swap(characteristic_, rhs.characteristic_);
}
```

4. Теперь поместите следующий макрос в раздел `private`:

```
BOOST_COPYABLE_AND_MOVABLE(person_info)
```

5. Напишите копирующий конструктор.
6. Напишите копирующее присваивание, принимая параметр как

```
BOOST_COPY_ASSIGN_REF(person_info)
```

7. Напишите конструктор перемещения и присваивание перемещением, принимая параметр как `BOOST_RV_REF (person_info)`:

```
struct person_info {
    // Поля объявляются здесь;
    // ...
private:
    BOOST_COPYABLE_AND_MOVABLE(person_info)
public:
    // Чтобы было проще, мы предположим, что конструктор по умолчанию
    // person_info и функцию swap очень быстро и дешево вызывать;
    person_info();

    person_info(const person_info& p)
        : name_(p.name_)
        , second_name_(p.second_name_)
        , characteristic_(p.characteristic_)
    {}

    person_info(BOOST_RV_REF(person_info) person) {
        swap(person);
    }

    person_info& operator=(BOOST_COPY_ASSIGN_REF(person_info) person) {
        person_info tmp(person);
        swap(tmp);
        return *this;
    }

    person_info& operator=(BOOST_RV_REF(person_info) person) {
        person_info tmp(boost::move(person));
        swap(tmp);
        return *this;
    }

    void swap(person_info& rhs);
};
```

8. Теперь у нас есть переносимая быстрая реализация конструктора и оператора перемещения для класса `person_info`.

Как это работает...

Вот пример того, как можно воспользоваться move-семантикой:

```
int main() {
    person_info vasya;
    vasya.name_ = "Vasya";
    vasya.second_name_ = "Snow";

    person_info new_vasya(boost::move(vasya));
    assert(new_vasya.name_ == "Vasya");
    assert(new_vasya.second_name_ == "Snow");
    assert(vasya.name_.empty());
    assert(vasya.second_name_.empty());

    vasya = boost::move(new_vasya);
    assert(vasya.name_ == "Vasya");
    assert(vasya.second_name_ == "Snow");
    assert(new_vasya.name_.empty());
    assert(new_vasya.second_name_.empty());
}
```

Библиотека `Boost.Move` реализована очень эффективно. Когда используется компилятор C++11, все макросы для эмуляции rvalues разворачиваются в обычные rvalue C++11, в противном случае (на компиляторах C++03) rvalues эмулируются.

Дополнительно...

Вы обратили внимание на вызов функции `boost::swap`? Это действительно полезная служебная функция, которая сначала ищет функцию `swap` в пространстве имен переменной (в нашем примере это пространство имен `other::`), и если соответствующей функции нет, она использует `std::swap`.

См. также

- Дополнительную информацию о реализации эмуляции можно найти на сайте Boost и в исходниках библиотеки `Boost.Move` по адресу http://boost.org/libs/move;
- библиотека `Boost.Utility` содержит функцию `boost::swap`, и у нее имеется множество полезных функций и классов. На странице http://boost.org/libs/utility вы найдете документацию по ней;
- рецепт «*Инициализация базового класса членом класса-наследника*» главы 2 «Управление ресурсами»;
- рецепт «*Создание некопируемого класса*»;
- в рецепте «*Создание некопируемого, но перемещаемого класса*» есть больше информации о библиотеке `Boost.Move` и приводятся примеры того, как можно использовать перемещаемые объекты в контейнерах переносимым и эффективным способом.

Создание некопируемого класса

Вы почти наверняка сталкивались с определенными ситуациями, когда классу принадлежат некие ресурсы, которые не следует копировать по техническим причинам:

```
class descriptor_owner {
    void* descriptor_;

public:
    explicit descriptor_owner(const char* params);

    ~descriptor_owner() {
        system_api_free_descriptor(descriptor_);
    }
};
```

Компилятор C++ для предыдущего примера сгенерирует копирующий конструктор и оператор присваивания, поэтому потенциальный пользователь класса `descriptor_owner` сможет написать и скомпилировать вот такие ужасные вещи:

```
void i_am_bad() {
    descriptor_owner d1("O_o");
    descriptor_owner d2("^_^");

    // Дескриптор d2 не был правильно освобожден
    d2 = d1;

    // деструктор d2 освободит дескриптор
    // деструктор d1 попытается освободить уже освобожденный дескриптор
}
```

Подготовка

Все, что требуется для этого рецепта, – базовые знания C++.

Как это делается...

Чтобы избежать таких ситуаций, был придуман класс `boost::noncopyable`. Если вы наследуете от него свой собственный класс, копирующий конструктор и оператор присваивания не будут сгенерированы компилятором C++:

```
#include <boost/noncopyable.hpp>
class descriptor_owner_fixed : private boost::noncopyable {
    // ...
```

Теперь пользователь не сможет делать плохие вещи:

```
void i_am_good() {
    descriptor_owner_fixed d1("O_o");
    descriptor_owner_fixed d2("^_^");

    // Компиляции не будет
    d2 = d1;
```

```
    // И здесь компиляции не будет
    descriptor_owner_fixed d3(d1);
}
```

Как это работает...

Искушенный читатель заметит, что можно достичь точно такого же результата:

- написав копирующий конструктор и оператор присваивания для `descriptor_owning_fixed`;
- определив их без фактической реализации;
- явно удалив их, используя синтаксис C++11 `= delete`.

Да, вы правы. В зависимости от возможностей вашего компилятора класс `boost::noncopyable` выбирает самый подходящий способ сделать класс некопируемым.

`boost::noncopyable` также служит хорошей документацией для вашего класса. С ним никогда не возникает таких вопросов, как «Определено ли тело копирующего конструктора где-то еще?» или «Есть ли у него нестандартный копирующий конструктор (с неконстантным ссылочным параметром)?».

См. также

- Рецепт «*Создание некопируемого, но перемещаемого класса*» даст вам идеи о том, как разрешить уникальное владение ресурсом в C++03 путем его перемещения;
- вы можете найти много полезных функций и классов в официальной документации библиотеки `Boost.Core` по адресу http://boost.org/libs/core;
- рецепт «*Инициализация базового класса членом класса-наследника*» главы 2 «Управление ресурсами»;
- рецепт «*Использование эмуляции перемещения в C++11*».

Создание некопируемого, но перемещаемого класса

Теперь представьте следующую ситуацию: у нас есть ресурс, который нельзя скопировать, который должен быть правильно освобожден в деструкторе, и нужно вернуть его из функции:

```
descriptor_owner construct_descriptor()
{
     return descriptor_owner("Construct using this string");
}
```

На самом деле такие ситуации можно обойти, используя метод `swap`:

```
void construct_descriptor1(descriptor_owner& ret)
{
    descriptor_owner("Construct using this string").swap(ret);
}
```

Однако такой обходной путь не позволяет нам использовать класс `descriptor_owner` в контейнерах. Да и выглядит подобное решение ужасно!

Подготовка

Настоятельно рекомендуется, чтобы вы были по крайней мере знакомы с основами rvalue-ссылок C++11. Также рекомендуется прочитать рецепт «*Использование эмуляции перемещения в C++11*».

Как это делается...

Те читатели, которые используют стандарт C++11, уже знают о классах, которые можно перемещать, но не копировать (например, `std::unique_ptr` или `std::thread`). Используя такой подход, мы можем создать класс `descriptor_owner`, который можно перемещать, но не копировать:

```
class descriptor_owner1 {
    void* descriptor_;

public:
    descriptor_owner1()
      : descriptor_(nullptr)
    {}

    explicit descriptor_owner1(const char* param);

    descriptor_owner1(descriptor_owner1&& param)
        : descriptor_(param.descriptor_)
    {
        param.descriptor_ = nullptr;
    }

    descriptor_owner1& operator=(descriptor_owner1&& param) {
        descriptor_owner1 tmp(std::move(param));
        std::swap(descriptor_, tmp.descriptor_);
        return *this;
    }

    void clear() {
        free(descriptor_);
        descriptor_ = nullptr;
    }

    bool empty() const {
        return !descriptor_;
    }

    ~descriptor_owner1() {
        clear();
    }
};

// GCC компилирует это в C++11 и более поздних режимах.
descriptor_owner1 construct_descriptor2() {
    return descriptor_owner1("Construct using this string");
}
```

```
void foo_rv() {
    std::cout << "C++11\n";
    descriptor_owner1 desc;
    desc = construct_descriptor2();
    assert(!desc.empty());
}
```

Это будет работать только на компиляторах, совместимых с C++11. Подходящий момент для Boost.Move! Давайте внесем изменения в наш пример, чтобы его можно было использовать на компиляторах C++03.

Чтобы написать перемещаемый, но некопируемый тип в переносимом синтаксисе, нужно выполнить следующие простые шаги:

1. Поместить макрос BOOST_MOVABLE_BUT_NOT_COPYABLE(имя класса) в раздел private:

```
#include <boost/move/move.hpp>

class descriptor_owner_movable {
    void* descriptor_;

    BOOST_MOVABLE_BUT_NOT_COPYABLE(descriptor_owner_movable)
```

2. Написать перемещающий конструктор и перемещающее присваивание, принимая параметр как BOOST_RV_REF(имя класса):

```
public:
    descriptor_owner_movable()
        : descriptor_(NULL)
    {}

    explicit descriptor_owner_movable(const char* param)
        : descriptor_(strdup(param))
    {}

    descriptor_owner_movable(
        BOOST_RV_REF(descriptor_owner_movable) param
    ) BOOST_NOEXCEPT
        : descriptor_(param.descriptor_)
    {
        param.descriptor_ = NULL;
    }

    descriptor_owner_movable& operator=(
        BOOST_RV_REF(descriptor_owner_movable) param) BOOST_NOEXCEPT
    {
        descriptor_owner_movable tmp(boost::move(param));
        std::swap(descriptor_, tmp.descriptor_);
        return *this;
    }

    // ...
};
```

```
descriptor_owner_movable construct_descriptor3() {
    return descriptor_owner_movable("Construct using this string");
}
```

Как это работает...

Теперь у нас есть перемещаемый, но некопируемый класс, который можно использовать даже в компиляторах C++03 и с контейнерами из `Boost.Containers`:

```
#include <boost/container/vector.hpp>
#include <your_project/descriptor_owner_movable.h>

int main() {
     // Приведенный ниже код будет работать на компиляторах C++11 и C++03
     descriptor_owner_movable movable;
     movable = construct_descriptor3();
     boost::container::vector<descriptor_owner_movable> vec;
     vec.resize(10);
     vec.push_back(construct_descriptor3());

     vec.back() = boost::move(vec.front());
}
```

К сожалению, контейнеры стандартной библиотеки C++03 по-прежнему не смогут его использовать (вот почему в предыдущем примере мы использовали вектор из `Boost.Containers`).

Дополнительно...

Если вы хотите использовать `Boost.Containers` на компиляторах C++03, а контейнеры стандартной библиотеки – на компиляторах C++11, можете выполнить следующий простой трюк. Добавьте в свой проект файл заголовка следующего содержания:

```
// your_project/vector.hpp
// Указание лицензии и авторских прав

// include guards
#ifndef YOUR_PROJECT_VECTOR_HPP
#define YOUR_PROJECT_VECTOR_HPP

// Содержит макрос BOOST_NO_CXX11_RVALUE_REFERENCES
#include <boost/config.hpp>

#if !defined(BOOST_NO_CXX11_RVALUE_REFERENCES)
// rvalue-ссылки доступны
#include <vector>

namespace your_project_namespace {
  using std::vector;
} // your_project_namespace

#else
// rvalue-ссылки недоступны
#include <boost/container/vector.hpp>
```

```
namespace your_project_namespace {
  using boost::container::vector;
} // your_project_namespace

#endif // !defined(BOOST_NO_CXX11_RVALUE_REFERENCES)
#endif // YOUR_PROJECT_VECTOR_HPP
```

Теперь вы можете включить в код `<your_project/vector.hpp>` и использовать вектор из пространства имен `your_project_namespace`:

```
int main() {
    your_project_namespace::vector<descriptor_owner_movable> v;
    v.resize(10);
    v.push_back(construct_descriptor3());
    v.back() = boost::move(v.front());
}
```

См. также

- В рецепте «*Уменьшение размера кода и повышение производительности пользовательского типа в C++11*» главы 10 «Сбор информации о платформе и компиляторе» содержится дополнительная информация о `noexcept` и `BOOST_NOEXCEPT`;
- более подробную информацию о библиотеке `Boost.Move` можно найти на сайте Boost: http://www.boost.org/libs/move.

Использование алгоритмов C++14 и C++11

В C++11 есть несколько новых классных алгоритмов в заголовочном файле `<algorithm>`. В C++14 и C++17 их стало еще больше. Если вы застряли на компиляторе для стандарта, предшествующего C++11, то вам необходимо писать эти алгоритмы с нуля. Например, если вы хотите вывести ASCII-символы от 65 до 125, то должны написать следующий код:

```
#include <boost/array.hpp>

boost::array<unsigned char, 60> chars_65_125_pre11() {
    boost::array<unsigned char, 60> res;

    const unsigned char offset = 65;
    for (std::size_t i = 0; i < res.size(); ++i) {
         res[i] = i + offset;
    }

    return res;
}
```

Подготовка

Для этого рецепта требуются базовые знания C++, а также базовые знания библиотеки `Boost.Array`.

Как это делается...

Библиотека Boost.Algorithm содержит все новые алгоритмы C++11 и C++14. Используя ее, можно переписать предыдущий пример следующим образом:

```
#include <boost/algorithm/cxx11/iota.hpp>
#include <boost/array.hpp>

boost::array<unsigned char, 60> chars_65_125() {
    boost::array<unsigned char, 60> res;
    boost::algorithm::iota(res.begin(), res.end(), 65);
    return res;
}
```

Как это работает...

Как вы, наверное, знаете, в библиотеке Boost.Algorithm есть отдельный заголовочный файл для каждого алгоритма. Просто откройте нужный заголовочный файл и используйте необходимую функцию.

Дополнительно...

Скучно, когда есть библиотека, которая просто реализует алгоритмы из стандарта C++. Это не новаторство; это не в традициях Boost! Вот почему вы можете найти в Boost.Algorithm функции, которые не являются частью стандарта C++. Вот, например, функция, которая преобразует данные в шестнадцатеричное представление:

```
#include <boost/algorithm/hex.hpp>
#include <iterator>
#include <iostream>

void to_hex_test1() {
    const std::string data = "Hello word";
    boost::algorithm::hex(
        data.begin(), data.end(),
        std::ostream_iterator<char>(std::cout)
    );
}
```

Предыдущий код выводит следующее:

```
48656C6C6F20776F7264
```

Что еще интереснее, все функции в Boost.Algorithm имеют дополнительные перегрузки, которые вместо двух итераторов принимают первым параметром диапазон. **Диапазон** – это концепция от **Ranges TS**. Массивы и контейнеры с функциями .begin() и .end() соответствуют концепции диапазона. Зная это, предыдущий пример можно сократить:

```
#include <boost/algorithm/hex.hpp>
#include <iterator>
#include <iostream>
```

```
void to_hex_test2() {
    const std::string data = "Hello word";
    boost::algorithm::hex(
        data,
        std::ostream_iterator<char>(std::cout)
    );
}
```

C++17-библиотека Boost.Algorithm скоро будет расширена новыми алгоритмами и функциями C++20, такими как алгоритмы, которые можно использовать в constexpr-функциях. Следите за этой библиотекой, поскольку однажды она может стать готовым решением проблемы, с которой вы имеете дело.

См. также

- В официальной документации для библиотеки Boost.Algorithm содержится полный список функций и краткие описания для них: http://boost.org/libs/algorithm;
- экспериментируйте с новыми алгоритмами в режиме онлайн: http://apolukhin.github.io/Boost-Cookbook.