пятое издание

А. В. СТОЛЯРОВ

ВВЕДЕНИЕ В ЯЗЫК СИ++

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Учебное пособие Андрея Викторовича Столярова «Введение в язык Си++», опубликованное в издательстве МАКС Пресс в 2020 году, называемое далее «Произведением», защищено действующим российским и международным авторско-правовым законодательством. Все права на Произведение, предусмотренные законом, как имущественные, так и неимущественные, принадлежат его автору.

Настоящая Лицензия устанавливает способы использования электронной версии Произведения, право на которые предоставлено автором и правообладателем неограниченному кругу лиц, при условии безоговорочного принятия этими лицами всех условий данной Лицензии. Любое использование Произведения, не соответствующее условиям данной Лицензии, а равно и использование Произведения лицами, не согласными с условиями Лицензии, возможно только при наличии письменного разрешения автора и правообладателя, а при отсутствии такого разрешения является противозаконным и преследуется в рамках гражданского, административного и уголовного права.

Автор и правообладатель настоящим **разрешает** следующие виды использования данного файла, являющегося электронным представлением Произведения, без уведомления правообладателя и без выплаты авторского вознаграждения:

1. Воспроизведение Произведения (полностью или частично) на бумаге путём распечатки с помощью принтера в одном экземпляре для удовлетворения личных бытовых или учебных потребностей, без права передачи воспроизведённого экземпляра другим лицам;

2. Копирование и распространение данного файла в электронном виде, в том числе путём записи на физические носители и путём передачи по компьютерным сетям, с соблюдением следующих условий: (1) **все воспроизведённые и передаваемые любым лицам экземпляры файла являются точными копиями оригинального файла** в формате PDF, при копировании не производится никаких изъятий, сокращений, дополнений, искажений и любых других изменений, включая изменение формата представления файла; (2) **распространение и передача копий другим лицам производится исключительно *бесплатно*, то есть при передаче не взимается никакое вознаграждение ни в какой форме**, в том числе в форме просмотра рекламы, в форме платы за носитель или за сам акт копирования и передачи, даже если такая плата оказывается значительно меньше фактической стоимости или себестоимости носителя, акта копирования и т. п.

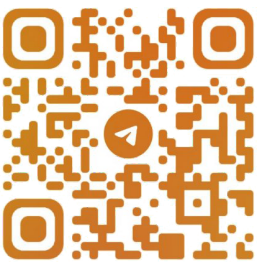
Любые другие способы распространения данного файла при отсутствии письменного разрешения автора запрещены. В частности, **запрещается**: внесение каких-либо изменений в данный файл, создание и распространение искаженных экземпляров, в том числе экземпляров, содержащих какую-либо часть произведения; распространение данного файла в Сети Интернет через веб-сайты, оказывающие платные услуги, через сайты коммерческих компаний и индивидуальных предпринимателей, а также **через сайты, содержащие рекламу любого рода**; продажа и обмен физических носителей, содержащих данный файл, даже если вознаграждение значительно меньше себестоимости носителя; включение данного файла в состав каких-либо информационных и иных продуктов; распространение данного файла в составе какой-либо платной услуги или в дополнение к такой услуге. С другой стороны, **разрешается** дарение (бесплатная передача) носителей, содержащих данный файл, бесплатная запись данного файла на носители, принадлежащие другим пользователям, распространение данного файла через бесплатные децентрализованные файлообменные P2P-сети и т. п. Ссылки на экземпляр файла, расположенный на официальном сайте автора, разрешены без ограничений.

А. В. Столяров *запрещает* Российскому авторскому обществу и любым другим организациям производить любого рода лицензирование любых его произведений и осуществлять в интересах автора какую бы то ни было иную связанную с авторскими правами деятельность без его письменного разрешения.

А. В. Столяров

ВВЕДЕНИЕ В ЯЗЫК СИ++

издание пятое,
исправленное и дополненное



Москва – 2020

УДК 519.68+004.43
ББК 32.973.26-018.1
С81

Столяров А. В.
С81 Введение в язык Си++: Учебное пособие. – 5-е изд., испр. и доп. – Москва: МАКС Пресс, 2020. – 156 с.: ил.
ISBN 978-5-317-06294-1

В пособии освещаются основы объектно-ориентированного программирования на примере языка Си++. Рассматривается только ядро языка в его классическом варианте; стандартная библиотека Си++ оставлена читателю для самостоятельного изучения. В примерах используются возможности стандартной библиотеки языка Си. Курс построен в предположении, что язык Си читателю уже известен, что позволяет излагать материал путём плавного перехода от Си к Си++ с постепенным введением новых понятий.

Для студентов программистских специальностей, преподавателей и всех желающих освоить объектно-ориентированное программирование.

Ключевые слова: язык Си++, объектно-ориентированное программирование, абстрактные типы данных.

УДК 519.68+004.43
ББК 32.973.26-018.1

Учебное издание
СТОЛЯРОВ Андрей Викторович
ВВЕДЕНИЕ В ЯЗЫК СИ++
Рисунок и дизайн обложки Елены Доменновой
Напечатано с готового оригинал-макета
Подписано в печать 10.12.2019 г.
Формат 60x90 1/16. Усл.печ.л. 9,75. Тираж 150 экз. Изд. № 292.
Издательство ООО "МАКС Пресс"
Лицензия ИД № 00510 от 01.12.99 г.
11992 ГСП-2, Москва, Ленинские горы,
МГУ им. М.В.Ломоносова, 2-й учебный корпус, 527 к.
Тел. 939-3890, 939-3891. Тел./Факс 939-3891
Отпечатано в полном соответствии с качеством
предоставленных материалов в ООО «Фотоэксперт»
115201, г. Москва, ул. Котляковская, д. 3, стр. 13.

ISBN 978-5-317-06294-1

Содержание

Предисловие

Текст этой книжки впервые увидел свет в 2008 году и трижды переиздавался — в 2011, 2012 и 2018 гг. Тираж издания 2012 года расходился медленно, последние экземпляры были распроданы лишь к 2014 году. Потом в 2015 году возник проект учебника «Программирование: введение в профессию», в который этот текст должен был войти в качестве одной из частей. К настоящему моменту вышло три тома «Введения в профессию» и планируется, что текст о языке Си++ станет частью следующего тома, четвёртого по счёту, под общим названием «Парадигмы»; проблема в том, что со сроками выхода этого тома вышла некоторая заминка. Издание 2018 года было предпринято, когда стало понятно, что текст про Си++ в достаточной степени востребован и свободно доступный электронный вариант решает проблему лишь частично (так, я неоднократно видел переплетённые распечатки); тираж разошёлся быстрее, чем можно было ожидать, а поскольку рукопись четвёртого тома «Введения в профессию» всё ещё не готова к печати, очередное переиздание «Введения в язык Си++» отдельной книжкой выглядит шагом вполне логичным.

За несколько лет, прошедших между третьим и четвёртым изданием, мир ощутимо изменился: группа международных террористов, по недоразумению называющихся комитетом по стандартизации Си++, развернула весьма бурную и эффективную деятельность по окончательному уничтожению этого языка. Вышедшие последовательно «стандарты» С++11, С++14 и, наконец, С++17 не переставали удивлять публику: каждый раз казалось, что более мрачного и безумного извращения придумать уже нельзя, и каждый раз выход очередного «стандарта» наглядно демонстрировал, что всё возможно; ожидающийся С++20 как будто специально задуман как наглядное подтверждение, что предела этому процессу нет, разве что Си++ всё-таки окочурится. Если под «языком Си++» понимать С++17 или тем паче С++20, то о применении такого инструмента на практике не может быть никакой речи, т. е. с языком Си++ следует попрощаться, устроить торжественные похороны и поискать альтернативу; впрочем, то же самое можно сказать про все его «стандарты», начиная с самого первого, принятого в 1998 году — строго говоря, язык Си++ как уникальное явление был уничтожен именно тогда.

Проблема, к сожалению, в том, что альтернативы появляться не спешат. Нельзя сказать, что в мире вообще не создаются новые языки программирования, но по каким-то неведомым причинам ни один из этих новых языков просто в силу своего исходного дизайна не может претендовать на роль заменителя Си++ и тем более чистого Си. При

этом рвение, с каким «стандартизаторы» последовательно убивают именно эти два языка программирования, заставляет подозревать, что в IT-индустрии здравый смысл как явление окончательно умер.

Примечательно здесь, пожалуй, вот что. Автору этих строк представляется очевидным, что язык Си (чистый Си) смог стать тем, чем он стал, и занять его нынешнюю нишу благодаря двум своим очевидным свойствам: во-первых, жёсткой и очевидной границе между самим языком и его библиотекой, сколь бы «стандартной» она ни была, и, во-вторых, достижимости zero runtime, т. е. возможности использования созданных компилятором объектных модулей без поддержки со стороны поставляемых с компилятором библиотек. В отсутствие любого из этих свойств язык программирования оказывается неприменим для программирования на «голом железе» (т. е. для ядер операционных систем и для прошивок микроконтроллеров) и, как следствие, не может претендовать на универсальность. Тем удивительнее, сколь упорно создатели «стандартов» пытаются изничтожить оба этих свойства, причём как в Си++, так и в чистом Си.

Подавляющее большинство «современных программистов» реально не видят принципиальной разницы между Си++ и, к примеру, тем же Питоном или Джавой, а все «новые технологии» (включая и новые «стандарты») предпочитают встречать с каким-то нездоровым восторгом, полностью отключив свои способности к критическому мышлению. К счастью, кроме этого большинства, существует также и меньшинство, думать пока не разучившееся, в противном случае мировая IT-индустрия, скорее всего, уже давно перестала бы приносить хоть какую-то пользу. Сейчас, спустя два десятка лет после получения STL'ем «официального статуса», в числе тех, кто его принципиально не использует, можно обнаружить, например, такого «монстра», как проект Qt; но, пожалуй, интереснее выглядит библиотека FLTK, в тексте которой принципиально не применяются не только возможности «стандартной библиотеки» Си++, но и шаблоны как таковые, и обработка исключений, и даже пространства имён (`namespaces`), не говоря уже о «возможностях» из новомодных «стандартов».

Судя по всему, в условиях, когда язык Си++ изувечен «стандартизаторами», адекватных альтернатив не предвидится, а программировать на чём-то всё же нужно, сознательное использование таких усечённых подмножеств остаётся последним и единственным вариантом для тех, кто сохраняет разборчивость в выборе инструментов. Мне остаётся лишь пожелать успехов на этом нетривиальном пути всем моим читателям.

Методические замечания

Если вы собираетесь использовать эту книжку, чтобы *изучать* Си++, то методические рассуждения вам вряд ли будут понятны — этот текст предназначен не для тех, кто учится, а для тех, кто учит. Конечно, читать его никто вам не запрещает; но когда вы утратите понимание, о чём вообще идёт речь — не слишком огорчайтесь и просто пропустите остаток текста до следующего заголовка.

Эта короткая книжка, посвящённая языку Си++, не похожа на другие книги о том же предмете — ни тем, как построено её содержание, ни даже своим размером. Привычный «современный» учебник по Си++ обычно представляет собой внушительный том на добрую тысячу страниц, под завязку забитый информацией, причём практически все книги на эту тему, какие можно найти в книжных магазинах, с ходу обрушивают на читателя всевозможные премудрости из так называемой стандартной библиотеки Си++, такие как `iostream`, а то и вовсе контейнеры; при этом не объясняется, что это такое и как оно сделано, потому что и перегрузка символов инфиксных операций, и шаблоны — это тема для существенно более позднего разговора. Кроме того, большинство авторов книг по Си++, судя по всему, гордятся своей способностью расписывать Си++ таким, каким его видят стандартизаторы, едва ли не раньше, чем выйдет очередной «стандарт». Напротив, в *этой* книге сознательно игнорируется и «стандартная библиотека» Си++, и все (именно так — *все*) «возможности», которые успели напридумывать вошедшие в неуместный раж авторы «стандартов».

Изменения Си++, навязываемые миру вредоносными комитетами, представляют собой не «развитие языка», а его глубочайшую деградацию; будучи в этом полностью убеждённым, я, тем не менее, хотел бы сейчас попытаться временно абстрагироваться от части своих убеждений, с которыми согласны (естественно) далеко не все. Как ни странно, совершенно не обязательно разделять моё мнение о технических стандартах, чтобы построить начальное изучение Си++ так же, как это делаю я, поскольку для отказа от раннего рассмотрения стандартной библиотеки и новшеств из «современных стандартов» есть множество иных причин.

Хотелось бы с самого начала подчеркнуть, что **между началом обучения программированию и серьёзным разговором об абстрактных типах данных и тем более об объектно-ориентированном стиле должно пройти не менее полутора–двух лет**. Начинающего программиста можно считать готовым к этому этапу, когда программы, которые он пишет (при этом непременно сам), пе-

ревалят за две-три тысячи строк[1]. Где-то там — после второй тысячи строк кода — расположен предел, на котором количество переходит в качество, нелинейно растущая сложность требует решительных мер по инкапсуляции деталей, а иерархичность предметной области — любой! — внезапно начинает выпирать из всех мыслимых щелей. Человек, никогда не писавший программ существенного объёма, просто *не поймёт, о чём идёт речь*.

К сожалению, в наше время довольно часто встречаются попытки использовать Си++ для обучения новичков с нуля. Чистый Си в «нулевых» новичков, что называется, *не лезет*, и тому есть очень простая причина: указатели. Каждый, кто хоть раз пытался оформить в мозгу обучаемого эту сущность и при этом интересовался, что в результате получается (а не просто «начитывал» материал, который превышает возможности аудитории), знает, насколько этот барьер в действительности высок. При этом в процессе изучения чистого Си операция взятия адреса требуется на первом же занятии, просто чтобы прочитать «с клавиатуры» исходные данные; но реальность такова, что объяснить человеку, который не написал ещё ни одной программы, что это за «`&x`», чем оно отличается от просто «`x`» и зачем оно понадобилось при вызове `scanf`, *невозможно*, и ни слова, ни иллюстрации, ни танцы с бубнами тут не помогут. И когда до не слишком опытного преподавателя это, наконец, доходит, Си++ начинает выглядеть заманчиво просто потому, что там не нужен `scanf`, а `iostream` никаких операций взятия адреса не требует.

Как автор книги, я искренне надеюсь, что этот случай — не ваш. Здесь и далее текст построен в предположении, что Си++ изучается на уже сформированной базе, и уж во всяком случае с чистым Си у обучаемых никаких вопросов не осталось. Для преподавателей, использующих Си++ в обучении новичков, моя книга заведомо бесполезна (как, впрочем, и любая другая); бесполезна она и для их ни в чём не повинных учеников. Должен отметить, что мне регулярно приходится исправлять последствия подобного «обучения», и в большинстве случаев с этим ничего толком не получается — обычно такие случаи безнадёжны.

Позволю себе озвучить один результат личных наблюдений, который, возможно, кому-то из учителей и преподавателей поможет пересмотреть своё отношение к происходящему. Если обучаемому, не умеющему работать с динамическими массивами и всевозможными списками на низком уровне — например, на Паскале или чистом Си — дать в руки шаблоны STL, то в качестве одного из результатов мы получаем (практически всегда) следующее утверждение: *лучше всегда использовать `vector`, а не `list`, поскольку в нём есть удобная операция индексирования, а всё остальное, вроде добавления*

[1] Каждая! Суммарный объём написанного кода должен быть намного больше, и я вынужден признать, что не готов дать его оценку.

элементов в середину, тоже прекрасно работает. Реальность такова, что объяснить разницу между `vector` и `list` человеку, не имеющему собственного практического опыта работы с указателями и низкоуровневыми динамическими данными, *невозможно* — вообще, то есть никак. Если же научить человека работать с контейнерами, то низкоуровневые структуры данных он не будет знать и уметь *никогда*, на сознательное понижение уровня абстрагирования используемых инструментов мотивации хватает единицам. Большинству новичков такое обучение закрывает путь в серьёзное программирование, оставляя возможной лишь всевозможную низкоквалифицированную деятельность вроде веб-разработки или «программирования» под бухгалтерские системы.

Если у кого-то из читателей в этом месте возник вопрос, какой же язык тогда следует использовать первым — то мой ответ будет краток: альтернативы Паскалю в этой роли нет и не предвидится. Подробно этот тезис разобран в предисловиях к первому тому «Введения в профессию».

Итак, предположим, что обучаемый тем или иным способом научился складывать отдельные синтаксические конструкции в работающие программы и, больше того, освоил программирование на чистом Си, в том числе — что очень важно — указатели, но, конечно, не только их; его программы перевалили за тот рубеж сложности, когда АТД и ООП уже не будут пустым звуком (если что-то из этого не так, продолжать чтение моей книги бессмысленно). Си++ — язык, несомненно, важный; для профессионала, претендующего на высокую квалификацию, его знание необходимо. Что же теперь?

Даже если забыть про странную последовательность изложения, характерную для «современных» учебников по Си++, останется — и никуда не денется — их объём. Будь в нашем распоряжении три-четыре *семестра* на изучение одного только Си++, мы бы, возможно, смогли выстроить некий учебный курс, охватывающий все премудрости, утрамбованные в эти гроссбухи. Проблема, однако, в том, что такого количества времени обычно просто нет.

Даже если бы времени было достаточно, всё равно лучше не начинать изучение языка Си++ с правил использования шаблонов стандартной библиотеки, как это предлагает большинство авторов во главе с самим Страуструпом. При таком обучении даже самые способные студенты не видят границы между языком и его библиотекой, в результате чего Си++ воспринимается просто как *ещё один язык высокого уровня*, каковых и без него хватает. Уникальные возможности языка Си++ остаются в тени. Когда на все эти объективные сложности накладывается ещё и нехватка времени, очень часто после прослушивания курса по Си++ такие слова, как «класс», «наследование» или тем более «полиморфизм» так и остаются для студентов пустым звуком, зато они при этом точно знают, что в начале программы надо вместо `#include <stdio.h>` написать `#include <iostream>`, потом — всенепременнейшим образом! — зага-

дочное «`using namespace std;`», вместо `printf` нужно использовать хитрые значки и слово `cout`, а всякие занудные списки можно больше не писать, потому что есть волшебное слово `list<>`.

Так или иначе, если начать изучение Си++ с контейнеров, то наиболее сложные для освоения концепции остаются для обучаемого «за кадром», у него так и не возникает понимания, что такое Си++ и зачем он нужен. Такое «обучение языку Си++» представляет собой не просто *пустую* трату учебного времени, которого и так не слишком много; результат может быть много хуже, нежели нулевой. Мне доводилось видеть претендентов на вакансию «программист на Си++», которые указывают в резюме знание Си++ и несколько лет опыта практического программирования на этом языке, а на собеседовании не могут сказать, что такое конструктор, не говоря уже о «сложных материях» вроде чисто виртуальных функций.

Представляется по меньшей мере странным тратить дефицитное время на изучение экзотических закорючек, когда его не хватает даже на объяснение наиболее фундаментальных концепций. В самом деле, изучить самостоятельно возможности `iostream`, да и контейнеры STL — гораздо проще, чем самостоятельно понять, что такое наследование и зачем оно нужно. Итак, простая мысль, которую мне хотелось бы сейчас донести до читателя, уже знающего Си++ и намеренного научить этому языку кого-то ещё, состоит в следующем. **Даже если вы не согласны с моим отношением к стандартам и их авторам, есть смысл *сначала* показать обучаемым базовые концепции Си++, АТД и ООП**, то есть именно то, что излагается в этой книге; те материи, которые я вообще не считаю достойными изучения, вы можете по крайней мере *оставить на потом*.

При взгляде на оглавление книги может возникнуть ощущение беспорядочности в выборе последовательности тем, особенно в первой половине книги. На самом деле именно такая, а не иная последовательность подачи материала имеет достаточно простое объяснение. В качестве основных целей при создании данного курса рассматривалось ознакомление студента с четырьмя темами: (1) средства инкапсуляции и описания абстрактных типов данных; (2) обработка исключительных ситуаций; (3) наследование, статический и динамический полиморфизм и (4) обобщённое программирование (шаблоны). Предлагаемые студенту новые концепции при этом сложны сами по себе, поэтому по возможности изучение специфических инструментов Си++, таких, например, как ссылки или перегрузка имён функций, откладывалось на как можно более позднее время. В результате перегрузка имён рассматривается непосредственно пе-

ред переопределением символов стандартных операций и введением нескольких конструкторов в одном классе; ссылки рассматриваются непосредственно перед введением конструктора копирования, и т. д.

Ну и последнее. При построении практических занятий хотелось бы рекомендовать использование операционной системы семейства Unix — например, Linux или FreeBSD. Обучение следует всегда начинать с простых программ, а простые программы немыслимы в условиях отсутствия культуры консольных приложений. Наличие консольных программ в Windows не решает проблему, поскольку такие программы кажутся студентам неполноценными, игрушечными, и не возникает ощущения овладения настоящим инструментом. Напротив, в ОС Unix все программы являются консольными, включая и те, которые работают с графическими окнами, так что даже самые простые программы оказываются в определённом смысле «настоящими».

Несколько слов для изучающих программирование

Книжка, которую вы читаете — пожалуй, самая короткая из книг по языку Си++ и объектно-ориентированному программированию. При этом, как можно заметить, объём пособия всё-таки остаётся значительным; материал, кое-как вместившийся в эти полторы сотни страниц, соответствует примерно шести–восьми лекциям.

Прежде всего мне хотелось бы выразить надежду, что вы не пытаетесь *начать* изучение программирования с Си++. Смею вас заверить, это всё равно не получится; если же научить азам программирования с использованием Си++ в роли первого языка вас пытается *кто-то другой* (например, учитель в школе), то эта книжка будет для вас абсолютно бесполезна; всё же попробую дать один совет — постарайтесь от такого горе-учителя сбежать, и как можно дальше, а если это невозможно — то хотя бы поберегите свои мозги.

В нормальной ситуации между началом изучения программирования и началом изучения Си++ должно пройти не меньше полутора–двух лет, а для кого-то времени может потребоваться и больше. Чтобы понять, готовы ли вы к восприятию предлагаемого здесь материала, ответьте на три вопроса:

- уверенно ли вы знаете язык Си (чистый Си, без «плюсов»)?
- умеете ли вы в Си использовать указатели для построения списков и деревьев?
- писали ли вы когда-нибудь программы объёмом 2000 строк и больше?

Если хотя бы на один из вопросов ваш ответ отличается от чёткого и уверенного «да», отложите эту книжку подальше. Кстати, вы теперь знаете, когда надо будет снова к ней вернуться. Если же вы без колебаний ответили положительно на все три вопроса, то я бы советовал вам дочитать это предисловие до конца; возможно, вы всё же предпочтёте найти какую-нибудь другую книгу для изучения Си++.

Подход к изучению Си++, на котором основана эта книжка, резко отличается от «общепринятого»; если вы попытаетесь сравнить её с любой книгой по Си++, найденной в книжном магазине, то у вас может возникнуть ощущение, что речь идёт о двух разных языках программирования.

Большинство нынешних программистов не мыслит использования Си++ в отрыве от стандартной библиотеки шаблонов (STL) и не видит ничего плохого во всё более и более изощрённых (хотя лучше будет сказать — *извращённых*) возможностях, предлагаемых выходящими с завидной регулярностью — каждые три года — «новыми стандартами». Между тем уже с появлением в 1998 году самого первого «стандарта» Си++ полное описание всех возможностей языка и его «стандартной библиотеки» превратилось в книгу такого объёма, что ею вполне можно было при желании воспользоваться в качестве оружия против врагов (в смысле чисто механическом). «Современные» версии Си++ вообще невозможно описать в одной книге; самое интересное, что делать это, пожалуй, ко всему ещё и *бесполезно*, поскольку такая книга не нашла бы подходящего читателя: начиная с С++11, получающееся у стандартизаторов *нечто* стало принципиально недоступно для изучения, так как многие возможности, втиснутые в этот «стандарт», решительно невозможно объяснить человеку, не имеющему (уже!) солидного практического опыта работы на Си++. Новичок, даже умеющий программировать на других языках, просто не поймёт, о чём идёт речь. Последовавшие за этим С++14 и С++17 превратили язык в безобразную кучу непойми чего, лишив его всякой внутренней логики, и, судя по всему, процесс этот останавливать никто не собирается — готовящийся сейчас С++20 будет намного хлеще своих предшественников.

Между тем исходно Си++ представлял собой интереснейшее явление в мире программирования — единственный в своём роде язык *произвольного* уровня, то есть такой, который позволял заниматься как низкоуровневым (почти ассемблерным), так и сколь угодно высокоуровневым программированием, причём все нужные здесь и сейчас абстракции высокого уровня можно было тут же и создать, не полагаясь на «доброго дядю». Логику построения языка (в *той* его версии) нельзя было назвать совсем безупречной, но она хотя

бы существовала, и, более того, на все её недочёты можно было не обращать внимания как на нечто не слишком важное.

В «современном» Си++ всего этого уже давно не видно под горой семантического хлама. Следует подчеркнуть, что изначальный Си++ никуда не делся, его возможности по-прежнему используются (хоть в реализации той же пресловутой стандартной библиотеки) — но уловить существовавшую когда-то концептуальную структуру практически невозможно, слишком много чужеродных возможностей этому мешает.

В этой книжке Си++ описан практически таким, каким он был когда-то давно, до того, как за него принялись безответственные разработчики «стандартов». Несомненно, знакомство с материалом книги никак не сможет вам помешать в будущем освоить и STL, и «новые стандарты»; но время, потраченное на этот материал, совершенно точно не пропадёт зря — понимание глубинной сути Си++ придаст вам уверенности и позволит к изучению «современного» Си++ подойти осознанно, понимая, что происходит.

А теперь самое, пожалуй, важное. Все усилия ваших преподавателей и ваши собственные пропадут впустую, если вы не свыкнетесь с одной простой мыслью: **нет и не может быть иного способа изучения программирования, чем САМОСТОЯТЕЛЬНОЕ написание программ на компьютере**. Смотреть, как пишут другие, пытаться разбираться в чужих программах — занятие совершенно бессмысленное. Точно так же бессмысленно писать программы под чью-то диктовку. Программа будет вашей только в том случае, если вы напишете её самостоятельно — возможно, заглядывая в справочники, но без помощи других людей.

Домашняя страница этой книги в сети Интернет расположена по адресу `http://www.stolyarov.info/books/cppintro`. Здесь вы можете получить тексты примеров программ, приведённых в этой книге, а также электронную версию самой книги.

1. Введение

1.1. Парадигмы программирования, ООП и АТД

Становление концепции ***объектно-ориентированного программирования*** обычно связывают с языком Симула-67, который предназначался для имитационного моделирования; сам термин «объектно-ориентированное программирование» впервые появился несколько позже — в середине 70-х годов XX столетия в проекте Smalltalk, который также известен другим новшеством — оконным интерфейсом пользователя.

Языки программирования, с которыми вам приходилось встречаться до сих пор, относятся к категории императивных языков программирования. Программа на императивных языках воспринимается как последовательность команд, изменяющих значения переменных и производящих другие действия (отсюда название парадигмы «императивное программирование», от слова «императив» — приказ, команда). Помимо императивного программирования, существуют такие парадигмы, как логическое программирование, где программа воспринимается как набор логических высказываний, а выполнение — как доказательство или опровержение некоторого высказывания; функциональное программирование, где программа представляется как набор математических функций, а исполнение программы представляет собой вычисление некоторой главной функции. Сам термин «парадигма программирования» означает способ восприятия человеком (программистом) компьютерной программы и её процесса исполнения. Надо заметить, что парадигмы относятся скорее к мышлению программиста, чем к средствам языков программирования, но язык при этом может стимулировать, поощрять, а в некоторых случаях — и навязывать определённый стиль мышления, поэтому, конечно, игнорировать изобразительные средства избранного языка программирования при разговоре о парадигмах было бы неправильно.

Объектно-ориентированное программирование представляет собой ещё одну парадигму программирования. Осмысливая свою программу в соответствии с этой парадигмой, мы прежде всего представляем данные в виде некоторых ***объектов*** — «чёрных ящиков». Внутреннее устройство объекта извне недоступно (и в ряде случаев может быть просто неизвестно — например, если данный объект реализован другим программистом). Всё, что можно сделать с объектом — это послать ему сообщение и получить ответ. Так, операция $2 + 3$ в терминах объектно-ориентированного программирования выглядит как «мы посылаем двойке сообщение *прибавь к себе тройку*,

а она отвечает нам, что получилось 5». Вполне возможно, что, получив сообщение, объект произведёт ещё и какие-то действия, сменит своё внутреннее состояние или пошлёт сообщение другому объекту.

Объекты, внутреннее устройство которых одинаково, образуют ***классы***. В прагматическом смысле класс — это описание внутреннего устройства объекта; при создании нового объекта указывается, к какому классу он принадлежит (объектом какого класса он будет являться). Имея описание класса, можно создать произвольное количество[2] объектов этого класса. Более того, можно взять имеющийся класс и на его основе создать новый класс, в чём-то похожий на старый и проявляющий его свойства, но при этом всё-таки отличающийся от него, как правило, в сторону усложнения (хотя и не всегда). Эта возможность называется ***наследованием***.

Недоступность деталей реализации объекта за пределами его описания позволяет снизить общую сложность программы по принципу «разделяй и властвуй». В применении к программированию этот принцип называется ***инкапсуляцией*** и состоит в разделении всей программы на подсистемы, каждая из которых реализуется более-менее независимо и может разрабатываться без учёта деталей реализации других подсистем; учитывать приходится лишь то, как подсистемы «выглядят извне», то есть какие функции и прочие возможности в них предусмотрены для взаимодействия с другими подсистемами. Конечно, инкапсуляция присутствует не только в объектно-ориентированных языках, но если в других языках основной единицей инкапсуляции обычно является *модуль*, то в ООП мы используем инкапсуляцию на уровне отдельных объектов или (как в Си++) классов, что в ряде случаев позволяет резко снизить общую сложность многих модулей.

Часто бывает так, что несколько различных классов способны обрабатывать некоторый общий для них всех набор сообщений. В таком случае говорят, что эти классы поддерживают общий ***интерфейс***; во многих объектно-ориентированных языках программирования интерфейсы описываются в виде классов специального вида, есть такая возможность и в Си++.

С объектно-ориентированным программированием часто (и безосновательно) смешивают другую парадигму программирования — ***абстрактные типы данных***. *Абстрактным* называют такой тип данных, для которого неизвестна его внутренняя организация, а известен лишь некий набор базовых операций; именно так мы воспринимаем, например, тип `FILE`, вводимый стандартной библиотекой Си

[2]Строго говоря, иногда можно встретить класс, объект которого может существовать только в единственном экземпляре, и даже такие классы, для которых вообще нельзя создавать объекты, но это скорее исключение из правил.

для высокоуровневой работы с файлами — мы знаем, что можно описать переменную типа `FILE*` (т. е. указатель на `FILE`), что с ней можно работать, используя функции `fopen`, `fclose`, `fputc`, `fgetc` и т. п., но при этом нас не интересует, что́ в действительности представляет собой тип `FILE`. Если подумать, можно обнаружить, что к абстрактным относятся встроенные типы языка Си для обработки чисел с плавающей точкой — `float`, `double` и `long double`, поскольку в языке Си нет никаких средств, зависящих от конкретного представления «плавучих» чисел. В то же время целочисленные типы отнести к абстрактным не получится, ведь для них язык предоставляет побитовые операции, то есть внутреннее устройство целых чисел на уровне языка зафиксировано и доступно.

Путанице между ООП и АТД, несомненно, способствует наличие в обеих парадигмах неких требований «закрытости внутреннего устройства», но сходство на этом заканчивается. При работе с абстрактными типами данных не идёт никакой речи ни об «обмене сообщениями», ни о «внутреннем состоянии», ни тем более о наследовании или загадочном «полиморфизме». В большинстве случаев применение АТД не приводит к серьёзным изменениям в восприятии программы и её выполнения, господствующая парадигма остаётся традиционной — императивной, тогда как правильный подход к ООП заставляет программиста полностью поменять стиль мышления.

Ключевое различие между объектами в смысле ООП и абстрактными данными можно проиллюстрировать тем фактом, что абстрактные типы данных можно присваивать[3] и это никак не противоречит избранной модели мышления, тогда как в «чистом» ООП присваивание оказывается чем-то чужеродным. В самом деле, обычных переменных «чистое ООП» не предусматривает, там существуют только объекты; присваивание (безотносительно конкретного языка программирования) есть не что иное как указание сделать один объект таким же, как некий другой, то есть, присвоив один объект другому, мы будем точно знать, что теперь внутреннее состояние этих двух объектов одинаково, но ведь ООП запрещает какие-либо предположения о внутреннем состоянии объектов!

Надо признать, впрочем, что даже в таких объектно-ориентированных «до мозга костей» языках, как Smalltalk и Eiffel, присваивание всё же присутствует.

В языке Си++ представлены средства как для ООП, так и для создания АТД, причём сложно сказать, какая из этих двух парадигм поддержана «лучше». Усугубляет запутанное положение ещё и тот факт, что для обеспечения «закрытости» (как объектов, так и АТД) Си++ вводит единый инструмент — так называемую *защиту*,

[3]Из этого правила тоже есть исключения. Например, файловые переменные языка Паскаль не присваиваются; впрочем, остаётся открытым вопрос, можно ли считать эту сущность абстрактным типом данных.

и вдобавок на происходящее накладывает заметный отпечаток исходная императивная сущность чистого Си. К тому же ООП и АТД не исчерпывают новшеств, введённых в Си++ по сравнению с чистым Си; так, *обработка исключений* вообще никакого отношения к ООП не имеет, и у многих программистов этот факт вызывает недоверчивое удивление, поскольку в Си++ исключения тесно связаны с объектами и наследованием. Между ООП и «всем остальным» язык Си++ сам по себе никакой чёткой границы не проводит. Встречаются программисты, вообще не умеющие использовать ООП, но при этом уверенные, что то, что они делают — это ООП и есть.

Мы постараемся своевременно напоминать читателю о том, что в действительности абстрактные типы данных — это ещё не ООП. Сразу же оговоримся, что абстрактные типы данных сами по себе достаточно полезны, и эта польза не станет меньше от того, что их путают с объектами; но умение мыслить в терминах объектов и сообщений, то есть именно то, что составляет суть объектно-ориентированного программирования, тоже очень важно, в особенности при создании больших и сложных программных систем. Поэтому будет уместно посоветовать читателю регулярно вспоминать про объекты и сообщения, пока мышление в этих терминах не станет для него привычным делом.

1.2. От Си к Си++

Язык Си++ был предложен Бьёрном Страуструпом в начале восьмидесятых годов прошедшего столетия в качестве ответа на назревшую потребность в индустриальном объектно-ориентированном языке. Си++ представляет собой расширение языка Си; добавленные в язык инструменты обеспечивают поддержку как для объектно-ориентированного программирования, так и для целого ряда других парадигм — абстрактных типов данных, обобщённого программирования, обработки исключительных ситуаций и т. д.

Программы, написанные на чистом Си, в большинстве случаев будут корректны с точки зрения языка Си++ — но не всегда. Так, в Си++ присутствует целый ряд ключевых слов, которых не было в Си. Поэтому, например, программу, содержащую такое описание:

```
int try;
```

транслятор Си++ сочтёт ошибочной, ведь слово `try` — ключевое в Си++. Кроме того, в языке Си++ нет отдельного пространства имён для тегов структур; описание

```
struct mystruct {
```

```
    int a, b;
};
```

в программе на Си++ является описанием полноценного *типа* `mystruct`, тогда как на Си такое же описание означало бы лишь введение *тега структуры*. Иначе говоря, в Си++ мы можем описать переменную типа `mystruct`:

```
mystruct s1;
```

а в чистом Си нам приходилось использовать что-то вроде

```
struct mystruct s1;
```

(впрочем, Си++ поймёт и такое). С другой стороны, часто встречающиеся в старых программах на Си описания вида

```
typedef struct mystruct {
    int a, b;
} mystruct;
```

в программе на Си++ вызовут ошибку из-за возникающего конфликта имён; в языке Си такого конфликта не возникало, т. к. теги структур и имена типов относились к различным пространствам имён.

В ряде ситуаций компилятор Си++ ведёт себя строже, чем компилятор чистого Си, выдавая, например, ошибку там, где в Си возникло бы как максимум предупреждение. К таким ситуациям относятся, во-первых, вызовы необъявленных функций: в Си эта ситуация считается «нехорошей, но допустимой», так что, если забыть подключить какой-нибудь заголовочный файл, программа может благополучно оттранслироваться, хотя и с предупреждениями. В Си++ этот номер не проходит: если функция не описана, её вызов рассматривается как ошибка. Во-вторых, ошибкой будет неявное преобразование адресных выражений несовместимых типов: например, присваивание выражения типа `int*` переменной типа `double*` вызовет ошибку в Си++, тогда как в чистом Си выдавалось только предупреждение. Ошибкой в Си++ будет и неявное преобразование выражения типа `void*` в выражение другого адресного типа, в то время как компилятор чистого Си не выдаёт в такой ситуации даже предупреждения.

В чистом Си для обозначения «нулевого указателя», т. е. специального адресного значения, заведомо не соответствующего никакому адресу в оперативной памяти, используется макрос `NULL`. В Си++ на уровне спецификаций закреплено, что целочисленная константа «`0`» используется как для обозначения целого числа «ноль», так и для «нулевого указателя», причём вне всякой зависимости от того,

каково на данной архитектуре соответствующее «арифметическое» значение адреса. Конечно, макрос `NULL` использовать никто не запрещает (хотя бы из соображений совместимости с Си); правильнее будет сказать, что использовать его *не принято*.

В сравнении с чистым Си язык Си++ имеет ряд дополнительных возможностей, которые можно заметить ещё до введения средств, имеющих отношение к ООП или АТД. Отметим некоторые из них.

Для логических значений в Си++ введён специальный тип `bool`, состоящий из двух значений: `false` и `true`. Конечно, целые числа по-прежнему можно использовать в роли логических значений; более того, целому можно присвоить значение типа `bool`, при этом `false` превратится в `0`, а `true` — в `1`.

Символьные литералы вроде `'a'` или `'7'` в Си++ считаются константами типа `char`, тогда как в чистом Си они считались константами типа `int`.

Описание (переменных и типов) стало в Си++ ***оператором***, что позволяет вставлять его в произвольное место программы, а не только в начало блока; напомним, что в чистом Си нельзя описать переменную после того, как в блоке встретился хотя бы один оператор. Более того, в Си++ переменную можно описать *в заголовке цикла* `for`, примерно так:

```
for (int i = 0; i < 10; ++i)
```

Наряду с привычными «блочными» комментариями, которые заключаются в знаки «`/*`» и «`*/`», в Си++ введены ***строчные комментарии***, обозначаемые знаком «`//`»; компилятор проигнорирует весь текст, написанный после такого знака, вплоть до ближайшего символа перевода строки. Есть и другие отличия, о которых вы постепенно узнаете.

2. Методы, объекты и защита

В этой главе мы попытаемся показать читателю путь постепенного перехода от традиционного императивного стиля мышления к мышлению в терминах объектов и методов. Для этого мы, отталкиваясь от уже известных читателю средств чистого Си, шаг за шагом будем добавлять понятия и изобразительные средства, пришедшие из области объектно-ориентированного программирования, пока не придём к полноценному ***классу***, имеющему ***конструктор***, ***деструктор*** и ***методы***.

2.1. Функции-члены (методы)

Для поддержки объектно-ориентированного программирования в Си++ вводятся понятия ***функций-членов*** и ***классов***. О классах мы расскажем позднее, а пока попытаемся понять, что же такое «функция-член». Пусть нам потребовалась структура для представления комплексного числа через действительную и мнимую части:

```
struct str_complex {
    double re, im;
};
```

Введём теперь операцию вычисления модуля комплексного числа. На языке Си нам пришлось бы описать примерно такую функцию:

```
double modulo(struct str_complex *c)
{
    return sqrt(c->re*c->re + c->im*c->im);
}
```

Язык Си++ позволяет сделать то же самое несколько более элегантно, подчеркнув непосредственное отношение функции `modulo` к сущности комплексного числа. Функцию мы внесём *внутрь* структуры, сделав «как будто бы её частью»:

```
struct str_complex {
    double re, im;
    double modulo() { return sqrt(re*re + im*im); }
};
```

Обратите внимание, что в теле функции встречаются обращения к полям структуры прямо по именам, без уточнения, откуда такое имя берётся, как если бы это были простые переменные, а не поля. Теперь мы можем написать, например, такой код:

```
    str_complex z;
    double mod;
    z.re = 2.7;
    z.im = 3.8;
    mod = z.modulo();
```

Функция `modulo` называется *функцией-членом* или ***методом*** структуры `str_complex`. Иногда речь идёт о *методе объекта* (в данном случае — объекта `z`), при этом имеется в виду метод того типа, к которому принадлежит объект. Как видно из примера, метод вызывается не сам по себе, а *для конкретного объекта*, и как раз из этого

объекта берутся поля, когда в теле метода имеет место обращение к полю структуры по имени. В данном случае метод вызван для объекта `z`, так что в его теле упоминания идентификаторов `re` и `im` будут соответствовать полям `z.re` и `z.im`.

Вызов метода — это именно то, что в теории объектно-ориентированного программирования понимается под *отправкой сообщения объекту*. Иначе говоря, **термины «вызов метода» и «передача сообщения» являются синонимами**. Мы можем, следовательно, сказать, что в последней строчке вышеприведённого фрагмента кода мы *передали объекту* `z` *сообщение* `modulo`, означающее «посчитай свой модуль»; объект ответил на наше сообщение, причём в ответе содержалось число типа `double`, равное искомому модулю; получив этот ответ, мы присвоили его переменной `mod`.

2.2. Неявный указатель на объект

На самом деле вызов функции-члена (метода) объекта с точки зрения реализации представляет собой абсолютно то же самое, что и вызов обычной функции, первым параметром которой является адрес объекта. В частности, в предыдущем параграфе мы заменили внешнюю функцию `modulo`, получавшую адрес структуры, на метод `modulo`, описанный внутри структуры, а вызов с явной передачей адреса — на вызов метода для объекта; при этом *на уровне машинного кода никаких изменений не произошло*. Говорят, что функциям-членам при вызове их для объекта передаётся ***неявный параметр*** — адрес объекта, для которого функция вызывается.

К этому параметру при необходимости можно обратиться; именем ему служит ключевое слово `this`. Можно воспринимать `this` как локальную константу, имеющую тип `A*`, где `A` — имя описываемой структуры. Например, описанную в предыдущем параграфе версию структуры можно было бы переписать и так:

```
struct str_complex {
    double re, im;
    double modulo() {
        return sqrt(this->re*this->re + this->im*this->im);
    }
};
```

В данном конкретном случае это не имеет особого смысла, однако бывают и такие ситуации, в которых использование `this` оказывается необходимым; достаточно представить себе, что из метода нужно вызвать какую-либо функцию (обычную или метод другого объекта), аргументом которой должен стать как раз наш собственный объект, тот, для которого нас вызвали.

2.3. Защита. Понятие конструктора

Чтобы стать полноценным объектом, нашей структуре не хватает свойства закрытости. В самом деле, поля `re` и `im` доступны из любого места в программе, где доступна сама структура `str_complex`. Для того, чтобы скрыть детали реализации объекта, в языке Си++ введен механизм ***защиты***, который позволяет запретить доступ к некоторым частям структуры (как полям, так и функциям-методам) из любых мест программы, кроме тел функций-методов.

Для поддержания этого механизма в язык введены ключевые слова `public` и `private`, которыми в описании структуры могут быть помечены поля и методы, доступные извне структуры (`public:`) и, наоборот, доступные только из тел функций-методов (`private:`)[4]. Попытаемся переписать нашу структуру с использованием этих ключевых слов:

```
struct str_complex {
private:
    double re, im;
public:
    double modulo() { return sqrt(re*re + im*im); }
};
```

Теперь поля `re` и `im` доступны только из тела метода `modulo`. Легко заметить, однако, что пользоваться такой структурой мы не сможем, ведь в ней не предусмотрено никаких средств для задания значений полей `re` и `im`. Так, фрагмент кода, приведённый на стр. 20, попросту будет отвергнут компилятором, ведь поля теперь недоступны, в том числе и для присваивания. Решить проблему можно, введя соответствующий метод для задания значений полей. Описание нашей структуры могло бы выглядеть в этом случае, например, так:

```
struct str_complex {
private:
    double re, im;
public:
    void set(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double modulo()
        { return sqrt(re*re + im*im); }
};
```

а пример использования объекта мы теперь перепишем так:

```
    str_complex z;
```

[4]Несколько позже мы введём ещё и ключевое слово `protected`.

```
double mod;
z.set(2.7, 3.8);
mod = z.modulo();
```

Это решение имеет очень серьёзный недостаток[5]. Дело в том, что с момента объявления переменной `z` до вызова функции `set` наш объект (переменная `z`) оказывается в *неопределённом* состоянии, т. е. попытки его использовать будут заведомо ошибочны. Программист должен будет об этом помнить, что, помимо прочего, противоречит базовому принципу ООП: вне объекта не следует строить предположений о его внутреннем состоянии, и для «неопределённого» состояния вряд ли стоит делать исключение. Очевидно, было бы лучше произвести инициализацию объекта непосредственно в момент его создания. Более того, было бы логично *запретить* создание объекта в обход инициализации.

Язык Си++ позволяет задавать компилятору чёткие инструкции относительно действий, необходимых[6] при инициализации объекта. Для этого вводится ещё одно важное понятие — ***конструктор объекта***. Конструктор — это функция-метод специального вида, которая может, как и обычная функция, иметь параметры или не иметь их; тело этой функции как раз и представляет собой порядок действий, которые необходимо выполнить всякий раз, когда создаётся объект описываемого типа. Написав конструктор, мы тем самым «объясняем» компилятору, как — в соответствии с какой инструкцией — создавать новый объект данного типа, какие параметры должны быть для этого заданы и как воспользоваться значениями этих параметров.

Компилятор отличает конструкторы от обычных методов по имени, которое совпадает с именем описываемого типа (в данном случае структуры). Поскольку конструктор играет специальную роль и в явном виде обычно не вызывается[7], тип возвращаемого значения для конструктора указывать нельзя, он не возвращает никаких значений; в каком-то смысле результатом работы конструктора является сам объект, для которого его вызвали.

Проиллюстрируем сказанное, переписав структуру `str_complex`:

```
struct str_complex {
private:
```

[5] Более того, большинство компиляторов выдаст предупреждение при попытке компиляции такого объявления структуры.

[6] «Необходимых» в том числе и в буквальном смысле — то есть таких, которые *нельзя обойти*.

[7] На самом деле Си++ позволяет вызвать конструктор явным образом, но если вам это потребовалось — скорее всего вы делаете что-то очень странное; мы рассматривать эту возможность не будем.

```
    double re, im;
public:
    str_complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double modulo()
        { return sqrt(re*re + im*im); }
};
```

Введя конструктор (функцию-метод с именем `str_complex`), мы сообщили компилятору, что для создания объекта типа `str_complex` нужно знать два числа типа `double`, и задали набор действий, подлежащих выполнению при создании такого объекта. В этом наборе действий говорится в том числе и о том, как воспользоваться заданными числами типа `double`: первое из них использовать в качестве действительной части, второе — в качестве мнимой части создаваемого комплексного числа.

Вообще в семантике Си++ любая переменная создаётся с помощью конструктора. Во многих случаях компилятор считает конструктор существующим («неявно»), несмотря на то, что в программе конструктор не описан.

Код вычисления модуля теперь можно переписать вот так:

```
str_complex z(2.7, 3.8);
double mod;
mod = z.modulo();
```

Более того, для такой операции нам не обязательно описывать переменную, имеющую имя. Мы могли бы написать и так:

```
double mod = str_complex(2.7, 3.8).modulo();
```

Здесь мы создали ***анонимную переменную*** типа `str_complex` и для этой переменной вызвали метод `modulo`. Анонимным переменным мы позже посвятим отдельный параграф.

Сделаем важное замечание. После введения конструктора, имеющего параметры, компилятор откажется создавать объект типа `str_complex` без указания требуемых конструктором двух значений, то есть предыдущая версия кода, содержавшая описание

```
str_complex z;
```

компилироваться больше не будет. Если это создаёт неудобства, можно заставить компилятор снова считать такие объявления корректными; о том, как это делается, мы узнаем из §3.3.

Синтаксис создания переменной по заданному параметру допустим в Си++ и для переменных встроенных типов. Так, описание

```
int v(7);
```

означает абсолютно то же самое, что и привычное по языку Си описание переменной с *инициализатором*

```
int v = 7;
```

Напомним на всякий случай, что ***инициализация*** — это совсем не то же самое, что ***присваивание***: при присваивании мы меняем значение для переменной, которая уже существует, тогда как при инициализации — задаём начальное значение для переменной, которая прямо сейчас появится и которой до этого момента не было. В чистом Си инициализация и присваивание обозначаются одним и тем же символом — знаком равенства, но разница между ними становится очевидной, если вспомнить, что мы можем, например, задать начальное значение для массива, тогда как *присваивать* массивы нельзя. Кроме того, стоит вспомнить, что для глобальных переменных инициализация не требует никаких действий, выполняемых процессором, ведь при старте программы в память загружается образ, уже содержащий нужные значения; присваивание же, очевидно, всегда представляет собой именно действие.

В Си++ разница между инициализацией и присваиванием ещё более существенна, и мы в этом неоднократно убедимся. В частности, важно понимать, что конструкторы используются при инициализации и не имеют никакого отношения к присваиванию. Как мы узнаем из дальнейшего изложения, Си++ позволяет взять под контроль не только инициализацию объекта, но и присваивание ему значений, но для этого предусмотрены отдельные средства, конструкторы тут ни при чём.

Возвращаясь к обсуждению защиты, отметим ещё один немаловажный момент. **Единицей защиты в Си++ является не объект, а тип (в данном случае структура) целиком**. Это означает, что из тел методов мы можем обращаться к закрытым полям не только «своего» объекта (того, для которого вызван метод), но и вообще любого объекта того же типа.

2.4. Зачем нужна защита

Смысл механизма защиты часто оказывается непонятен программистам, начинающим осваивать объектно-ориентированное программирование. Попробуем пояснить его, основываясь на нашем примере.

Представим себе, что наша структура `str_complex` используется в большой программе, активно работающей с комплексными числами. Может случиться так, что в программе будет очень часто требоваться вычисление модулей комплексных чисел. Более того, может оказаться и так, что именно модуль комплексного числа требуется

нам даже чаще, чем его действительная и мнимая части. В такой ситуации, скорее всего, наша программа будет проводить значительную часть времени своего выполнения в вычислениях модулей. Обнаружив это, мы можем прийти к выводу, что в данной конкретной задаче удобнее хранить комплексные числа в полярных координатах, а не в декартовых, то есть в виде модуля и аргумента, а не в виде действительной и мнимой частей.

Если мы не применяли защиту, то, скорее всего, все модули нашей программы, в которых используются комплексные числа, содержат обращения к полям `re` и `im`. Если в такой программе изменить способ хранения комплексного числа, убрав поля `re` и `im` и введя вместо них, скажем, поля `mod` и `arg` для хранения модуля и аргумента (т. е. полярных координат), то все части программы, использовавшие нашу структуру, перестанут компилироваться и нам придётся их исправлять. Если программа достаточно большая (а современные программные проекты часто состоят из многих сотен и даже тысяч модулей), такое редактирование может потребовать существенных трудозатрат, приведёт к внесению в программу новых ошибок и т. п., так что, вполне возможно, нам придётся отказаться от изменений, несмотря на всю их полезность.

Допустим теперь, что мы использовали защиту и сделали все поля недоступными откуда бы то ни было, кроме методов нашей структуры. Конечно, при использовании комплексных чисел часто бывает нужно узнать отдельно действительную и мнимую части числа, но для этого можно ввести специальные методы; для полноты картины введём заодно функцию вычисления аргумента, которая будет использовать стандартную функцию `atan2` для определения соответствующего арктангенса:

```
struct str_complex {
private:
    double re, im;
public:
    str_complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double get_re() { return re; }
    double get_im() { return im; }
    double modulo() { return sqrt(re*re + im*im); }
    double argument() { return atan2(im, re); }
};
```

К полям `re` и `im` теперь не могут обращаться никакие части программы, кроме методов структуры, так что при переходе от хранения действительной и мнимой части к хранению полярных координат придётся переписывать только наши собственные методы, а весь

остальной текст программы, из скольких бы модулей он ни состоял и что бы ни делал с комплексными числами, сохранится без изменений и будет работать, как и раньше. Конечно, методы `get_re` и `get_im` теперь станут гораздо сложнее, чем были, зато упростятся методы `modulo` и `argument`:

```
struct str_complex {
private:
    double mod, arg;
public:
    str_complex(double re, double im) {
        mod = sqrt(re*re + im*im);
        arg = atan2(im, re);
    }
    double get_re() { return mod * cos(arg); }
    double get_im() { return mod * sin(arg); }
    double modulo() { return mod; }
    double argument() { return arg; }
};
```

В такой ситуации мы даже можем себе позволить иметь две реализации структуры `str_complex`, между которыми выбор осуществляется директивами условной компиляции. Можно будет, не меняя текста программы, откомпилировать её с использованием одной реализации, измерить быстродействие, откомпилировать программу с использованием другой реализации, снова измерить быстродействие, сравнить результаты измерений и принять решение, какую реализацию использовать. В случае, если используемые в программе алгоритмы изменятся и нам снова станет выгодно хранить комплексные числа в декартовых координатах, то для возврата к старой модели вообще не придётся ничего редактировать.

Более того, мы можем в некий момент решить, что и те, и другие вычисления производить довольно накладно, а памяти нам не жалко, и начать хранить в объекте как декартово, так и полярное представление числа:

```
struct str_complex {
private:
    double re, im, mod, arg;
public:
    str_complex(double a_re, double a_im) {
        re = a_re; im = a_im;
        mod = sqrt(re*re + im*im);
        arg = atan2(im, re);
    }
    double get_re() { return re; }
```

```
    double get_im() { return im; }
    double modulo() { return mod; }
    double argument() { return arg; }
};
```

Теперь в нашем объекте четыре поля, причём любые два из них можно вычислить, пользуясь значениями двух других, то есть значения наших полей не могут быть произвольными, а должны всегда находиться в некотором соотношении: в данном случае пары (`re`,`im`) и (`mod`,`arg`) должны задавать одно и то же комплексное число, хотя и разными способами. В таких случаях говорят, что хранимая информация ***избыточна***; в этом нет совершенно ничего страшного, если только мы можем обеспечить ***целостность*** информации, то есть гарантировать, что в нашей программе значения рассматриваемых полей всегда будут находиться в зафиксированном для них соотношении. Для обычной (открытой) структуры нам было бы тяжело это гарантировать, поскольку в любом месте программы мог бы появиться оператор, изменяющий только некоторые из взаимосвязанных значений и не изменяющий другие; если программа имеет существенный объём, и в особенности если над ней работают несколько программистов, уследить за правильностью использования такой структуры может оказаться проблематично. Когда же все поля закрыты, доступ к ним имеют только методы описываемой структуры, обычно находящиеся в рамках одного модуля. Объём такого модуля, как правило, невелик в сравнении со всей программой, ну а другие программисты либо вовсе не станут редактировать «чужой» модуль, либо, если такая необходимость всё же возникнет, для начала разберутся, как этот модуль должен функционировать.

Очень важно понять, что защита в языке Си++ предназначена не для того, чтобы защищать нас от врагов, но исключительно для защиты нас от самих себя, от собственных ошибок. Если задаться целью обойти механизм защиты, это можно сделать без особых проблем путём преобразования типов указателей или другими способами. Защита работает только в случае, если программисты не предпринимают целенаправленных действий по её обходу. Но обычно программисты этого не делают, поскольку понимают полезность механизма защиты и выгоды от его использования, а также и то, что попытки его обойти, скорее всего, приведут к внесению в программу ошибок.

Как уже говорилось, сокрытие деталей реализации некоторой части программы от всей остальной программы называется ***инкапсуляцией***. Защита, будучи непременным свойством объектно-ориентированного программирования, позволяет проводить инкапсуля-

цию на более глубоком уровне. Именно поэтому инкапсуляцию часто называют одним из «трёх китов» ООП.

2.5. Классы

Поскольку большинство внутренних полей объектов обычно закрыты, для описания объектов в Си++ используют специально введённый для этой цели тип составных переменных, называемый ***классом***. Класс — это тип переменной, напоминающий запись (структуру), но отличающийся тем, что к полям (членам) класса доступ по умолчанию есть только из методов самого этого класса.

Перепишем нашу реализацию комплексного числа в виде класса:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double get_re() { return re; }
    double get_im() { return im; }
    double modulo() { return sqrt(re*re + im*im); }
    double argument() { return atan2(im, re); }
};
```

Всё, что описано в классе до слова `public:`, компилятор рассматривает как детали реализации класса и запрещает доступ к ним отовсюду, кроме тел функций-членов данного класса. Слово `public:` меняет режим защиты с закрытого на открытый, т. е. всё, что описано после этого слова (в данном случае это функции-члены `Complex`, `modulo` и др.) будет доступно во всём тексте программы.

Важно помнить, что **модель защиты, включаемая по умолчанию — это единственное отличие классов от структур**. Больше они ничем друг от друга не отличаются, по крайней мере, с точки зрения компилятора Си++. Тем не менее, обычно программисты используют структуры для случаев, когда по смыслу поля должны быть открыты и доступны (например, при организации списков), а классы — для случаев, когда поля представляют собой детали реализации объекта, которые лучше всего скрыть.

2.6. Деструкторы

Наряду с конструкторами, контролирующими процесс *создания* (инициализации) объектов, в языке Си++ предусмотрены также де-

структоры, предназначенные для контроля над процессом *уничтожения* объекта.

Представим себе, что объект в ходе своей деятельности захватывает некий ресурс — например, открывает файл, выделяет динамическую память и т. п., причём об этом известно только самому объекту, т. е. на захваченный ресурс ссылаются только закрытые поля объекта (в примере с файлом это может означать, что дескриптор открытого файла хранится в закрытом поле объекта). Заметим теперь, что объект может в любой момент (неожиданно для себя) прекратить существование. Так, если объект — обычная локальная переменная в функции, то она исчезнет при возврате управления из этой функции; если создать объект в динамической памяти, то он перестанет существовать при освобождении этой памяти; даже если объект сделать глобальной переменной, то завершение программы тоже никто не отменял, а в программировании изредка встречаются и такие ресурсы, которые не освобождаются автоматически, даже когда перестаёт существовать программа, захватившая их. Но ведь о том, что наш объект что-то там захватил, *никто кроме него не знает*! Как следствие, только сам объект может освободить всё, что он себе забрал, так что ему жизненно необходима возможность «привести свои дела в порядок» перед исчезновением, когда бы и по каким бы причинам это исчезновение ни случилось.

Именно для этого предназначаются деструкторы. ***Деструктор*** — это метод класса, вызов которого автоматически вставляется компилятором в код в любой ситуации, когда объект прекращает существование. Функция-деструктор имеет имя, представляющее собой имя описываемого типа (класса или структуры), к которому спереди добавлен знак `~` (тильда); например, для класса `A` деструктор будет называться `~A`. Список параметров функции-деструктора всегда пуст, т. к. в языке отсутствуют средства передачи параметров деструктору. Деструктор играет специальную роль и, как и конструктор. в явном виде обычно не вызывается[8]; тип возвращаемого значения для деструктора не указывается — деструктор никогда не возвращает никаких значений.

Проиллюстрируем понятие деструктора на примере класса `File`, инкапсулирующего дескриптор файла[9]:

[8] В сноске 7 на стр. 7 мы упоминали, что Си++ позволяет вызвать конструктор явно, но делать этого не надо. То же самое относится и к деструкторам.

[9] Здесь и далее мы снабжаем примеры программного текста комментариями на русском языке. В учебном пособии мы можем себе позволить такую вольность, но помните, что **в текстах реальных программ допустим только один язык — английский**, а русские буквы, как и вообще любые символы, не входящие в набор ASCII, в программах встречаться не должны, и к комментариям это тоже относится.

```
class File {
    int fd; // Дескриптор. -1 означает, что файл не открыт
public:
    File() { fd = -1; }
        // На момент создания объекта мы ещё ничего не открыли

    bool OpenRO(const char *name) {
        fd = open(name, O_RDONLY);
        return (fd != -1);
    }   // метод пытается открыть файл на чтение,
        // возвращает true в случае успеха,
        // false в случае неудачи

    // ...
    // ... методы работы с файлом ...
    // ...

    ~File() { if(fd!=-1) close(fd); }
         // Деструктор закрывает файл, если он открыт
};
```

Деструктор будет вызван в любой ситуации, когда объект типа `File` прекращает существование. Например, если объект был описан как локальный в функции, то при возврате из функции (в том числе и досрочном вызове оператора `return`) для этого объекта отработает деструктор. Вообще **при создании объекта ровно один раз отрабатывает конструктор, при уничтожении объекта ровно один раз отрабатывает деструктор**. За выполнением этого правила следит компилятор.

3. Абстрактные типы данных в Си++

Защита вместе с конструкторами и деструкторами, которые мы рассмотрели в предыдущей главе, используются как для создания *объектов* в терминах объектно-ориентированного программирования, так и для синтеза абстрактных типов данных; что касается методов (функций-членов), то для ООП они обязательны, тогда как для абстрактных типов данных это лишь один из возможных способов описания набора операций над вводимым типом.

В этой главе мы рассмотрим средства, которые вообще не имеют никакого отношения к парадигме объектно-ориентированного программирования, но это не делает их менее важными. Возможность ввести свои версии арифметических операций для пользовательских типов данных, контроль объектов за собственным созданием, ко-

пированием, присваиванием и ликвидацией, уникальная концепция типа-ссылки, позволяющая вынести в систему типов понятие *леводопустимого выражения* — всё это практически с самого начала стало визитной карточкой языка Си++.

Рассматривая все эти инструменты единым комплексом, мы можем заметить, что главный достигнутый результат здесь — это возможность для программиста ввести сколь угодно сложную абстракцию, работа с которой будет выглядеть так же, как мы привыкли работать с элементарными встроенными типами, при желании полностью игнорируя детали реализации. Имеющиеся в Си++ средства создания абстрактных типов данных обладают выразительной мощью, близкой к принципиально возможному максимуму.

При правильной организации программы трудоёмкость её написания на Си++ может оказаться в разы ниже, чем создание такой же программы на чистом Си, и основной выигрыш по трудозатратам здесь дают именно средства генерации абстрактных типов данных, рассматриваемые ниже.

3.1. Перегрузка имён функций; декорирование

Для понимания целого ряда возможностей, которые мы будем постепенно вводить в следующих параграфах, нужно иметь представление о свойстве языка Си++, называемом ***перегрузкой имён функций***. Сразу оговоримся, что эта возможность не имеет никакого отношения ни к ООП, ни к АТД и вообще к каким бы то ни было парадигмам; это сугубо техническое решение, позволившее снять некоторые проблемы, которых вообще-то можно было изначально не допускать; как мы вскоре увидим, перегрузка функций сама порождает достаточно ощутимые технические проблемы.

Перегрузка имён состоит в том, что в рамках одной области видимости Си++ позволяет ввести *несколько* различных функций, имеющих одно имя, но предполагающих разное количество и/или типы параметров. Например, будет вполне корректен следующий код:

```
void print(int n) { printf("%d\n", n); }
void print(const char *s) { printf("%s\n", s); }
void print() { printf("Hello world\n"); }
```

При обработке вызова функции компилятор определяет, какую функцию с таким именем следует вызвать, используя количество и типы фактических параметров. Например:

```
print(50);                // вызывается print(int)
print("Have a nice day"); // вызывается print(const char*)
print();                 // версия без параметров
```

Введение перегруженных функций может привести к совершенно неожиданным последствиям. Так, следующий код сам по себе корректен:

```
void f(const char *str)
{
    printf("This is a string: %s\n", str);
}
void f(float f)
{
    printf("This is a floating point number: %f\n", f);
}
```

Компилятор вполне справится с вызовами `f("string")` и `f(2.5)`, поскольку никаких разночтений тут не возникает. Более того, даже вызов `f(1)` будет успешно откомпилирован, так как целое можно неявно преобразовать к типу `float`, но не к типу `const char*`. Однако вызов `f(0)` будет расценен компилятором как ошибочный, поскольку компилятор с совершенно одинаковым успехом может рассматривать `0` и как константу типа `float`, и как адресную константу («нулевой указатель»), а оснований выбрать тот, а не иной вариант у него нет. Между тем, если бы в программе присутствовала только одна из двух вышеописанных функций `f` (любая!), вызов `f(0)` был бы заведомо корректен.

Отметим один интересный казус. Формально говоря, перегрузка функций есть одно из проявлений ***статического полиморфизма***. С полиморфизмом этого вида, *встроенным в язык*, читатель неоднократно сталкивался, но, скорее всего, не задумывался о происходящем в таких терминах. Строго говоря, полиморфизм — это способность одинаковых конструкций языка (чаще всего выражений, хотя и не обязательно) обозначать различные действия в зависимости от типов задействованных переменных и значений. Например, выражение «`a+b`» во многих языках, в том числе и в чистом Си, и в Паскале, может означать сложение двух целых чисел, сложение двух чисел с плавающей точкой (читатели, знакомые с языком ассемблера, точно знают, что это совсем другая операция), а в некоторых языках может и вовсе обозначать конкатенацию двух строк. Изучая Си, мы столкнулись с тем, что выражение вида «`a[i][j]`» может вычисляться совершенно по-разному: одно дело, если `a` представляет собой указатель на указатель, и совсем другое — если `a` является именем двумерного массива (либо, что фактически то же самое, значением мозгодробительного типа «указатель на массив»).

В отличие от таких случаев, перегрузка функций — это полиморфизм, вводимый пользователем, а не встроенный в язык. И тем

не менее, ни к объектно-ориентированному программированию, ни к абстрактным типам данных перегрузка прямого отношения не имеет.

Появление в Си++ перегрузки имён функций ведёт к определённым трудностям при компоновке программ из модулей, написанных на разных языках — или, если говорить честно, при использовании в одной программе модулей, написанных на Си++, на чистом Си и (гораздо реже) на языке ассемблера[10]. Если в ваши планы не входит использование отдельных модулей на чистом Си, можете пропустить остаток этого параграфа — но лучше прочитайте, чтобы хотя бы примерно знать, где вас могут поджидать очередные грабли.

Как мы знаем (см. т. 2, §3.7), объектные файлы, получающиеся в результате компиляции из отдельных модулей, содержат образы областей памяти, помеченные ***метками***, и указания для редактора связей о том, что в определённые места нужно подставить итоговые адреса, которые получатся из меток, содержащихся в других модулях. Редактор связей по сути примитивен, он не знает ничего не только о перегрузке имён функций, но и вообще о функциях, для него функция — это просто образ области памяти. Компиляторы чистого Си в качестве метки для кода функции обычно используют имя этой функции (а для глобальных переменных — имена этих переменных); компилятор Си++ с глобальными переменными обходится точно так же, но с функциями он так поступить не может, ведь «благодаря» перегрузке имён в программе может оказаться больше одной функции с заданным именем.

Здесь на сцене появляется очередной монстр, который по-английски называется *name mangling*; наиболее часто это переводится на русский словосочетанием «***декорирование имён***», которое, увы, совершенно не отражает сути. Между прочим, слово *mangling* правильнее было бы перевести как «коверканье» — именно так оно переводится в других (непрограммистских) контекстах. Программисты, разумеется, в повседневной речи используют транслитерацию «*манглинг*», и труднопроизносимость этого слова их не останавливает.

Как бы мы ни называли это явление, суть его в том, что для обозначения функции в объектном файле компилятор использует не её имя, а некое неудобоваримое чёрт-те что, содержащее закодированную информацию обо всём профиле функции, то есть о её имени и типах всех её параметров. Например, имя функции

```
double foo(double a, int b);
```

наш компилятор g++ превратит в «`_Z3foodi`»; здесь `_Z` — это «волшебный» префикс, обозначающий «декорированное» имя, затем следует десятичное число, обозначающее длину исходного имени функции (в нашем случае 3, поскольку имя `foo` состоит из трёх букв), потом идёт собственно имя, а за ним — закодированные типы параметров. Для встроенных типов всё довольно просто — как мы видим, `double` превратилось в букву `d`, `int` — в `i`; с пользовательскими типами компилятору приходится изворачиваться сильнее —

[10] Если вы знаете людей, которые используют в программе на Си++ модули из какого-нибудь ещё языка, кроме Си и ассемблера, сообщите о них автору книги, ему было бы интересно на них посмотреть.

например, параметр типа `const struct str1*` превратился бы в «`PK4str1`». Забегая вперёд, отметим, что декорированное имя функции, параметрами которой служат типы, инстанциированные из шаблонов, обычно очень хочется *развидеть*.

Пикантности ситуации добавляет ещё и то, что разные компиляторы (а в некоторых особо патологических случаях — даже разные версии одного компилятора) могут использовать различные алгоритмы декорирования — разумеется, несовместимые между собой. Впрочем, в этом плане в последние годы наметился определённый прогресс — во всяком случае, g++ не меняет схему декорирования, начиная с версий 3.*, и ту же самую схему используют несколько других компиляторов, включая, например, clang. Так или иначе, **библиотеки, написанные на Си++, можно переводить в объектный код только для использования с тем же компилятором**, совместимости с другими компиляторами никто пообещать не может (хотя иногда она и есть). Эта проблема никак не затрагивает тех (несомненно разумных) людей, которые используют и распространяют библиотеки исключительно в форме исходного кода, ну а тех (не очень хороших) людей, кто по каким-то причинам исходные тексты своих библиотек распространять не хочет, заставляет собирать объектные варианты своих библиотек под каждый компилятор; следует признать, что и в этом ничего слишком трудного нет, ведь компиляторов Си++ не так много.

Хуже другое: видя *один и тот же заголовок функции*, компиляторы Си и Си++ используют *разные* имена для объектного кода этой функции. Если взять модуль, написанный на чистом Си и содержащий функцию вроде приведённой выше `foo`, откомпилировать его компилятором чистого Си, а его заголовочный файл, содержащий заголовок той же самой функции, подключить с помощью `#include` из модуля, написанного на Си++, компилируемого компилятором Си++ и при этом содержащего обращение к функции `foo`, итоговая программа не пройдёт компоновку. В самом деле, в объектном модуле, полученном компилятором чистого Си, функция будет называться просто `foo`, тогда как компилятор Си++ построит обращение к ней через имя `_Z3foodi`; редактор связей попытается найти в предоставленных ему файлах такую метку, но, понятное дело, не найдёт.

К счастью, Си++ предусматривает решение для этой проблемы. Если заголовки некоторых функций заключить в «тело» директивы `extern "C"` примерно так:

```
extern "C" {
    // ...
    double foo(double a, int b);
    // ...
}
```

— то для этих функций компилятор будет применять соглашения, характерные для Си; на самом деле конкретный язык задаётся строковым литералом после слова `extern`, в данном случае «`"C"`»; впрочем, большинство компиляторов поддерживает только один такой «внешний» язык — Си. Декорирование имён для этих функций будет отключено.

Основных способов применения директивы `extern "C"` два. Если автор модуля, написанного на Си, о нас не позаботился, проще всего будет заключить в `extern "C"` *саму директиву* `#include`, подключающую заголовочный файл, так что в результате компилятор увидит всё его содержимое уже внутри `extern "C"`. Выглядит это примерно так:

```
extern "C" {
#include "foo.h"
}
```

Если же мы пишем модуль на чистом Си, но исходно предполагаем, что он будет использоваться в программах, написанных как на Си, так и на Си++, будет вполне осмысленно в начале и в конце заголовочного файла разместить начало и окончание директивы `extern "C"`. Естественно, нужно позаботиться о том, чтобы их видел только компилятор Си++, а компилятор чистого Си не видел — в чистом Си такой директивы, собственно говоря, нет. Но и этот вопрос легко решается благодаря макросимволу `__cplusplus`, который всегда определён, когда используется компилятор Си++. Заголовочный файл `foo.h` с учётом защиты от повторного включения (см. т. 2, §4.11.4) может выглядеть примерно так:

```
#ifndef FOO_H_SENTRY
#define FOO_H_SENTRY

#ifdef __cplusplus
extern "C" {
#endif

// ...

double foo(double a, int b);

// ...

#ifdef __cplusplus
}
#endif

#endif
```

К декорированию имён компилятор вынужден прибегать не только из-за перегрузки функций, но и из-за имён, вложенных в *области видимости*; позже мы увидим, что такими областями являются, например, классы и структуры: их методы имеют «сложную» форму имён. Однако с этой проблемой можно было бы справиться, не ломая совместимость с Си, когда речь идёт о *простых функциях* (не методах классов). Например, метод `f` из класса `C` мог бы на уровне объектного кода иметь имя `C@@f` или даже просто `C::f`, если линкер допускает двоеточие в именах меток (впрочем, обычно это не так).

Перегрузка имён функций в этом плане кажется довольно неудачной идеей. Введена она в основном по двум причинам: во-первых, чтобы иметь возможность описать в одном классе или структуре *несколько конструкторов*, а во-вторых, как мы будем обсуждать позже, чтобы можно было перегружать символы стандартных операций для типов, введённых пользователем, для чего потребовалось позволить *нескольким* разным функциям иметь одно и то же имя, например, `operator+`. Но обе эти проблемы могли бы быть решены без введения перегрузки: никто не мешает (теоретически) давать конструкторам разные имена, например, просто помечая их специальным словом «`constructor`», а функции-операции вроде `operator+` могли бы быть не функциями, а чем-то вроде макросов, либо можно было потребовать, чтобы функции с именами такого вида всегда были «подставляемыми» (inline) и не были бы, как следствие, видны линкеру; если операция предполагает сложную реализацию, её можно было бы выполнить в функции с обычным именем, а из `operator+` её просто вызвать.

Мы в очередной раз видим, что язык Си++ не идеален; увы, достойная альтернатива ему не спешит появляться. Так или иначе, уроки эпохи Си++ можно было бы учесть при создании нового языка программирования — хотя, судя по происходящему в мире IT (особенно по новым «стандартам» Си++), рассчитывать на это в ближайшие годы не приходится.

3.2. Переопределение символов стандартных операций

Вернёмся к нашему примеру — классу, описывающему комплексное число. В реальной задаче мы вряд ли захотим ограничиться только операциями взятия модуля и разложения на компоненты (действительную и мнимую части, модуль и аргумент). Скорее всего, нам потребуются также сложение, вычитание, умножение и деление.

Язык Си++ предоставляет возможности записи таких действий с помощью привычных символов арифметических операций. Кроме того, логично предусмотреть возможность узнать действительную и мнимую части комплексного числа, если вдруг нам это понадобится. Для этого дополним наш класс ещё несколькими функциями-членами, имена которых будут выглядеть несколько экзотически. Эти функции как раз и будут описывать действия, которые должен по нашему замыслу означать символ той или иной арифметической операции. Отметим, что методы в этом примере будут обращаться к закрытым полям не только «своего» объекта, но и объекта, переданного как параметр. Это не нарушает защиту: как уже говорилось на стр. 25, единицей защиты является не объект, а класс или структура как таковые. Итак, пишем:

```
class Complex {
    double re, im;
```

```
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    double modulo() { return sqrt(re*re + im*im); }
    double argument() { return atan2(im, re); }
    double get_re() { return re; }
    double get_im() { return im; }
    Complex operator+(Complex op2) {
        Complex res(re + op2.re, im + op2.im);
        return res;
    }
    Complex operator-(Complex op2) {
        Complex res(re - op2.re, im - op2.im);
        return res;
    }
    Complex operator*(Complex op2) {
        Complex res(re*op2.re - im*op2.im,
                    re*op2.im + im*op2.re);
        return res;
    }
    Complex operator/(Complex op2) {
         double dvs = op2.re*op2.re+op2.im*op2.im;
         Complex res((re*op2.re + im*op2.im)/dvs,
                     (im*op2.re - re*op2.im)/dvs);
         return res;
    }
};
```

Поясним, что слово `operator` является зарезервированным (ключевым) словом языка Си++. Стоящие подряд две лексемы `operator` и `+` образуют *имя функции*, которую можно вызвать и как обычную функцию:

```
a = b.operator+(c);
```

— однако, в отличие от обычной функции, появление функции-операции позволяет записать то же самое в более привычном для математика виде:

```
a = b + c;
```

Теперь мы можем, к примеру, узнать, каков будет модуль суммы двух комплексных чисел:

```
Complex c1(2.7, 3.8);
Complex c2(1.15, -7.1);
double m = (c1+c2).modulo();
```

Как мы увидим позже, аналогичным образом в языке Си++ можно переопределить символы любых операций, включая присваивания, индексирование, вызов функции и прочую «экзотику». Существуют только две[11] операции, которые нельзя переопределить: это тернарная ***условная операция*** (`a ? b : c`) и операция выборки поля из структуры или класса (точка). Интересно, что операция выборки поля по указателю («стрелка») переопределяема, хотя и несколько экзотическим способом. Ко всем этим вопросам мы вернёмся позже.

Несомненно, возможность вводить арифметические операции для пользовательских типов — это ещё одно проявление ***статического полиморфизма***, как и рассмотренная в предыдущем параграфе перегрузка функций. Мы уже упоминали, что функции, имя которых содержит слово `operator`, стали одной из причин, по которым в Си++ вообще появилась перегрузка имён функций; как мы увидим позже, такие функции не всегда вводятся как методы классов, обычная функция тоже может иметь такое «странное» имя и вводить пользовательский смысл для арифметической операции; различать операции, имеющие одно и то же обозначение (например, `+`), компилятор вынужден по *типам аргументов* — то есть ему приходится на основании информации о типах выбирать одну из многих функций, а это и есть, собственно говоря, перегрузка. Но если про перегрузку функций в общем виде мы говорили, что она не имеет никакого отношения к интересующим нас парадигмам (и вообще к каким бы то ни было парадигмам), то перегрузка символов стандартных операций — это весьма существенный элемент поддержки пользовательских абстрактных типов данных.

Довольно неприятная терминологическая путаница возникает из-за английского слова `operator`, которое так и хочется перевести на русский как «оператор», прочитав его «как написано». Увы, делать этого нельзя, ведь русское слово «оператор» в программировании имеет совершенно иное значение: так мы называем законченные структурные единицы текста программы, такие как «оператор ветвления» (`if`), «операторы циклов» (`for`, `if`, `while`) и другие управляющие конструкции. Напомним, что в Си (и в Си++, естественно, тоже) можно превратить в *оператор* произвольное арифметическое выражение, добавив к нему точку с запятой; такой оператор называется «оператором вычисления выражения ради побочного эффекта». В частности, когда мы вызываем функцию, игнорируя при этом её возвращаемое значение, мы применяем именно этот оператор.

Английское слово *operator* обозначает сущность совершенно иную — то, что мы по-русски обычно называем «операциями»: всевозможные арифметические операции (сложение, вычитание и т. п.), операции сравнения, поби-

[11] Некоторые авторы включают в этот список ещё и «операцию» раскрытия области видимости; это не вполне корректно, поскольку символ раскрытия области видимости **не является операцией**. К этому вопросу мы вернёмся в §3.13.

товые и логические операции, операции с адресами и указателями, а если речь идёт о Си и его потомках — то ещё и операции присваивания. Иначе говоря, по-английски *operator* — это нечто такое, что может встречаться в *выражении* наряду с константами, переменными и скобками.

Ничего удивительного в именно таком использовании слова *operator* нет. Читатель, знакомый с высшей математикой, несомненно, встречал такие термины, как «линейный оператор», «оператор дифференциирования» и т. п., которыми обычно обозначаются отображения из некоторого пространства в него само, т. е. попросту *функции*, имеющие некое рассматриваемое пространство одновременно в качестве области определения и области значений. Когда речь идёт о таких вот — математических — «операторах», соответствующим английским термином окажется как раз слово *operator*. Ну а арифметические операции, понятное дело, с этими *operator*'ами состоят в самом близком родстве, так что для обозначения (в математике) символов вроде плюса, минуса или деления англоговорящие математики используют тот же самый термин; естественно, переняли этот термин и программисты.

Что же касается структурных единиц программного текста, называемых по-русски «операторами», то этой сущности соответствует совершенно иной английский термин: *statement* (буквально переводится как «утверждение»). Как так получилось, что английское *statement* переводят на русский словом «оператор», кто это первым придумал, создав для грядущих поколений русскоязычных программистов неприятную терминологическую проблему — вопрос, конечно, интересный; к сожалению, сделать с этим уже ничего нельзя, термин «оператор» намертво врос в русскую программистскую лексику именно в этом совершенно нелогичном для него значении.

3.3. Конструктор умолчания. Массивы объектов

Возвращаясь к нашему классу `Complex`, напомним (см. стр. 24), что описать переменную типа `Complex` привычным нам образом без всяких параметров нельзя, т. к. для единственного конструктора класса `Complex` требуются два параметра. Так, следующий код будет некорректен:

```
Complex sum; // ОШИБКА - нет параметров для конструктора
sum = c1+c2;
```

Кроме того, у нас возникнут сложности при создании массивов комплексных чисел; обычный массив заранее заданного размера мы создать сможем, но только при указании инициализатора, имеющего весьма нетривиальный синтаксис; забегая вперёд, отметим, что *динамический* массив комплексных чисел мы создать не сможем вообще никак, поскольку синтаксис языка не позволяет задать параметры для конструкторов каждого элемента массива.

Снять возникшие проблемы позволяет введение ещё одного конструктора. Напомним, что компилятор считает конструктором функ-

цию-член класса (или структуры), имя которой совпадает с именем класса (или структуры). Благодаря свойству перегрузки функций в Си++ мы можем ввести в одной области видимости несколько функций с одним и тем же именем; это относится, разумеется, и к конструкторам. Нужно только, чтобы функции, имеющие одинаковые имена (в данном случае — конструкторы), различались количеством и/или типом параметров. Итак, добавим в класс `Complex` ещё один конструктор:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
       { re = a_re; im = a_im; }
    Complex() { re = 0; im = 0; }

//....
```

Теперь мы можем описывать переменные типа `Complex`, не задавая никаких параметров. Поэтому конструктор, имеющий пустой список параметров, называют ***конструктором по умолчанию***. Его наличие делает возможными, в частности, следующие описания:

```
Complex z3; // используем конструктор по умолчанию
Complex v[50]; // конструктор будет вызван 50 раз
               // т.е. для каждого элемента
```

3.4. Конструкторы преобразования

При создании программ на языках, обладающих типизацией, нередко возникает потребность использовать значение одного типа там, где по смыслу предполагается значение другого типа. Один из простейших примеров такой ситуации — сложение (или другая арифметическая операция) числа с плавающей точкой и целого числа. В языке Си в подобных случаях производится так называемое *неявное преобразование типов*. Так, если мы опишем переменную `a` как целочисленную, а переменную `b` как имеющую тип `float`, и после этого используем в программе выражение `a + b`, компилятор *неявно* преобразует значение переменной `a` к типу `float`, и только после этого выполнит сложение. Точно так же, если в программе описана функция `void f(float)`, мы можем записать вызов `f(25)`, и компилятор сочтет такой код корректным, т. к. «знает», каким образом из целого числа сделать число с плавающей точкой; иначе говоря, в языке Си присутствует правило преобразования значений типа `int` в значения типа `float`. Естественно, язык Си++ также позволяет

производить подобные неявные преобразования, но, помимо этого, в нём есть средства указания правил преобразования для *типов данных, введённых пользователем*. Одним из таких средств являются *конструкторы преобразования*.

Очевидно, что задать правило преобразования значений типа `A` в значения типа `B` — это то же самое, что задать инструкцию по созданию объекта типа `B` по имеющемуся значению типа `A`. В самом деле, в разобранном выше примере компилятор сначала строил значение типа `float`, используя в качестве отправной точки имеющееся значение типа `int`; разумеется, компилятору для этого необходимо «знать», как это делать. Как отмечалось в §2.3, инструкции компилятору Си++ по созданию объектов класса на основе заданных параметров даются путем описания конструкторов с соответствующими параметрами. Таким образом, если ввести в классе `B` конструктор, получающий параметр типа `A`, это как раз и будет инструкция по созданию объекта типа `B` по имеющемуся значению типа `A`. Представляется поэтому вполне логичным использовать такие инструкции для *выполнения неявного преобразования типов*.

Итак, конструктор, который получает на вход ровно один параметр, имеющий тип, отличный от описываемого, называется ***конструктором преобразования***[12] и используется компилятором не только в случае явного создания объекта, но и для проведения неявного преобразования типов. Проиллюстрируем сказанное на примере введённого ранее класса `Complex`. Ясно, что по смыслу комплексных чисел любое действительное число может быть преобразовано к комплексному добавлением нулевой мнимой части точно так же, как целое число преобразуется к числу с плавающей точкой добавлением нулевой дробной части. Чтобы выразить это соотношение между действительными и комплексными числами, дополним класс следующим конструктором:

```
Complex(double a) { re = a; im = 0; }
```

С одной стороны, этот конструктор позволяет нам описывать комплексные числа с указанием одного действительного параметра, например:

```
Complex c(9.7);
```

С другой стороны, если в программе имеется функция

```
void f(Complex a);
```

[12] Если только специальное значение такого конструктора не отменить директивой `explicit`.

мы можем вызвать её для действительного параметра:

```
f(2.7);
```

Благодаря наличию в классе `Complex` конструктора преобразования компилятор сочтет такой вызов корректным; для его обработки с помощью конструктора преобразования будет создан ***временный объект*** типа `Complex`, который и будет подан на вход функции `f`.

3.5. Ссылки

Понятие *ссылки*, которому посвящён этот параграф, оказывается ключевым для понимания дальнейшего материала. Между тем, ничего похожего нет ни в чистом Си, ни в других языках программирования, так что ссылки часто вызывают у начинающих определённые трудности. Постарайтесь поэтому подойти к изучению этого параграфа особенно внимательно, а при возникновении вопросов обязательно задайте их вашему преподавателю или кому-то ещё, кто сможет вам объяснить происходящее.

Ссылка в Си++ — это особый вид объектов данных, реализуемый путём хранения адреса какой-то переменной, но, в отличие от указателя, ссылка семантически эквивалентна той переменной, на которую она ссылается. Иначе говоря, любые операции над ссылкой будут на самом деле производиться над той переменной, адрес которой содержится в ссылке. Надо отметить, что это относится и к присваиваниям, и к взятию адреса — вообще ко всем операциям, какие могут быть; саму ссылку, таким образом, вообще невозможно изменить, её значение (переменная, на которую она ссылается) задаётся в момент создания ссылки и остаётся неизменным в течение всего срока её существования.

Синтаксически тип данных «ссылка» описывается аналогично указателю, только вместо символа «`*`» используется символ «`&`»[13]. Проиллюстрируем сказанное простым примером:

```
int i;       // целочисленная переменная
int *p = &i; // указатель на переменную i
int &r = i;  // ссылка на переменную i

i++;         // увеличить i на 1
(*p)++;      // то же самое через указатель
r++;         // то же самое по ссылке
```

[13]Важно понимать, что в данном случае символ «`&`» не имеет ничего общего с операцией взятия адреса! Почему автор языка Си++ Б. Страуструп выбрал именно такое обозначение — вопрос открытый.

Обычно говорят о «переменных ссылочного типа» наравне с переменными других типов, хотя ссылки не вполне корректно называть «переменными», поскольку изменить значение *самой ссылки* нельзя: всё, что мы с ней попытаемся сделать, на самом деле будет сделано с той переменной, на которую ссылка ссылается.

Из-за неизменности ссылок **переменную ссылочного типа нельзя описать без инициализации, то есть без задания начального значения**. Такое описание было бы заведомо бессмысленным, ведь если не задать значение ссылке с самого начала, то потом мы никаким способом осмысленное значение в ссылку уже не занесём. Пользуясь случаем, напомним ещё раз, что инициализация и присваивание — это совершенно разные сущности.

Описав в нашем примере ссылку на переменную `i`, мы фактически ввели *синоним* имени `i` — имя `r`, обозначающее тот же самый объект данных. Такое использование ссылок может показаться бессмысленным и действительно встречается очень редко. По-настоящему возможности ссылочного типа в Си++ раскрываются при передаче ссылок в функции и возврате их из функций в качестве значения. Так, благодаря ссылочному типу в нашем распоряжении оказывается отсутствовавший в языке Си механизм передачи параметров по ссылке[14]. Допустим, нам нужно написать функцию, находящую максимальный и минимальный элементы заданного массива чисел типа float. На языке Си эта функция выглядела бы так:

```
void max_min(float *arr, int len, float *min, float *max)
{
    int i;
    *min = arr[0];
    *max = arr[0];
    for(i=1; i<len; i++) {
        if(*min>arr[i])
            *min = arr[i];
        if(*max<arr[i])
            *max = arr[i];
    }
}
```

а её вызов — например, так:

```
    float a[500];
    float min, max;
    // ...
    max_min(a, 500, &min, &max);
```

[14] Читатель, скорее всего, уже знаком с передачей параметров по ссылке: в языке Паскаль такие параметры называются параметрами-переменными (var-parameters).

Поскольку из функции нужно вернуть больше одного значения, приходится использовать возврат через параметры. Между тем в языке Си все параметры передаются *по значению*, поэтому нам приходится вручную *имитировать* выходные параметры, передавая значение адреса переменной, подлежащей модификации; поэтому при вызове функции мы вынуждены применять операцию взятия адреса («`&`»). В самой функции вместо имени переменной мы вынуждены использовать леводопустимое выражение разыменования, т. е. ставить перед идентификаторами `min` и `max` символ операции разыменования «`*`», чтобы преобразовать адрес переменной в саму переменную.

С использованием ссылок и код функции, и её вызов обретают более ясный вид:

```
void max_min(float *arr, int len, float &min, float &max)
{
    int i;
    min = arr[0];
    max = arr[0];
    for(i=1; i<len; i++) {
        if(min>arr[i])
            min = arr[i];
        if(max<arr[i])
            max = arr[i];
    }
}

    // ...
    max_min(a, 500, min, max);
```

Параметры `min` и `max`, объявленные на сей раз как ссылки, семантически представляют собой синонимы неких целочисленных переменных, так что все операции, которые производятся над `min` и `max`, на самом деле происходят над теми переменными, на которые эти параметры ссылаются. При вызове функции мы не применяем никаких операций взятия адреса, поскольку (с семантической точки зрения) нам для построения синонима нужна сама переменная, а не её адрес.

Ещё более интересные возможности открывает использование ссылочного типа как типа возвращаемого значения функции. Допустим, нам нужно произвести поиск целочисленной переменной в составе сложной структуры данных (например, искомая переменная является полем структуры, которая, в свою очередь, является элементом массива и т. п.), а затем либо использовать значение найденной переменной, либо выполнить над ней то или иное присваивание. В такой ситуации поиск переменной можно выделить в отдельную

функцию, возвращающую ссылку на найденную переменную. Если такая функция имеет профиль

```
int &find_var(/*params*/);
```

то допустимы будут, например, такие действия:

```
int x = find_var(/*...*/) + 5;
find_var(/*...*/) = 3;
find_var(/*...*/) *= 10;
find_var(/*...*/)++;
int y = ++find_var(/*...*/);
```

3.6. Константные ссылки

Модификатор `const` уже, скорее всего, знаком нам по языку Си, хотя изначально он появился именно в Си++, а в Си проник гораздо позднее, причём так и не был признан авторами языка Си. При описании адресных типов модификатор `const` позволяет указать, что область памяти, на которую указывает описываемый указатель, не подлежит изменению. Например:

```
const char *p;
    // p указывает на неизменяемую область памяти
p = "A string";
    // всё в порядке, переменная p может изменяться
*p = 'a';
    // ОШИБКА! Нельзя менять область памяти,
    // на которую указывает p
p[5] = 'b';     // Это также ОШИБКА
```

Подчеркнём ещё раз, что `const char *p` — это именно *указатель на константу*, а не *константный указатель*. Чтобы описать константу адресного типа, необходимо расположить ключевые слова в другом порядке:

```
char buf[20];
char * const p = buf+5;
     // p - константа-указатель, указывающая на
     // шестой элемент массива buf (buf[5])

p++;        // ОШИБКА, значение p не может быть изменено
*p = 'a';   // Всё в порядке, изменяем значение buf[5]
p[5] = 'b'; // Всё в порядке, изменяем значение buf[5+5]
```

Аналогичным образом дело обстоит со ссылочными типами:

```
int i;
const int &r = i; // ссылка на константу
int x = r+5; // всё в порядке
i = 7;       // тоже всё в порядке
r = 12;  // ОШИБКА, значение по ссылке r нельзя менять

const int j = 5;
int &jr = j;  //  ОШИБКА, нельзя ссылаться
              //  на константу обычной ссылкой
const int &jcr = j; // всё в порядке
```

Константные ссылки позволяют передавать в функции в качестве параметра адрес переменной вместо копирования значения, при этом не позволяя эту переменную менять, как и при обычной передаче по значению. Это может быть полезно, если параметр представляет собой класс или структуру, размер которой существенно превышает размер адреса. Так, если в ранее описанном классе `Complex` вместо

```
Complex operator+(Complex op2)
    { return Complex(re+op2.re, im+op2.im); }
```

написать

```
Complex operator+(const Complex &op2)
    { return Complex(re+op2.re, im+op2.im); }
```

семантика кода не изменится, но физически (на уровне машинного кода) вместо копирования двух полей типа `double` будет происходить передача адреса существующей переменной и обращение по этому адресу.

3.7. Ссылки как семантический феномен

Не следует недооценивать важность ссылок как семантического изобретения. В определённом смысле именно ссылки — это самое важное новшество языка Си++; всё остальное, что содержит этот язык, ранее встречалось в других языках. При этом введение ссылок открывает очень интересные перспективы, и в этом параграфе будет сделана попытка эти перспективы обозначить.

При описании языка Си обычно вводится понятие ***леводопустимого выражения*** (англ. *lvalue*), изначально обозначавшее всевозможные выражения, допустимые *слева* от знака присваивания (отсюда название). Результат вычисления такого выражения не просто представляет собой значение, а идентифицирует некую область памяти, в которой располагается значение. Простейшим примером леводопустимого выражения служит *имя переменной*, но этим, конечно же, дело не ограничивается. Так, если `p` — некий указатель типа,

отличного от `void`, то `*p` — леводопустимое выражение; если `v` — некий массив (или, опять же, адрес), то `v[i]` будет леводопустимым; если `q` — адрес структуры, содержащей поле `f`, то `q->f` — леводопустимое выражение, и так далее. Общий принцип тут такой: результатом вычисления леводопустимого выражения в действительности становится некая *область памяти*, и если от выражения требуется значение, то оно берётся из этой области памяти; когда леводопустимое выражение стоит слева от присваивания, значение от него не требуется, а требуется сама область памяти как таковая, в которую заносится новое значение. Когда леводопустимое выражение оказывается операндом таких операций, как `+=` или `++`, оно используется и как значение, и как область памяти.

С проникновением в язык Си модификатора `const` термин *lvalue* утратил свой изначальный смысл; например, если описать в программе что-то вроде

```
const int width = 80;
```

то выражение, состоящее из идентификатора `width`, будет считаться *lvalue*, хотя слева от присваивания располагаться уже не может. В современных описаниях Си «леводопустимые» выражения делят на *изменяемые* и *неизменяемые* (англ. *modifiable and non-modifiable lvalues*); понятности описания Си всё это отнюдь не способствует.

Поскольку *lvalue* всегда, по крайней мере, представляет собой идентифицированную область памяти, то можно назвать и общее свойство для всех *lvalue*-выражений: от них можно взять адрес, т. е. применить к ним унарную операцию «`&`». К сожалению, даже это свойство нельзя считать определяющим: взятие адреса можно применить к имени функции, а с некоторых пор — и к имени массива (в результате получается адресное выражение типа «указатель на массив»; того, кто это придумал, очень хочется убить), но ни то, ни другое не является леводопустимым.

В чистом Си леводопустимость — это свойство конкретного выражения, не типа, не переменной, не функции, а именно выражения. Ссылки, которые Страуструп придумал для Си++, замечательны тем, что позволяют оперировать понятием леводопустимости на уровне системы типов. Для этого достаточно считать, что идентификатор переменной, имеющей, например, тип `int`, сам по себе представляет выражение не типа `int`, а типа *ссылка на* `int`, и то же самое сказать про остальные случаи леводопустимых выражений: если `p` имеет тип `int*`, то `*p` — это не `int`, а `int&` (ссылка на `int`), если `f` — поле структуры, имеющее тип `double`, то `s.f` или `q->f` — выражения типа `double&`, если `str` — массив элементов типа `char`, то `str[i]` имеет тип `char&`, и т. д.

Если принять такую терминологию, потребность в отдельном понятии «леводопустимого выражения» отпадает: вместо этого можно

говорить, что слева от присваивания, а равно и в других «модифицирующих» случаях могут применяться ссылки и только они, если же ссылка применяется в выражении, в котором требуется *значение*, то ссылка автоматически преобразуется к своему значению (т. е. выполняется извлечение значения из памяти).

Нетрудно видеть, что наличие константных ссылок позволяет отразить разницу между пресловутыми *modifiable lvalues* и *non-modifiable lvalues*: первым соответствуют обычные ссылки, вторым — константные.

Остаётся лишь сожалеть, что столь простая и стройная семантическая концепция не была замечена создателями «стандартов» Си++: вместо того, чтобы избавиться от странных терминов *lvalue* и *rvalue*, заменив их ссылками, стандартизаторы, наоборот, ввели ещё *prvalues*, *xvalues* и вообще напрочь запутали описание языка.

Интересно, что, например, авторы компилятора gcc, судя по выдаваемой этим компилятором диагностике (именно, ошибок при поиске подходящей функции), потенциал ссылок в роли замены понятия *lvalue* заметили и воспользовались им.

На случай, если всё изложенное выше про «семантический феномен» показалось читателю чем-то вроде отвлечённой философии, приведём один небезынтересный пример. Пусть у нас есть две целочисленные переменные `x` и `y` и нам понадобилось присвоить какое-то значение (скажем, находящееся в переменной `z`) той из них, значение которой в настоящий момент меньше. Конечно, это можно сделать с помощью обычного `if`:

```
if(x < y)
    x = z;
else
    y = z;
```

В Си++ благодаря ссылкам можно сделать так:

```
(x < y ? x : y) = z;
```

В чистом Си такое не проходит, поскольку имена `x` и `y` в условном выражении соответствуют *значениям* этих переменных, а не им самим. Аналогичную конструкцию сделать всё же можно, но труднее:

```
*(x < y ? &x : &y) = z;
```

3.8. Константные методы

Представим себе, что мы описали некий класс (пусть он называется `A`), после чего создали функцию, получающую параметром *константный* адрес объекта такого класса:

```
void f(const A* ptr)
{
    // ...
}
```

Внутри функции объект, на который указывает `ptr`, будет *константным*, то есть его будет нельзя изменять. Если класс `A` описывает полноценный объект, то все его поля скрыты, то есть взаимодействие с объектом возможно только через методы. В то же время метод может, вообще говоря, изменить внутреннее состояние (то есть значения скрытых полей) объекта, нарушив требование о его неизменности. Поэтому для обеспечения неизменности объекта компилятор вынужден запрещать вызовы методов.

Такая же ситуация возникнет, естественно, при передаче параметром *константной ссылки*, при возврате константных адресов и ссылок из функций, а равно и в случае, если переменная-объект сама по себе изначально описана с модификатором `const` (хотя последнее используется сравнительно редко). Итак, объект вроде бы есть, но всё, что мы можем с ним делать — это вызывать его методы, а поскольку метод может изменить объект, компилятор нам вызывать методы не позволит. Получается, что с таким объектом мы вообще никак не сможем работать, если не предпримем специальных мер.

Для работы с константными объектами в языке Си++ предусмотрены ***константные методы***. Константным считается метод, после заголовка которого (но *перед* телом, если таковое присутствует) поставлен модификатор `const`:

```
class C1 {
    // ...
    void method(int a, int b) const
        { /* .... */ }
    // ...
 };
```

В теле такого метода поля объекта доступны, но запрещены действия, изменяющие их или способные привести к их изменению, в том числе присваивания полям новых значений, присваивания адресов этих полей неконстантным указателям, передача их в функции по неконстантным ссылкам и т. п. Иначе говоря, относительно константного метода известно, что он заведомо не может изменить состояние объекта (значения его полей). Поэтому константные методы можно без опаски вызывать для объектов, которые нельзя изменять. Например, такой код:

```
void f(const Myclass *p)
```

```
{
    p->my_method();
}
```

будет корректным только в случае, если в классе `Myclass` метод `my_method` объявлен как константный.

При написании программ **рекомендуется все методы, которые по своему смыслу не должны изменять состояние объекта, обязательно помечать как константные, тем самым разрешая вызывать их для константных объектов**. В частности, в описанном ранее классе `Complex` все методы, кроме конструкторов, никаких изменений в поля класса не вносят. Чтобы с объектами типа `Complex` было удобнее работать, следует пометить все методы класса модификатором `const`:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im)
        { re = a_re; im = a_im; }
    Complex(double a_re)
        { re = a_re; im = 0; }
    Complex() { re = 0; im = 0; }
    double modulo() const
        { return sqrt(re*re + im*im); }
    double argument() const
        { return atan2(im, re); }
    double get_re() const { return re; }
    double get_im() const { return im; }
    Complex operator+(const Complex &op2) const
        { return Complex(re+op2.re, im+op2.im); }
    // ...
};
```

В теле константного метода не допускаются (для того же объекта) вызовы методов, не являющихся константными. Причина такого ограничения очевидна: неконстантный метод может изменить объект, а это внутри константного метода делать запрещено. Это можно объяснить и иначе. **Внутри константного метода произвольного класса или структуры `C` указатель `this` имеет тип `const C *`, а не просто `C *`**; вызов неконстантного метода означал бы, что мы передаём как неконстантный параметр значение, которое у нас самих константное, то есть мы снимаем модификатор `const`, а это делать компилятор запрещает.

3.9. Операции работы с динамической памятью

Известно, что язык Си сам по себе не включает средств работы с динамической памятью; создание и уничтожение динамических структур данных вынесено в библиотеку и производится обычно с помощью функций `malloc`, `realloc` и `free`. В языке Си++ одновременно с выделением и освобождением памяти в некоторых случаях, а именно, при создании и удалении объекта типа структуры или класса, имеющего конструкторы и/или деструкторы, нужно выполнять дополнительные действия, заданные этими конструкторами и деструкторами. Кроме того, при создании динамических объектов с помощью конструктора иного, нежели конструктор по умолчанию, необходима возможность указания параметров конструктора.

Функции `malloc` и `free` ничего не знают о конструкторах, деструкторах и параметрах, поэтому для создания и удаления объектов они непригодны. Более того, информацией о конструкторах и деструкторах обладает только компилятор, поэтому в языке Си++ вообще невозможно вынести работу с динамической памятью из языка в библиотеку без введения дополнительных средств.

Автор языка Си++ Бьёрн Страуструп решил пойти более простым путем и внёс в язык соответствующие синтаксические конструкции для создания и удаления объектов. Для создания в динамической памяти одиночного объекта (переменной произвольного типа) в языке Си++ используется операция `new`, в которой нужно указать имя типа создаваемого объекта. Например:

```
int *p;
p = new int;
```

Если создаваемый объект принадлежит типу, имеющему конструктор, и возникает необходимость передать конструктору параметры, то эти параметры указываются в скобках после имени типа. Так, создание объекта описанного ранее типа `Complex` (см. §2.5) с указанием действительной и мнимой частей может выглядеть так:

```
Complex *p;
p = new Complex(2.4, 7.12);
```

Для удаления используется операция `delete`:

```
delete p;
```

Для создания и удаления динамических массивов используются специальные *векторные формы* операций `new` и `delete`, синтаксически отличающиеся наличием квадратных скобок:

```
int *p = new int[200]; // массив из 200 целых чисел
delete [] p;           // удаление массива
```

Эти формы отличаются тем, что соответствующие конструкторы и деструкторы вызываются для *каждого элемента массива*. Стоит заметить, что векторная форма операции `new` не имеет синтаксических средств для передачи параметров конструкторам, поэтому для создания массива элементов типа класс или структура *необходимо наличие у этого типа конструктора по умолчанию* (см.§3.3).

Важно помнить, что **объекты в динамической памяти, созданные с помощью векторной формы операции `new`, нельзя удалять с помощью обычной формы операции `delete` и наоборот**. Дело в том, что реализация менеджера динамической памяти вправе выделять память под обычные переменные и под массивы из разных областей динамической памяти, имеющих, возможно, различную организацию служебных структур данных. Также не следует удалять с помощью `delete` объекты, созданные функцией `malloc`, и наоборот, не следует удалять объекты, созданные операциями `new`, с помощью `free`.

Скорее всего ваш компилятор и прилагающаяся к нему библиотека построены так, что смешение разных способов выделения и освобождения памяти никаких негативных последствий не повлечёт. Это, однако, не повод так делать: при переходе на другой компилятор или даже на другую версию того же самого компилятора всё может внезапно сломаться.

3.10. Конструктор копирования

Рассмотрим следующую ситуацию. В реализации некоторого класса (назовем его `Cls1`) нам потребовался динамический массив, который мы создаем в теле конструктора класса; естественно, в деструктор следует поместить оператор для уничтожения этого массива.

```
class Cls1 {
    int *p;
public:
    Cls1() { p = new int[20]; }
    ~Cls1() { delete [] p; }
    // ...
};
```

Теперь предположим, что кто-то создает в программе копию объекта класса `Cls1`. Такое может произойти, например, если объект окажется передан *по значению* в качестве параметра функции. Например:

```
void f(Cls1 x)
{
    // ...
}
int main()
{
    // ...
    Cls1 c;
    f(c);
    //...
}
```

Проанализируем происходящее со структурами данных при вызове функции `f`. Локальная переменная `x` является копией объекта `c`. Копия любого объекта данных создается путем обычного побитового копирования, если не указать иного. Следовательно, при копировании объекта класса `Cls1` скопирован окажется *указатель* на динамический массив; иначе говоря, у нас появятся два объекта, использующие один и тот же экземпляр динамического массива: оригинал объекта `c` и его локальная копия `x`. Возникшая ситуация показана на рис. 1.

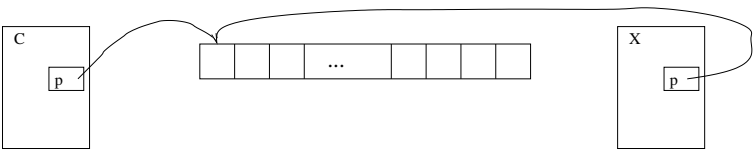


Рис. 1. Схема структуры данных после побитового копирования объекта

Даже если такое разделение не приведёт к немедленным ошибкам (например, функция `f` может изменить объект `x`, что отразится на внутреннем состоянии объекта `c`, чего мы могли и не ожидать), в любом случае при выходе из функции `f` отработает деструктор для объекта `x`, который уничтожит динамический массив, в результате чего объект `c` окажется в заведомо ошибочном состоянии, поскольку будет содержать указатель на уничтоженный массив. К возникновению ошибки теперь приведёт любое действие с объектом `c`, если же никаких действий не предпринимать, то ошибка возникнет при уничтожении объекта `c`, когда деструктор попытается вновь уничтожить уже уничтоженный массив.

Очевидно, что для объектов класса `Cls1` побитовое копирование нас не устраивает и необходимо проинструктировать компилятор о том, как именно создавать копии этих объектов, чтобы всё оставалось *корректно*. В языке Си++ для этого предусмотрен специальный

вариант конструктора, называемый ***конструктором копирования***. Конструктор копирования имеет ровно один параметр, причём тип этого параметра представляет собой *ссылку на объект данного (описываемого) типа — класса или структуры*; в большинстве случаев эту ссылку снабжают модификатором `const`, чтобы показать, что при создании копии исходный объект не изменится. Конструктор копирования, таким образом, представляет собой инструкцию компилятору, как создать объект данного типа, уже имея один такой объект — то есть, попросту говоря, *как создать копию имеющегося объекта*. Снабдим конструктором копирования наш класс `Cls1`:

```
class Cls1 {
    int *p;
public:
    Cls1() { p = new int[20]; }
    Cls1(const Cls1& a) {
        p = new int[20];
        for(int i=0; i<20; i++)
            p[i] = a.p[i];
    }
    ~Cls1() { delete [] p; }
    // ...
};
```

Теперь ситуация при вызове функции `f` будет выглядеть так, как показано на рис. 2 — у каждого объекта будет свой экземпляр массива `p`.

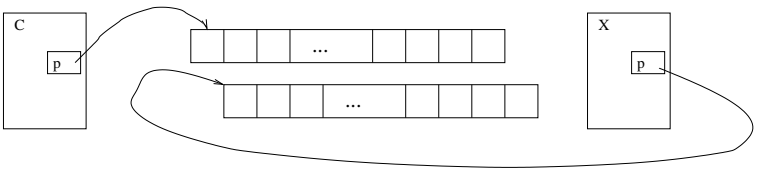


Рис. 2. Схема структуры данных после применения конструктора копирования

3.11. Временные и анонимные объекты

С анонимными и временными объектами мы уже встречались ранее (см. §2.3, стр. 24; §3.4, стр. 43). Анонимные объекты обычно применяются в случае, когда объект создаётся, чтобы быть использованным лишь один раз; как правило, в такой ситуации давать объекту имя не хочется, к тому же код с использованием анонимного объекта оказывается более лаконичным и наглядным. Для введения в

выражение анонимного объекта используют имя его конструктора, снабжённое списком фактических параметров (возможно, пустым). Пусть, например, у нас есть объекты `t` и `z` класса `Complex` и нам нужно занести в переменную `t` значение `z`, помноженное на мнимую единицу. Можно сделать это так:

```
Complex imag_1(0, 1);
t = z * imag_1;
```

а можно воспользоваться анонимным объектом, что гораздо удобнее:

```
t = z * Complex(0, 1);
```

С временными объектами ситуация несколько сложнее. В отличие от анонимных объектов, которые программист вводит в выражение в явном виде, объекты временные компилятор порождает самостоятельно, без их явного упоминания, хотя мы и можем в большинстве случаев однозначно предсказать появление временного объекта. Самый простой случай такого появления — передача параметра в функцию с использованием конструктора преобразования. Так, в примере на стр. 42 мы вызывали функцию `f`, предполагающую параметр типа `Complex`, от фактического аргумента, представляющего собой обычное число с плавающей точкой; используя введённый нами конструктор преобразования, компилятор создал *временный объект* типа `Complex`, который и был в итоге передан в функцию `f`.

Рассмотрим не столь очевидный случай появления временного объекта. Пусть у нас имеются переменные `a`, `b` и `c`, имеющие тип `Complex`, и мы пытаемся вычислить их сумму, например, так:

```
t = a + b + c;
```

Операции сложения в этом операторе будут выполняться, как обычно, слева направо, причём второе сложение в качестве своего «левого» аргумента получит результат первого сложения, который, очевидно, имеет тип `Complex`, то есть *является объектом типа* `Complex`. Ясно, что эту сумму `a` и `b` компилятор должен представить в виде объекта, чтобы *для него* вызвать операцию сложения, передав объект `c` в качестве параметра. Как вы уже догадались, компилятор создаёт для этого временный объект. Этот пример демонстрирует частный случай более общей ситуации, когда компилятор вынужден порождать временные объекты, а именно — при использовании в выражении вызова некоторой функции, которая возвращает значение типа объект, причём возврат производится именно по значению (а не, например, по ссылке). В данном случае в роли такой функции выступает `operator+`.

Несмотря на очевидное различие, анонимные и временные объекты имеют между собой много общего; анонимные объекты можно считать частным случаем временных[15]. В первую очередь отметим *время жизни* временных и анонимных объектов: они существуют до момента окончания вычисления выражения, содержащего их, после чего уничтожаются; например, в операторе вычисления выражения ради побочного эффекта временные объекты просуществуют до окончания выполнения оператора — «до точки с запятой». Если соответствующий класс имеет деструктор, этот деструктор будет вызван. Из этого правила есть одно исключение: если временный или анонимный объект был использован в качестве инициализатора ссылки, то такой объект будет существовать до тех пор, пока существует ссылка.

Второе правило, которое необходимо знать и учитывать, состоит в том, что **на временный или анонимный объект нельзя ссылаться неконстантной ссылкой**. Это ограничение особенно заметно при передаче параметров в функции: если функция принимает параметр типа «объект» по значению или по ссылке на константу, то в качестве такого параметра может быть использован временный или анонимный объект, тогда как если параметр передаётся по неконстантной ссылке, то временные и анонимные объекты для такого параметра не подходят. Это правило при ближайшем рассмотрении оказывается вполне логичным. В самом деле, наличие параметра-ссылки, если эта ссылка не обозначена как константная, обычно подразумевает, что соответствующий параметр используется как *выходной*, то есть предполагается, что через этот параметр функция *возвращает* некоторую информацию. Помещать такую информацию во временный объект, очевидно, бессмысленно.

Из этого можно сделать очевидный методологический вывод: используйте модификатор `const` для всех ссылок, для которых это возможно. Неконстантные ссылки в качестве параметров функций следует использовать тогда и только тогда, когда функция заведомо должна модифицировать переменную, переданную через такой параметр, то есть параметр используется для передачи информации из функции к вызывающему — например, когда функция должна возвратить больше одного значения.

3.12. Значения параметров по умолчанию

Язык Си++ позволяет при объявлении функции (т. е. в том месте единицы трансляции, где впервые фигурирует прототип функции)

[15]Более того, Страуструп в своих книгах вообще не делает между ними явного различия, называя «временными объектами» обе рассматриваемые сущности.

задать для некоторых или всех её параметров ***значения по умолчанию***, в результате чего при вызове функции можно будет указать меньшее количество параметров. Недостающие параметры компилятор подставит сам. Например, опишем следующую функцию:

```
void f(int a = 3, const char *b = "string", int c = 5);
```

Теперь возможны такие вызовы:

```
f(5, "name", 10);
f(5, "name"); // то же, что f(5, "name", 5);
f(5);         // то же, что f(5, "string", 5);
f();          // то же, что f(3, "string", 5);
```

На значения по умолчанию накладываются определенные ограничения. Для функции может быть задано значение по умолчанию любого количества её параметров, но при этом *все параметры функции, следующие в списке параметров за первым, имеющим значение по умолчанию, должны также иметь значение по умолчанию.* Например, следующие описания корректны:

```
void f(int a = 0, int b = 10, int c = 20);
void f(int a, int b = 10, int c = 20);
void f(int a, int b, int c = 20);
```

а следующие — некорректны:

```
void f(int a = 0, int b, int c = 20); // ОШИБКА
void f(int a = 0, int b = 0, int c);  // ОШИБКА
void f(int a = 0, int b, int c);      // ОШИБКА
     // b и c должны иметь умолчание, т.к. его имеет a

void f(int a, int b = 10, int c);     // ОШИБКА
     // c должен иметь умолчание, т.к. его имеет b
```

Это ограничение введено в язык, чтобы упростить компилятору поиск подходящей функции. Если при вызове указано n фактических параметров, компилятор сопоставляет их с n *первыми* формальными параметрами функции. Так, если задана функция

```
int f(int a, const char *str = "name", int *p = 0);
```

то следующий фрагмент будет ошибочен:

```
int x;
f(5, &x);
```

поскольку фактический параметр `&x` имеет тип `int *`, а сопоставлен он будет строго со вторым параметром функции `f`, который имеет тип `const char *`. Кроме того, выражение, задающее значение по умолчанию, также может стать причиной ошибки. Рекомендуется использовать в качестве таковых только *константные выражения*, то есть такие, которые компилятор может вычислить во время компиляции; хотя в языке такого ограничения в явном виде нет, его вводят некоторые компиляторы.

В литературе встречается утверждение, что параметры по умолчанию представляют собой ещё одно проявление статического полиморфизма. На самом деле это утверждение сомнительно, поскольку никакого отношения к *различным типам аргументов* этот механизм не имеет; впрочем, вопрос тут, как водится, терминологический, а термины можно определять по-разному.

Введение параметров по умолчанию приводит к необходимости уточнения формулировок, связанных с конструкторами специальных видов. В §§3.3, 3.4 и 3.10 мы рассматривали специфические конструкторы, говоря, что:

- конструктор без параметров воспринимается компилятором как *конструктор по умолчанию*;
- конструктор с одним параметром, имеющим тип, отличный от описываемого, воспринимается компилятором как *конструктор преобразования*;
- конструктор с одним параметром, имеющим тип «ссылка на описываемый класс или структуру», воспринимается компилятором как *конструктор копирования*.

Учитывая возможность задания значений параметров по умолчанию, следует говорить, что компилятор воспримет как конструктор специального вида такой конструктор, который *допускает его вызов* с соответствующими параметрами. Так, мы могли бы в классе `Complex` описать всего один конструктор, который бы служил и конструктором по умолчанию, и конструктором преобразования, и обычным конструктором от двух аргументов:

```
Complex(double a_re = 0, double a_im = 0)
    { re = a_re; im = a_im; }
```

В самом деле, такой конструктор может быть вызван и без параметров, то есть как конструктор по умолчанию, и с одним параметром типа `double`, то есть как конструктор преобразования из типа `double`.

3.13. Описание метода вне класса. Области видимости

До сих пор все описываемые нами методы состояли из одной-двух строк кода. Конечно, так получается далеко не всегда, а размер тела метода, вообще говоря, не ограничен, как не ограничен и размер тела обычной функции[16].

Описание класса при наличии в нём нескольких методов значительного объёма может стать (в силу своих размеров) совершенно недоступным для понимания. Чаще всего программисты читают описания классов, чтобы узнать, как с этими классами работать. Ясно, что для этого нужно знать список (публичных) методов. Пока описание класса целиком умещается на экране, список его методов можно охватить одним взглядом; если же в классе описаны методы с громоздкими телами, для ознакомления с заголовками методов может потребоваться долгое перелистывание кода туда и обратно, причём в процессе поиска заголовка одного метода программист может успеть забыть про другие, так что изучение класса становится занятием долгим и утомительным.

Есть и ещё одна проблема с телами методов, которая возникает при написании многомодульных программ. В языке Си мы помещали описания структур, необходимые более чем в одном модуле, в заголовочные файлы. В Си++ с классами и структурами ситуация совершенно такая же: если мы хотим использовать некоторый класс или структуру в нескольких модулях, следует поместить описание этого класса или структуры в заголовочный файл и включить этот файл директивой `#include "..."` во все нужные модули. Если при этом класс или структура содержит тела методов, это будет означать, что код, составляющий их тела, окажется откомпилирован в **каждом** из модулей. Если таких модулей много, в итоговом исполняемом файле окажется значительное количество дублируемого кода: методы нашего класса (один и тот же код!) будут присутствовать в таком количестве копий, сколько модулей используют наш класс.

Обе проблемы — громоздкость описания класса и дублирование объектного кода методов — решаются вынесением тел методов за пределы описания класса (называемого также ***заголовком клас-***

[16]Это, впрочем, не означает, что написание громоздких функций в чём-то правильно. Чем функция длиннее, тем труднее стороннему программисту понять, что она делает. Многие программисты придерживаются точки зрения, что всё описание любой функции (тело вместе с заголовком) должно укладываться в 25 строк или в крайнем случае быть чуть-чуть больше. Такую функцию можно охватить одним взглядом. Если код функции разбух и стал гораздо больше, из неё следует выделить логические части и оформить их в виде отдельных функций, то есть разбить одну функцию на несколько. Всё это справедливо и для методов.

са). При этом в заголовке класса оставляется только прототип (заголовок) функции-метода, то есть тип возвращаемого значения, имя метода, список формальных параметров и (возможно) модификатор `const`, после чего вместо тела метода ставится точка с запятой, как и после обычного прототипа функции. Тело функции-метода при этом описывается в другом месте, за пределами заголовка класса, возможно даже, что в другом файле — обычно так случается, если заголовок класса вынесен в заголовочный файл; тела методов при этом описываются в файле реализации соответствующего модуля, так что при трансляции модулей, *включающих* этот заголовочник, компилятор видит заголовок класса, но не видит реализацию его методов.

При описании функции-метода за пределами заголовка класса необходимо указать компилятору, что речь идёт именно о методе определённого класса, а не о простой функции. Это делается с помощью ***символа раскрытия области видимости***, в роли которого в Си++ выступает два двоеточия «`::`» (иногда программисты называют этот символ «четвероточием»). Например, если мы описываем некий класс `C1` и в нём есть конструктор по умолчанию и некоторые методы `f` и `g`, вынесение описания методов за пределы класса может выглядеть так:

```
class C1 {
    // ...
public:
    C1();
    void f(int a, int b);
    int g(const char *str) const;
};

// ... //

C1::C1()
{
    // тело конструктора
}

void C1::f(int a, int b)
{
    // тело метода f
}

int C1::g(const char *str) const
{
    // тело метода g
}
```

Смысл «раскрытия области видимости» можно пояснить следующим образом. Имена полей и методов локализованы в классе: если в классе есть метод с именем `f`, то вне класса мы можем использовать идентификатор `f` для других целей, либо вовсе не использовать его; появление идентификатора `f` вне класса не имеет никакого отношения к методу `f`, описанному в классе. Вместе с тем в некоторых случаях имя метода `f` всё же появляется вне класса именно как имя этого метода — например, при вызове его для объекта класса. Таким образом, в отличие от, например, локальных переменных в функциях, имена членов класса *доступны* вне класса (конечно, если они не закрыты механизмом защиты), но только если имеются указания на то, что требуется имя из класса; при вызове метода *для объекта* таким указанием считается тип этого объекта.

Говорят поэтому, что класс (или структура) представляет собой ***область видимости***. В области видимости имеются свои имена, не конфликтующие с именами вне её даже при совпадении идентификаторов. Имена, локализованные в области видимости, от этого не становятся, вообще говоря, недоступны; они доступны, но только при наличии указаний на область видимости. Символ раскрытия области видимости как раз и предназначен для таких указаний. Можно сказать, что один и тот же метод имеет внутри класса `C1` (то есть в телах его методов) имя `f`, а вне класса — имя `C1::f`.

Символ раскрытия области видимости применяется не только для описания методов вне заголовка класса. Если опустить имя области видимости слева от символа «`::`», подразумевается *глобальная* область видимости; например, если в вашем классе есть метод `func` и при этом в вашей программе есть обычная функция с таким же именем, вы можете к ней обратиться из методов вашего класса, назвав её «`::func`». С другими случаями применения символа «`::`» мы столкнёмся позже при рассмотрении статических полей и методов.

Сделаем ещё одно важное замечание. Часто можно столкнуться с утверждением, что «четвероточие» — это якобы *операция* (и, в частности, её включают в список операций, которые нельзя переопределять). Чтобы опровергнуть это утверждение, достаточно заметить, что ни слева, ни справа от символа «`::`» не может стоять никакое *выражение*: ни имя области видимости, ни имя поля или метода не могут быть *вычислены*, их можно указывать только в явном виде. Если считать «`::`» операцией, она окажется «нуль-арной», то есть операцией без операндов, но таких «операций» обычно всё-таки не бывает. Логичнее считать, что символ «`::`» никакого отношения к операциям не имеет, это просто символ.

3.14. «Подставляемые» функции (inline)

К нынешнему моменту у читателя могло сложиться ощущение, что объектно-ориентированное программирование вместе с абстрактными типами данных — это жутко неэффективно из-за того, что к закрытым полям классов и структур приходится обращаться через функции-методы, а ведь вызов функции предполагает помещение в стек её параметров (как минимум указателя `this`), создание и последующую ликвидацию стекового фрейма и всё такое прочее. Для мало-мальски нетривиальных функций всё это не создаёт проблем, поскольку операции, связанные с техническим исполнением вызова функции и возврата из неё, обычно отнимают ничтожное время в сравнении с выполнением тела функции; но ведь мы неоднократно видели в примерах функции-методы, попросту возвращающие значение поля объекта, а скопировать поле из заданной области памяти можно одной командой `mov`!

Действительно, если бы компилятор оформлял все вызовы методов «по-честному», это могло бы привести к ощутимым потерям, но, к счастью, всё не так плохо. Дело в том, что компилятор Си++ некоторые функции считает «подставляемыми» (англ. *inline*; встречается также русский перевод «встраиваемые»). Когда в тексте программы встречается обращение к такой функции, компилятор вместо обычного вызова *подставляет машинный код тела функции*. После этого оптимизатор, работающий на уровне машинных команд, выкидывает всё лишнее, так что итоговый код становится ничуть не менее эффективным, чем если бы мы обращались к полям объекта без всяких вызовов методов — собственно говоря, на уровне машинного кода никаких вызовов и не происходит.

Конечно, такое имеет смысл лишь для функций с очень коротким телом. Вспомним теперь, что́ мы обсуждали в предыдущем параграфе: только очень короткие тела методов стоит оставлять в заголовке класса, а все прочие выносить за его пределы. Сопоставив одно с другим, мы сможем лучше понять логику, стоящую за следующим соглашением: **компилятор Си++ пытается обрабатывать как «подставляемые» все методы, тела которых описаны непосредственно в заголовке класса**.

Мы можем предложить компилятору обрабатывать в качестве «подставляемой» любую функцию, для этого достаточно перед её описанием поставить ключевое слово `inline`:

```
inline int f(int x)
{
   //...
}
```

Естественно, сделать это можно и с методом, тело которого мы решили вынести за пределы заголовка класса, например, чтобы не загромождать этот заголовок.

С inline-функциями связано несколько интересных моментов, которые стоит понимать. Прежде всего отметим, что ни описание тела метода в заголовке класса, ни даже явным образом указанная директива `inline` в действительности ни к чему компилятор не обязывают: если по тем или иным причинам он считает, что функцию с таким телом будет эффективнее обрабатывать как обычную вызываемую функцию, а не как «подставляемую», то именно так он с ней и поступит.

Второй момент не столь очевиден. Если в одном из ваших модулей вводится inline-функция, которую вы хотите сделать доступной для других модулей, то её *описание* — полностью, вместе с телом — следует вынести в заголовочный файл. Чтобы понять, почему это так, достаточно вспомнить, что редактор связей при сборке программы из объектных модулей расставляет адреса, по которым модули обращаются к объектам (переменным и функциям, или, говоря шире, областям памяти), введённым в других модулях; больше никаких изменений в код редактор связей внести не может, в том числе, естественно, не может он и подставить вместо вызова функции какой-то там фрагмент кода, будь он хоть трижды телом этой функции. Следовательно, подстановку кода inline-функции должен выполнить компилятор во время компиляции, но ведь при компиляции одного модуля компилятор не видит текста других модулей — собственно говоря, в этом и состоит суть *раздельной трансляции*. Всё, что знает компилятор о других модулях — это то, что написано в подключённых заголовочных файлах, они как раз для этого и создаются; так что, коль скоро компилятору для подстановки inline-функции нужно, очевидно, знать её тело, то это тело следует вынести в заголовочник.

Любопытно, что компилятор может создать как обычную, так и «подставляемую» версию для одной и той же функции. Например, компилятор попросту вынужден генерировать обычную (вызываемую) подпрограмму в случае, если где-то в вашей программе от этой подпрограммы берут адрес и заносят его в указатель на функцию. Впрочем, ни это, ни возможный отказ компилятора рассматривать функцию как «подставляемую» в любом случае не может повлиять на семантику итоговой программы — только на размер машинного кода.

Так или иначе, об inline-функциях стоит помнить хотя бы для того, чтобы не бояться потерять эффективность из-за вызовов коротких методов.

3.15. Инициализация членов класса в конструкторе

До сих пор мы задавали значения полей объекта с помощью обычного оператора присваивания в теле конструктора. Такой способ может оказаться неприемлемым; действительно, никто не мешает объявить в классе поле, имеющее, в свою очередь, тип класс, причём не имеющий конструктора по умолчанию. Рассмотрим эту ситуацию подробнее. Пусть `A` — класс, единственный конструктор которого требует на вход два параметра типа `int`:

```
class A {
    // ...
public:
    A(int x, int y) { /*...*/ }
    // ...
};
```

Опишем теперь класс `B`, в котором есть поле типа `A`. Для простоты картины будем считать, что в классе `B` нам нужен только конструктор по умолчанию (для других конструкторов ситуация будет абсолютно аналогична):

```
class B {
    A a;
public:
    B();
    // ...
};
```

Рассмотрим теперь тело конструктора `B`. В нём (то есть во время его работы) **уже** доступно поле `a`, а для этого, как мы знаем, должен был отработать конструктор класса `A`. Но ведь единственный конструктор класса `A` требует параметров! Как же их ему передать?

Чтобы решить возникшую проблему, в Си++ введён специальный синтаксис для ***инициализации полей объекта***. При описании конструктора между его заголовком и началом тела (открывающей фигурной скобкой) можно поставить двоеточие, после которого через запятую перечислить вызовы конструкторов для некоторых или всех полей класса. В рассматриваемом примере это будет выглядеть так:

```
B::B() : a(2, 3) { /*...*/ }
```

Здесь мы указываем компилятору, что поле `a` следует инициализировать («сконструировать») с помощью конструктора от двух параметров (`2` и `3`). Обнаружив такой код, компилятор вставит вызов

соответствующего конструктора класса `A` непосредственно в начале тела конструктора `B`. Отметим, что так можно инициализировать любые поля, а не только поля типа класс. В частности, для нашего типа `Complex` конструкторы могли бы выглядеть и так:

```
class Complex {
    double re, im;
public:
    Complex(double a_re, double a_im) : re(a_re), im(a_im) {}
    Complex(double a_re) : re(a_re), im(0) {}
    Complex() : re(0), im(0) {}
    // ...
```

Сделаем ещё одно важное замечание. **Инициализаторы полей должны следовать в списке после двоеточия в том же порядке, в котором сами поля описаны в классе**. Иное, формально говоря, не является ошибкой, но хороший компилятор обязательно выдаст предупреждение.

3.16. Перегрузка операций простыми функциями

В классе `Complex` мы перегружали символы арифметических операций в виде функций-методов. Благодаря наличию в классе конструктора преобразования возможно, например, прибавить к комплексному числу действительное:

```
Complex z, t;
// ...
z = t + 0.5;
```

(при этом константа `0.5` будет преобразована к объекту `Complex` с помощью конструктора преобразования). В то же время следующая операция окажется ошибочной:

```
z = 0.5 + t; // ошибка!
```

Дело тут в том, что метод `operator+` может быть вызван только для объекта класса `Complex`, а константа `0.5` таковым не является. В отличие от аргументов функций, объекты, для которых вызываются методы, компилятор не преобразует.

Эту проблему также можно решить. Дело в том, что операцию сложения двух комплексных чисел можно представить не только как метод самого комплексного числа (с одним параметром, по принципу «прибавь к себе этот параметр и скажи, что получится»), но и как стороннюю функцию, которая по двум заданным комплексным числам выдаёт другое комплексное число. Итак, уберём `operator+` из класса `Complex`, а вне класса напишем следующее:

```
Complex operator+(const Complex& a, const Complex& b)
{
    return Complex(a.get_re() + b.get_re(),
                   a.get_im() + b.get_im());
}
```

Теперь мы можем написать:

```
Complex z, t;
// ...
z = t + 0.5;
z = 0.5 + t;
```

и никаких ошибок это не вызовет.

Можно считать, что выражение `a + b` компилятор пытается превратить в одно из двух следующих выражений:

`a.operator+(b)` или `operator+(a, b)`

и то же самое справедливо почти для всех бинарных операций; с унарными операциями ситуация аналогична, например, `~a` превращается в одно из

`a.operator~()` или `operator~(a)`

Интересно заметить, что приоритета ни тот, ни другой вариант не имеют; если вы предусмотрите в вашей программе и метод, и внешнюю функцию для одной и той же операции (и для одних и тех же типов операндов), компилятор откажется делать выбор между ними и при попытке использовать операцию в инфиксной форме выдаст ошибку.

Перегрузка символов стандартных операций в виде отдельных функций (а не методов в классах) может оказаться полезной также и в том случае, если нам необходимо ввести операцию для объектов некоторого класса, который мы по тем или иным причинам не можем изменить; например, этот класс может быть частью библиотеки, созданной другим программистским коллективом, и от неё у нас может не быть исходных текстов (к сожалению, несмотря на успехи движения Open Source, такие ситуации пока что не редкость).

3.17. Дружественные функции и классы

В некоторых случаях бывает полезно сделать исключения из запретов, налагаемых механизмом защиты. В языке Си++ класс или структура, имеющие защищённую часть, могут объявить ту или иную функцию своим «другом» (friend); в этом случае из тела такой ***дружественной функции*** все детали реализации класса или

структуры будут доступны. Можно объявить «дружественным» также целый класс или структуру; эффект от этого будет такой же, как если бы мы объявили дружественными все методы дружественного класса. Иначе говоря, если в классе `A` будет заявлено, что класс `B` является для него дружественным, то во всех методах класса `B` будут доступны все детали реализации класса `A`.

Применять механизм дружественных функций и классов следует с осторожностью. Мы обсуждали в §2.4, что защита деталей реализации класса — механизм очень полезный; поэтому необдуманные исключения из него могут нанести существенный вред. Одна из ситуаций, в которых применение механизма дружественных функций можно считать практически безопасным — это вынос перекрытых символов стандартных операций за пределы класса, как это предлагалось в предыдущем параграфе. В теле функции `operator+` мы были вынуждены пользоваться методами `get_re` и `get_im`, поскольку из функции, не являющейся формально методом класса `Complex`, прямого доступа к полям `re` и `im` нет. Однако использование методов доступа (так называемых аксессоров) не всегда удобно, к тому же их может попросту не быть; часто бывает так, что некоторые детали реализации не следует делать доступными даже через метод.

Ситуацию можно исправить с помощью механизма дружественных функций. Для начала в заголовок класса `Complex` вставим директиву `friend`, объявив функцию `operator+` дружественной. Для этого нужно записать прототип дружественной функции, предварив его директивой `friend`:

```
class Complex {
    friend Complex operator+(const Complex&, const Complex&);
    //...
```

Теперь мы можем переписать функцию `operator+`, используя напрямую поля складываемых объектов без обращения к методам `get_re` и `get_im`:

```
Complex operator+(const Complex& a, const Complex& b)
{
    return Complex(a.re + b.re, a.im + b.im);
}
```

Сравните это описание с тем, которое мы приводили на стр. 67.

Конечно, дружественной может быть и обычная функция:

```
class Cls1 {
    friend void f(int, const char *);
    //...
```

```
};

void f(int, const char *)
{
    // здесь можно использовать закрытые поля Cls1
}
```

Чтобы сделать дружественным сразу целый класс, следует после слова `friend` написать «`class` *имя*»:

```
class A {
    friend class B;
    //...
};
```

Использовать такой вариант «дружбы» следует с особенной осторожностью, только в ситуации, когда понятия, описываемые обоими классами, тесно связаны между собой; желательно сначала задать себе вопрос, нельзя ли обойтись без `friend`. Бóльшим злом, чем использование «дружественных» отношений, можно считать разве что снятие защиты с внутренних полей или введение публичных методов доступа к деталям, которые по смыслу должны быть скрыты: ясно, что «дружественность» всё-таки ограничивает объём кода, который придётся просмотреть при изменении реализации класса, так что использование `friend` предпочтительнее полного отказа от защиты. Позже мы рассмотрим примеры, в которых применение «дружественных» классов оказывается оправданным.

В заключение обсуждения директивы `friend` отметим один довольно популярный миф; из текста некоторых распространённых пособий по Си++ их читатели делают вывод, что (якобы) символы стандартных операций для класса можно переопределять либо функцией-членом (методом), либо функцией-другом. Не верьте! **Функция, не являющаяся методом класса и переопределяющая для объектов класса символ стандартной операции (то есть поименованная с использованием слова `operator`), *никоим образом не обязана* быть классу другом или кем-то ещё.** Просто такие функции сравнительно часто объявляют дружественными классу из соображений удобства их написания; но это совершенно не обязательно.

3.18. Переопределение операций присваивания

Как уже говорилось выше, в Си++ можно переопределить («перегрузить») символ любой операции, участвующей в выражениях, за

исключением операции выборки поля (точка) и тернарной условной операции. При этом некоторые операции при их перегрузке имеют определённые особенности, которые необходимо учитывать.

Начнём с переопределения операций присваивания. Напомним, что присваивание является в языках Си и Си++ *операцией*, а не оператором, как в Паскале и большинстве других языков: запись вида «`a =` *выражение*» сама по себе является выражением, имеющим значение, и, может входить в состав других выражений. То же самое можно сказать и про операции присваивания, совмещённые с арифметическими действиями, такие как `+=`, `-=`, `<<=`, `|=` и др. Операции присваивания, как и другие операции, могут быть переопределены для классов и структур, введённых пользователем; но на них действует ограничение: **операции присваивания можно переопределять только как методы класса или структуры**, а определять их в виде обычных функций нельзя.

Обычно при переопределении операций присваивания учитывают, что обычное присваивание возвращает значение, которое только что было присвоено. Из этих соображений из операции присваивания возвращают либо копию объекта, либо (что предпочтительнее) константную ссылку на объект, для которого операция была вызвана. Это, однако, не обязательно: язык Си++ не требует именно такого оформления операций присваивания. Можно, в частности, сделать операцию присваивания функцией типа `void`, то есть не возвращающей никакого значения. Например, для нашего класса `Complex` мы могли бы написать такие операции:

```
class Complex {
    // ...
    const Complex& operator=(const Complex& c)
        { re = c.re; im = c.im; return *this; }
    const Complex& operator+=(const Complex& c)
        { re += c.re; im += c.im; return *this; }
    // ...
};
```

или, если нас не слишком волнуют семантические традиции, можно было бы написать и так:

```
class Complex {
    // ...
    void operator=(const Complex& c)
        { re = c.re; im = c.im; }
    void operator+=(const Complex& c)
        { re += c.re; im += c.im; }
    // ...
};
```

Здесь стоит ответить на один вопрос, часто возникающий у студентов. Операции +=, -= и другие подобные им являются с точки зрения языка Си++ полностью самостоятельными, то есть, например, операция += никак не связана ни с операцией =, ни с операцией +. Если мы опишем в некотором классе операции = и +, это само по себе не даст нам возможности применять к объектам этого класса операцию +=, она должна быть описана отдельно. В принципе она может делать что-то совершенно иное, нежели последовательно применённые = и +, ведь компилятор никак не может проверить соответствие одного и другого; но лучше от таких трюков воздержаться, чтобы не создавать в программе лишних ребусов для разгадывания.

Аргумент операции присваивания не обязан иметь тот же тип, что и описываемый класс. Так, для комплексных чисел мы могли бы определить и присваивание комплексной переменной действительного числа:

```
class Complex {
    // ...
    void operator=(double x) { re = x; im = 0; }
    // ...
};
```

Операция присваивания, аргумент которой представляет объект того же класса или ссылку на такой объект, как в предыдущем примере, имеет одну важную особенность: такая операция *генерируется неявно*, если её не описать. Неявные методы мы подробно обсудим в следующем параграфе.

3.19. Методы, возникающие неявно

Если описать в программе на Си++ структуру или даже класс, не содержащий (по крайней мере на первый взгляд) никаких конструкторов, то переменную такого типа всё же окажется возможным создать. Это вполне понятно с позиций здравого смысла: ведь структуры в Си++ часто используются в их исходной роли, пришедшей из языка Си, то есть в качестве обычной структуры данных, безо всяких методов. Между тем семантика языка Си++ подразумевает, что каждый экземпляр структуры или класса является объектом, а для каждого объекта при его создании вызывается конструктор. Возникающее противоречие решается введением понятия ***неявного конструктора***.

Неявный конструктор — это конструктор, который генерируется компилятором автоматически, несмотря на отсутствие соответствующего конструктора в коде, описывающем класс или структуру. Компилятор Си++ неявно генерирует только два вида конструкторов:

конструкторы по умолчанию (то есть конструкторы без параметров) и конструкторы копирования. При этом **конструктор копирования неявно генерируется для любого класса или структуры, в которых программист не описал конструктор копирования явно**, то есть получается, что конструктор копирования на самом деле присутствует (явно или неявно) вообще в любом классе или структуре. Неявный конструктор копирования производит копирование наиболее очевидным способом: поля, которые сами имеют конструкторы копирования, копируются с помощью этих конструкторов, прочие поля — побитовым копированием.

С **конструктором по умолчанию** ситуация чуть сложнее: он **генерируется неявно, если программист не описал в структуре или классе вообще ни одного конструктора**. Если в классе или структуре явно описать хотя бы один конструктор (любой), компилятор не будет генерировать неявный конструктор по умолчанию, поскольку сочтёт, что программист взял заботу о конструировании в свои руки. Именно поэтому в примерах, которые рассматривались в §2.3, мы могли описывать переменные типа `str_complex` «по-сишному», пока не ввели первый конструктор, после чего возможность описания переменных этого типа без указания параметров нами была утрачена до тех пор, пока мы (уже сами, явно) не определили конструктор по умолчанию.

Неявная версия конструктора по умолчанию использует для инициализации полей класса, в свою очередь, конструкторы по умолчанию, определённые для соответствующих типов. Можно считать, что для встроенных типов, таких как числа или указатели, конструкторы по умолчанию тоже есть, просто они ничего не делают.

Аналогично обстоят дела и с деструкторами. Считается, что деструктор есть в любом классе и любой структуре, даже если мы его там не описали: в этом случае компилятор создаст его неявно. В простейшем случае такой неявный деструктор не будет делать ничего, но если в нашем классе или структуре есть поля, имеющие нетривиальные деструкторы, то наш неявный деструктор будет содержать их вызовы.

Последний метод, генерируемый неявно — это операция присваивания объекту того же типа. Это вполне понятно, ведь в чистом Си можно было присваивать между собой переменные, имеющие один структурный тип, и эта возможность была унаследована в Си++. Если говорить точнее, то, если в некотором классе `A` не описать явным образом операцию присваивания с параметром, имеющим тот же тип `A` или ссылку на него, то компилятор неявно создаст операцию со следующим профилем:

```
class A {
   // ...
   A& A::operator=(const A& other);
   // ...
};
```

Такая неявным образом возникшая операция будет использовать для копирования содержимого каждого поля определённую для этого поля его собственную операцию присваивания, если она у него есть. Интересно, что для некоторых полей операции присваивания может не найтись — например, если поле имеет тип «ссылка» или объявлено как *константное*. В этом случае компилятор не сможет сгенерировать неявную операцию присваивания.

Сделаем ещё одно важное замечание. Поскольку конструктор копирования может быть сгенерирован неявно, отсутствие явно описанного конструктора копирования не означает невозможности создания копии объекта. Существует, однако, **способ запретить копирование объектов некоторого класса**: для этого достаточно **описать конструктор копирования** явно, но сделать это **в приватной части класса**. Такой приём часто применяют для классов, объекты которых по смыслу копироваться не должны, чтобы исключить случайные ошибки, связанные, например, с передачей их по значению. Ясно, что объект, для которого копирование запрещено, не может быть ни передан по значению в функцию, ни возвращён из функции; это не исключает, разумеется, передачи по ссылке. Интересно, что применение объекта, копирование которого запрещено, в роли *поля* другого объекта сделает невозможной генерацию неявного конструктора копирования для этого второго объекта, так что его копирование будет запрещено автоматически; впрочем, это не мешает создать конструктор копирования явно.

Аналогичный приём можно применить и для запрещения присваивания объектов: достаточно описать операцию присваивания с аргументом «константная ссылка на объект описываемого класса» **в приватной части класса**:

```
class A {
    // ...
private:
    void operator=(const A& ref) {} // no assignments
};
```

Иногда в приватную часть класса убирают деструктор; этому приёму мы посвятим §5.11.

3.20. Переопределение операции индексирования

Операция извлечения элемента из массива, обозначаемая квадратными скобками, как известно, является арифметической операцией над указателем и целым числом; в языке Си выражение «`a[b]`» полностью эквивалентно выражению «`*(a+b)`», что лишний раз подчёркивает арифметическую сущность индексирования. В языке Си++ операцию индексирования можно переопределить для объектов класса или структуры, тем самым заставив объект в некоторых случаях выглядеть синтаксически похожим на обычный массив или даже просто исполнять роль массива. Подобно операциями присваивания, **операция индексирования может быть переопределена только как метод класса или структуры**.

Для примера опишем класс, объект которого представляет собой массив целых чисел с заранее неизвестным размером, который при обращении к несуществующим (пока) элементам автоматически увеличивается в размерах. Хранить элементы массива будем в динамически создаваемом массиве, скрытом в приватной части класса. Исходно создадим массив размером 16 элементов, а при возникновении такой необходимости будем удваивать его размеры до тех пор, пока нужный нам индекс не станет допустимым.

Заметим, что при попытке копирования объекта такого класса возникнет проблема с разделением одного и того же массива в динамической памяти между двумя объектами класса; подробно эта проблема описана в §3.10). В нашей упрощенной версии мы просто запретим присваивание и копирование, описав фиктивные конструктор копирования и операцию присваивания в приватной части. Описание полноценного конструктора копирования и полноценной операции присваивания оставим читателю в качестве упражнения. Для начала напишем заголовок класса:

```
class IntArray {
    int *p;             // указатель на хранилище
    unsigned int size; // текущий размер хранилища
public:
    IntArray() {
        size = 16;
        p = new int[size];
    }
    ~IntArray() { delete[] p; }
    int& operator[](unsigned int idx);
private:
    void Resize(unsigned int required_index);
        // запретим копирование и присваивание
    void operator=(const IntArray& ref) {}
```

```
    IntArray(const IntArray& ref) {}
};
```

Тела конструктора и деструктора имеют сравнительно небольшой размер, поэтому мы совершенно спокойно можем оставить их внутри заголовка класса. Тело операции индексирования, напротив, будет состоять из нескольких строк, поэтому мы опишем его отдельно. Для лучшей ясности его реализации мы предусмотрели также вспомогательную функцию `Resize`, которая будет осуществлять изменение размера массива. Поскольку эта функция, очевидно, является деталью реализации и не предназначена для пользователя (с точки зрения пользователя наш массив представляется бесконечным), мы скрыли эту функцию в приватной части класса.

Читатель, возможно, обратил внимание на тип возвращаемого значения операции индексирования. Функция `operator[]` возвращает *ссылку* на соответствующий элемент массива, чтобы сделать возможным как выборку значения из массива, так и присваивание его элементам новых значений. Опишем теперь тело операции индексирования:

```
int& IntArray::operator[](unsigned int idx)
{
    if(idx >= size)
        Resize(idx);
    return p[idx];
}
```

Нам осталось описать функцию `Resize`:

```
void IntArray::Resize(unsigned int required_index)
{
    unsigned int new_size = size;
    while(new_size <= required_index)
        new_size *= 2;
    int *new_array = new int[new_size];
    for(unsigned int i = 0; i < size; i++)
        new_array[i] = p[i];
    delete[] p;
    p = new_array;
    size = new_size;
}
```

Теперь мы можем в программе использовать, например, такие операторы:

```
    IntArray arr;
```

```
arr[500] = 15;
arr[1000] = 30;
arr[10] = arr[500] + 1;
arr[10]++;
```

Операция индексирования должна иметь ровно один параметр, но этот параметр, вообще говоря, может быть любого типа. Это позволяет создавать «массивы», использующие в качестве индекса текстовые строки или даже объекты других классов. Хранить элементы массива мы также можем любым удобным нам способом, например, в виде списка (в некоторых случаях это может быть оправдано). Можно организовать объект, выглядящий как массив, но хранящий свои элементы в файле на диске, так что операция индексирования реально будет представлять собой операцию чтения или записи в файл; встречаются и другие применения.

3.21. Переопределение операций ++ и --

Переопределение операций инкремента и декремента имеет свои особенности, обусловленные тем, что в Си и Си++ каждая из операций `++` и `--` допускает две формы: префиксную (`++i`) и постфиксную (`i++`). Строго говоря, эти две формы представляют собой *различные* операции. Если бы не это, можно было бы переопределять инкремент и декремент как обычные унарные операции, то есть вне классов в виде функции от одного аргумента, либо в классе/структуре в виде метода без параметров.

Чтобы отличать друг от друга переопределение префиксной и постфиксной формы, в Си++ принято соглашение, выглядящее довольно неожиданно. Поскольку префиксная форма более «традиционна» для унарных операций, обычным способом переопределяется именно она. Иначе говоря, если мы переопределяем соответствующие операции введением метода в классе или структуре, то метод с именем `operator++` или `operator--` без параметров определяет *префиксную* форму соответствующей операции (например, `++i`); аналогично, префиксную форму операции инкремента (декремента) задаёт и глобальная функция (не метод), имеющая имя `operator++` (`operator--`) и *один* параметр. Что касается *постфиксной* формы, то она переопределяется функцией с тем же именем `operator++` (`operator--`), но имеющей один дополнительный (фиктивный) параметр типа `int`. Наличие параметра благодаря перегрузке имён функций позволяет иметь две функции с одним и тем же именем; параметр, введённый исключительно ради этого различия, реально никогда не используется. Для переопределения постфиксной формы операции инкремента или декремента мы можем воспользоваться либо методом

класса/структуры с именем `operator++` (`operator--`) и *одним* параметром (фиктивным), либо глобальной функцией с таким же именем, имеющей *два* параметра (второй из них фиктивный). Например, пусть класс `A` описан следующим образом:

```
class A {
public:
    void operator++() { printf("first\n"); }
    void operator--() { printf("second\n"); }
    void operator++(int) { printf("third\n"); }
    void operator--(int) { printf("fourth\n"); }
};
```

Рассмотрим теперь фрагмент кода:

```
    A a;
    ++a;   // first
    a++;   // third
    --a;   // second
    a--;   // fourth
```

В результате выполнения этого фрагмента будут напечатаны (в столбик) слова `first`, `third`, `second` и `fourth` — именно в таком порядке.

В рассмотренном примере наши операции ничего не возвращали; между тем исходно префиксная и постфиксная формы этих операций различаются именно возвращаемым значением. При этом операция в префиксной форме возвращает *новое* значение переменной, к которой она применена, что позволяет выдержать семантику этой операции, вернув сам объект или (для оптимизации) константную ссылку на него. С операцией в постфиксной форме дела обстоят несколько хуже: согласно классической семантике она должна вернуть *старое* значение, в то время как объект уже имеет *новое* значение. Чтобы выдержать семантику в этой форме, приходится в теле функции `operator++(int)` создавать временную (локальную) копию объекта, а возвращать значение, сохранённое в этой копии, приходится обязательно *по значению*, ведь возвращать ссылку на локальную копию было бы ошибкой: копия исчезнет в момент завершения работы функции, то есть *до того, как вызывающая функция успеет воспользоваться возвращённым результатом*. Проиллюстрируем сказанное на примере операций инкремента для класса, инкапсулирующего целочисленную переменную и повторяющего её семантику:

```
class MyInt {
    int i;
public:
```

```
MyInt(int x) : i(x) {}
const MyInt& operator++() { i++; return *this; }
MyInt operator++(int)
    { MyInt tmp(*this); i++; return tmp; }
// ...
```

В этом примере объект класса занимает столько же памяти, сколько и обычная целочисленная переменная, так что двойное копирование объекта (при создании переменной `tmp` и при возврате значения) не приводит к существенным потерям. Однако для более сложных классов педантичное следование классической семантике для постфиксных операций `++` и `--` может обойтись неоправданно дорого.

3.22. Переопределение операции `->`

Операция `->` переопределяется, пожалуй, наиболее экзотическим способом. Напомним, что эта операция в языке Си означает выборку поля из структуры, на которую указывает заданный адрес. В Си++ она означает практически то же самое; естественно, с её помощью можно также обращаться и к методам, а работать она может как со структурами, так и с классами. Перегрузка этой операции обычно требуется нам, если мы хотим создать объект, ведущий себя подобно указателю, но при этом выполняющий некоторые дополнительные действия.

Отметим для начала один неочевидный факт. Операция `->`, несмотря на её внешний вид, является *унарной*, то есть имеет всего один аргумент (указатель). Действительно, пусть у нас описана структура

```
struct s1 {
    int a, b;
};
```

и имеется указатель на неё:

```
s1 *p = new s1;
```

Теперь обращение к полю `a` будет выглядеть так: `p->a`. Здесь как будто бы два операнда, `p` и `a`; но ведь `a` — это *имя поля*, которое никоим образом не является самостоятельным выражением, не имеет ни типа, ни значения! Иначе говоря, на месте `a` не может стоять ничего, кроме имени поля. Язык не содержит никаких выражений, *вычисляющихся* в значение, способное заменить имя поля. Единственным полноценным операндом в выражении `p->a` остаётся адрес, представленный указателем `p`: в отличие от `a`, на месте `p` может стоять выражение произвольной сложности, вычисляющее значение типа `s1*`

(адрес структуры `s1`). Итак, мы имеем дело с *унарной* операцией, полным именем которой можно считать `->a` (операция выборки поля `a`). Можно при желании считать, что символ `->` задаёт целое семейство операций — столько же, сколько в структуре есть полей (а для Си++ — ещё и методов). Впрочем, будучи семантически безупречным, на практике такое рассмотрение нам не понадобится; здесь мы приводим его только для иллюстрации сказанного.

Вернёмся к вопросу о переопределении операции `->`. Наряду с многими другими операциями, операция `->` переопределяется исключительно методами класса или структуры, то есть не может быть переопределена отдельной функцией. Чтобы избежать рутинной работы по переопределению отдельных операций для выборки каждого поля, в Си++ принято неочевидное, но вполне работающее соглашение: операция `->` определяется методом `operator->`, не имеющим параметров (как и обычные унарные операции), но при этом метод обязан возвращать либо *указатель на некую другую структуру или класс*, либо объект (или ссылку на объект), для которого, в свою очередь, операция `->` переопределена как метод. При использовании перегруженной операции `->` компилятор вставляет в код последовательно вызовы методов `operator->`, пока очередной метод не окажется возвращающим обычный указатель на структуру или класс; именно по этому указателю и производится в итоге выборка заданного поля.

Попробуем проиллюстрировать сказанное на примере. Пусть у нас есть структура `s1` и нам нужен класс, объекты которого будут вести себя как указатели на `s1`, но при исчезновении такого «указателя» (например, при завершении функции, в которой он описан как локальная переменная) объект, на который он указывает, будет уничтожаться. Начнём описание класса:

```
class Pointer_s1 {
    s1 *p;
public:
```

Будем предполагать, что хранящийся внутри объекта простой указатель на `s1` может иметь нулевое значение, что будем расценивать обычным образом как отсутствие объекта; естественно, в этом случае удалять ничего не будем. Опишем теперь конструктор и деструктор:

```
    Pointer_s1(s1 *ptr = 0) : p(ptr) {}
    ~Pointer_s1() { if(p) delete p; }
```

Для удобства работы введём операцию присваивания обычного адреса:

```
    s1* operator=(s1 *ptr) {
        if(p)
            delete p;
        p = ptr;
        return p;
    }
```

Заметим, что побитовое копирование и присваивание объектов класса `Pointer_s1` заведомо приведёт к ошибкам, так как один и тот же объект типа `s1` в этих случаях будет удаляться дважды. Во избежание этого запретим присваивание и копирование объектов класса `Pointer_s1`, убрав конструктор копирования и соответствующую операцию присваивания в приватную часть:

```
private:
    Pointer_s1(const Pointer_s1&) {}
        // copying prohibited
    void operator=(const Pointer_s1&) {}
        // assignments prohibited
```

Опишем теперь операции, которые превратят объекты нашего класса в подобие указателя на `s1`, а именно, операции разыменования (унарную `*`) и выборки поля (`->`), и завершим описание класса:

```
public:
    s1& operator*() const { return *p; }
    s1* operator->() const { return p; }
};
```

Такой класс удобно использовать, например, внутри функций. Так, если в начале некоторой функции описать объект класса `Pointer_s1`, то ему можно будет присваивать адреса новых экземпляров `s1`, при этом он будет каждый раз удалять старый экземпляр, а когда работа функции завершится, автоматически удалит последний из экземпляров `s1`:

```
int f()
{
    Pointer_s1 p;
    p = new s1;
    p->a = 25;
    p->b = p->a + 36;
    // ...
    // при завершении f память будет освобождена
}
```

3.23. Переопределение операции вызова функции

Операция вызова функции, синтаксически представляющая собой постфиксную операцию, символом которой служат круглые скобки (возможно, содержащие список параметров), может быть, как и другие операции, переопределена. Поскольку вызов функции обозначается круглыми скобками, имя функции, которая переопределяет эту операцию, будет состоять из слова `operator` и круглых скобок; как обычно, после имени функции записывается список формальных параметров. Например, операция вызова функции без параметров записывается так:

```
void operator()() { /* тело */ }
```

Подобно операциям присваивания и индексирования, вызов функции переопределяется только методом класса.

Приведём пример. Опишем класс `Fun`, для которого переопределены операции вызова функции без параметров, а также с одним и двумя целочисленными параметрами:

```
class Fun {
public:
    void operator()()
        { printf("fun0\n"); }
    void operator()(int a)
        { printf("fun1: %d\n", a); }
    void operator()(int a, int b)
        { printf("fun2: %d %d\n", a, b); }
};
```

Теперь мы можем написать следующий фрагмент кода:

```
Fun f;
f(); f(100); f(25, 36);
```

В результате выполнения этого фрагмента будет напечатано:

```
fun0
fun1: 100
fun2: 25 36
```

Классы, для которых определена операция вызова функции — иначе говоря, классы, объекты которых можно использовать подобно именам функций — часто называют ***функторами***.

3.24. Переопределение операции преобразования типа

Рассмотрим следующий фрагмент кода:

```
int i;
double d;
// ...
i = d;
```

В строчке, содержащей присваивание, задействована (неявно) ***операция преобразования типа выражения***, позволяющая в данном случае построить значение типа `int` на основе имеющегося значения типа `double`. Аналогичную возможность неявного преобразования можно предусмотреть и для классов, введённых программистом. Один из способов этого мы уже знаем — это конструктор преобразования (см. §3.10); но в некоторых случаях применить конструктор преобразования не удаётся. В частности, конструктором можно задать преобразование из базового типа в тип, описанный пользователем, но не наоборот. Кроме того, иногда бывает по каким-то причинам невозможно изменить описание некоторого класса, но в программе нужно преобразование в объекты этого класса. Бывают и другие (довольно экзотические) ситуации, когда мы не можем задать способ преобразования путём модификации того типа, *к которому* производится преобразование, но при этом у нас есть возможность изменить (дополнить) описание *того типа, значения которого* подлежат преобразованию. В таких случаях применяют переопределение операции преобразования типа.

Операция неявного преобразования типов определяется методом, имя которого состоит из слова `operator` и имени типа, к которому необходимо преобразовывать тип выражения. Тип возвращаемого значения для такой функции не указывается, так как он определяется именем. Например:

```
class A {
    //...
public:
    //...
    operator int() const { /* ... */ }
};
```

Наличие в классе A операции преобразования к `int` делает возможным, например, такой код:

```
A a;
```

```
int x;
// ...
x = a;
```

Естественно, операция преобразования может быть переопределена только как метод класса или структуры.

Учтите, что чрезмерное увлечение переопределениями операции преобразования приводит, как правило, к тому, что компилятор находит больше одного способа преобразования из одного типа к другому и в результате не применяет ни одного из них, выдавая ошибку. Поэтому на практике переопределение операции преобразования типа почти не используется; для получения значений нужного типа из объектов классов или структур обычно применяют специально для этого вводимые методы (обыкновенные, без слова `operator`), вызываемые явно. Пример крайне редкой ситуации, когда операция преобразования типа оказывается действительно нужна, рассмотрен в §3.25.

3.25. Пример: разреженный массив

Под *разреженным массивом* понимается массив относительно большого объёма (например, состоящий из нескольких миллионов элементов), большинство элементов которого равны одному и тому же значению, чаще всего нулю, и лишь некоторые элементы, количество которых очень мало в сравнении с размером массива, от этого значения отличаются. Естественно, в такой ситуации целесообразно хранить в памяти лишь те элементы массива, значения которых отличны от нуля (или другого «общего» значения).

Попробуем написать на Си++ такой класс, который ведёт себя как целочисленный массив потенциально бесконечного размера[17] в том смысле, что к объекту этого класса применима операция индексирования (квадратные скобки), однако объект реально хранит только те элементы, значения которых отличаются от нуля. Назовём этот класс `SparseArrayInt`. Для простоты картины будем считать, что количество ненулевых элементов массива настолько незначительно, что даже линейный поиск среди них может нас устроить по быстродействию. Это позволит хранить ненулевые элементы массива в виде списка пар «индекс/значение». Все детали реализации, включая и структуру, представляющую звено списка, скроем в приватной части класса, чтобы в будущем можно было безболезненно сменить реализацию на более сложную.

[17] На самом деле мы будем использовать в качестве индекса тип `unsigned long`; значения этого типа могут достигать $2^{32} - 1$, т. е. чуть больше четырёх миллиардов.

Основная проблема при реализации разреженного массива возникает в связи с поведением операции индексирования. Выражение «элемент массива» может встречаться как в правой, так и в левой части присваиваний, и в обоих случаях используется одна и та же функция-метод класса `SparseArrayInt` — `operator[]`. Если бы все элементы массива хранились в памяти, как это было в примере из §3.20, можно было бы просто возвратить ссылку на место в памяти, где хранится элемент, соответствующий заданному индексу. С разреженным массивом так действовать не получится, поскольку в большинстве случаев элемент в памяти не хранится. При этом просто возвратить нулевое значение (без всякой ссылки) нельзя, ведь тогда выражение, содержащее нашу операцию индексирования, будет нельзя применять в левой части присваивания.

Итак, что же делать в случае, если операция индексирования вызвана для индекса, для которого элемент в памяти не хранится, поскольку равен нулю? Одно из возможных решений — завести в памяти соответствующий элемент, пусть даже и с нулевым значением, и вернуть ссылку на него. Это, конечно, позволит выполнять присваивания, но приведёт к тому, что при каждом обращении к нулевым элементам (в том числе на чтение, а не на запись) такие элементы будут заводиться в памяти, и вскоре окажется, что изрядное количество памяти занято теми самыми нулевыми значениями, хранения которых требовалось избежать.

Схему можно несколько усовершенствовать, если каждый раз при вызове операции индексирования сохранять в приватной части объекта значение индекса или даже адрес заводимого звена списка. Тогда при следующем вызове можно будет проверить, хранит ли запись, заведённая на предыдущем вызове, число, отличное от нуля, и если нет, то удалить её, освободив память. У такого варианта реализации есть, однако, свой недостаток: на заведение и удаление элементов будет тратиться изрядное количество времени.

В идеале хотелось бы иметь возможность вызывать разные функции-методы для случаев, когда происходит обращение к элементу массива на чтение и когда индексирование используется для присваивания элементу нового значения. В первом случае можно было бы тогда возвращать просто значение элемента или ноль, если элемента в памяти нет; во втором случае можно было бы без особых проблем создавать новый элемент и возвращать ссылку на поле, хранящее значение; это, впрочем, не исключает необходимости проверки на следующем вызове, не был ли предыдущему элементу присвоен ноль (в том числе для случаев, когда элемент уже существовал). К сожалению, возможности введения разных методов индексирования для этих двух случаев в Си++ нет, как нет и возможности определить

из тела операции индексирования, вызвана она из левой части присваивания или нет. Это вполне можно считать недостатком Си++.

Впрочем, такой механизм можно *смоделировать*; именно так мы и поступим. Никто не мешает нам описать небольшой вспомогательный класс, объекты которого как раз и будет возвращать наша операция индексирования. Объект этого класса будет просто хранить всю имеющую отношение к делу информацию, а в нашем случае такой информации немного: адрес главного объекта (то есть самого разреженного массива), с которым нужно работать, и значение индекса, переданное в качестве параметра операции индексирования.

Имея эту информацию и доступ к объекту массива, наш вспомогательный объект в зависимости от ситуации может произвести любые действия, которые в этой ситуации могут потребоваться. Что же касается определения, какая из возможных ситуаций имеет место, то у вспомогательного объекта для этого есть существенно больше возможностей, чем было в теле операции индексирования: ведь все операции, применяемые к результату операции индексирования, как раз и будут на самом деле применены к этому объекту. Например, достаточно определить для нашего вспомогательного класса операции преобразования к типу `int` и присваивания, чтобы сделать возможным и корректным следующий код:

```
SparseArrayInt arr;
int x;
//...
x = arr[500];  // чтение
arr[300] = 50; // запись (возможно создание элемента)
arr[200] = 0;  // удаление элемента, если таковой есть
```

К сожалению, недостаток есть и у такой реализации. Проблема в том, что обычное присваивание — это далеко не единственная операция, способная изменить значение целочисленной переменной. Пользователь вполне может попытаться использовать, например, такие выражения:

```
arr[15] += 100;
x = arr[200]++;
y = --arr[200];
```

Чтобы все подобные операции работали, нам придётся для нашего вспомогательного объекта определить их все явным образом, что потребует изрядного объёма работы: языки Си и Си++ поддерживают десять операций присваивания, совмещённых с арифметическими действиями (`+=`, `-=`, `<<=` и т. п.), плюс к тому четыре операции инкремента/декремента (`++` и `--` в префиксной и постфиксной форме). В

нашем примере мы ограничимся описанием операции += и двух форм операции ++. Остальные подобные операции читатель без труда опишет самостоятельно по аналогии.

Прежде чем продемонстрировать заголовок класса SparseArrayInt, отметим, что для удобства реализации всех модифицирующих операций целесообразно в классе вспомогательного объекта ввести две внутренние функции. Функция Provide будет отыскивать в списке элемент, соответствующий заданному индексу, и возвращать ссылку на поле, содержащее значение; если соответствующего элемента в списке нет, функция будет его создавать. Вторая функция, Remove, будет удалять из списка элемент с заданным индексом, если таковой в списке присутствует. Вспомогательный класс назовём Interm от английского *intermediate*. С учётом этого заголовок класса будет выглядеть так:

```
class SparseArrayInt {
    struct Item {
        int index;
        int value;
        Item *next;
    };
    Item *first;
public:
    SparseArrayInt() : first(0) {}
    ~SparseArrayInt();
    class Interm {
        friend class SparseArrayInt;
        SparseArrayInt *master;
        int index;
        Interm(SparseArrayInt *a_master, int ind)
            : master(a_master), index(ind) {}
        int& Provide();
        void Remove();
    public:
        operator int();
        int operator=(int x);
        int operator+=(int x);
        int operator++();
        int operator++(int);
    };
    friend class Interm;

    Interm operator[](int idx)
        { return Interm(this, idx); }
private:
    SparseArrayInt(const SparseArrayInt&) {}
```

```
    void operator=(const SparseArrayInt&) {}
};
```

В самом классе `SparseArrayInt`, как можно заметить, оказывается не так много функций: бóльшая часть реализации оказывается вытеснена в методы вспомогательного класса. Единственным методом основного класса, тело которого в силу размеров не включено в заголовок класса, будет деструктор, который должен освободить память от элементов списка:

```
SparseArrayInt::~SparseArrayInt()
{
    while(first) {
        Item *tmp = first;
        first = first->next;
        delete tmp;
    }
}
```

Опишем теперь методы класса `SparseArray::Interm`, в которых и заключена основная функциональность нашего разреженного массива:

```
int SparseArrayInt::Interm::operator=(int x)
{
    if(x == 0)
        Remove();
    else
        Provide() = x;
    return x;
}
int SparseArrayInt::Interm::operator+=(int x)
{
    int& location = Provide();
    location += x;
    int res = location;
    if(res == 0)
        Remove();
    return res;
}
int SparseArrayInt::Interm::operator++()
{
    int& location = Provide();
    int res = ++location;
    if(location == 0)
        Remove();
    return res;
```

```
}
int SparseArrayInt::Interm::operator++(int)
{
    int& location = Provide();
    int res = location++;
    if(location == 0)
        Remove();
    return res;
}
SparseArrayInt::Interm::operator int()
{
    Item* tmp;
    for(tmp = master->first; tmp; tmp = tmp->next) {
        if(tmp->index == index) {
            return tmp->value;
        }
    }
    return 0;
}
```

Осталось описать вспомогательные функции `Provide` и `Remove`:

```
int& SparseArrayInt::Interm::Provide()
{
    Item* tmp;
    for(tmp = master->first; tmp; tmp = tmp->next) {
        if(tmp->index == index)
            return tmp->value;
    }
    tmp = new Item;
    tmp->index = index;
    tmp->next = master->first;
    master->first = tmp;
    return tmp->value;
}
void SparseArrayInt::Interm::Remove()
{
    Item** tmp;
    for(tmp = &(master->first); *tmp; tmp = &(*tmp)->next) {
        if((*tmp)->index == index) {
            Item *to_delete = *tmp;
            *tmp = (*tmp)->next;
            delete to_delete;
            return;
        }
    }
}
```

3.26. Статические поля и методы

Иногда бывает полезно иметь переменные и методы, которые, с одной стороны, доступны только из класса и/или воспринимаются как его часть, но, с другой стороны, не привязаны ни к какому из объектов и могут использоваться даже тогда, когда ни одного объекта данного класса не существует. В языке Си++ такие члены класса называются ***статическими*** и обозначаются ключевым словом `static`[18].

Статическое поле класса представляет собой, по сути, обычную переменную, время жизни которой совпадает с временем выполнения программы — точно так же, как у глобальных переменных. При этом правила видимости для статического поля класса точно такие же, как и для обычных полей: статическое поле входит в область видимости класса, так что при упоминании статического поля вне методов класса требуется использовать символ раскрытия области видимости «`::`»[19]; на статическое поле распространяются механизмы защиты, так что, если его расположить в приватной части класса, доступ к такому полю будет возможен только из методов класса и друзей класса. Если же статическое поле оставить в публичной части, мы получим просто глобальную переменную с «хитрым» именем.

Сущность статических полей класса можно пояснить и иначе: статическое поле — это такое «хитрое» поле, которое существует в одном экземпляре для всего класса, вне зависимости от количества объектов этого класса; даже если ни одного объекта нет, статическое поле всё равно существует, коль скоро оно описано в классе. Получается, что статическое поле, в отличие от обычного, не является составной частью *объекта* и принадлежит *классу* как единому целому.

Статическое поле описывается в классе точно так же, как и обычное поле класса, только с добавлением спереди слова `static`:

```
class A {
    //...
    static int the_static_field;
    //...
};
```

[18]Не путайте их с локальными переменными и функциями модулей, а также с локальными переменными функций, которые сохраняют своё значение при повторном входе в функцию; как и в языке Си, в Си++ эти сущности тоже обозначаются словом `static`, но практически ничего общего не имеют со статическими членами класса.

[19]Мы уже встречались с символом раскрытия области видимости в §3.13 на стр. 61.

Нужно учесть очень важное отличие статического поля от обычного: если описание обычного поля само по себе уже означает, что в объекте под это поле будет выделено место, то описание статического поля означает лишь, что *место под это поле будет выделено где-то в одном из модулей программы.*

Читатель, хорошо знакомый с языком Си, может заметить здесь прямую аналогию с объявлениями глобальных переменных с директивой `extern`. Чтобы понять, почему описание статического поля в классе считается именно объявлением, а не описанием, подумайте, что будет, если заголовок класса будет вынесен в заголовочный файл, а сам этот файл — включён из нескольких модулей.

Итак, чтобы завершить создание статического поля, нужно **вне класса** поместить его описание:

```
int A::the_static_field = 0;
```

Обычно определения статических полей снабжают инициализацией, как в нашем примере, хотя это и не обязательно. Обратите внимание, что, если ваше описание класса вынесено в заголовочный файл, то определения статических полей следует **обязательно поместить в файл реализации одного из модулей**, ни в коем случае не в заголовочник!

Повторим, что введённое таким образом статическое поле будет существовать в программе **в единственном экземпляре и в течение всего времени выполнения программы** вне всякой зависимости от того, будут ли в вашей программе заводиться объекты класса `A` и в каких количествах. Если поместить объявление статического поля в приватной части класса, то соответствующее имя будет доступно только в методах класса и в «дружественных» функциях. Объявление можно поместить и в открытую часть класса, тогда оно будет доступно отовсюду, а обратиться к нему можно будет как через существующий объект, так и без всякого объекта с помощью явного раскрытия области видимости:

```
A a;
a.the_static_field = 15;  // правильно
A::the_static_field = 15; // тоже правильно
```

Следует отметить, что статические поля схожи с глобальными переменными в том числе и тем, что в них программа может *накапливать глобальное состояние*, в результате чего для стороннего наблюдателя поведение одних и тех же функций окажется изменяющимся по неочевидным законам. Кроме того, статические поля могут существенно затруднить масштабирование программы. Так, если вам зачем-то понадобился список объектов определённого класса и

вы ввели для этой цели статический указатель на первый элемент такого списка, то это означает, что в программе такой список может быть только один, а если когда-нибудь понадобится два таких списка (из объектов одного и того же типа), программу придётся очень серьёзно переделать.

Поэтому **использование статических полей рекомендуется в одной и только одной ситуации: когда такое поле представляет собой константу, то есть его значение никогда не изменяется во время работы программы**. Например, статическим полем можно представить какой-нибудь крупный массив, содержащий *неизменные* данные, необходимые для работы объектов данного класса, но не требующиеся нигде за его пределами. Примером такого массива могут служить, например, таблица переходов между состояниями конечного автомата в классе, реализующем этот конечный автомат; массив строк с возможными диагностическими сообщениями; таблица соответствия пользовательских команд (строк) вызываемым функциям, и т. п.

Статический метод — это особый вид функции-метода, которая, являясь методом класса и имея доступ к его закрытым деталям реализации, при этом **вызывается независимо от объектов класса**. Первоначально статические методы предназначались для работы со статическими полями, но получившийся механизм нашёл в итоге существенно более широкий спектр применений. Описание статического метода аналогично описанию обычного метода, но перед таким описанием ставится ключевое слово `static`, как в следующем примере:

```
class Cls {
    //...
    static int TheStaticMethod(int a, int b);
    //...
};
```

Обращение к статическому методу, как и к статическому полю, возможно как через объект класса, так и без такового, с помощью символа раскрытия области видимости:

```
Cls::TheStaticMethod(5, 15); // всё правильно
Cls c;
c.TheStaticMethod(5, 15); // так тоже можно
```

Поскольку статическая функция-метод может быть вызвана без объекта, у неё, как следствие, отсутствует неявный параметр `this` (см. §2.2). Кроме всего прочего это означает, что такая функция не может просто так обращаться, как другие методы, к полям объекта,

ведь объекта у неё нет. Вызывать нестатические функции-методы статическая функция тоже просто так не может, поскольку нестатические методы должны вызываться для конкретного объекта. Ситуация кардинально меняется, если статическая функция тем или иным способом всё-таки получает доступ к объекту своего класса. Такое вполне может произойти и никоим образом не противоречит определению статического метода: действительно, никто не мешает передать объект через один из явно обозначенных параметров; вполне возможно получить доступ к объекту своего класса через глобальные переменные или с помощью глобальных функций; наконец, **статическая функция вполне может создать объект сама**.

Итак, несмотря на отсутствие указаний на конкретный объект при вызове статической функции, она в некоторых случаях может получить доступ к объекту своего класса. И тогда **статическая функция, как и любой метод класса, может обращаться к закрытым (приватным) полям и методам объекта** (действительно, как уже говорилось ранее, единицей защиты в Си++ является не объект, а класс или структура целиком). Наличие в Си++ статических методов позволяет в ряде случаев применять весьма изящные приёмы программирования. Например, мы вполне можем убрать в закрытую часть класса все имеющиеся конструкторы, запретив таким образом создание объектов данного класса извне его самого, и поручить создание объектов статическому методу, который можно вызвать, не имея ни одного объекта.

4. Обработка исключительных ситуаций

Прежде чем начать рассказ об обработке исключений в Си++, мы должны предостеречь читателя от возможных последствий некритического восприятия этого инструмента. Общая идея исключительных ситуаций, как вы увидите чуть позже, чрезвычайно полезна, но то, как этот инструмент представлен в языке Си++, вызывает определённое недоумение. Чтобы реализовать передачу управления в строгом соответствии со спецификацией Си++, компилятор вынужден идти на неочевидные и дорогостоящие трюки — вставлять в программу большое количество практически бесполезного машинного кода, использовать громоздкую «невидимую» библиотечную поддержку, снижать быстродействие; коль скоро мы упомянули быстродействие, отметим, что, если уж ваша программа перешла в состояние обработки исключительной ситуации, то происходить это будет по компьютерным меркам просто *безумно* медленно.

Многие программисты, даже не относящие себя к консерваторам, подобным автору этих строк, считают использование исключений недопустимым; большинство компиляторов Си++ позволяет полностью отключить этот механизм. Возможно, при минималистическом подходе, который мы приняли в нашей книге в отношении Си++, логичнее было бы вообще не рассказывать об исключениях, и такой вариант, смеем вас заверить, рассматривался весьма серьёзно, но, несмотря на всё сказанное, эта глава в книге сохранилась, и этому есть целый ряд причин.

Прежде всего следует отметить, что использование исключений чрезвычайно сильно снижает трудоёмкость написания программ, во всяком случае, если сравнивать с программами, в которых, несмотря на отсутствие механизма исключений, все возникающие ошибочные ситуации скрупулёзно проверяются и обрабатываются. Можно ли ради такого выигрыша в трудоёмкости пожертвовать эффективностью — вопрос, на который следует отвечать, исходя из особенностей конкретной решаемой задачи.

Но есть здесь и другой момент, пожалуй, даже более важный. Недостатки, присущие механизму исключений языка Си++, обусловлены не только и не столько самой идеей исключений, сколько довольно странным подходом, который избрал для воплощения этой идеи Бьёрн Страуструп. Обработка исключений как таковая существует не только в Си++, и далеко не всегда она столь же убийственна для эффективности полученной программы. Можно легко представить себе другой подход, который ничуть не снизил бы выразительной мощи исключений, но при этом такой, который не заставлял бы программистов ломать копья по поводу допустимости его применения.

Для читателей, которые знают, о чём идёт речь, поясним: если бы конкретная исключительная ситуация идентифицировалась в Си++ не *типом объекта*, как это сделано сейчас, а *адресом переменной* произвольного типа, глобальной, динамической или даже стековой, для которой программист обязан гарантировать, что она не перестанет существовать ни к моменту возбуждения исключения, ни к моменту его обработки, то обработка исключений не тянула бы за собой динамического монстра, именуемого *системой идентификации типов во время исполнения* (RTTI), а пометка и размотка стека могли бы работать проще и быстрее.

С учётом всего этого можно сказать, что знакомство с обработкой исключений будет читателю полезно, даже если он примет для себя решение никогда не использовать её в программах на Си++. Существуют другие языки, предусматривающие аналогичные средства. Кроме того, в будущем возможно (и, пожалуй, неизбежно) появление какого-то нового языка программирования, и, возможно, в нём всё-таки учтут негативный опыт Си++.

Сразу хотелось бы отметить ещё один момент. Если вы всё же решите использовать исключения в своих программах, то, во всяком случае, **никогда не применяйте механизм обработки исключительных ситуаций для *штатного* выхода из вложенных управляющих конструкций или глубоких цепочек вызовов функций**. Когда в вашей программе возникла та или иная ошибка, вам уже обычно всё равно, сколько времени уйдёт на её обработку: в большинстве случаев выход из положения требует вмешательства пользователя (человека), а его «перетормозить» надо ещё исхитриться. Совсем другое дело, если никакой ошибки нет. Исключения работают медленно до такой степени, что программа, использующая их для штатного возврата управления из вложенного вычислительного контекста, может (легко!) проводить в обработке исключений процентов этак 95 всего своего времени работы: коллеги рассказывали автору про реальные случаи, когда быстродействие программы повышалось более чем на порядок (конкретно — в семнадцать раз) из-за замены выхода по исключению на простой `return` и несколько глобальных флажков.

Как говорится, кто предупреждён — тот вооружён.

4.1. Ошибочные ситуации и проблемы их обработки

Любая сколько-нибудь нетривиальная программа содержит фрагменты, которые в некоторых обстоятельствах не могут отработать корректно. Например, программа, анализирующая содержимое заданного файла, не сможет работать, не сумев открыть файл на чтение; программа, работающая по компьютерной сети, не сможет работать, если не работает сеть или если недоступен нужный сервер; функция, производящая сложные вычисления, не может корректно их завершить, если в ходе вычислений потребовалось деление на ноль или, например, вычисление логарифма по единичному основанию, и т. п. Подобные ситуации называют ошибочными, однако же это совершенно не обязательно означает, что ошибся программист, писавший программу. В случае с файлом «виноват» в возникновении ошибочной ситуации, скорее всего, пользователь, в случае с сетью — обслуживающий персонал сети или сервера; деление на ноль может быть следствием ошибки программиста, но может стать также и результатом некорректно сформированных исходных данных.

Иначе говоря, мы при написании программ часто сталкиваемся со случаями, когда успешная работа нашей программы зависит от внешних условий, которые мы сами гарантированно обеспечить не

можем. В таких случаях приходится предусматривать в программе *обработку ошибок*.

Проверить все нужные условия обычно несложно. Существенно сложнее может оказаться следующий вопрос: а что же делать, если условия оказались неудовлетворительны, — не открылся файл, не установилось соединение, в делителе оказался ноль, — то есть возникла та самая ошибочная ситуация? Многие студенты в такой ситуации поступают просто, дёшево и сердито: печатают какое-нибудь сообщение (очень часто просто `"ERROR"`) и завершают выполнение программы, например, вызовом `exit`. В реальной жизни такой вариант, как правило, недопустим. Чтобы понять причины этой недопустимости, представьте себе, что вы долго набирали текст в каком-нибудь текстовом редакторе, потом при сохранении *случайно* ввели неправильное имя директории или, например, попытались осуществить запись на защищённый носитель. Если бы автор текстового редактора обрабатывал ошибки «по-студенчески», программа редактора бы немедленно завершилась, уничтожив все результаты вашей работы. Маловероятно, что такое могло бы вам понравиться.

Более того, не всегда и не везде можно так вот просто «напечатать сообщение». Например, при программировании оконного приложения под MS Windows никакого потока стандартного вывода в распоряжении программиста нет, так что вместо печати приходится создавать модальный диалог с соответствующим текстом и кнопками. Под ОС Unix всё не так плохо, поток стандартного вывода есть всегда, есть даже специальный поток для вывода сообщений об ошибках; проблема только в том, что далеко не во всех случаях результаты вывода в эти потоки кто-то читает, и в некоторых случаях следует вместо них пользоваться системой журнализации. Если же написанный нами код работает в ядре операционной системы или это прошивка для какого-нибудь микроконтроллера, то возможность «выдать сообщение об ошибке» может попросту отсутствовать как явление.

Кроме того, не все ошибочные ситуации фатальны. В вышеупомянутом примере с редактором текстов в ошибке, допущенной пользователем, нет вовсе ничего страшного: нужно просто констатировать факт ошибки и попросить пользователя ввести другое имя файла. Точно так же может не быть фатальной ошибка, связанная с установлением сетевого соединения, ведь во многих случаях можно обратиться к серверу-дублёру[20]. Даже ситуация с делением на ноль в некоторых случаях может оказаться нефатальной, как, например, при исследовании некоторых некорректно поставленных задач: обыч-

[20]Такое возможно при обращениях к системе доменных имён (DNS), а в некоторых случаях — и при отправке электронной почты, и в других ситуациях тоже.

но в этих случаях головная программа «знает», как можно скорректировать исходные данные, чтобы избежать деления на ноль, и нужную комбинацию ищет методом проб и ошибок. Наконец, один и тот же код может использоваться в совершенно разных программах, причём для одних ошибка может оказаться фатальной, для других — не стоящей даже упоминания. Например, многие программы выполняют поиск нужного файла в разных директориях, просто пытаясь его открыть и продолжая поиск дальше после каждой неудачи; в этом случае ошибка вообще перестаёт быть ошибкой, тогда как в какой-нибудь другой программе ошибка при открытии файла делает дальнейшее выполнение бессмысленным.

Из всего вышесказанного можно сделать один очень серьёзный и важный вывод: **принимать решение о том, что делать при возникновении ошибочной ситуации — это прерогатива головной программы**. Завершать программу, а также и выполнять тем или иным способом выдачу сообщений может только функция `main` и те из вызываемых ею функций, которые для этого специально предназначены. Что же касается кода, не попавшего в эту категорию, то его дело в случае возникновения ошибки — оповестить об этом головную программу, и не более того.

Звучит это достаточно просто, однако все, кто имеет опыт написания даже не очень сложных программ, знают, с каким трудом это воплощается на практике. В простых языках программирования, таких как Си или Паскаль, приходится из функций в случае возникновения ошибок возвращать специальные значения, а в точке вызова функции, соответственно, проверять, не вернула ли вызванная функция значение, соответствующее ошибке. В большинстве случаев при этом вызвавшая функция, в свою очередь, вынуждена возвращать признак ошибки и т. д. Представьте себе теперь, что функция `f1` вызвала функцию `f2`, та, в свою очередь, функцию `f3` и т. д., а в функции, скажем, `f10` возможно возникновение ошибочной ситуации. Тогда нужно предусмотреть значение, возвращаемое в случае этой ошибки, в функции `f9` сделать проверку и в случае ошибки, в свою очередь, возвратить нечто ошибочное, и так во всех функциях.

Теоретически именно так всё и должно делаться, все возможные ошибочные ситуации должны проверяться и т. д., но на практике аккуратное соблюдение требований по обработке ошибок может оказаться настолько трудоёмким, что программисты прибегают к известному «алгоритму решения всех проблем», а именно — (1) поднять вверх правую руку и (2) резко махнуть ею, одновременно произнося что-то вроде «а, ладно». Есть даже шуточная фраза на эту тему: «никогда не проверяйте на ошибки, которые вы не знаете, как обработать». Разумеется, такое обычно даром не проходит. Ошибка,

которую не стали обрабатывать в надежде, что она никогда не произойдёт, проявится в самый неподходящий момент, причём чем дальше находится заказчик и чем больше денег он заплатил, тем выше вероятность, что наша программа сломается именно у него. Именно поэтому во многих языках предусмотрены специальные возможности для обработки ошибочных ситуаций. В языке Си++ соответствующий механизм называется ***обработкой исключений*** (англ. ***exception handling***).

4.2. Общая идея механизма исключений

Отметим ещё один недостаток использования специальных возвращаемых значений для индикации ошибки. В некоторых случаях среди всех значений возвращаемого функцией типа просто нет ни одного неиспользуемого; отличный пример этого — функция `atoi`, переводящая строковое представление целого числа в значение типа `int`. Возвращает она, естественно, число типа `int`, а поскольку любое такое число имеет строковое представление, то и вернуть (при отсутствии каких-либо ошибок) она может, вообще говоря, любое значение типа `int`. В то же время возможно, что в строке, поданной `atoi` на вход, содержится текст, не являющийся представлением какого-либо числа (например, состоящий из букв, а не из цифр). Разработчики функции решили в такой ситуации за неимением лучшего возвращать ноль, но это не решает проблему, ведь получается, что функция возвращает одно и то же и для строки, содержащей белиберду, и для строки `"0"`, которая вполне корректна. Это приводит нас к идее *исключения*, которое представляет собой *особый способ завершения функции*: если в языках, не поддерживающих механизм исключений, функция могла только вернуть одно из возможных значений, то в языках, поддерживающих исключения, функция может либо вернуть значение, либо *возбудить исключение*.

Отметим заодно, что исключения вовсе не являются составной частью парадигмы объектно-ориентированного программирования, как это почему-то часто утверждается. Напротив, они скорее относятся к *функциональному программированию*; небезызвестная функция `call/cc`, поддерживаемая во многих функциональных языках, представляет собой *обобщение* механизма исключений; в языке Common Lisp, не поддерживающем `call/cc`, при этом представлены механизмы, по смыслу аналогичные средствам обработки исключений Си++.

При возбуждении исключения программа переходит в особое состояние, в котором все активные на настоящий момент функции досрочно завершаются одна за другой, пока не найдётся такая, которая может справиться с данным типом исключительных ситуаций. Чтобы понять, о чём идёт речь, вспомним ситуацию, рассматривав-

шуюся в предыдущем параграфе: функция `f1` вызвала функцию `f2`, та, в свою очередь, функцию `f3` и т. д. вплоть до `f10`, где и возникает ошибка. При этом нам совершенно не обязательно делать какие-либо проверки в функциях `f9`, `f8` и т. д., вполне достаточно пометить, например, функцию `f1` как умеющую справляться с данным типом ошибок. Тогда при возникновении ошибки во время исполнения `f10` будет завершена досрочно и она сама, и вызвавшая её функция `f9`, и `f8`, и так вплоть до `f1`, которая и обработает ошибку. Иначе говоря, если некая функция `g` вызывает другую функцию `h`, а эта последняя возбуждает исключение, для которого в `g` нет обработчика, то это эквивалентно тому, как если бы функция `g` сама возбудила то же самое исключение.

Иногда всю обработку ошибок делают в функции `main`. Она для этого удобна, поскольку остаётся активной всё время исполнения программы[21] и, как следствие, может обработать исключение, возбуждённое практически в любой части программы.

4.3. Возбуждение исключений

При возникновении ситуации, трактуемой как ошибочная и требующая обработки в качестве исключительной, следует *возбудить исключение* с помощью оператора `throw` (англ. *бросить*). У этого оператора имеется один параметр, в качестве которого может выступать выражение почти что произвольного[22] типа, в том числе как любого базового (`int`, `float`, ...), так и производного (указатель и т. п.), а равно и определённого пользователем, включая, что немаловажно, классы и структуры. Тип выражения в операторе `throw` будем называть ***типом исключения***, а значение выражения — ***значением исключения***.

Для примера рассмотрим функцию, которая принимает на вход имя (текстового) файла, а возвращает количество строк (т. е. символов перевода строки) в этом файле. Ситуация, когда файл не удалось открыть, для такой функции оказывается заведомо исключительной; в этом случае в языке Си нам пришлось бы возвращать специальное значение (например, `-1`), а при вызове проверять, не его ли вернула функция. В Си++ это можно сделать проще:

```
unsigned int line_count_in_file(const char *file_name)
```

[21] Строго говоря, это не совсем так, поскольку есть ещё, например, конструкторы и деструкторы глобальных объектов, которые отрабатывают, соответственно, до и после функции `main`, но в некотором приближении можно этим пренебречь.

[22] Строго говоря, это не совсем так: в роли объекта исключения не могут выступать типы, для которых недоступны конструктор копирования и/или деструктор; почему это так, станет понятно позднее.

```
{
    FILE *f = fopen(file_name, "r");
    if(!f)
        throw "couldn't open the file";
    int n = 0;
    int c = 0;
    while((c = fgetc(f)) != EOF)
        if(c == '\n')
            n++;
    fclose(f);
    return n;
}
```

В этом примере мы в качестве исключения «бросили» строку (точнее, адрес её начала, т. е. значение типа `const char*`). Мы могли бы «бросить» выражение другого типа; например, следующий оператор «бросает» исключение типа `int`:

```
throw 27;
```

Чаще, однако, в роли исключений выступают объекты специально введённых для этой цели классов; разговор об этом впереди.

4.4. Обработка исключений

Обработку исключительных ситуаций следует предусматривать, естественно, только в тех местах, в которых мы можем с соответствующей ситуацией справиться. Так, если дальнейшая работа после возникновения исключительной ситуации требует обращения к пользователю, то обработку такой ситуации следует вставить в одну из функций, наделённых правом вести диалог с пользователем. Как уже говорилось, часто обработчик помещают в функцию `main`, но, конечно, не всегда.

Общий принцип организации обработки исключений таков. Из программы выделяется некоторый блок кода, исполнение которого *предположительно* может привести к возникновению исключительных ситуаций. Этот блок заключается в специальное синтаксическое обрамление и снабжается одной или несколькими инструкциями о том, как следует действовать при возникновении исключения того или иного типа. При этом такие инструкции действуют в отношении исключений соответствующих типов, **возникших в программе с момента входа в помеченный блок и до момента выхода из него**, в том числе в функциях, вызванных из блока прямо или косвенно. Глубина вызовов в данном случае роли не играет.

Для пометки блока с целью обработки исключительных ситуаций в Си++ используется ключевое слово `try` (англ. *попытаться*), за которым следует собственно блок, то есть последовательность операторов, заключённая в фигурные скобки. Сразу после `try`-блока помещают один или больше ***обработчиков исключений***. Обработчик исключений синтаксически несколько напоминает функцию с одним параметром, хотя это только внешнее сходство; на самом деле обработчик начинается с ключевого слова `catch` (англ. *поймать*), сразу после него ставятся круглые скобки, внутри которых — описание формального параметра (точно так, как это делается при описании функции с одним параметром). Тип параметра задаёт тип обрабатываемого исключения, а по имени параметра можно получить доступ к значению исключения.

Для примера припомним функцию из предыдущего параграфа, подсчитывающую количество строк в файле, и на основе этой функции напишем программу целиком. Будем считать, что функция `line_count_in_file` находится в отдельном модуле, так что в программе нам потребуется только её прототип.

```
#include <stdio.h>

unsigned int line_count_in_file(const char *file_name);

int main(int argc, char **argv)
{
    if(argc<2) {
        fprintf(stderr, "No file name\n");
        return 1;
    }
    try {
        int res = line_count_in_file(argv[1]);
        printf("The file %s contains %d lines\n",
               argv[1], res);
    }
    catch(const char *exception) {
        fprintf(stderr, "Exception (string): %s\n",
                exception);
        return 1;
    }
    return 0;
}
```

Эта программа, если всё будет в порядке, напечатает имя файла и количество строк в нём; если же открыть файл не удастся, функция `line_count_in_file` переведёт программу в состояние исключительной ситуации, а соответствующий обработчик будет найден в

функции `main`. Ни остаток функции `line_count_in_file`, ни остаток `try`-блока в этом случае выполняться не будут, управление окажется сразу передано в обработчик (`catch`-блок).

В этом примере исключение может возникнуть в функции, непосредственно вызванной из `try`-блока, но на самом деле это не важно: с таким же успехом оно могло возникнуть в функции, вызванной из `line_count_in_file`, и т. д. на любую глубину.

Обратите внимание, что мы в данном случае «ловим» исключение `const char*`, а не просто `char*`, поскольку именно такой тип (адрес константы типа `char`) имеет в Си и Си++ строковый литерал — строка, заключённая в двойные кавычки. Если убрать модификатор `const`, исключение поймано не будет.

В рассмотренном примере мы предусмотрели всего один обработчик исключения, хотя язык Си++ допускает произвольное их количество (не менее одного). Если бы из нашего `try`-блока вызывалось больше функций и потенциально они могли бы привести к исключительным ситуациям других типов, мы могли бы написать что-то вроде следующего:

```
try {
    // ...
}
catch(const char *x) {
    // ...
}
catch(int x) {
    // ...
}
```

Здесь важно понимать несколько простых правил. Во-первых, **обработчики (`catch`-блоки) рассматриваются по порядку, один за другим, причём сработать может только один из них или ни одного**. Это значит, в частности, что писать два обработчика одного типа или типов, сводимых один к другому[23], нет никакого смысла.

Во-вторых, **обработчик может поймать только исключение, возникшее во время работы соответствующего `try`-блока**. В частности, обработчик не может поймать исключение, которое выбросил один из предшествующих обработчиков, относящихся к тому же `try`-блоку.

Наконец, **если исключение не поймано ни одним из обработчиков, оно «выбрасывается» дальше, как если бы фрагмент кода, в котором исключение возникло, вовсе не был об-**

[23] Правила преобразования типов при обработке исключений несколько отличаются от правил, применяемых при вызове функций; мы рассмотрим эти правила в одном из следующих параграфов.

рамлён никаким `try`-блоком. На пути одного исключения может оказаться несколько `try`-блоков, как бы вложенных друг в друга (во всяком случае, в смысле временны́х периодов исполнения), и в некоторых из них может не оказаться подходящего обработчика. В таком случае досрочное завершение активных функций и уничтожение стековых фреймов продолжается дальше, пока не будет найден `try`-блок с подходящим обработчиком.

В обработчиках исключений можно использовать специальную форму оператора `throw`, а именно — написать этот оператор без параметров. Это означает «бросить то исключение, которое только что было поймано». Так делают в случаях, когда полностью справиться с возникшей ситуацией в данном месте программы невозможно, но прежде чем бросить исключение дальше, нужно произвести ещё какие-то действия. Примеры таких ситуаций рассматриваются в следующем параграфе.

4.5. Обработчики с многоточием

В некоторых случаях оказывается осмысленным обработчик, способный поймать *произвольное исключение независимо от его типа*. Такой обработчик записывается так же, как и обычный, только вместо типа исключения и имени параметра в скобках указывается многоточие. Конечно, значение исключения в этом случае использовано быть не может, но в некоторых случаях это и не важно. Например, если мы подключаем к нашей программе модули, написанные другими программистами, и не уверены в том, что нам известны все типы исключений, генерируемые такими модулями, может быть неплохой идея расположить в функции `main` примерно такой `try`-блок:

```
int main()
{
    try {
        // здесь пишем весь код главной функции
        return 0;
    }
    catch(const char *x) {
        fprintf(stderr, "Exception (string): %s\n", x);
    }
    catch(int x) {
        fprintf(stderr, "Exception (int): %d\n", x);
    }
    catch(...) {
        fprintf(stderr, "Something strange caught\n");
    }
    return 1;
```

```
}
```

Обработчики с многоточием делают особенно актуальной форму `throw` без параметров; её использование позволяет поймать исключение произвольного типа, выполнить какие-то действия, после чего бросить исключение дальше. Это может оказаться полезным, если в нашем коде локально захватываются те или иные ресурсы (выделяется память, открываются файлы и т. п.) и их следует освободить, прежде чем функция окажется завершена. Рассмотрим для примера функцию, в которой выделяется динамический массив целых чисел:

```
void f(int n)
{
    int *p = new int[n];
    // весь остальной код функции
    delete[] p;
}
```

Если во время выполнения кода, находящегося между `new` и `delete`, возникнет исключение, то `delete` не будет выполнен и массив, на который указывал `p`, будет продолжать занимать память, то есть станет мусором. Проблема снимается, если весь этот код заключить в `try`-блок, после которого есть обработчик для исключений произвольного типа, который, прежде чем бросить исключение дальше, освобождает память:

```
void f(int n)
{
    int *p = new int[n];
    try {
        // весь остальной код функции
    }
    catch(...) {
        delete[] p;
        throw;
    }
    delete[] p;
}
```

4.6. Объект класса в роли исключения

Обычно при обнаружении исключительной ситуации желательно передать обработчику максимум информации о возникших трудностях. Встроенные типы данных плохо пригодны для этого. Действительно, в рассмотренном выше примере было бы неплохо передать обработчику не только текстовое сообщение «couldn't open the file»,

но и имя файла, и значение переменной `errno` сразу после выполнения `fopen`, которое может помочь в анализе проблемы. Мы, однако, ограничились в примере одной строковой константой, чтобы не формировать строку, содержащую всю перечисленную информацию — ведь это потребовало бы значительного увеличения кода и создало бы определённые трудности с освобождением памяти от такой строки, поскольку саму строку пришлось бы создавать динамически.

Кроме того, есть и ещё одна проблема. Программа может использовать несколько библиотек, причём изготовленных разными производителями; также программа может быть разделена на несколько подсистем, создаваемых более или менее независимо. Весьма желательно иметь некий универсальный способ разделения исключений по признаку их возникновения в той или иной подсистеме программы или библиотеке. Встроенные типы для этого не подходят совсем, поскольку идея использования тех же текстовых строк в качестве исключений может прийти в голову одновременно авторам сразу всех используемых нами библиотек и половины наших подсистем.

Именно поэтому чаще всего в качестве типа исключения выступает специально для этой цели описанный класс, а значением исключения — объект такого класса. С одной стороны, в объект класса можно поместить всю информацию, какая нам только может показаться полезной для обработчика; действительно, никто ведь не мешает описать в классе столько полей, сколько нам захочется. С другой стороны, каждая подсистема и каждая библиотека в этой ситуации, скорее всего, введёт свои собственные классы для представления исключений. В головной программе мы сможем обрабатывать такие классы отдельными обработчиками, что позволит разделить обработку исключений, сгенерированных в разных подсистемах программы.

Отметим один очень важный момент. Как правило, выбрасываемый в качестве исключения объект создаётся локально. Локальные объекты, естественно, исчезают вместе со стековым фреймом создавшей их функции; как следствие, тот объект, который «поймается» в обработчике исключения, заведомо не может быть тем же экземпляром объекта, который фигурировал в операторе `throw`, и является, как нетрудно догадаться, его копией. Поэтому **практически всегда в классе, используемом для исключений, обязан присутствовать конструктор копирования**.

Для примера опишем класс, объект которого было бы удобно использовать в качестве исключения в нашей функции `line_count_in_file`. Объект класса будет хранить имя файла, текстовый комментарий (этот комментарий поможет понять, в каком месте программы произошла ошибка) и будет запоминать значение глобальной переменной `errno` на момент создания объекта. Для ко-

пирования строк мы опишем в приватной части класса вспомогательную функцию `strdup`[24]; поскольку доступ к объекту ей не нужен, опишем её с квалификатором `static`.

```
class FileException {
    int err_code;
    char *filename;
    char *comment;
public:
    FileException(const char *fn, const char *cmt);
    FileException(const FileException& other);
    ~FileException();
    const char *GetName() const { return filename; }
    const char *GetComment() const { return comment; }
    int GetErrno() const { return err_code; }
private:
    static char *strdup(const char *str);
};
```

Описания конструкторов, деструктора и функции `strdup` вынесем за пределы заголовка класса:

```
FileException::FileException(const char *fn, const char *cmt)
{
    err_code = errno;
    filename = strdup(fn);
    comment = strdup(cmt);
}
FileException::FileException(const FileException& other)
{
    err_code = other.err_code;
    filename = strdup(other.filename);
    comment = strdup(other.comment);
}
FileException::~FileException()
{
    delete[] filename;
    delete[] comment;
}
char* FileException::strdup(const char *str)
{
    char *res = new char[strlen(str)+1];
    strcpy(res, str);
    return res;
}
```

[24]Эта функция будет полностью аналогична одноимённой функции из стандартной библиотеки Си, но будет использовать для выделения памяти `new[]`, а не `malloc`.

Теперь мы можем изменить начало функции `line_count_in_file` (см. стр. 98) на следующее:

```
unsigned int line_count_in_file(const char *file_name)
{
    FILE *f = fopen(file_name, "r");
    if(!f)
        throw FileException(file_name,
            "couldn't open for line counting");
  // ...
```

Обработчик (`catch`-блок) для такого исключения может быть, например, таким:

```
    catch(const FileException &ex) {
        fprintf(stderr, "File exception: %s (%s): %s\n",
                ex.GetName(), ex.GetComment(),
                strerror(ex.GetErrno()));
        return 1;
    }
```

4.7. Автоматическая очистка

В примере, рассмотренном на стр. 103, мы были вынуждены перехватывать и потом заново «выбрасывать» исключения произвольного вида, чтобы обеспечить корректное удаление локально выделенной динамической памяти. К счастью, так приходится действовать не всегда. Работа с исключениями изрядно облегчается тем, что **компилятор гарантирует вызов деструкторов для всех локальных объектов, у которых деструкторы есть, прежде чем функция завершится — по исключению ли или обычным путём**.

Пусть, например, имеется класс `A`, снабжённый деструктором (непустым, иначе не интересно), и описана функция

```
void f()
{
    A a;
    //..
    g();
    //..
}
```

В момент завершения работы функции `f` для объекта `a` будет вызван деструктор, причём произойдёт это вне зависимости от того, как именно `f` завершится — в том числе и если функция `g` «выбросит» исключение и именно оно станет причиной завершения `f`. Это свойство Си++ называется ***автоматической очисткой***.

4.8. Преобразования типов исключений

Тип исключения, указанный в обработчике (`catch`-блоке), не всегда точно совпадает с типом выражения, использованного в операторе `throw`, однако правила преобразования типов в данном случае отличаются от правил, действующих при вызове функций. Одно из основных отличий тут в том, что целочисленные типы при обработке исключений не преобразуются друг к другу, то есть, например, обработчик с указанным типом исключения `int` не поймает исключение типа `char` или `long`.

Допустимые преобразования проще будет явно перечислить. Во-первых, любой тип может быть преобразован к ссылке на этот тип, то есть если оператор `throw` использует выражение типа `A`, то такое исключение может быть обработано `catch`-блоком вида `catch(A &ref)`. Во-вторых, неконстантные указатели и ссылки могут быть преобразованы к константным, то есть, например, если мы бросим исключение типа `char*`, то оно может быть поймано не только как `char*`, но и как `const char*`. В обратную сторону преобразование запрещено, так что обработчик типа `char *` (без модификатора `const`) не может поймать выражение типа `const char*`, и, в частности, не может обработать исключение, «выброшенное» с использованием строкового литерала, например такое:

```
throw "I can't work";
```

Последний (и, возможно, самый важный) вид преобразований имеет отношение к наследованию и полиморфизму. К обсуждению этой темы мы вернёмся после рассказа о наследовании.

5. Наследование и полиморфизм

Большая часть возможностей Си++, рассмотренных нами выше, не имеют, как это неоднократно подчёркивалось, прямого отношения к объектно-ориентированному программированию — кроме разве что методов, вызов которых рассматривается как «техническая реализация» передачи сообщения объекту. Впрочем, если для поддержки объектов ограничиться только методами и инкапсуляцией, получится не вполне полноценный вариант ООП, который часто даже называют иначе — *object-based programming* (а не *object-oriented*).

Настала пора познакомиться с фирменной особенностью ООП — ***наследованием***, а заодно и с теми вариантами полиморфизма, которые имеются в виду при перечислении «трёх китов ООП» (в отличие, опять же, от статического полиморфизма, с которым мы неоднократно сталкивались и который никакого отношения к ООП не имеет).

5.1. Иерархические предметные области

Очень часто, и особенно в более-менее крупных компьютерных программах, объекты предметной области оказывается возможно естественным образом объединить в некие категории, причём объекты каждой категории могут подразделяться на подкатегории и т. д., образуя таким образом ***иерархию типов объектов***. Например, в программе, моделирующей дорожное движение, могут рассматриваться объекты категории «участник дорожного движения», причём эта категория подразделяется на подкатегории «пешеходы» и «транспортные средства»; транспортные средства в свою очередь делятся на «велосипеды», «гужевые повозки» и «механические транспортные средства», эти последние делятся на трамваи, толлейбусы и автомобили, автомобили — на грузовые и легковые, и т. д.

В такой иерархии каждая категория объектов, в том числе и такая, которая сама подразделяется на категории, обладает некоторыми свойствами, специфичными для данной категории, причём этими же свойствами обладают и все объекты из её подкатегорий. Так, все автомобили имеют, по-видимому, характеристики «объём двигателя», «тип топлива», «объём топливного бака» и «количество топлива в баке», у них может заканчиваться топливо, над ними должна быть определена операция «заправка топлива». Все эти характеристики не имеют никакого смысла для объектов, не входящих в категорию «автомобили» — ни для трамваев, ни для велосипедов, ни тем более для пешеходов. С другой стороны, у грузовых автомобилей есть характеристика «объём кузова» и операции «погрузка» и «разгрузка», которых нет у автомобилей, не являющихся грузовыми. Наконец, такие характеристики, как текущая скорость и направление движения, очевидно, присущи каждому объекту рассматриваемой иерархии, вплоть до пешеходов.

Как видим, у нас возникают структуры данных, имеющие, с одной стороны, различный набор полей и обрабатывающих функций, но, с другой стороны, имеющие и некоторые общие поля, и определённые общие операции. Существуют весьма различные подходы к решению возникшего противоречия. При работе на языке Си чаще всего выходят из положения описанием структур данных, у которых первые несколько полей совпадают; при этом приходится то и дело изменять типы указателей, а это, как мы знаем, ведёт к труднообнаружимым ошибкам. Объектно-ориентированные языки программирования предлагают более корректный подход, называемый обычно ***наследованием***.

Объяснить наследование как технический приём не так просто; мы попытаемся применить не вполне традиционный подход, начав

с рассмотрения объекта как простой структуры данных (каковой он на самом деле и является, просто мы об этом не всегда помним). Методы мы подключим к делу потом.

5.2. Наследование структур и полиморфизм адресов

В терминах структур данных, расположенных в памяти, наследование представляет собой добавление новых полей данных к ранее описанной структуре. Такое добавление представляет собой *уточнение* сведений об описываемом объекте, и, таким образом, **наследование рассматривается как переход от общего к частному**. Пусть, например, у нас имеется структура данных, описывающая персону (человека) и содержащая поля для имени, пола и года рождения:

```
struct person {
    char name[64];
    char sex; // 'm' or 'f'
    int year_of_birth;
};
```

Пусть теперь нам понадобилась структура данных, описывающая *студента* (частный случай персоны). Относительно студента нас может интересовать код специальности, номер курса и показатель успеваемости (средний балл). Вместе с тем, несомненно, студент — это тоже человек, так что ему присущи и свойства, общие для всех персон: имя, пол и год рождения; иначе говоря, мы можем перейти от общего (персона) к частному (студент), *добавив* уточняющие сведения, и сделать это можно с помощью механизма наследования. На Си++ это записывается так:

```
struct student : person {
    int code;
    int year;              // курс
    float average;
};
```

Если теперь описать переменную типа `student`, она будет иметь все свойства структуры `person` плюс дополнительные поля:

```
    student s1;
    strcpy(s1.name, "John Doe");
    s1.sex = 'm';
    s1.year_of_birth = 1989;
```

```
s1.code = 51311;
s1.year = 2;
s1.average = 4.75;
```

Структура `person` в данном случае называется ***базовой*** или ***родительской***, а структура `student` — ***унаследованной*** или ***порождённой***, иногда ***дочерней***. Также употребляются термины «предок» и «потомок» (в данном случае соответственно для `person` и `student`).

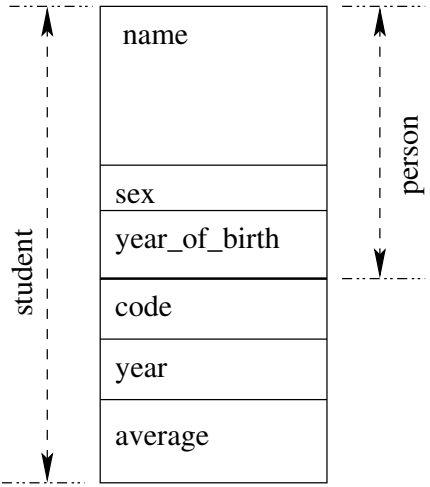


Рис. 3. Унаследованная структура данных

Расположение полей переменной `s1` в памяти показано на рис. 3. Стоит специально отметить, что поля, относящиеся к части структуры `student`, унаследованной от `person`, располагаются на тех же местах (по тем же смещениям относительно начала), где они располагались бы в структуре `person`. Получается, что мы можем при необходимости работать с переменной `s1` точно так, как если бы она была переменной типа `person`, а не типа `student` Важно понимать, что обратное неверно: в структуре `person` отсутствуют поля, которые ожидались бы от структуры `student`. В связи с этим **Си++ разрешает неявное преобразование адреса переменной порождённого типа в адрес объекта родительского типа**, в нашем примере `student*` в `person*`. Соответствующие преобразования разрешены как для указателей, так и для ссылок. Например, следующие строки полностью корректны:

```
student s1;
person *p;
```

```
    p = &s1;
    person &ref = s1;
```

Если в программе описана функция, принимающая параметром указатель или ссылку на объект типа `person`, ей без каких-либо сложностей можно передать соответственно указатель или ссылку на объект типа `student`:

```
void f(person &pers) { ... }
    // ...
    student s1;
    // ...
    f(s1); // корректно!
```

Это свойство предков, потомков и их адресов называется ***полиморфизмом адресов*** (реже — полиморфизмом ссылок); в отличие от рассмотренных ранее проявлений *статического* полиморфизма, этот новый его вид имеет к объектно-ориентированному программированию самое прямое отношение. Разрешённое в объектно-ориентированных языках неявное преобразование адресов (указателей) и ссылок от типа-потомка к типу-предку мы будем называть ***преобразованием по закону полиморфизма***.

Мы подразумеваем, что полиморфизм, какой бы его вид мы ни рассматривали, состоит в том, что операция или действие, обозначаемое одним и тем же символом, корректно выполняется для операндов или параметров различных типов. Часто встречается немного иное понимание полиморфизма: что одно и то же обозначение используется для *разных* действий, выбираемых в зависимости от типов параметров. Если полиморфизм понимать так, то наш «полиморфизм адресов» перестанет быть, собственно говоря, полиморфизмом, ведь здесь, напротив, действия (технически) выполняются совершенно одинаковые вне зависимости от типа; при этом рассмотренные ранее виды полиморфизма такому пониманию вполне соответствуют.

Так или иначе, применяемая терминология может различаться от источника к источнику.

5.3. Методы и защита при наследовании

Поскольку наследование, несомненно, является составной частью объектно-ориентированного программирования, чаще всего оно применяется для типов, имеющих функции-члены (методы), и в роли таких типов классы выступают намного чаще, чем структуры. Технически это никоим образом не требование, ведь структуры от классов, как мы уже знаем (см. §2.5), отличаются только моделью защиты по умолчанию — и более ничем. Естественно, наследование возможно как для структур, так и для классов, причём классы можно наследовать от структур и наоборот; просто, как мы отмечали ранее, для

описания объектов в смысле ООП программисты стараются использовать именно классы, а не структуры, чтобы показать, что здесь имеется в виду именно эта парадигма. В §1.1 мы отмечали, что само слово *класс* изначально относится к терминологии ООП — это множество объектов, устроенных одинаково, а с прагматической точки зрения — описание устройства объекта; в этом смысле, даже если для создания объекта (в смысле ООП) мы воспользуемся структурой, полученный тип следует считать именно классом, если не в терминах языка Си++, то в терминах ООП. В дальнейшем изложении мы будем использовать слово «класс» для обозначения всех типов объектов, если таковые относятся к парадигме ООП, не уточняя, что это (в терминах Си++) может быть как класс, так и структура.

Считается, что построение класса-потомка на основе базового класса *может быть* (хотя и не обязано, и в большинстве случаев не является) *частной деталью реализации*, о которой окружающий мир знать не должен. В связи с этим в Си++ различают наследование открытое (public) и закрытое (private). Модель защиты для конкретного случая наследования указывается в заголовке класса непосредственно перед названием класса-предка, например:

```
class B : public A {
   /*...*/
};
```

или

```
class C : private A {
   /*...*/
};
```

Если в первом случае все свойства класса `A` (его открытые методы и поля, если такие в нём есть) будут доступны для объектов класса `B` отовсюду, то во втором случае — только из методов класса `C`, а вся остальная программа вообще не будет знать, что класс `C` унаследован от `A`. Всё это работает и для структур.

Модель защиты для наследования можно не указывать, как мы это делали в примере на стр. 109. В этом случае будут действовать умолчания: для структуры наследование по умолчанию открытое (public), для класса — закрытое (private). **В реальной жизни потребность в закрытом наследовании возникает крайне редко**, поэтому при описании наследуемых классов обычно указывают слово `public`, а при описании наследуемых структур не указывают ничего, полагаясь на умолчания.

Заметим теперь, что для базового класса порождённый класс ничуть не «лучше» всей остальной программы и, как и любой недружественный фрагмент кода, не должен иметь доступа к деталям реализации класса. Поэтому, очевидно, закрытые поля и методы базового класса не будут доступны порождённым классам.

В реальных задачах часто возникает потребность в таких деталях интерфейса класса, которые предназначены только и исключительно для его потомков. Для таких случаев наряду с режимами `public` и `private` вводится ещё и третий режим защиты, `protected` (*защищённый*). Поля и методы, помеченные словом `protected`, будут доступны в самом классе (то есть в его методах), в дружественных функциях, а также в методах потомков данного класса[25]; во всей остальной программе доступ к ним будет запрещён.

Важно понимать, что поля и методы, имеющие режим защиты `protected`, не могут считаться в полном смысле слова деталями реализации, которые не касаются никого, кроме данного класса. Разница здесь достаточно принципиальна. Детали, помещённые в закрытую (private) часть класса, можно безболезненно изменять, исправляя при этом только методы самого класса и дружественные функции, которые опять-таки перечислены в явном виде в заголовке класса. Иначе говоря, переделывая приватную часть класса, мы всегда знаем, где проходит граница кода, который может при этом «сломаться» и который нам, возможно, придётся исправлять. Напротив, особенности реализации класса, помеченные как `protected`, могут использоваться в самых неожиданных частях программы: достаточно кому-то где-то описать ещё одного потомка нашего класса, о котором мы не подозреваем. Поэтому, в отличие от приватных полей и методов, поля и методы с режимом `protected` должны обязательно документироваться, а к их изменению следует подходить столь же осторожно, как и к изменению открытой (публичной) части класса. Можно сказать, что `protected` — это хоть и особый, но всё же вид *открытых* особенностей класса.

Если мы работаем с объектом-потомком, в его классе-предке введён некий метод и нам этот метод доступен, то есть не скрыт от нас защитой, мы можем вызывать такой метод для объекта-потомка точно так же, как и для объекта-предка, причём, если не предпринять специальных мер, методы предка не будут ничего знать о том, что их вызывают для объекта-потомка, то есть будут работать так

[25] Для непосредственных потомков это верно всегда; если же речь идёт о *потомке потомка* или о ещё более дальнем «родстве», может оказаться так, что в цепочке наследников встретилось наследование закрытое, и тогда потомок может вообще не знать, что его предок построен на основе другого класса; естественно, такой потомок не будет иметь доступа ни к каким членам «засекреченного» дальнего предка, даже к публичным — и к защищённым тоже.

же, как и для объектов базового класса. Например, если мы сделаем класс «автомобиль» и предусмотрим для него операцию «заправка бензином», а потом породим от него класс «грузовик» с дополнительными свойствами, то операция «заправка бензином» будет доступна и для грузовика, причём реализация этой операции не будет ничего знать про особенности грузовиков и будет работать с объектом типа «грузовик» точно так же, как и с объектом типа «автомобиль».

Здесь работает описанный в предыдущем параграфе адресный полиморфизм в применении к параметру `this` (см. §2.2). В самом деле, пусть класс `B` унаследован от класса `A`, в котором есть функция (метод) `f`. Тогда указатель `this` в методе `f` имеет тип `A*` (или `const A*`, если метод константный). При вызове `f` для объекта класса `B` указатель `this` должен иметь тип `B*`, но, как мы знаем, `B*` разрешено неявно преобразовывать в `A*` по закону полиморфизма.

В терминах ООП это означает, что потомок умеет отвечать на все виды сообщений, предусмотренные для его предков; чтобы объекту можно было отправить такое сообщение (т. е. вызвать метод предка), достаточно иметь доступ к этому методу, то есть чтобы он не был скрыт защитой. Иначе говоря, с объектом-наследником можно (при желании) работать точно так же, как с объектом-предком, вообще не обращая внимания на тот факт, что мы имеем дело с объектом другого типа. Часто можно даже встретить утверждение, что **объект класса-потомка является также и объектом класса-предка**; учитывая, что на идеологическом уровне, как мы уже отмечали, наследование есть переход от общего к частному, объект-потомок оказывается *частным случаем* класса-предка. В самом деле, разве грузовик не является частным случаем автомобиля?

С другой стороны, как видно из рис. 3, объект порождённого класса *содержит* в себе объект базового класса в качестве своей части. Никакого противоречия тут нет, это просто две разные модели восприятия (можно даже сказать, что это разные парадигмы) или, если угодно, разные точки зрения: одна — точка зрения реализации средств языка (реализаторская семантика), вторая — точка зрения логики проектирования программы (пользовательская семантика). К этому вопросу мы ещё вернёмся.

5.4. Конструирование и деструкция наследника

Конструкторы и деструкторы заслуживают особого упоминания при появлении наследования в программе. С какой бы точки зрения мы ни рассматривали порождённый объект, очевидно, что в момент его создания также создаётся и объект базового класса, причём неважно, считаем мы его «частью» порождённого объекта или же его «другой ипостасью». Таким же точно образом при уничтожении порождённого объекта исчезает и базовый объект. Следова-

тельно, при создании объекта порождённого класса должен отработать и конструктор базового класса, а при уничтожении такого объекта — деструктор базового класса. При этом, очевидно, в телах конструктора и деструктора порождённого класса должны быть доступны все части объекта, для которого они вызываются. Базовый объект, таким образом, должен быть инициализирован *раньше*, а уничтожен — *позже* объекта порождённого. Получается, что время существования объекта класса-наследника в его «базовой» ипостаси как бы чуть-чуть больше, чем время существования его же в качестве объекта своего собственного типа.

С деструктором дела обстоят достаточно просто: компилятор автоматически вставляет в код деструктора порождённого класса (в самый его конец) вызов деструктора базового класса, так что сначала отработает тело деструктора порождённого класса, а затем — тело деструктора базового класса. С конструктором всё тоже могло бы быть просто (сначала вызвать тело конструктора базового класса, потом тело конструктора класса порождённого), если бы не то обстоятельство, что конструкторы (в общем случае) могут требовать входных параметров. С похожей проблемой мы уже сталкивались при рассмотрении классов, имеющих поля типа класс (см. §3.15). Решается проблема в данном случае совершенно аналогично: в описании конструктора порождённого класса между заголовком и телом вставляется список инициализаторов, начинающийся с инициализатора базового класса (обозначаемого в данном случае именем класса) с указанием всех нужных параметров конструктора. Так, если `A` — базовый класс, конструктору которого требуются два параметра типа `int`, `B` — класс, унаследованный от `A`, снабжённый конструктором по умолчанию и имеющий поле `int i`:

```
class A {
   // ...
public:
   A(int p, int q) { /* ... */ }
   // ...
};

class B : public A {
   int i;
public:
   B();
   // ...
};
```

— то описание его конструктора может выглядеть так:

```
B::B() : A(2, 3), i(4)
{
    /* ... */
}
```

где 2 и 3 — параметры для конструктора базового класса, 4 — начальное значение поля i.

5.5. Виртуальные функции; динамический полиморфизм

До сих пор мы предполагали, что объект-потомок будет реагировать на сообщения, определённые для его предка, *точно так же*, как на них реагировал бы объект-предок. В ряде случаев это не отвечает потребностям моделируемой предметной области: бывает так, что на те или иные сообщения потомку желательно реагировать иначе, каким-то своим собственным способом. Достичь этого позволяет механизм ***виртуальных функций***; с его помощью автор класса-предка может предоставить потомкам возможность модификации поведения отдельных методов, причём как полной, так и частичной. Чтобы понять, как это происходит и зачем всё это нужно, мы для начала рассмотрим пример, а затем поясним, как соответствующий механизм устроен.

Итак, допустим, что перед нами стоит задача, связанная с компьютерной графикой, и нужно описать сцену (то есть набор графических элементов) в виде некоего списка или массива объектов, задающих графические объекты различного типа, например, отдельные пиксели, линии, окружности и т. п. Для начала опишем класс, объекты которого будут представлять в нашей сцене простейшие графические примитивы — отдельные пиксели. Ясно, что у пикселя имеются две координаты (их обычно задают числами с плавающей точкой) и цвет (целое число). Будем считать, что основные действия с графическим объектом, в том числе и с пикселями — это показать объект на экране (мы обозначим это действие функцией Show), убрать его с экрана (Hide) и переместить его в новую позицию (Move).

```
class Pixel {
    double x, y;
    int color;
public:
    Pixel(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) {}
    void Show();
    void Hide();
```

```
    void Move(double nx, double ny);
};
```

Конкретика описания тел функций `Show` и `Hide` зависит от используемой графической библиотеки, правил пересчёта координат и т. п.; мы на этом останавливаться не будем, просто предположим, что эти функции где-то описаны. Используя их, мы очень легко можем описать функцию `Move`, ведь перемещение состоит в том, что объект сначала убирают с экрана, потом меняют его координаты и наконец снова показывают:

```
void Pixel::Move(double nx, double ny)
{
    Hide();
    x = nx;
    y = ny;
    Show();
}
```

Предположим теперь, что нам нужен также объект «окружность». Понятно, что такой объект обладает теми же свойствами (координаты и цвет) и в дополнение к ним характеризуется ещё радиусом. Поэтому мы воспользуемся уже имеющимся классом `Pixel` в качестве базового[26], а класс `Circle` от него *унаследуем*. Конечно, методам класса `Circle` потребуется знать координаты и цвет, хотя бы для того, чтобы уметь рисовать заданную окружность на экране. В нашем примере мы решим вопрос доступа самым простым (хотя и не самым лучшим) способом: сделаем поля базового класса защищенными (`protected`), а не закрытыми; для этого вставим в самое начало описания класса `Pixel` директиву `protected`:

```
class Pixel {
protected:
    double x;
    //...
```

Теперь мы можем перейти к описанию класса `Circle` и для начала опишем поле, которого нам не хватает, чтобы превратить точку в окружность:

```
class Circle : public Pixel {
    double radius;
public:
```

[26] Здесь мы несколько нарушаем принципы объектно-ориентированного проектирования, поскольку окружность, очевидно, не является частным случаем точки. Мы исправим эту оплошность в одном из следующих параграфов.

Опишем теперь конструктор для класса `Circle`. При этом нам придётся принять на один параметр больше, чем в конструкторе класса `Pixel`. Кроме того, поскольку единственный конструктор базового класса требует указания трёх параметров, нужно будет задать параметры для вызова конструктора базового класса, как это обсуждалось на стр. 115. Окончательно описание выйдет таким:

```
Circle(double x, double y, double rad, int color)
    : Pixel(x, y, color), radius(rad) {}
```

Разумеется, в объекте необходимо описать функции `Show` и `Hide`, предназначенные для рисования и стирания с экрана окружности, которые будут отличаться от функций, предназначенных для одиночного пикселя; мы, как и при рассмотрении класса `Pixel`, воздержимся от описания конкретных тел этих функций, ограничившись их заголовками:

```
void Show();
void Hide();
```

Нетерпеливый читатель может предположить, что сейчас мы будем описывать функцию `Move`, аналогичную той, что описана выше для класса `Pixel`. Вместо этого мы предложим немного подумать. Легко заметить, что *функция `Move` для нового класса окажется абсолютно такой же, как и для класса базового*, то есть нам потребуется написать не только точно такой же заголовок, но и точно такое же, с точностью до последней буквы, тело функции! Это и неудивительно, ведь вне всякой зависимости от формы графической фигуры её перемещение по экрану производится абсолютно одинаково, в три шага: стереть, изменить координаты, нарисовать в новом месте.

Возникает естественный вопрос, нельзя ли использовать для класса `Circle` ту же самую функцию `Move`, которая уже описана для класса `Pixel`, не создавая новой версии этой функции для `Circle`. Легко заметить, что (если не предпринять специальных мер) простой вызов функции `Move` для объекта класса `Circle`, конечно, возможен (ведь это же публичная функция базового класса, а наследование произведено по публичной схеме), но *не приведёт к нужным нам результатам*, и вот почему. Во время трансляции тела функции `Pixel::Move` компилятор может ничего не знать о существовании потомков у класса `Pixel`, которые к тому же вводят свои версии функций `Show` и `Hide`. Поэтому компилятор, естественно, вставляет в машинный код функции `Pixel::Move` обычные вызовы функций `Pixel::Show` и `Pixel::Hide`, то есть инструкции `call` с жёстко заданными исполнительными адресами, не подразумевающие никаких изменений.

Если теперь вызвать функцию `Move` для объекта класса `Circle`, она для стирания с экрана объекта вызовет функцию `Hide` в той её версии, в которой она описана для класса `Pixel`. Эта функция выполнит действия по «стиранию» с экрана объекта-точки, тогда как стереть на самом деле нужно окружность. То же самое произойдёт с функцией `Show`: вместо окружности в новой позиции на экране будет отрисована точка. Ясно, что это совсем не то, что нам нужно.

Тем не менее, кажется странным и неправильным описывать для класса `Circle` функцию, слово в слово повторяющую другую, которая уже имеется в базовом классе. И здесь нам на помощь приходит механизм ***виртуальных функций***. Кратко говоря, виртуальная функция (или виртуальный метод) — это функция, описывающая такое действие над объектом, относительно которого предполагается, что аналогичное действие будет определено и для объектов классов-потомков, но, возможно, для потомков оно будет выполняться иначе, чем для базового класса. В терминах теории ООП можно сказать, что виртуальным методом задаётся реакция объекта класса на некоторый тип сообщений в случае, если:

- предполагается, что у данного класса будут классы-потомки;
- объекты классов-потомков будут способны получать сообщение того же типа;
- объекты некоторых или всех потомков будут реагировать на эти сообщения иначе, чем это делает объект класса-предка.

Язык Си++ позволяет объявить в качестве виртуальной любую функцию-метод, кроме конструкторов и статических методов. Для этого перед заголовком функции ставится ключевое слово `virtual`. В отличие от вызовов обычных функций, вызовы функций виртуальных обрабатываются компилятором в предположении, что тип объекта, для которого вызывается функция, может отличаться от базового класса, и что для этого объекта может потребоваться вызов совсем другой функции, нежели для базового класса. В частности, если в классе `Pixel` поставить слово `virtual` перед каждой из функций `Show` и `Hide`, то при компиляции тела функции `Move` компилятор будет знать, что объект, на который указывает параметр `this`, может быть как объектом самого класса `Pixel`, так и объектом какого-нибудь его класса-потомка, причём для такого объекта может потребоваться вызов *других версий* функций `Show` и `Hide`, введённых в соответствующем потомке. Код, сгенерированный компилятором в такой ситуации, будет несколько сложнее, чем для обычного вызова функции, но зато он будет вызывать именно функцию, соответствующую типу объекта.

Попробуем понять, как (технически) это достигается. Если в классе описана хотя бы одна виртуальная функция, то компилятор вставляет во все объекты этого класса невидимое поле, называемое ***указателем на таблицу виртуальных методов*** (англ. *virtual method table pointer*, *vmtp*). Для всего класса создаётся (в одном экземпляре) неизменяемая ***таблица виртуальных методов***, содержащая указатели на каждую из описанных в классе виртуальных функций. Когда компилятор встречает вызов виртуальной функции, он вставляет в объектный код инструкции извлечь из объекта значение поля `vmtp`, затем обратиться по полученному адресу к таблице виртуальных методов и из неё взять адрес нужной функции, после чего обратиться к функции, используя адрес.

Объект класса-потомка содержит, как мы знаем, все поля, присущие классу-предку. Это касается и невидимого поля `vmtp`, причём объекты класса-потомка имеют в этом поле иное значение, нежели объекты класса-предка. Для каждого класса-потомка компилятор создаёт (опять-таки в единственном экземпляре) свою собственную таблицу виртуальных методов, содержащую адреса соответствующих версий виртуальных функций; именно адрес этой таблицы заносится в поле `vmtp`. За инициализацию `vmtp` отвечает конструктор класса, в начало которого компилятор вставляет соответствующие команды.

Итак, объявим функции `Show` и `Hide` виртуальными в базовом классе:

```
class Pixel {
    // ...
    virtual void Show();
    virtual void Hide();
    // ...
};
```

Это позволит использовать функцию `Move` для любого из классов, унаследованных от `Pixel`, если только для этих классов описаны собственные (правильно работающие) функции `Show` и `Hide`.

Прежде чем завершить описание классов `Pixel` и `Circle`, отметим, что в классе, в котором имеется хотя бы одна виртуальная функция, рекомендуется всегда явно описывать деструктор и объявлять его виртуальным. Зачем это нужно, мы расскажем в §5.9. Сейчас просто сделаем это, чтобы компилятор не выдавал предупреждения. Окончательно заголовки наших классов примут следующий вид:

```
class Pixel {
protected:
```

```
    double x, y;
    int color;
public:
    Pixel(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) {}
    virtual ~Pixel() {}
    virtual void Show();
    virtual void Hide();
    void Move(double nx, double ny);
};

class Circle : public Pixel {
    double radius;
public:
    Circle(double x, double y, double rad, int color)
        : Pixel(x, y, color), radius(rad) {}
    virtual ~Circle() {}
    virtual void Show();
    virtual void Hide();
};
```

Отметим, что в описании класса `Circle` слово `virtual` можно опустить; функции, совпадающие по профилю с виртуальными функциями базового класса, объявляются виртуальными автоматически.

Виртуальные функции можно вызывать не только из других методов того же класса, как в нашем примере; компилятор сгенерирует код для вызова через таблицу виртуальных методов при *любом* обращении к виртуальной функции, если тип объекта, для которого её вызывают, хотя бы теоретически может меняться[27]. Конечно, когда объект описан в виде обычной переменной типа класс и мы обращаемся к его методу, пусть и виртуальному, напрямую через имя этой переменной и точку, то есть в программе написано что-то вроде

```
A a;
// ...
a.func();   // всегда вызывается A::func; механизм
            // виртуальности не задействован
```

— то такое обращение, очевидно, бессмысленно реализовывать через виртуальность, ведь тип переменной известен во время компиляции и стать другим не может. Иное дело, если обращение к виртуальной функции записано через адрес объекта (ссылку или указатель). Это касается в том числе вызовов методов из тел других методов, где

[27] На самом деле из этого правила есть одно важное исключение, которое мы рассмотрим позже в §5.13.

объект, как мы знаем, идентифицируется указателем `this` — но *не ограничивается* методами.

Так, если рассматривать классы из нашего примера, то где бы у нас в программе ни появился какой-нибудь указатель типа `Pixel*` или `Circle*`, вызовы через такой указатель функций `Show` и `Hide` компилятор оформит как виртуальные, предполагая, что указатель во время исполнения может содержать адрес не только объекта базового класса, но и какого-нибудь его потомка, в том числе такого, о котором компилятору на момент компиляции ничего не известно; то же самое касается ссылок типа `Pixel&` и `Circle&`. Скажем, если мы пишем какую-нибудь функцию, имеющую параметр `p` типа `Pixel*`, то, решив показать переданный нам объект на экране, мы можем выполнить

```
p->Show();
```

и не волноваться о том, адрес объекта какого типа нам передали — ведь вызван в любом случае будет метод нужного класса.

5.6. Чисто виртуальные методы и абстрактные классы

Как отмечалось в сноске на стр. 117, наследование класса «окружность» от класса «точка» нарушает принципы объектно-ориентированного проектирования, поскольку окружность не является частным случаем точки. Попробуем исправить ситуацию. Для этого заметим, что и точка, и окружность — частные случаи *геометрических фигур*, причём можно считать, что каждая геометрическая фигура обладает цветом и имеет координаты *точки привязки*. Для обычной точки в качестве точки привязки выступает она сама, для окружности точкой привязки будет её центр. Для других фигур точку привязки можно выбрать разными способами; так, для какого-нибудь прямоугольника это может быть либо центр пересечения диагоналей, либо одна из вершин, и т. п.

Такое представление об абстрактном понятии геометрической фигуры позволяет нам указать единый для всех фигур алгоритм передвижения фигуры по экрану: стереть фигуру с экрана, изменить координаты точки привязки, отрисовать фигуру в новом месте. С этим алгоритмом мы уже знакомы, он был реализован в функции `Move` на стр. 117.

Теперь ясно, как нужно изменить архитектуру нашей библиотеки классов, чтобы привести её в соответствие с основными принципами объектно-ориентированного программирования. Понятия «точка» и «окружность» не являются частными случаями друг друга, но оба

они являются частным случаем понятия «геометрическая фигура». Поэтому, если мы опишем класс для представления абстрактной геометрической фигуры и от него унаследуем оба класса `Pixel` и `Circle`, такая архитектура будет полностью удовлетворять требованиям теории объектно-ориентированного проектирования.

Прежде чем приступать к описанию класса, представляющего геометрическую фигуру, отметим ещё один важный момент. Описывая в предыдущем параграфе классы для точек и окружностей, мы не писали конкретных тел для методов `Show` и `Hide`, но предполагали при этом, что в рабочей программе тела этих методов будут описаны (с учетом конкретной платформы разработки, используемой графической библиотеки и т. п.). Но для абстрактной геометрической фигуры тела методов `Show` и `Hide` описать *невозможно*; действительно, как можно нарисовать на экране фигуру, не зная, как она выглядит?!

Несмотря на это, мы точно знаем, как написать функцию `Move` в предположении, что для классов-потомков будут правильно описаны методы `Show` и `Hide`. Иначе говоря, мы знаем, что все объекты классов-потомков данного класса должны уметь получать сообщение определённого типа, но мы не можем при описании базового класса описать какую бы то ни было реакцию на такие события, поскольку таковая реакция полностью зависит от типа нашего потомка. При этом для описания некоторых других (более общих) методов базового класса нам требуется знание о том, что реакция на соответствующие события будет предусмотрена во всех наших потомках.

Специально для таких случаев в Си++ введены так называемые ***чисто виртуальные функции*** (англ. *pure virtual functions*). Описывая в классе чисто виртуальную функцию, программист информирует компилятор о том, что функция с таким профилем будет существовать во всех классах-потомках, что под неё нужно зарезервировать позицию в таблице виртуальных функций, но при этом сама функция (её тело) для базового класса описываться не будет, так что значение адреса этой функции в таблице виртуальных функций следует оставить нулевым. Синтаксис описания чисто виртуальной функции таков:

```
class A {
    // ...
    virtual void f() = 0;
    // ...
 };
```

Видя на месте тела функции специальную лексическую последовательность «`= 0;`», компилятор воспринимает функцию как чисто виртуальную[28].

Класс, в котором есть хотя бы одна чисто виртуальная функция, называется ***абстрактным классом***. Полезно будет запомнить, что **компилятор не позволяет создавать объекты абстрактных классов**. Единственное назначение абстрактного класса — служить базисом для порождения других классов, в которых все чисто виртуальные функции будут конкретизированы. Если в порождённом классе не описывается хотя бы одна из функций, объявленных в базовом классе как чисто виртуальные, компилятор считает, что функция осталась чисто виртуальной; такой класс-потомок считается абстрактным, как и его предок.

Подытожим наши рассуждения. Чтобы соответствовать принципам объектно-ориентированного программирования, нам следует описать класс, представляющий абстрактную геометрическую фигуру, обладающую цветом и координатами точки привязки, но не обладающую конкретной формой; назовём этот класс `GraphObject`. Классы `Pixel` и `Circle` нужно переписать, унаследовав от `GraphObject`. В классе `GraphObject` методы `Show` и `Hide` мы объявлены как чисто виртуальные, так что сам этот класс будет абстрактным:

```
class GraphObject {
protected:
    double x, y;
    int color;
public:
    GraphObject(double ax, double ay, int acolor)
        : x(ax), y(ay), color(acolor) {}
    virtual ~GraphObject() {}
    virtual void Show() = 0;
    virtual void Hide() = 0;
    void Move(double nx, double ny);
};
```

Описание функции `Move` мы не приводим, поскольку оно слово в слово повторяет описание метода `Pixel::Move` на стр. 117. Напомним, что в теле функции `Move` вызываются функции `Show` и `Hide`. То, что тела для этих функций нами не заданы, не создаёт никаких проблем, поскольку для любого класса-потомка, объекты которого можно бу-

[28] Такой синтаксис трудно назвать удачным, особенно если учесть, что никакое число, кроме нуля, использоваться здесь не может; тем не менее, именно таков синтаксис в языке Си++. Вопрос о причинах этого следовало бы задать лично Бьёрну Страуструпу, но, скорее всего, он на него не ответит.

дет создавать, таблица виртуальных методов будет содержать адреса конкретных функций, описанных в этом классе-потомке.

Заголовки классов `Pixel` и `Circle` будут теперь выглядеть так:

```
class Pixel : public GraphObject {
public:
    Pixel(double x, double y, int col)
        : GraphObject(x, y, col) {}
    virtual ~Pixel() {}
    virtual void Show();
    virtual void Hide();
};
class Circle : public GraphObject {
    double radius;
public:
    Circle(double x, double y, double rad, int color)
        : GraphObject(x, y, color), radius(rad) {}
    virtual ~Circle() {}
    virtual void Show();
    virtual void Hide();
};
```

Для этих классов, в отличие от класса `GraphObject`, необходимо описать конкретные тела методов `Show` и `Hide`, иначе программа не пройдёт этап линковки.

5.7. Виртуальность в конструкторах и деструкторах

Вызов виртуальных функций в телах конструкторов и деструкторов сопряжён с одной достаточно нетривиальной особенностью, связанной с конструированием и последующим деструктированием в нашем объекте невидимого поля, содержащего адрес таблицы виртуальных функций. Естественно, это поле заполняет конструктор; но, как несложно догадаться, он делает это *после того, как остальные подобъекты нашего объекта уже сконструированы*, в том числе сконструирован и предок. Между тем, конструктор предка тоже, естественно, содержит код, инициализирующий указатель на таблицу виртуальных методов, а поскольку про потомков он ничего не знает, в поле `vmtp` он занесёт адрес *своей* таблицы.

Получается, что, коль скоро в классе вообще имеются виртуальные методы, **во время выполнения тела конструктора виртуальные функции вызываются те, которые описаны для данного класса, вне зависимости от того, объект какого класса**

на самом деле конструируется. Для симметрии аналогичный эффект присутствует также и в телах деструкторов; после завершения своего тела деструктор «деинициализирует» указатель на виртуальную таблицу — физически это означает, что в поле `vmtp` записывается адрес таблицы виртуальных методов предка.

В частности, из конструкторов и деструкторов вообще нельзя вызывать методы, описанные в данном объекте как чисто виртуальные: вызывать будет попросту некого. Не стоит, как мы видим, уповать и на то, что объект-потомок сможет как-то скорректировать действия нашего конструктора или деструктора за счёт переопределения вызываемых из них виртуальных методов. При возникновении любых сомнений на этот счёт лучше будет вообще воздержаться от обращения к виртуальным методам из конструкторов и деструкторов.

5.8. Наследование ради конструктора

В практическом программировании часто применяют один упрощённый случай наследования, при котором класс-потомок отличается от предка только набором конструкторов, то есть он не вводит ни новых методов, ни новых полей. Объекты такого класса создаются из соображений экономии объёма кода, чтобы не повторять одни и те же действия при конструировании однотипных объектов. Чтобы проиллюстрировать сказанное на примере, для начала мы введём ещё один класс-потомок класса `GraphObject`, представляющий ломаную линию, а затем опишем графический объект «квадрат» как частный случай ломаной линии.

Напомним, что каждая геометрическая фигура в нашей системе имеет точку привязки; ломаную при этом проще всего хранить в виде списка координатных пар, задающих смещение каждой вершины ломаной относительно точки привязки. Для организации такого списка мы в закрытой части класса опишем структуру, задающую элемент списка. Исходно будем создавать ломаную, не имеющую ни одной вершины, а для добавления новых вершин будем использовать метод, который назовём `AddVertex`[29]. Напишем заголовок класса:

```
class PolygonalChain : public GraphObject {
    struct Vertex {
        double dx, dy;
        Vertex *next;
```

[29]Это лучше, чем пытаться тем или иным способом передать координаты вершин в конструктор, поскольку мы не знаем заранее количество вершин, так что для передачи их в качестве параметра конструктора пришлось бы в том месте, где создаётся объект, формировать некую динамическую структуру данных (массив либо список), что потребовало бы дополнительных усилий.

```
    };
    Vertex *first;
public:
    PolygonalChain(double x, double y, int color)
        : GraphObject(x, y, color), first(0) {}
    virtual ~PolygonalChain();
    void AddVertex(double adx, double ady);
    virtual void Show();
    virtual void Hide();
};
```

Функция `AddVertex` будет (для экономии усилий) добавлять новую вершину в начало списка, а не в конец; линия при этом будет изображаться на экране в обратном порядке, но это, естественно, ничего не изменит:

```
void PolygonalChain::AddVertex(double ax, double ay)
{
    Vertex *tmp = new Vertex;
    tmp->dx = ax;
    tmp->dy = ay;
    tmp->next = first;
    first = tmp;
}
```

Поскольку наш объект использует динамическую память, ему нужен деструктор. Напишем его:

```
PolygonalChain::~PolygonalChain()
{
    while(first) {
        Vertex *tmp = first;
        first = first->next;
        delete tmp;
    }
}
```

Как и раньше, мы воздержимся от написания тел функций `Show` и `Hide`, но будем предполагать, что это сделано.

Пусть теперь нам нужен класс для представления квадрата, стороны которого параллельны осям координат, а длина стороны задаётся параметром конструктора. Ясно, что такой квадрат представляет собой частный случай ломаной, описываемой классом `PolygonalChain`. Если в качестве точки привязки выбрать левую нижнюю вершину квадрата, а длину стороны квадрата обозначить буквой a, то ломаная должна начаться в точке привязки (что соответствует вектору $(0,0)$), пройти через точки $(a,0)$, (a,a), $(0,a)$ и

вернуться в точку $(0, 0)$; всего, таким образом, ломаная будет содержать пять вершин, причём первая и последняя будут совпадать, чтобы сделать ломаную замкнутой.

Чтобы не приходилось каждый раз для представления квадрата писать шесть строк кода (одну для описания объекта `PolygonalChain`, остальные для добавления вершин), можно описать класс (мы назовём его `Square`), который будет унаследован от `PolygonalChain`, причём отличаться будет только конструктором:

```
class Square : public PolygonalChain {
public:
    Square(double x, double y, double a, int color)
        : PolygonalChain(x, y, color)
    {
        AddVertex(0, 0);
        AddVertex(0, a);
        AddVertex(a, a);
        AddVertex(a, 0);
        AddVertex(0, 0);
    }
};
```

Подчеркнём ещё раз, что больше ничего описывать для квадрата не нужно, всё остальное сделают методы базового класса.

5.9. Виртуальный деструктор

Ранее при обсуждении виртуальных функций на стр. 120 мы отметили, что в классе, имеющем хотя бы одну виртуальную функцию, деструктор тоже следует сделать (объявить) виртуальным, но не объяснили причины этого. Попробуем сделать это сейчас.

При активном использовании полиморфизма нередко возникает ситуация, когда нужно применить оператор `delete` к указателю, имеющему тип «указатель на базовый класс», притом что указывать он может и на объект потомка. В этой ситуации требуется, естественно, вызвать деструктор, соответствующий типу уничтожаемого объекта, а не указателя. Так, допустим, мы описали указатель на класс `GraphObject`:

```
GraphObject *ptr;
```

Теперь вполне корректным будет такое присваивание:

```
ptr = new Square(27.3, 37.7, 0xff0000, 10.0);
```

Теперь наш указатель имеет тип «указатель на `GraphObject`», но реально указывает на объект класса `Square`. Если теперь потребуется уничтожить этот объект, мы можем без всяких опасений выполнить

```
delete ptr;
```

Поскольку указатель имеет тип `GraphObject*`, а деструктор класса `GraphObject` виртуальный, компилятор произведёт вызов деструктора через таблицу виртуальных методов уничтожаемого объекта. В результате этого будет вызван неявный деструктор для класса `Square`, который вызовет деструктор класса `PolygonalChain`, а тот, в свою очередь, — деструктор класса `GraphObject`. Если бы мы не объявили деструктор класса `GraphObject` как виртуальный, компилятор произвёл бы жёсткий вызов деструктора по типу указателя, то есть был бы вызван только деструктор класса `GraphObject`. Между тем уничтожаемый объект класса `Square` порождает и использует список в динамической памяти (из пяти элементов), и если деструктор класса `PolygonalChain` не будет вызван, то эти элементы превратятся в «мусор», то есть будут по-прежнему занимать память, не принося никакой пользы.

Объявление деструктора как виртуального практически не приводит к расходу памяти: таблица виртуальных методов увеличивается на один слот, что составляет несколько лишних байтов на каждый новый *класс* (не объект!). При этом сам факт наличия в классе виртуальных функций указывает на то, что при работе с объектами класса будет активно использоваться полиморфизм. При описании класса в большинстве случаев трудно предсказать, будут ли объекты потомков такого класса уничтожаться оператором `delete`, применяемым к указателю, имеющему тип «указатель на предка». Решив, что такое удаление нам не понадобится, мы рискуем в дальнейшем забыть об этом и получить трудную для обнаружения ошибку. Именно поэтому считается, что **деструктор любого класса, имеющего хотя бы одну виртуальную функцию, следует объявлять как виртуальный, не задумываясь о том, понадобится это в программе или нет; многие компиляторы выдают предупреждение, если этого не сделать**.

5.10. Ещё о полиморфизме

Приведём ещё один пример использования полиморфизма. Пусть мы создаём графическую сцену[30], состоящую из разных графических объектов — точек, окружностей, многоугольников и, возмож-

[30] Напомним, что под сценой в компьютерной графике обычно понимается весь набор графических объектов, видимых одновременно.

но, каких-то других элементов, представляемых объектами классов, унаследованных от `GraphObject`. При этом на момент написания программы мы не знаем, сколько и каких именно объектов будет в сцене; так бывает, если описание сцены нужно получить из внешнего источника (например, прочитать из файла), либо если сцена генерируется во время исполнения программы (например, случайным образом, что часто используется во всевозможных программах-«скринсейверах»).

Допустим, в некий момент в программе нам всё же становится известно, сколько объектов будет содержать сцена. Это позволит использовать для хранения всей сцены динамически создаваемый массив указателей на объекты потомков `GraphObject`. Пусть, например, количество объектов сцены будет храниться в переменной `scene_length`, а указатель на сам массив мы назовём просто `scene`:

```
int scene_length;
GraphObject **scene;
```

Когда переменная `scene_length` тем или иным способом получит значение (например, оно будет прочитано из файла), можно будет завести массив:

```
scene = new GraphObject*[scene_length];
```

Теперь благодаря полиморфизму оказываются корректны, например, следующие присваивания (при условии, конечно, что `i` не превышает `scene_length`):

```
scene[i] = new Pixel(1.25, 15.75, 0xff0000);
// ...
scene[i] = new Circle(20.9, 7.25, 0x005500, 3.5);
// ...
scene[i] = new Square(55.0, 30.5, 0x008080, 10.0);
// ...
```

и тому подобные. У нас получился массив указателей типа `GraphObject*`, каждый из которых на самом деле указывает на некоторый объект *класса-потомка*. Обратим внимание на то, что нам может вовсе никогда больше не потребоваться знать, на объекты какого типа указывают конкретные указатели в нашем массиве. Вне зависимости от конкретных типов мы вполне можем перемещать объекты по экрану, гасить их и снова отрисовывать, ведь методы `Show`, `Hide` и `Move` доступны для класса `GraphObject`, а значит, могут быть вызваны по указателю типа `GraphObject*` без уточнения типа объекта. Точно таким же образом благодаря наличию виртуального деструктора можно уничтожить все объекты сцены, а потом и саму сцену:

```
for(int i=0; i<scene_length; i++)
    delete scene[i];
delete [] scene;
```

Подобные ситуации часто возникают в более-менее сложных программах. Поскольку конкретные методы, которые нужно вызывать, становятся известны только во время исполнения программы, такой вид полиморфизма называется ***динамическим полиморфизмом***; он становится возможным благодаря механизму виртуальных функций и является, в конечном итоге, их предназначением.

5.11. Приватные и защищённые деструкторы

Убирая деструктор из публичного доступа, мы можем получить своеобразные эффекты, иногда довольно полезные на практике. Если деструктор описать в секции `private:`, это будет означать, что уничтожение объекта нашего класса возможно только в его методах и дружественных функциях; деструктор в секции `protected:` добавит к этому ещё потомков нашего класса.

Объект класса, имеющего приватный деструктор, за пределами его методов и «друзей» *нельзя будет создать в виде простой (локальной или глобальной) переменной*, ведь при этом компилятор должен будет вставить в код вызов деструктора (для локальных переменных — при завершении функции, для глобальных — при завершении выполнения программы), но мы ему это делать запретили. Этот эффект иногда используют, чтобы создать тип объекта, всегда размещаемого в динамической памяти; обычно такие объекты при тех или иных обстоятельствах удаляют себя сами — например, можно предусмотреть для этого в классе специальный метод, что-то вроде

```
class A {
    ~A() {} // no destruction, dectructor is private
public:
    // ...
    void Disappear() { delete this; }
};
```

Отметим, что **приватный деструктор полностью исключает возможность наследования от такого класса**, если только заранее не описать всех наследников в качестве друзей. Напротив, защищённый (`protected`) деструктор явным образом указывает на то, что наш класс задуман как заготовка для создания наследников и что использование его самого по себе не предполагается; как показывает практика, это действует надёжнее, чем соответствующая фраза в документации.

Если все наследники нашего класса тоже будут вводить свои деструкторы в защищённой части, мы получим целую полиморфную иерархию классов, объекты которых могут существовать исключительно в динамической памяти.

Привести пример ситуации, в которой что-то подобное может потребоваться, не так просто: для этого нужно весьма подробно описать довольно сложную предметную область. Делать этого мы не будем, ограничившись замечанием, что автору книги такие ситуации встречались, и неоднократно; полезно иметь в виду сам факт существования подобных возможностей, чтобы суметь воспользоваться ими, когда ваша практическая деятельность подбросит вам подходящую задачу.

5.12. Перегрузка функций и сокрытие имён

Допустим, мы описали класс `A` и от него унаследовали класс `B`, и при этом в *обоих классах* есть поля или методы, названные одним и тем же идентификатором (например, `x`). Вообще-то так писать в большинстве случаев не надо (кроме случая переопределения виртуальной функции), но если вы всё-таки это написали, полезно будет помнить одно важное правило языка Си++: **введение поля или метода с именем `x` в порождённом классе *скрывает* любые поля или методы базовых классов, имеющие такое же имя**[31]. Если речь идёт в обоих случаях об имени поля или о функциях-методах с одинаковым профилем (т. е. одинаковым количеством и типами параметров), правило сокрытия оказывается достаточно очевидным. Очевидно оно и для случая, когда в базовом классе имеется функция с именем `x`, а в порождённом вводится поле `x`, или наоборот — ведь для имён полей в Си++ перегрузка не предусмотрена. Например:

```
class A { // ...
public:
    void f(int a, int b);
};
class B : public A {
    double f;  // метод f(int, int) теперь скрыт
};
```

Если теперь создать объект класса `B`, вызвать метод `f` мы с ходу не сможем:

[31] Мы не рассматриваем в нашей книге множественное наследование, но на всякий случай отметим, что если поля или методы с одинаковыми именами появились в двух базовых классах одного порождённого класса, то в таком порождённом классе сокрытию подвергнутся имена из *обоих* базовых классов.

```
B b;
b.f(2, 3);   // ОШИБКА!!! Метод f скрыт
```

Однако «скрыт» не значит «недоступен»; в классе `B` по-прежнему присутствует метод `f`, унаследованный от класса `A`, просто его вызов нужно выполнять с явным указанием области видимости:

```
b.A::f(2, 3);   // всё в порядке
```

Поначалу такая конструкция создаёт жутковатое впечатление, но это быстро пройдёт; в целом здесь всё вполне логично, ведь, как мы отмечали в §3.13, `A::f` — это не что иное как *имя функции*.

Наиболее неочевидным проявлением описанного правила становится то, что появившаяся в порождённом классе функция-метод с тем же именем, что и метод базового класса, **скрывает метод базового класса, даже если они различаются профилем**. Перегрузка функций нас в этом случае не спасает:

```
class A { // ...
public:
    void f(int a, int b);
};
class B : public A {
public:
    double f(const char *str);  // метод f(int, int) скрыт
};

B b;
double t = b.f("abrakadabra"); // всё в порядке
b.f(2, 3);                     // ОШИБКА!!!
b.A::f(2, 3);                  // всё в порядке
```

Всё это можно выразить одним простым правилом: **перегрузка имён функций действует только в рамках одной области видимости**, если же имена введены в различных (пусть и пересекающихся) областях видимости, принципы перегрузки на них не распространяются.

5.13. Вызов в обход механизма виртуальности

Явное указание области видимости (имени класса), использовавшееся в предыдущем параграфе для обращения к именам, которые компилятор от нас скрыл, имеет в Си++ ещё один эффект, довольно неожиданный на первый взгляд: *при вызове виртуального метода с явным указанием имени класса отключается механизм виртуальности*; функция-метод вызывается из того класса, имя которого

указано при её вызове, без оглядки на таблицу виртуальных методов. Рассмотрим для примера следующие классы:

```
class A {
public:
    virtual void f() { printf("first\n"); }
    void g() { f(); }    /* вызывается метод f в зависимости
                            от фактического типа объекта */
    void h() { A::f(); } /* всегда вызывается f из класса A */
};
class B : public A {
public:
    virtual void f() { printf("second\n"); }
};
```

Проиллюстрировать разницу между `f()` и `A::f()` поможет следующий фрагмент кода:

```
B b;
b.g();      /* печатается "second" */
b.h();      /* печатается "first" */
b.f();      /* печатается "second" */
b.A::f();   /* печатается "first" */
A *pa = &b;
pa->f();    /* печатается "second" */
pa->A::f(); /* печатается "first" */
```

Эта возможность требуется сравнительно редко, но иногда оказывается весьма полезной; наиболее частый вариант её использования — когда из «новой» версии виртуального метода, введённой классом-потомком, нужно вызвать «старую» версию, описанную для класса-предка.

5.14. Наследование как сужение множества

> Всякая селёдка — рыба, но не всякая рыба — селёдка.
>
> *А. Некрасов. Приключения капитана Врунгеля.*

Как уже говорилось, при описании объектно-ориентированного программирования можно использовать различные варианты терминологии. Так, термин «вызов метода объекта» эквивалентен термину «отправка сообщения объекту», просто эти термины относятся к разным терминологическим системам: о вызовах методов мы говорим при изучении практического применения ООП, тогда как передача

сообщения — термин, относящийся к теории ООП и к тем языкам программирования, которые полностью соответствуют этой теории (к таким языкам относится, например, Smalltalk).

Класс с теоретической точки зрения представляет собой *множество* объектов, удовлетворяющих определённым условиям. В этом смысле порождённый (наследуемый, или дочерний) класс представляет собой *подмножество*. В самом деле, согласно закону полиморфизма объект порождённого класса может быть использован в качестве объекта базового класса, то есть, попросту говоря, является одновременно и объектом порождённого, и объектом базового класса; в то же время объект базового класса вовсе не обязан быть объектом класса порождённого.

Можно прийти к тем же выводам и иначе. Наследование представляет собой *уточнение* свойств объекта, или, иначе говоря, переход от общего случая к частному. Ясно, что множество частных случаев есть подмножество множества случаев общих. Получается, что в терминах множеств наследование описывается отношением включения ($A \supseteq B$). Естественным следствием такого рассмотрения является термин ***подкласс*** (англ. *subclass*) для обозначения порождённого класса и термин ***надкласс*** или ***суперкласс*** (англ. *superclass*) — для обозначения базового.

Такая терминология часто порождает определённую путаницу. Дело тут в том, что объект порождённого класса (*подкласса*) мало того, что памяти занимает заведомо не меньше (а при добавлении новых полей — совершенно точно больше), нежели объект класса базового (*суперкласса*), но ещё и *содержит в себе объект базового класса*, т. е. объект суперкласса оказывается *подобъектом* объекта подкласса. Получается, что мы рассматриваем одновременно два отношения вложенности, и они мало того что разные, они оказываются *направлены противоположно*: объект базового класса вложен в объект порождённого класса (чисто технически), а вот сами классы (уже как множества объектов) «вложены», наоборот, порождённый в базовый.

Вся эта путаница обусловлена применением двух разных терминологических систем в одном месте. Когда речь идёт о суперклассах и подклассах — это значит, что используется теоретико-множественная терминология. Когда же речь заходит об используемой памяти и подобъектах — очевидно, что разговор идёт в терминах реализаторской (прагматичной) точки зрения. Мы уже встречались с этим дуализмом в §5.3 (см. комментарий на стр. 114). Обе терминологические системы активно используются и имеют право на существование; вы на практике можете столкнуться как с одним вариантом терминологии, так и с другим, а в некоторых случаях — и с их сме-

шением, как в вышеприведённом примере. Поэтому желательно понимать, что означают термины обеих систем.

5.15. Операции приведения типа

В процессе программирования часто приходится изменять тип выражения. Иногда это делается *неявно*, как, например, в случае сложения целочисленного значения со значением дробным (с плавающей точкой). В иных случаях (как, например, при изменении типа указателя) приходится явно указывать компилятору новый тип выражения. В языке Си это делалось с помощью ***операции преобразования типа***, записываемой как унарная операция, символ которой есть имя типа, заключённое в круглые скобки, как, например, в следующем выражении:

```
int x;
char *p = (char*)&x;
```

Здесь значение выражения `&x`, имеющее тип `int*`, приводится к типу `char*`. Операция приведения типа *опасна* в том смысле, что её применение позволяет при желании обойти любые ограничения, вводимые системой типизации, включая, например, запреты на запись в константные области памяти и даже защиту данных в классах. Необдуманное применение преобразования типов приводит к запутыванию программы и, в конечном счёте, к трудновыявляемым ошибкам.

Для снижения негативного эффекта операции приведения типов, а также для поддержки полиморфного программирования в языке Си++ вводятся четыре дополнительные операции, предназначенные для преобразования типа выражения. Эти операции имеют достаточно нетривиальный синтаксис: сначала записывается ключевое слово, задающее операцию (`static_cast`, `dynamic_cast`, `const_cast` или `reinterpret_cast`), затем в угловых скобках ставится имя нового типа и, наконец, в круглых скобках записывается само выражение, тип которого необходимо изменить, например:

```
Square *sp = static_cast<Square*>(scene[i]);
```

В отличие от операции приведения типов в языке Си, которая применялась для любых случаев «ручного» (явного) изменения типа, каждая из операций Си++ предназначена для своего случая. Так, операция `const_cast` позволяет снять или, наоборот, установить сколько угодно модификаторов `const`[32]; попытка сделать с её помощью любое другое изменение типа вызовет ошибку при компиляции:

[32] Также эта операция работает с модификатором `volatile`, который мы в нашем курсе не рассматриваем.

```
int *p;
const int *q;
const char *s;
// ...
q = p;  // можно без преобразования
p = q;  // ошибка! снятие const
p = const_cast<int*>(q); // правильно
p = const_cast<int*>(s); // ошибка!
```

Отметим, что наличие в языке операции `const_cast` не отменяет опасности такого преобразования. Реальная потребность в обходе константной защиты возникает крайне редко; прежде чем применять преобразование, подумайте, всё ли вы правильно делаете, не забыли ли вы, например, пометить словом `const` функцию-метод, не изменяющую состояние объекта, и т. п. **Для применения операции `const_cast` нужны очень веские причины, и сакраментальное «без неё не работает» такой причиной считать нельзя.** В некоторых программистских коллективах на каждое применение `const_cast` нужно получить личное разрешение руководителя разработки.

Операция `static_cast` предназначена для работы с наследуемыми объектами и позволяет преобразовать указатель или ссылку в направлении, противоположном закону полиморфизма, т. е. от базового класса к порождённому. Попытка произвести любое другое преобразование вызовет ошибку. Приведём примеры:

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C { /* ... */ };
// ...
    A *ap;
    B *bp;
    C *cp;
    // ...
    ap = bp; // можно без преобразования
    bp = ap; // ошибка!
    bp = static_cast<B*>(ap); // допустимо
    cp = static_cast<C*>(ap); // ошибка, A и C не родственны
```

Естественно, делать это следует только когда мы *действительно* уверены, что по данному адресу расположен объект именно того типа, к которому мы намерены преобразовывать; иное в большинстве случаев приведёт к ошибкам, причём часто к таким, на локализацию которых уходит много времени.

Операция `reinterpret_cast` позволяет произвести любое преобразование (чего угодно во что угодно), если только компилятор по-

нимает, как это сделать (в частности, преобразовать объекты структур разных типов друг к другу не получится, поскольку непонятно, как такое преобразование производить). Фактически эта операция эквивалентна операции языка Си, которая обозначается именем типа, взятым в круглые скобки. Рекомендуется, однако, применять именно `reinterpret_cast`, а не операцию Си, поскольку такие преобразования требуют особого внимания, а выражение с использованием `reinterpret_cast` лучше заметно в тексте программы, чем имя типа в скобках.

Несколько особое место занимает операция `dynamic_cast`. Три операции, которые мы рассмотрели выше, служат для управления системой контроля типов, т. е. для управления компилятором, и не порождают действий, осуществляемых во время исполнения программы[33]. Операция `dynamic_cast`, в отличие от остальных, предполагает проведение нетривиальной проверки *во время исполнения* программы. Подобно операции `static_cast`, операция `dynamic_cast` предназначена для преобразования адресов объектов в направлении, противоположном закону полиморфизма, т. е. от адреса предка к адресу потомка. В случае со `static_cast` ответственность за корректность такой операции возлагается на программиста: именно программист тем или иным способом должен проверить, что преобразуемый адрес (будь то указатель или ссылка) указывает на объект нужного типа. Если же применить `dynamic_cast`, то она сама проведёт проверку и в случае, если преобразование некорректно (то есть по заданному адресу в памяти не находится объект нужного типа), вернёт нулевой указатель. На момент написания программы в общем случае тип объекта неизвестен, так что проверка проводится во время исполнения, т. е. *динамически*, отсюда название операции.

Проверка типа производится на основании значения указателя на таблицу виртуальных функций; дело в том, что такая таблица уникальна для каждого класса, имеющего виртуальные методы, т. е. её адрес однозначно идентифицирует класс объекта. Как следствие, `dynamic_cast` может работать только с классами (или структурами), имеющими виртуальные функции. В некоторых источниках такие классы называют ***полиморфными***, что не совсем корректно: как мы видели, полиморфизм в определённом смысле работает и для классов, не имеющих виртуальных функций.

Отметим ещё один немаловажный момент. Обычно реализации `dynamic_cast` весьма неэффективны по времени исполнения — по-

[33]Кроме преобразования между целыми числами и числами с плавающей точкой, что требует неких действий; иногда компилятору приходится изменить и численное значение указателя, но в нашей книге не рассматриваются механизмы, порождающие такую необходимость.

просту говоря, работают очень медленно. Поэтому злоупотреблять этой операцией не следует. В действительности будет лучше вообще без неё обойтись.

Преобразование dynamic_cast умеет работать не только с указателями, но и со ссылками; при этом понятия «нулевой ссылки» в природе не существует, поэтому при отрицательном результате проверки такой dynamic_cast выбрасывает некое «стандартное исключение». Чтобы обработать его, придётся подключить заголовочный файл стандартной библиотеки, а дальше, как говорят, коготок увяз — всей птичке пропасть. В действительности нет *никаких* причин применять dynamic_cast к ссылкам: если у вас ссылка, а не указатель, примените к ней операцию взятия адреса, превратив её тем самым в простой указатель, выполните dynamic_cast, проверьте результат на неравенство нулю и после чего разыменуйте его обратно.

5.16. Иерархии исключений

При обсуждении преобразований типов выражений в обработчиках исключительных ситуаций (см. стр. 107) мы отметили, что одним из наиболее важных видов преобразования является преобразование по закону полиморфизма, однако подробное обсуждение этого отложили, поскольку на тот момент ещё не было введено наследование.

Возвращаясь к этому вопросу, заметим, что третий и последний вид допустимых преобразований от типа выражения в операторе `throw` к типу, указанному в заголовке `catch` — это преобразование адреса (т. е. указателя или ссылки) объекта-потомка к соответствующему адресному типу объекта-предка. Если мы опишем два класса, унаследовав один от другого:

```
class A { /* ... */ };
class B : public A { /* ... */ };
```

— то обработчик вида

```
catch(const A& ex) { /* ... */ }
```

сможет ловить исключения *обоих* типов, т. е. результат как оператора «`throw A(...);`», так и «`throw B(...);`».

Это свойство используется для создания *иерархий исключительных ситуаций*. Например, мы можем поделить все ошибки, возникающие в какой-либо программе или библиотеке, на следующие категории:

- ошибки, возникающие по вине пользователя:
  - синтаксические ошибки при вводе (например, буквы там, где ожидается число);

  - неправильно указано имя файла;
  - неправильно введённый пароль;
  - недопустимая комбинация требований (например, одновременное требование упорядочивания по возрастанию и по убыванию);

- ошибки, обусловленные средой выполнения:

  - переполнение диска;
  - отсутствие файлов, необходимых для работы;
  - прочие ошибки операций ввода-вывода;
  - недостаток оперативной памяти;
  - ошибки при работе с сетью;
  - и т. п.

- ситуации, свидетельствующие об ошибке в самой программе и требующие её исправления (например, переменная принимает значение, которое вроде бы принимать не должна).

Опишем теперь класс `Error`, соответствующий понятию «любая ошибка». В полном соответствии с принципами объектно-ориентированного программирования перейдём от общего к частному, унаследовав от класса `Error` подклассы `UserError`, `ExternalError` и `Bug` для обозначения соответственно пользовательских ошибок, внешних ошибок и ошибок в программе. От класса `UserError`, в свою очередь, унаследуем классы `IncorrectInput`, `WrongFileName`, `IncorrectPassword` и т. д.

После этого обработчик вида

```
catch(const IncorrectPassword& ex) { /* ... */ }
```

будет обрабатывать только исключения, связанные с неправильным паролем, тогда как обработчик вида

```
catch(const UserError& ex) { /* ... */ }
```

будет реагировать на любые ошибки пользователя, что же касается обработчика

```
catch(const Error& ex) { /* ... */ }
```

то он сможет «поймать» вообще любое исключительное событие из нашей иерархии. Подчеркнём, что **такое преобразование работает только для адресов**, а не для объектов как таковых; именно поэтому мы в нашем примере в заголовках `catch` использовали ссылки.

6. Шаблоны

В программировании часто возникают ситуации, в которых простейшим и очевидным решением оказывается написание нескольких почти одинаковых (а иногда и совсем одинаковых) фрагментов кода. Известно, что в таких случаях пойти «очевидным» путём — идея крайне неудачная, ведь тексты программ приходится изменять, исправлять ошибки, добавлять новые возможности, и если некий фрагмент кода будет существовать более чем в одном экземпляре, вносить изменения придётся синхронно в каждый из экземпляров. Практика показывает, что рано или поздно мы про какой-нибудь из экземпляров забудем, исправив все кроме него; впрочем, даже если бы не это, механически дублировать одни и те же правки в нескольких местах программы оказывается делом утомительным и неприятным.

Если фрагменты, которые хочется написать для быстрого решения возникшей задачи, различаются только некоторыми *значениями* (или не различаются вовсе), бороться с дублированием кода легко: достаточно вынести «сомнительный» фрагмент в отдельную функцию, и вместо нескольких почти одинаковых кусков кода в нашей программе появятся вызовы этой функции; различаться они будут значениями параметров, причём заметить отличия при чтении такого кода будет, разумеется, гораздо проще, чем если бы приходилось каждый раз сверять большие фрагменты. Аналогичным образом обстоят дела при различии по именам переменных: мы можем в таком случае написать функцию, в которую будет передаваться адрес соответствующей переменной.

Но что делать, если нам потребовались почти одинаковые фрагменты, различия между которыми не сводятся к *значениям выражений*? Самый простой и часто встречающийся пример такой ситуации — различие *типов* переменных и выражений; как поступить, если нам потребовалось выполнить одни и те же действия, но в одном случае — над переменными типа `int`, а в другом — над переменными типа `double`? В языке Си в таких ситуациях приходится прибегать к определению макросов, причём часто — многострочных. Каждый программист, написавший хотя бы один многострочный макрос на Си, знает, какое это утомительное и неблагодарное занятие. Если в определение макроса вкралась ошибка, найти её будет непросто, причём даже не сразу становится понятно, что ошибка кроется где-то в макросе, поскольку компилятор в сообщении об ошибке указывает не на тело макроса, а на то место, где макрос вызван. Вообще, макропроцессор языка Си, из соображений совместимости включённый и в Си++, представляет собой средство опасное в применении и крайне неудобное, что многократно отмечают многие авторы, вклю-

чая Страуструпа. Однако вред от дублирования кода оказывается ещё больше, так что при работе на Си в ряде случаев иного выхода не остаётся.

В языке Си++ из большинства таких положений можно выйти, используя ***шаблоны***, которые делятся на шаблоны функций и шаблоны классов. Общая идея тех и других состоит в том, что мы пишем как бы заготовку для функции или класса, которая при конкретизации некоторых параметров может превратиться в настоящую функцию или, соответственно, класс. В качестве параметров шаблона чаще всего выступают имена типов выражений, но это могут быть и значения целого типа, а в некоторых специфических случаях и адресные выражения. Разумеется, один шаблон можно использовать для создания нескольких разных функций или классов, отличающихся друг от друга только значениями некоторых параметров. Важно понимать, что **шаблон — это ещё не код, это лишь заготовка для кода**. Вполне допустимо восприятие шаблонов как этаких «умных макросов» — как минимум в том смысле, что это фрагменты текста программы, из которых неким преобразованием (с подстановкой параметров) получаются другие фрагменты текста программы, которые уже компилируются как обычно. Надо сказать, что Страуструп тщательно отрицает связь шаблонов с макросами — но, видимо, он под макросами понимает только то странное недоделанное нечто, которое мы имеем в чистом Си, тогда как в действительности макропроцессирование как явление намного сложнее.

Можно считать, что **шаблоны представляют собой ещё один вид полиморфизма** — так называемый ***параметрический полиморфизм***. Поскольку этот вид полиморфизма реализуется на стадии компиляции, его следует рассматривать как частный случай статического полиморфизма, и, как и другие случаи статического полиморфизма, к ООП он никакого отношения не имеет, что, конечно же, не делает его менее важным.

6.1. Шаблоны функций

Рассмотрим для примера функцию, сортирующую массив целых чисел методом «пузырька»[34]:

```
void sort_int(int *array, int len)
{
    for(int start=0; ; start++) {
        bool done = true;
```

[34]Конечно, этот метод неэффективен для массивов заметного размера, но для примера он вполне подойдёт, и к тому же на массивах из десятка элементов ничего эффективнее простого «пузырька» пока что не придумано.

```
        for(int i=len-2; i>=start; i--)
            if(array[i+1] < array[i]) {
                int tmp = array[i];
                array[i] = array[i+1];
                array[i+1] = tmp;
                done = false;
            }
        if(done)
            break;
    }
}
```

Пусть теперь нам потребовалась такая же функция для сортировки массива из чисел типа `double`. Использовать имеющуюся функцию мы, понятное дело, не сможем: числа с плавающей точкой совершенно иначе сравниваются и имеют другой размер. Если же мы всё-таки напишем функцию `sort_double`, она будет отличаться от `sort_int` всего в двух местах: в типе параметра `array` и в типе временной переменной `tmp`, больше нигде. Поскольку дублировать код таким образом совершенно недопустимо, приходится как-то выкручиваться. В языке Си для этого пришлось бы создать с помощью директивы `#define` многострочный макрос с одним параметром, задающим как раз тип элементов сортируемого массива и, соответственно, тип переменной `tmp`.

В Си++ проблемы, подобные вышеописанной, можно решить с помощью ***шаблонов***. Как уже говорилось, шаблон — это своего рода «заготовка» для функции; сам по себе шаблон функцией не является, но может быть в неё превращён, если указать значения параметров. В данном случе параметром шаблона будет тип элементов массива. Опишем этот шаблон:

```
template <class T>
void sort(T *array, int len)
{
    for(int start=0; ; start++) {
        bool done = true;
        for(int i=len-2; i>=start; i--)
            if(array[i+1] < array[i]) {
                T tmp = array[i];
                array[i] = array[i+1];
                array[i+1] = tmp;
                done = false;
            }
        if(done)
            break;
    }
```

```
}
```

Поясним, что ключевое слово `template` указывает компилятору, что далее последует шаблон. Затем в угловых скобках перечисляются ***параметры шаблона***, причём слово `class` означает на самом деле *произвольный тип*, а не только класс. Таким образом, наш шаблон имеет один параметр (с именем `T`), в качестве которого ожидается указание некоего типа. Далее следует тело функции, в котором идентификатор `T` используется в качестве типа. Ясно, что такое тело невозможно откомпилировать в объектный код, поскольку неизвестно, какой тип обозначается именем `T`, как выполнять индексирование в массиве из таких элементов (ведь размер элемента тоже неизвестен), как выполнять присваивание и сравнение. Но компилировать написанное мы и не собираемся — ведь мы же договорились, что пишем не функцию, а лишь заготовку для неё. Эта заготовка (шаблон) имеет имя `sort`.

Чтобы теперь заставить компилятор построить функцию на основе шаблона, достаточно указать конкретный тип, который будет использоваться вместо `T`. Это делается тоже с помощью угловых скобок; выражение `sort<int>` обозначает функцию для сортировки целочисленных массивов, полученную из шаблона `sort` с использованием типа `int` в качестве значения параметра `T`:

```
int a[30];
// ...
sort<int>(a, 30);
```

Таким же точно образом `sort<double>` обозначает функцию для сортировки массивов из чисел типа `double`. Более того, шаблон годится для сортировки массивов из элементов произвольного типа, нужно только, чтобы для этого типа существовал конструктор по умолчанию, были определены операция `<`, используемая в нашем шаблоне для сравнения, и операция присваивания.

Получение функции из шаблона называется ***инстанциацией*** шаблона. Компилятор инстанциирует каждую функцию только один раз, т. е. если в нашем модуле трижды встретится вызов функции `sort<int>`, код для неё будет сгенерирован лишь единожды.

Интересно, что в ряде случаев инстанциированную функцию можно вызвать, не указывая параметры шаблона; в частности, вызов

```
sort(a, 30);
```

будет вполне корректным. По типу параметра `a` компилятор «догадается», что имеется в виду именно `sort<int>`. Эта возможность называется *автоматическим выводом аргументов шаблона*; забегая вперёд, отметим, что для шаблонов классов такой возможности нет.

6.2. Шаблоны классов

Чтобы понять, зачем нужны шаблоны классов, вспомним пример, который мы приводили в §3.25. Напомним, что мы рассматривали *разреженный массив целых чисел*. При этом мы написали класс `SparseArrayInt` и подчинённый ему класс `Interm`. Если нам потребуется теперь разреженный массив чисел другого типа или, скажем, разреженный массив символьных строк, то код класса `SparseArrayInt` придётся полностью дублировать с минимальными изменениями, что, как мы уже говорили, недопустимо. Существенно правильнее будет превратить существующий код для целочисленного массива в шаблон массива произвольного типа. Перепишем заголовок класса, приведённый на стр. 86, в виде шаблона:

```
template <class T>
class SparseArray {
    struct Item {
        int index;
        T value;
        Item *next;
    };
    Item *first;
public:
    SparseArray() : first(0) {}
    ~SparseArray();
    class Interm {
        friend class SparseArray<T>;
        SparseArray<T> *master;
        int index;
        Interm(SparseArray<T> *a_master, int ind)
            : master(a_master), index(ind) {}
        T& Provide(int idx);
        void Remove(int idx);
    public:
        operator T();
        T operator=(int x);
        T operator+=(int x);
        T operator++();
        T operator++(int);
    };
    friend class Interm;

    Interm operator[](int idx)
        { return Interm(this, idx); }
private:
    SparseArray(const SparseArray<T>&) {}
    void operator=(const SparseArray<T>&) {}
```

```
};
```

Обратите внимание, что везде к слову SparseArray мы добавляем параметр <T>, за исключением имени класса и имён конструкторов и деструкторов. Дело тут в том, что относительно самого класса (т. е. шаблона) компилятор и так знает, что описывается шаблон, и то же самое можно сказать про конструкторы и деструкторы; когда же речь идёт о типах параметров в функциях, о типах указателей, и, наконец, о конкретном дружественном классе — то теоретически в качестве таковых могут выступать любые классы, в том числе созданные из этого же шаблона, но с параметром, отличным от T. Поэтому тип приходится указывать полностью. Функции-методы нашего шаблона класса тоже придётся описывать как шаблоны. Например, описание деструктора примет следующий вид:

```
template <class T>
SparseArray<T>::~SparseArray()
{
    while(first) {
        Item *tmp = first;
        first = first->next;
        delete tmp;
    }
}
```

а операцию присваивания из класса Interm нужно будет описать так:

```
template <class T>
T SparseArray<T>::Interm::operator=(T x)
{
    if(x == 0)
        Remove(index);
    else
        Provide(index) = x;
    return x;
}
```

Обратите внимание, что при раскрытии области видимости нам приходится в явном виде указывать, какой *класс* (а не шаблон класса) мы имеем в виду, то есть писать SparseArray<T>, а не просто SparseArray. Это и понятно: соответствующий метод будет присутствовать в каждом классе, построенном по нашему шаблону, но ведь это будут *разные* методы (хотя и построенные по одному и тому же *шаблону* метода). *Вообще, везде, где по смыслу предполагается имя типа, мы должны при использовании шаблона указать значения для параметров шаблона, чтобы получить тип*; имя шаблона

класса без угловых скобок и параметров используется только в трёх случаях: в начале описания шаблона (когда, собственно говоря, *задаётся* имя шаблона), при описании конструктора и при описании деструктора. Описать шаблоны для остальных методов `SparseArray` предоставим читателю самостоятельно в качестве упражнения.

6.3. Специализация шаблонов

Язык Си++ позволяет задать свой (отличный от общего) вид шаблона для частных случаев его параметров — попросту говоря, правило вида «в таком случае генерировать вот так, во всех остальных случаях — в соответствии с общим шаблоном».

Допустим, нам захотелось использовать шаблон функции сортировки, приведённый на стр. 143, для сортировки массива указателей на строки (указателей типа `char*`), чтобы расположить соответствующие строки в лексикографическом («алфавитном») порядке. Проблема в том, что шаблон в той его версии, которая нами рассматривалась, использует для сравнения элементов операцию «<», а эта операция хотя и определена для указателей, но никакого отношения к алфавитному порядку строк не имеет. Решение, однако, оказывается достаточно простым. Для начала заменим знак «меньше» на вызов функции, которую назовём, например, `sort_less`:

```
template <class T>
void sort(T *array, int len)
{
    for(int start=0; ; start++) {
        bool done = true;
        for(int i=len-2; i>=start; i--)
            if(sort_less(array[i+1], array[i])) {
                T tmp = array[i];
                array[i] = array[i+1];
                array[i+1] = tmp;
                done = false;
            }
        if(done)
            break;
    }
}
```

Саму функцию `sort_less` опишем тоже с помощью шаблона:

```
template <class T>
bool sort_less(T a, T b)
{
```

```
    return a < b;
}
```

В результате для всех типов, для которых имеется операция «меньше», шаблон `sort` будет продолжать работать так же, как работал до переделки; функцию `sort_less` компилятор будет каждый раз генерировать автоматически как обычное сравнение. Нам осталось только предусмотреть особый случай для сравнения элементов типа `char*`, и тут мы как раз и прибегнем к ***явной специализации***.

Описание случая явной специализации начинается, как обычно, со слова `template`, но угловые скобки после него оставляются пустыми, чтобы показать, что это, с одной стороны, шаблон, но, с другой стороны, *этот* шаблон пишется для частного случая, не зависящего от дополнительных параметров. Далее записывается имя шаблонной функции, причём в некоторых случаях[35] требуется вместе с именем указать все параметры шаблона (в нашем случае — один параметр). Всё вместе выглядит так:

```
template<>
bool sort_less<const char*>(const char *a, const char *b)
{
    return strcmp(a, b) < 0;
}
```

Для компилятора это означает примерно следующее: «будь любезен, шаблон `sort_less` для случая `sort_less<const char*>` обрабатывай в соответствии с вот этим текстом шаблона, а не каким-либо иным».

Шаблон класса тоже можно подвергнуть специализации. Например, если нам потребуется разреженный массив элементов типа `bool`, мы можем, конечно, воспользоваться шаблоном из §6.2, но, как легко заметить, такая реализация окажется несколько странной: ведь тип `bool` имеет всего два значения, а для разреженного массива это значит, что, если элемент вообще хранится в объекте, то этот элемент имеет значение `true` (поскольку если бы он имел значение `false`, он бы в объекте не хранился: массив-то разреженный). Следовательно, элементы списка `Item` для этого случая логично бы сделать состоящими из двух, а не из трёх полей — хранить номер элемента и указатель на следующий элемент, а значение не хранить, поскольку оно и так известно. Больше того, множество целых чисел нам может показаться удобнее хранить в массиве, а не в списке, и т. д. Механизм специализаций позволяет создать отдельное описание шаблона `SparseArray` для случая `T == bool`:

[35] В нашем случае это не обязательно, поскольку параметр шаблона выводится из типа параметров функции, так что параметр шаблона в имени функции можно опустить; но в случае, когда параметр шаблона с типами параметров функции не совпадает, указать его всё же придётся.

```
template <>
class SparseArray<bool> {
    // ... реализация булевского разреженного массива ...
};
```

Для шаблонов **классов** язык Си++ предусматривает ***частичную специализацию***, при которой специализирующий вариант шаблона сам по себе тоже зависит от параметров. Такой специализатор начинается со слова `template`, после которого следует *непустой* список параметров шаблона; при этом параметры в заголовке шаблона, вообще говоря, могут не совпадать (или совпадать не полностью) с параметрами описываемого шаблона; фактические параметры шаблона указываются в угловых скобках после его имени. В простейшем случае о частичной специализации речь идёт, если задаётся конкретное значение одного или нескольких, но *не всех* параметров шаблона. Так, если у нас имеется шаблон `Cls`, зависящий от двух параметров:

```
template <class A, class B>
class Cls {
    /* ... */
};
```

то можно задать специализированный вариант, например, для случая, когда параметр `B` есть тип `int`:

```
template <class X>
class Cls<X, int> {
    /* ... */
};
```

Более сложный случай частичной специализации возникает, когда в заголовке шаблона указывается некий тип (`class T`), а в описываемом типе используется указатель или ссылка на `T`, например:

```
template <class Z>
class Foo {
    /* общая реализация шаблона Foo */
};

template <class T>
class Foo<T*> {
    /* специальная реализация Foo для указателей */
};
```

В этом случае мы инструктируем компилятор, что шаблон `Foo` следует инстанциировать специальным способом, если в качестве его параметра задан тип-указатель, и обычным способом — если заданный параметр указателем не является.

Отметим, что **для шаблонов функций** (в отличие от шаблонов классов) **частичная специализация запрещена**, поскольку в сочетании с перегрузкой приводит к нежелательным последствиям. Ещё один немаловажный момент состоит в том, что в тексте программы специализированный (полностью или частично) вариант шаблона всегда должен находиться **после** общего.

6.4. Константы в роли параметров шаблона

Параметрами шаблонов могут быть не только типы, но и обычные константные выражения, чаще всего целочисленные. Многие авторы иллюстрируют эту возможность на примере задания границ массивов; так, мы могли бы в нашем шаблоне для сортировки массивов сделать размер массива параметром *шаблона*, а не функции:

```
template <class T, int len>
void sort(T *array)
{
    /* реализация сортировки */
}
```

вот только не совсем понятно, зачем это делать. Если, скажем, у нас имеются два целочисленных массива разного размера (например, 10 элементов и 15) и мы применим для их сортировки такой вот шаблон, то компилятор сгенерирует *две разные* функции: `sort<int,10>` и `sort<int,15>`, и эти функции будут представлять собой абсолютно одинаковые фрагменты исполняемого кода, отличающиеся только одной константой, тогда как если применять шаблон в том виде, в котором мы его писали ранее, функция будет *одна* (`sort<int>`).

Существенно интереснее (хотя и намного сложнее) будет другой пример. Зададимся целью создать объект, реализующий N-мерные массивы для произвольных значений N, которые к тому же будут автоматически изменять свои размеры по любому из измерений, подобно тому, как изменял свою длину массив `IntArray`, рассмотренный нами в §3.20. Поскольку нам могут понадобиться N-мерные массивы элементов разных типов, реализуем нашу задумку в виде шаблона класса с тремя параметрами: первый параметр будет задавать тип элементов массива, второй параметр (имеющий тип, совпадающий с типом элемента) укажет, какое исходное значение присваивать элементам массива при их создании, и, наконец, третий параметр — выражение типа `int` — будет задавать *количество измерений* массива. Шаблон будет описывать операцию индексирования, которая возвращает ссылку на объект, заданный таким же шаблоном с теми же фактическими параметрами, только с уменьшенным на единицу

количеством измерений. Ясно, что такой вариант не проходит для одномерного массива (не может же он, в самом деле, возвращать ноль-мерный массив, ведь такого не бывает), но здесь нам поможет частичная специализация: для случая количества измерений, равного одному, мы опишем специальный случай нашего шаблона, в котором операция индексирования будет возвращать ссылку на простой элемент (тип которого задан первым параметром шаблона).

С реализацией одномерного случая всё более-менее понятно, достаточно взять уже знакомый нам `IntArray` и переделать его в шаблон. Что касается всех многомерных случаев, то самый простой способ их реализации — взять всё тот же динамически расширяемый массив, элементами которого на сей раз будут выступать *указатели* на объекты, представляющие массив на единицу меньшей размерности: двумерный массив будет реализован как массив указателей на объекты одномерных массивов, трёхмерный — как массив указателей на двумерные, и т. д. Для экономии памяти указатели исходно будут нулевыми; соответствующие объекты $(N-1)$-мерных массивов будут создаваться только при первом обращении к ним.

Итак, в основе обоих случаев нашего массива (как одномерного, так и двумерного) оказывается одномерный массив, автоматически расширяющийся при необходимости. Дважды реализовывать эту сущность совершенно ни к чему, тем более что в нашем распоряжении имеется механизм шаблонов. Начнём с реализации динамически расширяющегося массива с произвольным типом элементов:

```
template <class T>
class Array {
    T *p;
    T init;
    unsigned int size;
public:
    Array(T in) : p(0), init(in), size(0) {}
    ~Array() { if(p) delete[] p; }
    T& operator[](unsigned int idx) {
        if(idx >= size) Resize(idx);
        return p[idx];
    }
    int Size() const { return size; }
private:
    void Resize(unsigned int required_index) {
        unsigned int new_size = size==0 ? 8 : size;
        while(new_size <= required_index)
            new_size *= 2;
        T *new_array = new T[new_size];
        for(unsigned int i = 0; i < new_size; i++)
```

```
            new_array[i] = i < size ? p[i] : init;
        if(p) delete[] p;
        p = new_array;
        size = new_size;
    }
       // запретим копирование и присваивание
    void operator=(const Array<T>& ref) {}
    Array(const Array<T>& ref) {}
};
```

Подробно комментировать этот шаблон мы не будем, поскольку реализация почти дословно повторяет реализацию класса `IntArray`, остановимся только на отличиях. Во-первых, мы добавили здесь поле `init`, значение которого, задаваемое в конструкторе, присваивается новым (не существовавшим ранее) элементам массива при его расширении. Во-вторых, размер массива исходно принимается нулевой, а не 16, как в `IntArray`. Ну и, конечно, класс преобразован в шаблон, что тоже существенно. Имея этот класс, мы можем легко реализовать общий случай нашего многомерного массива. Назовём этот шаблон `MultiMatrix` и напомним, что он должен работать для любого количества измерений, бóльшего единицы, а одномерный случай мы потом опишем путём специализации.

```
template <class T, T init_val, int dim>
class MultiMatrix {
    Array<MultiMatrix<T, init_val, dim-1>*> arr;
public:
    MultiMatrix() : arr(0) {}
    ~MultiMatrix() {
        for(int i=0; i < arr.Size(); i++)
            if(arr[i]) delete arr[i];
    }
    MultiMatrix<T, init_val, dim-1>&
    operator[](unsigned int idx) {
        if(!arr[idx])
            arr[idx] = new MultiMatrix<T, init_val, dim-1>;
        return *arr[idx];
    }
};
```

Мы воспользовались здесь шаблоном `Array` для построения массива *указателей* на элементы типа «$(N-1)$-мерный массив»; сам этот тип задаётся через сам же шаблон `MultiMatrix`, причём с теми же параметрами, что и у исходного шаблона, отличается лишь третий параметр, задающий количество измерений. Получается, что описание шаблона `MultiMatrix` в определённом смысле рекурсивно.

Объект массива указателей мы назвали `arr`. Напомним, что классы, построенные по шаблону `Array`, принимают в качестве параметра конструктора начальное значение, исходно присваиваемое элементам массива; в данном случае этот параметр — константа `0`, что означает нулевой указатель. Исходно все элементы массива нулевые, и только при первом обращении к соответствующему элементу создаётся объект $(N-1)$-мерного массива (см. оператор `if` в теле операции индексирования).

Наконец, опишем базисный случай — одномерный массив, то есть специализированный вариант шаблона `MultiMatrix` для случая, когда третий параметр шаблона равен единице. Это можно сделать гораздо проще, поскольку новый класс фактически представляет собой обёртку вокруг соответствующего объекта `Array` (он, как и раньше, называется `arr`). Сам объект класса `Array` можно сделать приватным полем; обёртка введёт операцию индексирования, реализованную через такую же операцию объекта-массива, и задаст аргумент его конструктору, больше от неё ничего не требуется:

```
template <class T, T init_val>
class MultiMatrix<T, init_val, 1> {
    Array<T> arr;
public:
    MultiMatrix() : arr(init_val) {}
    T& operator[](unsigned int idx) {
        return arr[idx];
    }
};
```

Можно поступить ещё лучше — *унаследовать* наш частный случай от `Array<T>`, тогда даже операцию индексирования описывать не придётся:

```
template <class T, T init_val>
class MultiMatrix<T, init_val, 1> : public Array<T> {
public:
    MultiMatrix() : Array<T>(init_val) {}
};
```

Полученный шаблон можно проверить, например, с помощью такой функции `main`:

```
int main()
{
    MultiMatrix<int, -1, 5> mm;
    mm[3][4][5][2][7] = 193;
    mm[2][2][2][2][2] = 251;
```

```
    printf("%d %d %d %d\n",
        mm[3][4][5][2][7], mm[2][2][2][2][2],
        mm[0][1][2][3][4], mm[1][2][3][2][1]);
    return 0;
}
```

Программа напечатает «`193 251 -1 -1`».

Параметром шаблона может быть константа любого целого типа, известная на момент компиляции; но целыми числами возможности параметров шаблонов не исчерпываются. В роли параметра шаблона могут выступать адресные выражения (указатели), но здесь действует целый ряд ограничений. Фактическим значением адресного параметра шаблона может быть только адрес глобальной переменной или функции[36], причём только в форме `&var`, `&f` или просто `f`, где `var` — имя глобальной переменной, `f` — имя функции. Недопустимо использовать сложные адресные выражения, нельзя использовать адреса локальных переменных (что и понятно, ведь их невозможно определить во время компиляции, когда происходит инстанциация шаблонов), нельзя использовать строковые литералы (строки в двойных кавычках; это тоже можно понять, ведь у двух одинаковых строковых литералов совершенно не обязан совпадать адрес, а если они появились в разных единицах трансляции — то их адреса наверняка будут разными). Впрочем, в реальной жизни очень редко применяется что-либо кроме типов; даже целочисленные параметры шаблонов — это скорее экзотика.

[36] Ещё можно использовать так называемый указатель на член класса, но мы эту сущность не рассматриваем.

Что дальше (вместо послесловия)

Это послесловие предназначено, пожалуй, только для тех, кто планирует стать профессиональным программистом.

Мы ограничились изучением языка Си++ как такового, полностью проигнорировав его стандартную библиотеку, а сам язык рассматривали в том виде, в котором он существовал до появления стандартов, и даже классические возможности Си++ рассмотрели не все: за рамками книги остались множественное наследование, пространства имён, указатели на методы классов и многое другое.

Несомненно, при приёме на работу едва ли не в любую коммерческую организацию от вас на собеседовании потребуют знания STL. Программирование на Си++ без STL возможно и, более того, позволяет получать более эффективные и качественные программы, но вам вряд ли удастся убедить вашего работодателя (будущего начальника) отказаться от STL, поскольку он, скорее всего, относится к числу программистов, изучавших Си++ после 1999 года. Более того, с неплохой вероятностью от вас потребуется владение возможностями Си++, которые мы оставили за кадром, в том числе сомнительными новшествами, которые внесли в язык пресловутые стандартизационные комитеты.

Если такое собеседование предстоит вам в течение ближайших двух недель, то у вас, очевидно, нет иного выхода, кроме как немедленно обрести недостающие знания. Для изучения STL можно воспользоваться практически любой книгой по Си++ (есть даже книги, специально посвящённые STL) и, конечно, компьютером, поскольку без практических навыков никакие знания в области программирования не будут ничего стоить. Про возможности из «новых стандартов» тоже не написал разве что ленивый; здесь стоит отметить, что чем новее стандарт, тем меньше вероятность, что от вас потребуют владения возможностями из него, так что С++11, возможно, стоит посмотреть подробно, С++14 пробежать «по верхам», а С++17 вообще проигнорировать.

Если же немедленно устраиваться на работу программистом в ваши планы пока не входит, я настоятельно советую не браться за изучение STL и не использовать его по крайней мере до тех пор, пока вы не напишете на Си++ одну-две практически применимые программы (под таковыми можно понимать программы, которые реально использует кто-то кроме их автора). После этого можно будет сказать, что основы языка Си++ вы знаете и к изучению STL подойдёте осознанно и с достаточным для этого опытом. Что касается новшеств от стандартизационного комитета, то ни одно из них не сделало язык лучше — но чтобы это оценить, опять-таки, нужен

опыт работы с классическими средствами Си++ и вообще опыт программирования. Пока у вас есть такая возможность, не обращайте внимания на «стандарты». Чем позднее вам придётся столкнуться с безумными стандартами Си++ и его кошмарной стандартной библиотекой, тем целостнее будет ваше восприятие происходящего.

В качестве дополнительного чтения можно посоветовать книгу Джеффа Элджера, которая в оригинале называется «C++ for real programmers», а в русском переводе вышла в серии «Библиотека программиста» под заголовком «Си++» [3]; куда при переводе делись остальные слова из оригинального названия — вопрос к издателям. Если вам удастся понять эту книгу, вас останется только поздравить: вы действительно поняли, что такое Си++. Стоит отметить, что Элджер ни словом не обмолвился про STL, хотя к тому времени STL уже существовал.

Список литературы

[1] B. Stroustrup. *The C++ Programming Language. Second edition.*. Addison-Wesley, Reading, Massachusets, 1991. Русский перевод: Бьярн Страуструп, Язык программирования C++, вторая редакция. В двух частях. Киев, «ДиаСофт», 1993.

[2] B. Stroustrup. *The design and evolution of C++.* Addison-Wesley, Reading, Massachusets, 1994. Русский перевод: Бьерн Страуструп, Дизайн и эволюция языка C++, М.: ДМК, 2000.

[3] J. Alger. *C++ for real programmers.* AP Professional, Boston, 1998. Русский перевод: Джефф Элджер, C++: библиотека программиста. СПб., Питер, 2001.

А.В.СТОЛЯРОВ

ВВЕДЕНИЕ В ЯЗЫК СИ++