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***.