Классика программирования

Бьерн Страуструп

Дизайн и эволюция языка C++

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

ББК 32.973.26-018.1
С80

Страуструп Б.
С80 Дизайн и эволюция C++: Пер. с англ. – М.: ДМК Пресс, 2016. – 446 с.: ил. (Серия «Для программистов»).

ISBN 978-5-97060-419-9

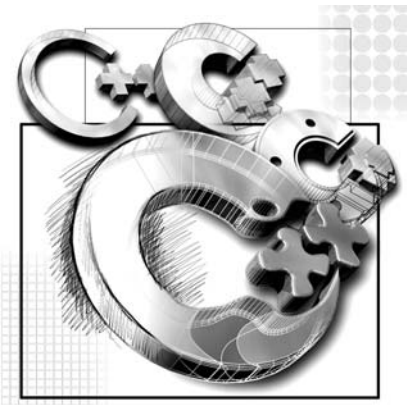
В книге, написанной создателем языка C++ Бьерном Страуструпом, представлено описание процесса проектирования и разработки языка программирования C++.

Здесь изложены цели, принципы и практические ограничения, наложившие отпечаток на структуру и облик C++, обсужден дизайн недавно добавленных в язык средств: шаблонов, исключений, идентификации типа во время исполнения и пространств имен. Автор анализирует решения, принятые в ходе работы над языком, и демонстрирует, как правильно применять «реальный объектно-ориентированный язык программирования».

Книга удобно организована, поучительна, написана с юмором. Описание ключевых идей даст начинающему пользователю ту основу, на которой позже он выстроит свое понимание всех деталей языка. Опытный программист найдет здесь обсуждение принципиальных вопросов проектирования, что позволит ему лучше понять язык, с которым он работает.

ISBN 978-0-201-54330-8 (англ.)
ISBN 978-5-97060-419-9 (рус.)

Содержание



Часть I

В части I описываются истоки C++ и его эволюция от C with Classes, а также принципы, определявшие развитие языка на протяжении этого периода и в последующее время. Здесь приводится хронология событий после выхода версии 1.0 и рассказывается о стандартизации. Также обсуждаются сферы применения C++ и другие вопросы.

Глава 1. Предыстория C++

Давным-давно, когда правило Зло!

Кристен Найгаард

1.1. Язык Simula и распределенные системы

Предыстория C++ – за пару лет до того, как мне пришла в голову мысль добавить к C некоторые возможности из Simula, – важна потому, что в это время выкристаллизовались критерии, позднее положенные в основу C++. Я работал над докторской диссертацией в лаборатории вычислительной техники Кембриджского университета в Англии. Тема – изучение альтернативных способов построения распределенных систем. Работа велась на базе Кембриджского компьютера CAP с его экспериментальной и постоянно эволюционирующей операционной системой [Wilkes, 1979]. Подробности этой работы и ее результаты [Stroustrup, 1979] к C++ отношения не имеют. Существенными оказались интерес к построению программного обеспечения из четко определенных модулей и тот факт, что основным средством для экспериментирования стал большой симулятор, который я написал для моделирования работы программ в распределенной системе.

Первая версия симулятора была написана на Simula [Birtwistle, 1979] и работала на компьютере IBM 360/165, установленном в вычислительном центре Кембриджского университета. Возможности Simula почти идеально подходили для моих целей. Особенно поразил тот факт, что концепции языка помогали мне размышлять над существом своей задачи. Концепция класса позволила отобразить понятия из предметной области на языковые конструкции настолько естественно, что мой код оказался понятнее, чем все, что я видел в других языках. То, что классы Simula могли использоваться в качестве сопрограмм, позволило мне легко выразить параллельность, присущую моему приложению. Например, объект класса `computer` было совсем просто заставить работать псевдопараллельно с другими объектами того же класса. Варианты понятий прикладного уровня выражались через иерархию классов. Так, виды компьютеров можно было представить как классы, производные от `computer`, а типы механизмов межмодульных коммуникаций – как классы, производные от класса `IPC`. Правда, этот прием я применял нечасто: для моего симулятора важнее было использовать классы с целью представления параллельности.

Во время написания и начальной отладки программы я смог в полной мере оценить выразительную мощь системы типов в Simula и способность компилятора находить ошибки типизации. Я заметил, что ошибки эти почти всегда являлись свидетельством невнимательности или изъяна проектирования. Последнее было

важнее всего и помогало мне больше, чем примитивные «сильно» типизированные языки, с которыми доводилось работать раньше. Так я пришел к уже звучавшему выше выводу, что система типов в языке Pascal не просто бесполезна – это смирительная рубашка, которая создает больше проблем, нежели решает, заставляя меня жертвовать чистотой дизайна ради удовлетворения причуд компилятора. Обнаруженный контраст между строгостью Pascal и гибкостью Simula оказался чрезвычайно важен при разработке C++. Концепция классов в Simula представлялась мне ключевым фактором, и с той поры я считаю, что при проектировании программ следует сосредотачивать свое внимание именно на классах. Я работал с Simula и прежде (во время учебы в университете города Аархус в Дании), но был приятно удивлен следующим: чем больше программа, тем очевидней польза от возможностей Simula. Механизмы классов и сопрограмм, а также исчерпывающий контроль типов гарантировали, что число ошибок увеличивается с ростом программы не более чем линейно (что было неожиданно). Напротив, программа работала, скорее, как набор очень маленьких программок, нежели как большой монолит, и поэтому писать, понимать и отлаживать ее было проще.

Однако реализация самого языка Simula не масштабировалась в той же степени, что и моя программа. В результате весь проект чуть не закончился крахом. В то время я пришел к выводу, что реализация Simula (в отличие от самого языка) была ориентирована на небольшие по объему программы, а для больших не приспособлена [Stroustrup, 1979]. На связывание отдельно скомпилированных классов уходила масса времени: на компиляцию 1/30 части программы и связывание ее с остальными, уже откомпилированными модулями тратилось больше времени, чем на компиляцию и связывание всей программы как монолита. Думаю, что, вероятнее всего, это была проблема используемого компоновщика, а не самого языка Simula, но это слабое утешение. Кроме того, производительность программы была такой низкой, что не оставляла надежд получить от симулятора хоть сколько-нибудь полезные данные. Плохие показатели производительности были обусловлены языком и его реализацией, а не приложением. Проблема накладных расходов является в Simula фундаментальной и неустранимой. Она коренится в некоторых особенностях языка и их взаимодействиях: проверке типов во время исполнения, гарантированной инициализации переменных, поддержке параллельности, сборке мусора для объектов, созданных пользователем, и записей активации процедур. Измерения показали: более 80% времени тратится на сборку мусора, хотя управление ресурсами брала на себя моделируемая система, так что мусор вообще не появлялся. Современные реализации Simula (15 лет спустя) стали лучше, но, по моим сведениям, увеличения производительности на порядок так и не достигнуто.

Чтобы не бросать проект, я переписал симулятор на BCPL и запускал его на экспериментальном компьютере CAP. Опыт кодирования и отладки на BCPL [Richards, 1980] оставил у меня неприятные воспоминания. Язык C по сравнению с BCPL – язык очень высокого уровня. Ни контроля типов, ни поддержки во время исполнения в BCPL здесь нет и в помине. Однако получившийся симулятор работал достаточно быстро и с его помощью я получил целый ряд полезных результатов. Они прояснили многие вопросы и легли в основу нескольких статей по операционным системам [Stroustrup, 1978, 1979b, 1981].

Расставшись с Кембриджем, я поклялся себе никогда больше не приступать к решению задачи, располагая такими неподходящими инструментами, как те, с которыми я намучился при проектировании и реализации симулятора. Для истории C++ важную роль сыграло составленное мной представление о «подходящем» инструменте для проектов такого масштаба, как большой симулятор, операционная система и аналогичные задачи системного программирования. Вот эти критерии:

- хороший инструмент должен предоставлять средства организации программ, подобные имеющимся в Simula: классы, форму их иерархии, поддержку параллельности и сильный (т.е. статический) контроль типов, основанный на классах. Эти критерии представлялись мне тогда (да и теперь) существенными для поддержки процесса проектирования, а не для реализации программы;
- необходимо, чтобы он генерировал программы, работающие так же быстро, как написанные на BCPL, и обладал способностью BCPL объединять раздельно откомпилированные модули в единую программу. Должно быть простое соглашение о связях, чтобы удалось объединять модули, написанные на разных языках, таких как C, Algol68, Fortran, BCPL, ассемблер и т.д. Иначе программист будет вынужден бороться с ограничениями, присущими какому-то одному языку;
- инструмент должен обеспечивать переносимую реализацию. Мой опыт показал, что «правильная» реализация, остро необходимая мне сейчас, будет готова «не раньше следующего года», да и то на компьютере, которого у меня нет. Отсюда следует, что должно быть несколько источников реализации инструмента (никакой монополист не сможет поддерживать всех пользователей «редких» машин, а также бедных студентов). Не должно быть также сложной системы поддержки времени исполнения, которую трудно переносить, и допустима лишь очень ограниченная зависимость инструмента от операционной системы.

Эти критерии еще не были четко сформулированы, когда я покидал Кембридж. Некоторые из них окончательно оформились лишь в ходе последующего осмысления собственного опыта, приобретенного при создании симулятора и программ, которые я писал в течение еще пары лет, а также опыта других людей. C++ в том виде, какой он принял к моменту выхода версии 2.0, полностью отвечает данным критериям; серьезные проблемы, с которыми я столкнулся при проектировании шаблонов и обработке исключений, связаны с отходом от некоторых из этих принципов. Я полагаю, что самой важной особенностью сформулированных выше правил является их слабая связь с нюансами конкретных языков программирования. Вместо этого они налагают определенные ограничения на решение.

Когда я работал в Кембридже, лабораторию вычислительной техники возглавлял Морис Уилкс (Maurice Wilkes). Помощь во всех технических вопросах мне оказывали мой руководитель Дэвид Уилер (David Wheeler) и Роджер Нидэм (Roger Needham). Мои знания в области операционных систем и интерес к модульности и межмодульным коммуникациям способствовали развитию C++.

Например, модель защиты в C++ базируется на концепции предоставления и передачи прав доступа; различие между инициализацией и присваиванием возникло благодаря размышлениям о способности переноса (transferring capabilities); концепция `const` берет начало от механизмов защиты от чтения/записи в аппаратных устройствах; механизм обработки исключений появился в связи с работой над отказоустойчивыми системами, выполненной группой под руководством Брайана Рэнделла (Brian Randell) в Ньюкасле в 70-х гг.

1.2. Язык C и системное программирование

На языке C я начал работать в Лондоне в 1975 г. и оценил его преимущества по сравнению с другими языками, которые принято называть языками для системного программирования, машинно-ориентированными или низкоуровневыми. Из таких языков мне были известны PL 360, Coral, Mary и другие, но в основном BCPL. Я не только пользовался BCPL, но однажды и реализовал его путем трансляции в промежуточный микрокод – O-код, так что хорошо представлял себе низкоуровневые аспекты, связанные с эффективностью языков такого класса.

Защитив диссертацию в Кембридже и получив работу в Bell Labs, я еще раз изучил C по книге Кернигана [Kernighan, 1978]. В то время я не был экспертом по C и рассматривал его в основном как самый современный и известный пример языка системного программирования. Лишь позже, приобретая собственный опыт и беседуя с коллегами – Стью Фельдманом (Stu Feldman), Стивом Джонсоном (Steve Johnson), Брайаном Керниганом и Деннисом Ричи, – я начал по-настоящему понимать C. Таким образом, общее представление о языках системного программирования значило для формирования C++ по меньшей мере столько же, сколько конкретные технические детали C.

Я довольно хорошо знал Algol68 [Woodward,1974] по работе над небольшими проектами в Кембридже и видел связь между конструкциями этого языка и C. Иногда мне казалось полезным рассматривать конструкции C как частные случаи более общих конструкций Algol68. Любопытно, что я никогда не рассматривал Algol68 в качестве языка системного программирования (несмотря на то что сам пользовался написанной на нем операционной системой). Подозреваю, причина здесь в том, что я считал очень важными переносимость, простоту связывания с программами на других языках и эффективность исполнения. Как-то раз я сказал, что Algol68 с классами, как в Simula, – это язык моей мечты. Однако в качестве практического инструмента C казался мне лучше, чем Algol68.

1.3. Немного об авторе книги

Говорят, что структура системы отражает структуру организации, в которой она была создана. В общем и целом я поддерживаю это мнение. Из него также следует, что если система есть плод работы одного человека, то она отражает склад его личности. Оглядываясь назад, я думаю, что на общую структуру C++ мое мировоззрение наложило такой же отпечаток, как и научные концепции, лежащие в основе отдельных его частей.

Я изучал математику, в том числе прикладную, поэтому защищенная в Дании кандидатская диссертация была посвящена математике и информатике. В результате я научился любить красоту математики, но предпочитал смотреть на нее, как на инструмент решения практических задач. Я искренне сочувствовал студенту, которого Евклид, по преданию, выгнал за вопрос «Но для чего нужна математика?» Точно так же мой интерес к компьютерам и языкам программирования носит в основном прагматический характер. Компьютеры и языки программирования можно оценивать как произведения искусства, но эстетические факторы должны дополнять и усиливать их полезные свойства, а не подменять их.

Больше 25 лет я увлекаюсь историей. Немалое время посвятил и изучению философии. Отсюда вполне осознанный взгляд на истоки моих интеллектуальных пристрастий. Если говорить о философских течениях, то мне, скорее, ближе эмпирики, чем идеалисты; мистиков я просто не понимаю. Поэтому Аристотеля я предпочитаю Платону, Юма – Декарту, а перед Паскалем склоняю голову. Всеобъемлющие «системы», такие, как у Платона или Канта, пленяют меня, но кажутся фундаментально порочными, поскольку, по-моему, они очень далеки от повседневного опыта и особенностей конкретного индивидуума.

Почти фанатичный интерес Кьеркегора к личности и его тонкое понимание психологии кажутся мне куда интереснее грандиозных схем и заботы обо всем человечестве, присущих Гегелю или Марксу. Уважение к группе, не подразумевающее уважения к ее членам, я не считаю уважением вовсе. Корни многих решений для C++ – в моем нежелании принуждать пользователей делать что бы то ни было жестко определенным образом. Из истории мы знаем, что вина за многие ужасные трагедии лежит на идеалистах, которые пытались заставить людей «делать так, чтобы им было хорошо». Кроме того, я считаю, что идеалисты склонны игнорировать неудобный опыт и факты, противоречащие их догмам или теории. Всякий раз, когда те или иные идеалы вступают в противоречие, а иногда и в тех ситуациях, где ученые мужи пришли к единодушному согласию, я предпочитаю давать программисту выбор.

Мои литературные вкусы еще раз подтверждают нежелание принимать решение только на основе теории и логики. В этом смысле C++ во многом обязан таким романистам и эссеистам, как Мартин А. Хансен, Альбер Камю и Джордж Оруэлл, которые никогда не видывали компьютера, и таким ученым, как Дэвид Грис, Дональд Кнут и Роджер Нидэм. Часто, испытывая искушение запретить какую-то возможность, которая лично мне не нравилась, я останавливался, ибо не считал себя вправе навязывать свою точку зрения другим людям. Я знаю, что многого можно добиться относительно быстро, если последовательно придерживаться логики и безжалостно выносить приговор «неправильному, устаревшему и нелогичному образу мыслей». Но при такой модели становятся очень велики человеческие потери. Для меня намного дороже принятие того факта, что люди думают и действуют по-разному.

Я предпочитаю медленно – иногда очень медленно – убеждать людей попробовать новые приемы и принять на вооружение те, которые отвечают их нуждам и склонностям. Существуют эффективные методы «обращения в другую веру» и «совершения революции», но я их боюсь и сильно сомневаюсь, что они так уж

эффективны в длительной перспективе и по большому счету. Часто, когда кого-то легко удается обратить в религию X, последующее обращение в религию Y оказывается столь же простым, а выигрыш от этого эфемерный. Я предпочитаю скептиков «истинно верующим». Мелкий, но неопровержимый факт для меня ценнее большинства теорий, а продемонстрированный экспериментально результат важнее груды логических аргументов.

Такой взгляд на вещи легко может привести к фаталистическому принятию status quo. В конце концов, нельзя приготовить омлет, не разбив яиц, – многие люди просто не хотят меняться, если это неудобно для их повседневной жизни, или стремятся хотя бы отложить перемены «до понедельника». Вот тут-то надо проявить уважение к фактам... и толику идеализма.

Состояние дел в программировании, как и вообще в мире, далеко от идеала, и многое можно улучшить. Я проектировал C++, чтобы решить определенную задачу, а не для того, чтобы что-то кому-то доказать, и в результате язык оказался полезным. В основе его философии лежала убежденность, что улучшений можно добиться путем последовательных изменений. Конечно, хотелось бы поддерживать максимальный темп изменений, улучшающих благосостояние людей. Но самое трудное – понять, в чем же состоит прогресс, разработать методику постепенного перехода к лучшему и избежать эксцессов, вызванных чрезмерным энтузиазмом.

Я готов упорно работать над внедрением в сознание идей, которые, по моему глубокому убеждению, принесут пользу людям. Более того, я считаю, что ученые и интеллектуалы должны способствовать распространению своих идей в обществе, чтобы они применялись людьми, а не оставались игрой изощренного ума. Однако я не готов жертвовать людьми во имя этих идей. В частности, я не хочу навязывать единый стиль проектирования посредством узко определенного языка программирования. Мысли и действия людей настолько индивидуальны, что любая попытка привести всех к общему знаменателю принесет больше вреда, чем пользы. Поэтому C++ сознательно спроектирован так, чтобы поддержать различные стили, а не показать единственный «истинный путь».

Принципы, которыми я руководствовался при проектировании C++, будут детально изложены в главе 4. Здесь вы обнаружите отголоски тех общих идей и идеалов, о которых говорилось только что.

Да, язык программирования – на редкость важная вещь, однако это всего лишь крохотная часть реального мира и поэтому не стоит относиться к нему чересчур серьезно. Необходимо обладать чувством меры и – что еще важнее – чувством юмора. Среди основных языков программирования C++ – богатейший источник шуток и анекдотов. И это неслучайно.

При обсуждении философских вопросов, равно как и возможностей языка легко скатиться на чрезмерно серьезный и нравоучительный тон. Если так произошло со мной, примите извинения, но мне хотелось объяснить свои интеллектуальные пристрастия, и думаю, что это безвредно – ну, почти безвредно. Да, кстати, мои литературные вкусы не ограничиваются только произведениями вышеназванных авторов, просто именно они повлияли на создание C++.

Глава 2. Язык C with Classes

Специализация – это для насекомых.
Р.А. Хайнлайн

2.1. Рождение C with Classes

Работа над тем, что впоследствии стало языком C++, началась с попытки проанализировать ядро UNIX, чтобы понять, как можно было бы распределить эту систему между несколькими компьютерами, соединенными локальной сетью. Этот эксперимент проходил в апреле 1979 г. в Центре исследований по вычислительной технике компании Bell Laboratories в Мюррей Хилл, штат Нью-Джерси. Вскоре было выявлено две подзадачи: как проанализировать сетевой трафик, порожденный распределенностью ядра, и как разбить ядро на отдельные модули. В обоих случаях требовалось выразить модульную структуру сложной системы и типичные способы обмена информацией между модулями. Задача была как раз из тех, к решению которых я зарекся приступать без соответствующих инструментов. Поэтому мне пришлось заняться разработкой подходящего инструментария в соответствии с критериями, выработанными в Кембридже.

В октябре 1979 г. был готов препроцессор, который я назвал Cpre. Он добавлял Simula-подобные классы к C. В марте 1980 г. препроцессор был улучшен настолько, что использовался в одном реальном проекте и нескольких экспериментальных. В моем архиве сохранилось упоминание о том, что к тому времени Cpre работал на 16 системах. Самым важным элементом этих проектов была первая серьезная библиотека на C++, которая поддерживала многозадачное программирование с применением сопрограмм [Stroustrup, 1980b], [Stroustrup, 1987b], [Shopiro, 1987]. Язык, подаваемый на вход препроцессора, получил название «C with Classes».

В период с апреля по октябрь я начал думать не об инструменте, а о языке, но C with Classes все еще рассматривался как расширение C, позволяющее выразить концепции модульности и параллельности. Однако важнейшее решение уже было принято. Хотя поддержка параллельности и моделирования в духе Simula и являлась основной целью C with Classes, язык не содержал никаких примитивов для выражения параллельности. Вместо этого была написана библиотека, поддерживающая необходимые стили параллельности. В ней использовалась комбинация наследования (иерархии классов) с возможностью определять функции-члены класса специального назначения, распознаваемые препроцессором. Обратите внимание на множественное число – «стили». Я считал – и считаю – особенно важным, чтобы с помощью языка можно было выразить несколько вариантов параллельной обработки. Есть множество приложений, которым параллельность необходима,

а преобладающей модели ее поддержки нет. Поэтому в случаях, когда параллельность нужна, ее следует реализовывать в библиотеках или в специализированном расширении так, чтобы одна конкретная форма не запрещала использование других.

Таким образом, на языке предлагались общие механизмы организации программ, а вовсе не поддержка конкретных предметных областей. Именно это и сделало C with Classes, позднее и C++, универсальным языком программирования, а не расширением C для специализированных приложений. Позже вопрос о выборе между поддержкой специализированных приложений и общим механизмом абстракций вставал неоднократно. И всякий раз принималось решение в пользу усовершенствования механизма абстракций. Поэтому в C++ нет ни встроенных типов для комплексных чисел, строк и матриц, ни прямой поддержки параллельности, устойчивости объектов, ни распределенных вычислений, ни сопоставления образцов и манипуляций на уровне файловой системы. Я упомянул лишь малую толику часто предлагавшихся расширений. Но существуют библиотеки, поддерживающие все эти возможности.

Предварительное описание C with Classes было опубликовано в виде технического отчета в Bell Labs в апреле 1980 г. [Stroustrup,1980] и в журнале SIGPLAN Notices [Stroustrup,1982]. Более подробный технический отчет *Adding Classes to the C Language: An Exercise in Language Evolution* [Stroustrup, 1980] напечатан в журнале *Software: Practices and Experience*. В данных работах заложена хорошая идея: описывать только те возможности, которые полностью реализованы и нашли применение. Такой подход соответствовал традициям Центра исследований по вычислительной технике компании Bell Laboratories. Я отошел от подобной тактики только тогда, когда назрела необходимость большей открытости относительно будущего C++, чтобы в свободной дискуссии по поводу эволюции языка могли принять участие многочисленные пользователи, которые не работали в AT&T.

C with Classes специально проектировался для обеспечения лучшей организации программ. Было решено, что с собственно «вычислительной частью» C и так справляется. Я считал принципиальным такой аспект: улучшение структуры не может достигаться за счет падения производительности по сравнению с C. Язык C with Classes не должен был уступать C во времени выполнения, компактности кода и данных. Однажды кто-то продемонстрировал систематическое трехпроцентное падение производительности новой версии по сравнению с C, вызванное наличием временных переменных, которые препроцессор C with Classes использовал в механизме возврата из функции. Недочет был признан неприемлемым, и причину быстро устранили. Аналогично, чтобы обеспечить совместимость формата хранения с C и, следовательно, избежать накладных расходов по памяти, в объекты классов не помещались никакие «служебные» данные.

Другой важной целью работы было стремление избежать ограничений на области использования C with Classes. Идеал (кстати, достигнутый) – C with Classes может применяться везде, где использовался C. Отсюда следовало, что новая версия ни в коем случае не должна была уступать C в эффективности, но эта эффективность не могла достигаться за счет отказа от некоторых, пусть даже уродливых особенностей C. Это соображение (если хотите, принцип) приходилось снова

и снова повторять тем людям (как правило, не работавшим постоянно с C with Classes), которые хотели сделать версию безопаснее, введя статический контроль типов а-ля Pascal. Альтернативу такой «безопасности» – вставку проверок в исполняемый код – решили реализовывать в отладочных средах. В самом языке нельзя было организовывать таких проверок, ибо тогда он безнадежно проиграл бы C по скорости и расходу памяти. Поэтому такого рода контроль в C with Classes включен не был, хотя в некоторых средах разработки для C++ он имеется, но только в отладочном режиме. Кроме того, пользователи могут сами производить проверки во время выполнения там, где сочтут целесообразным (см. раздел 16.10 и [2nd]).

C поддерживает низкоуровневые операции, например манипуляции с битами и выбор между разными размерами целых. Имеются в нем и такие средства (допустим, явное преобразование типов), которые позволяют намеренно обойти систему контроля типов. C With Classes, а затем и C++ сохранили низкоуровневые и небезопасные особенности C. Но в C++ систематически устраняется необходимость использования таких средств за исключением тех мест, где без них никак не обойтись. При этом небезопасные операции выполняются только по явному указанию программиста. Я искренне убежден, что не существует единственно правильного способа написать программу, и дизайнер языка не должен заставлять программиста следовать определенному стилю. С другой стороны, дизайнер обязан всячески поддерживать все разнообразие стилей и способов программирования, доказавших свою эффективность, а равно предоставить такие языковые возможности и инструментальные средства, которые уберегли бы программиста от хорошо известных ловушек.

2.2. Обзор языковых возможностей

Вот сводка тех средств, которые были включены в первую версию 1980 г.:

- классы (раздел 2.3);
- производные классы (но пока без виртуальных функций, раздел 2.9);
- контроль доступа – открытый/закрытый (раздел 2.10);
- конструкторы и деструкторы (раздел 2.11.1);
- функции, вызываемые при вызове и возврате (позже исключены, раздел 2.11.3);
- дружественные (`friend`) классы (раздел 2.10);
- контроль и преобразование типов аргументов функции (раздел 2.6).

В 1981 г. добавлены:

- встраиваемые (inline) функции (раздел 2.4.1);
- аргументы по умолчанию (раздел 2.12.2);
- перегрузка оператора присваивания (раздел 2.12.1).

Поскольку C with Classes был реализован с помощью препроцессора, описывать следовало лишь новые, отсутствовавшие в C возможности, а вся мощь C и так оставалась в распоряжении пользователей. В то время оба эти аспекта были должным образом оценены. Раз C – подмножество языка, то объем работы по поддержке и документированию намного уменьшается. Это особенно важно, поскольку на

протяжении нескольких лет мне приходилось заниматься документированием и поддержкой C with Classes и C++ одновременно с экспериментированием, проектированием и реализацией новых версий. Доступность всех возможностей C гарантировала, что по недосмотру или из-за моих предрассудков не будет введено никаких ограничений, которые лишат пользователей привычных средств. Естественно, переносимость на машины, где C уже имелся, обеспечивалась автоматически. Изначально C with Classes был реализован и эксплуатировался на DEC PDP/11, но вскоре был перенесен на DEC VAX и компьютеры на базе процессора Motorola 68000.

C with Classes все еще рассматривался как диалект C, а не как самостоятельный язык. Даже классы тогда назывались «абстрактными типами данных (abstract data type facility)» [Stroustrup, 1980]. Поддержка объектно-ориентированного программирования была провозглашена лишь после добавления в C++ виртуальных функций [Stroustrup, 1984].

2.3. Классы

Очевидно, самой важной чертой C with Classes, а позже и C++ была концепция класса. Многие ее аспекты видны из следующего примера [Stroustrup, 1980][1]:

```
class stack {
   char   s[SIZE];  /* массив символов */
   char*  min;      /* указатель на конец стека */
   char*  top;      /* указатель на вершину стека */
   char*  max;      /* указатель на начало выделенной области */
   void   new();    /* функция инициализации (конструктор) */
public:
   void   push(char);
   char   pop();
};
```

Класс – это определенный пользователем тип данных. Он содержит описания типов членов класса, то есть представление переменной данного типа (объекта класса), и набор операций (функций) для манипулирования такими объектами. Класс также определяет права доступа к своим членам со стороны пользователей. Функции-члены обычно определяются отдельно:

```
char stack.pop()
{
   if (top <= min) error("стек пуст");
   return *(--top);
}
```

Теперь можно определить и использовать объекты класса `stack`:

```
class stack s1, s2;          /* две переменные типа stack */
class stack * p1 = &s2;      /* p1 указывает на s2 */
```

[1] Я сохранил оригинальный синтаксис и стиль C with Classes. Отличия от C++ и современного стиля кодирования вряд ли у кого-то вызовут затруднения, а некоторым читателям будут интересны. Однако очевидные ляпы исправлены. Также добавлены комментарии, которых в исходном тексте не было.

```
class stack * p2 = new stack; /* p2 указывает на объект класса stack,
                                 размещенный в куче */

s1.push('h');    /* прямое обращение к объекту */
p1->push('s');   /* обращение к объекту по указателю */
```

Из этого примера видны несколько ключевых проектных решений:

- по аналогии с Simula язык C with Classes позволяет задать типы, из которых создаются переменные (объекты), тогда как, скажем, Modula описывает модуль как набор объектов и функций. В C with Classes класс – это тип (см. раздел 2.9), что стало главной особенностью C++. Но если слово `class` определяет в C++ пользовательский тип, то почему я не назвал его `type`? В основном потому, что я не люблю изобретать новую терминологию, а соглашения, принятые в Simula, меня в большинстве случаев устраивали;
- представление объектов пользовательского типа – часть объявления класса, что имеет далеко идущие последствия (см. разделы 2.4 и 2.5). Например, это означает следующее: настоящие локальные переменные пользовательского типа можно реализовать без использования кучи (ее еще называют динамической памятью) и сборки мусора. Отсюда также следует, что при изменении представления объекта все функции, использующие его напрямую (не через указатель), должны быть перекомпилированы. (См. раздел 13.2, где описаны средства C++ для определения интерфейсов, позволяющие избежать такой перекомпиляции.);
- контроль доступа используется еще на стадии компиляции для ограничения доступа к элементам представления класса. По умолчанию имена членов класса могут встречаться только в функциях, упомянутых в объявлении класса (см. раздел 2.10). Члены (обычно функции-члены), описанные в открытом интерфейсе класса – части, следующей за меткой `public:`, – могут использоваться и в других местах программы;
- для функций-членов указывается полный тип (включая тип возвращаемого значения и типы формальных аргументов). На основе данной спецификации производится статическая (во время компиляции) проверка типов (см. раздел 2.6). В то время это было отличием от C, где типы формальных аргументов не задавались в интерфейсе и не проверялись при вызове;
- определения функций обычно выносятся в другое место, чтобы класс больше походил на спецификацию интерфейса, чем на лексический механизм организации исходного кода. Это облегчает раздельную компиляцию функций-членов класса и внешней программы, которая их использует. Поэтому техники компоновки, традиционной для C, достаточно для поддержки C++ (см. раздел 2.5);
- функция `new()` является конструктором, для компилятора она имеет специальный смысл. Такие функции дают определенные гарантии относительно классов (см. раздел 2.11): конструктор – в то время он назывался `new`-функцией – обязательно будет вызван для инициализации каждого объекта своего класса перед его первым использованием;

- имеются указательные и неуказательные типы (они есть и в C, и в Simula). Указатели могут указывать на объекты как встроенных, так и пользовательских типов;
- как и в C, память для объектов может выделяться тремя способами: в стеке (автоматическая память), по фиксированному адресу (статическая память) и из кучи (динамическая память). Но в отличие от C, в C with Classes есть специальные операторы `new` и `delete` для выделения и освобождения динамической памяти (см. раздел 2.11.2).

Дальнейшую разработку C with Classes и C++ можно в значительной мере рассматривать как изучение последствий этих проектных решений, выявление их положительных и отрицательных сторон и устранение проблем, вызванных недостатками. Многие, но далеко не все следствия принятых решений были понятны уже в то время (работа [Stroustrup,1980] датирована 3 апреля 1980 г.). В данном разделе я попытаюсь объяснить, чту же было ясно уже тогда, и сошлюсь на разделы, где рассматриваются более отдаленные последствия и позднейшие реализации.

2.4. Эффективность исполнения

В Simula не может быть локальных или глобальных переменных типа класса, память для каждого объекта класса выделяется динамически оператором `new`. Измерения с помощью кембриджского симулятора убедили меня, что это основная причина неэффективности языка. Позднее Карел Бабчиски (Karel Babcisky) из Норвежского Вычислительного Центра представил данные о производительности Simula, подтвердившие мой вывод [Babcisky, 1984]. Уже по этой причине я хотел иметь глобальные и локальные переменные типа класса.

Кроме того, наличие разных правил для создания и области действия переменных встроенных и пользовательских типов просто неизящно, а в некоторых случаях я видел, как страдает мой стиль программирования от отсутствия в Simula локальных и глобальных переменных типа класса. Не хватало и возможности иметь указатели на встроенные типы в Simula. Эти наблюдения со временем превратились в определенное правило дизайна C++: пользовательские и встроенные типы должны вести себя одинаково по отношению к правилам языка. И сам язык, и инструментальные средства должны обеспечивать для них одинаковую поддержку. Когда формулировался этот критерий, поддержка встроенных типов была гораздо обширнее, но развитие языка C++ продолжалось, так что теперь данные типы поддержаны чуть хуже пользовательских (см. раздел 15.11.3).

В первой версии C with Classes отсутствовали встраиваемые (inline) функции, но вскоре они были добавлены. Основная причина их включения в язык – опасение, что из-за расходов на преодоление «защитного барьера» люди не захотят пользоваться классами для сокрытия представления. Так, в [Stroustrup,1982b] отмечено, что многие делают члены классов открытыми, дабы не расплачиваться за вызов конструктора в простых классах, где для инициализации достаточно одного-двух присваиваний. Толчком для включения в C with Classes встраиваемых

функций послужил проект, в котором для некоторых классов, связанных с обработкой в реальном времени, накладные расходы на вызов функций оказались неприемлемы. Чтобы воспользоваться преимуществами классов в такого рода приложениях, необходимо иметь возможность «бесплатно» преодолевать защиту доступа. Этого можно было добиться только сочетанием представления в объявлении класса с встраиванием вызовов открытых функций.

В связи с этим возникло следующее правило C++: недостаточно просто предоставить возможность, нужно также, чтобы плата за пользование ей была не слишком велика, то есть «приемлема на том оборудовании, которое есть у пользователей», а не «приемлема для исследователей, имеющих доступ к высокопроизводительному оборудованию» или «приемлема через пару лет, когда аппаратные средства подешевеют». C with Classes всегда рассматривался как язык, готовый к применению сейчас или в следующем месяце, а не как исследовательский проект, от которого можно ожидать отдачи через несколько лет.

2.4.1. Встраивание

Встраивание было признано важным для удобства работы с классами. Поэтому вопрос заключался не в том, стоит ли его реализовывать, а в том, кбк это сделать. Два аргумента убедили меня, что только программист должен решать, какие функции компилятор попытается встроить в код. Во-первых, у меня был печальный опыт работы с языками, где решение вопроса о встраивании оставлялось на усмотрение компилятора, поскольку он якобы «лучше знает». Однако на компилятор можно положиться только в том случае, если в него запрограммирована концепция встраивания и его представление об оптимизации по времени и памяти совпадает с моим. Опыт работы с другими языками показал, что встраивание, как правило, «будет реализовано в следующей версии», да и то в соответствии с внутренней логикой языка, которой программист не может эффективно управлять. Кроме того, C (а за ним C with Classes, и C++) организует раздельную компиляцию, так что компилятору всегда доступен только небольшой фрагмент всей программы (см. раздел 2.5). Встраивание функции, исходный код которой неизвестен, возможно только при наличии очень развитой технологии компоновки и оптимизации, но тогда такой технологии не было (нет и сейчас в большинстве сред разработки). Наконец, методы, использующие глобальный анализ кода, в частности автоматическое встраивание без поддержки со стороны пользователя, плохо адаптируются к большим программам. C with Classes проектировался для того, чтобы получать эффективный код при наличии простой переносимой реализации на наиболее распространенных системах. С учетом всего вышесказанного от программиста требуется помощь. Даже сегодня этот выбор кажется мне правильным.

В C with Classes допускалось только встраивание функций-членов. Единственным способом заставить компилятор встроить функцию было помещение ее в объявление класса. Например:

```
class stack {
   /* ... */
   char pop()
   {
      if (top <= min) error("стек пуст");
```

```
        return *--top;
    }
};
```

То, что при этом объявление класса несколько теряет наглядность, не осталось без внимания, но было сочтено правильным, поскольку препятствовало чрезмерному применению встраиваемых функций. Ключевое слово `inline` и возможность встраивать функции, не являющиеся членами, появились позже, уже в C++. Так, в C++ этот пример можно переписать иначе:

```
class stack {// C++
    // ...
    char pop();
};

inline char stack::pop()   // C++
{
    if (top <= min) error("стек пуст");
    return *--top;
}
```

Директива `inline` – лишь совет, который компилятор может игнорировать и часто так и поступает. Это вызвано логической необходимостью, поскольку можно написать рекурсивную встраиваемую функцию, а на этапе компиляции невозможно доказать, что рекурсия не окажется бесконечной. Попытка встраивания такой функции привела бы к зацикливанию компилятора. Придание слову `inline` статуса совета имеет и практическую пользу, поскольку позволяет автору компилятора обработать те случаи, когда встраивание невозможно, и просто отказаться от него.

Для C with Classes, как и для всех его преемников, было необходимо, чтобы встраиваемая функция имела в программе единственное определение. Определение такой функции, как `pop()`, приведенной выше, в разных единицах компиляции привело бы к хаосу, игнорированию системы контроля типов. Но в условиях раздельной компиляции трудно гарантировать, что в большой системе данное правило не нарушено. В C with Classes это не проверялось, и в большинстве реализаций C++ до сих пор нет гарантий, что встраиваемая функция не определена по-разному в разных единицах компиляции. Однако теоретическая проблема не переросла в практическую в основном потому, что встраиваемые функции обычно определяют в заголовочных файлах вместе с классами, а объявления классов в программе также должны быть уникальны.

2.5. Модель компоновки

Вопрос о том, как скомпоновать раздельно откомпилированные фрагменты программы, важен для любого языка программирования и до некоторой степени определяет возможности языка. На разработку C with Classes и C++ во многом повлияли следующие решения:

- раздельная компиляция может осуществляться с использованием стандартных компоновщиков (редакторов связей) для C/Fortran, применяемых на платформах UNIX и DOS;

- компоновка должна быть типобезопасной (не противоречить системе контроля типов);
- компоновка не должна нуждаться в какой бы то ни было базе данных (хотя ее использование в конкретных реализациях для повышения эффективности не запрещается);
- компоновка с фрагментами программы, написанными на других языках, например на C, Fortran или ассемблере, должна быть простой и эффективной.

Чтобы обеспечить непротиворечивость раздельной компиляции в C, используются заголовочные файлы, которые обычно дословно включаются в каждый исходный файл, где соответствующее объявление необходимо. В них помещаются объявления структур данных, функций, переменных и констант. Непротиворечивость обеспечивается за счет того, что в заголовочные файлы помещается вся необходимая информация, доступ к которой производится только путем включения этих файлов. C++ следует этой модели, но только до определенного момента.

В объявлении класса в C++ может (хотя и необязательно, см. раздел 13.2) быть описано размещение объекта. Это разрешено для того, чтобы упростить и сделать эффективным объявление истинно локальных переменных. Рассмотрим такую функцию:

```
void f()
{
   class stack s;
   int c;
   s.push('h');
   c = s.pop();
}
```

Реагируя на объявление класса `stack` (см. разделы 2.3 и 2.4.1), даже простейшая версия C with Classes сможет сгенерировать для этого примера код, где: динамическая память не используется, функция `pop()` встроена так, что ее вызов не связан с накладными расходами, а при обращении к `push()` вызывается невстраиваемая, отдельно скомпилированная функция. В этом отношении C++ напоминает язык Ada.

В то время я полагал, что можно найти какой-то компромисс между двумя подходами:

- отделение объявления интерфейса от реализации (как в Modula-2) в сочетании с подходящим инструментом (редактором связей);
- наличие единого объявления класса в сочетании с инструментом (анализатором зависимостей), который будет рассматривать интерфейс отдельно от деталей реализации с целью определить, в каких случаях нужна повторная компиляция.

Похоже, я недооценил сложность последнего решения, а сторонники первого подхода – его стоимость (с точки зрения переносимости и затрат во время исполнения).

Еще больше я осложнил жизнь пользователям C++, не объяснив должным образом, как можно воспользоваться производными классами для отделения

интерфейса от реализации. Разумеется, я пытался растолковать это (см. пример в [Stroustrup,1986, §7.6.2]), но почему-то не был понят. Думаю, причина неудачи в том, что мне никогда не приходила в голову простая мысль: многие (если не большинство) программистов, работая с C++, думают, что раз можно поместить представление прямо в объявление класса, описывающего интерфейс, то это обязательно нужно сделать.

Я не пытался создать инструменты типобезопасной компоновки для C with Classes, они появились лишь в версии C++ 2.0. Однако я помню разговор с Деннисом Ричи и Стивом Джонсоном о том, что безопасность с точки зрения типов при пересечении границ единиц компиляции должна была стать частью C. Просто не было возможностей гарантировать это для реальных программ, так что пришлось полагаться на инструменты типа `Lint` [Kernighan, 1984].

В частности, Стив Джонсон и Деннис Ричи утверждали, что в C предполагалось обеспечить эквивалентность имен, а не структур. Например, объявления

```
struct A { int x, y; };
struct B { int x, y; };
```

определяют два несовместимых типа `A` и `B`. Далее, объявления

```
struct C { int x, y; };    // в файле 1
struct C { int x, y; };    // в файле 2
```

определяют два разных типа с одним и тем же названием `C`, и компилятор, способный осуществлять сквозную проверку в разных единицах компиляции, должен был бы выдать ошибку «повторное определение». Это правило призвано снять некоторые проблемы при сопровождении программы. Подобные повторяющиеся объявления возникают, по всей вероятности, при копировании текста из одного файла в другой. Но после этой операции объявление вполне может измениться. И если, изменив его в одном файле, не сделать то же самое в другом, программа просто перестанет работать.

На практике C, а за ним и C++ гарантируют, что структуры типа `A` и `B`, приведенные выше, одинаково размещаются в памяти, так что их можно приводить друг к другу и использовать очевидным образом:

```
extern f(struct A*);
void g(struct A* pa, struct B* pb)
{
   f(pa);     /* правильно */
   f(pb);     /* ошибка: ожидается A* */

   pa = pb;             /* ошибка: ожидается A* */
   pa = (struct A*)pb;  /* правильно: явное преобразование */
   pb->x = 1;
   if (pa->x != pb->x) error("плохая реализация");
}
```

Эквивалентность имен – основополагающий принцип системы типов в C++, а правила совместимости размещения в памяти гарантируют возможность явных преобразований, используемых в низкоуровневых операциях. В других языках

для этого используется структурная эквивалентность. Я отдаю предпочтение эквивалентности имен, а не структур, потому что считаю такую модель наиболее безопасной. Поэтому мне было приятно узнать, что такое решение не противоречит C и не усложняет предоставление низкоуровневых услуг.

Так появилось «правило одного определения»: каждая функция (переменная, тип, константа и т.д.) должна иметь в C++ ровно одно определение.

2.5.1. Простые реализации

Желание иметь простую реализацию отчасти было вызвано необходимостью (для разработки C with Classes не хватало ресурсов), а отчасти обусловлено недоверием к излишне изощренным языкам и механизмам. Ранняя формулировка одной из целей проектирования C with Classes звучала так: «для реализации языка должно хватать алгоритмов не сложнее линейного поиска». Любое нарушение этого правила – например в случае перегрузки функций (см. раздел 11.2) – приводило к семантике, которая мне казалась слишком сложной. А зачастую и к сложностям при реализации.

Моей целью – памятуя опыт работы с Simula – было спроектировать язык достаточно простой для понимания, чтобы привлечь пользователей, и довольно простой в реализации, чтобы заинтересовать разработчиков компиляторов. С другой стороны, относительно несложная реализация должна была генерировать код, который не уступал бы C в корректности, скорости и величине. Пользователь, незнакомый с языком на практике и находясь в не очень «дружелюбной» среде разработки, тем не менее должен суметь воспользоваться компилятором в реальных проектах. Только при выполнении обоих этих условий можно было рассчитывать, что C with Classes, а позднее C++ выживут в конкуренции с C. Ранняя формулировка принципа звучала так: «C with Classes должен быть неприхотливым «сорняком» вроде C или Fortran, поскольку мы не можем позволить себе ухаживать за такой «розой», как Algol68 или Simula. Если мы создадим компилятор и на год уедем, то по возвращении хотелось бы увидеть его работающим хотя бы на нескольких системах. Этого не произойдет, если будет необходимо постоянное сопровождение или если простой перенос на новую машину займет больше недели».

Данный лозунг был частью философии – воспитывать в пользователях самостоятельность. Всегда, притом явно, ставилась задача вырастить местных экспертов по всем аспектам работы с C++. Кстати, большинство организаций вынуждено следовать противоположной стратегии – культивировать зависимость пользователей от услуг, приносящих доход центральной службе технической поддержки, консультантам или и тем, и другим одновременно. По моему мнению, здесь заключено фундаментальное отличие C++ от многих других языков.

Решение работать в довольно примитивной – и почти повсеместно доступной – среде, обеспечивающей лишь компоновку в стиле C, породило большую проблему, связанную с тем, что компилятор C++ в любой момент времени имел лишь частичную информацию о программе. Каждое предположение относительно программы может стать неверным, если завтра часть этой программы перепишут на каком-то другом языке (C, Fortran или ассемблере) и свяжут с остальными модулями, возможно, уже после того, как программа запущена в эксплуатацию.

Данная проблема имеет многообразные проявления. Компилятору очень трудно гарантировать, что:

- объект, переменная и т.п. уникальны;
- информация непротиворечива (в частности, типы не конфликтуют);
- объект, переменная и т.п. инициализированы.

Кроме того, в C есть лишь самая минимальная поддержка концепции раздельных пространств имен, так что проблемой становится загрязнение пространства имен из-за того, что части программ создаются разными людьми. Развивая C++, мы пытались ответить на все эти вызовы, не принося в жертву фундаментальную модель и технологию, которые обеспечили переносимость и эффективность; но во времена C with Classes мы просто полагались на заголовочные файлы в стиле C.

С принятием в качестве инструмента компоновщика, применяемого для программ на C, связано появление следующего правила: C++ – это просто еще один язык в системе, а не вся система. Другими словами, он выступает в качестве традиционного языка программирования и принимает фундаментальные различия между языком, операционной системой и другими важными компонентами работы программиста. Это ограничивает языковые рамки, что довольно трудно сделать для таких языков, как Smalltalk или Lisp, задумывавшиеся как всеобъемлющие системы или среды. Очень важно, чтобы часть программы на C++ могла вызывать части, написанные на других языках, и сама могла быть вызвана. Кроме того, раз C++ – просто язык, то он может применять инструменты, написанные для других языков.

Тот факт, что язык программирования и написанный на нем код должны быть лишь шестеренкой внутри большого механизма, принципиально важен для большинства пользователей, работающих над промышленными проектами. Тем не менее многие теоретики, педанты и пользователи из академических кругов, очевидно, не учитывали, насколько важно «мирное сосуществование» одного языка с другими и с системами. Я думаю, что в этом одна из основных причин успеха C++.

C with Classes почти совместим с C на уровне исходных текстов. Однако совместимость никогда не была стопроцентной. Например, `class` и `new` допустимы в C в качестве имен идентификаторов, но в C with Classes и его преемниках они являются ключевыми словами. Однако на уровне компоновки совместимость сохранена. Функции на C можно вызывать из C with Classes. Функции на C with Classes можно вызывать из C, а структуры одинаково размещаются в памяти, следовательно, передача как простых, так и составных объектов между функциями, написанными на разных языках, проста и эффективна. Такая совместимость по компоновке сохранена и в C++, за несколькими простыми и явно обозначенными исключениями, которые программист при желании легко может обойти (см. раздел 3.5.1). С годами я и мои коллеги пришли к выводу, что гораздо важнее совместимость на уровне компоновки, чем на уровне исходных текстов. По крайней мере, это верно, когда идентичный исходный код дает одни и те же результаты как в C, так и в C++, или не компилируется, или не связывается на одном из этих языков.

2.5.2. Модель размещения объекта в памяти

Базовая модель объекта имела огромное значение для дизайна C with Classes. Я всегда точно представлял себе, как располагается объект в памяти, и рассматривал воздействие различных языковых средств на объекты. Без понимания эволюции модели объекта нельзя понять эволюцию C++.

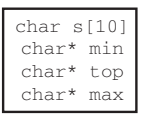
В C with Classes объект представлял собой просто C-структуру. Иными словами, объект

```
class stack {
   char s[10];
   char* min;
   char* top;
   char* max;
   void new();
public:
   void push();
   char pop();
};
```

хранился в памяти так же, как структура

```
struct stack {   /* сгенерированный C-код */
   char s[10];
   char* min;
   char* top;
   char* max;
};
```

то есть



Компилятор может добавлять некоторые заполнители до и после членов структуры для выравнивания, но в остальном размер объекта равен сумме размеров членов. Таким образом, расход памяти минимизируется.

Накладные расходы во время исполнения также сведены к минимуму за счет прямого отображения вызова функции-члена

```
void stack.push(char c)
{
   if (top>max) error("стек пуст");
   *top++ = c;
}

void g(class stack* p)
{
   p->push('c');
}
```