Содержание

С далекого 2002 года и до 2016 года .NET Framework оставался продуктом с закрытым исходным кодом. С появлением .NET Core разработчики смогли узнать, как платформа работает, какие технические решения были использованы в тех или иных местах. Одна из самых сложных подсистем .NET – это, вне всякого сомнения, сборщик мусора. Это тот элемент платформы, с которым так или иначе сталкивается каждый разработчик. При этом про сборщик мусора было известно довольно мало. Но и то немногое, что было известно, позволяло его успешно использовать. С другой стороны, механизмы управления памятью в .NET являются источником огромного количества мифов и недопониманий. Теперь, когда мы можем изучить исходный код, есть возможность развеять все мифы и понять, как устроена сборка мусора в .NET.

И вот вы держите в руках уникальную книгу. На сотнях страниц автор последовательно излагает всю информацию, необходимую для понимания работы с памятью на платформе .NET. Это книга не только про сборку мусора. Автор уделяет достаточно внимания и аппаратному уровню, и практическим аспектам программирования на платформе .NET, связанным с использованием памяти. Однако основной объем книги посвящен подробному разбору всех тонкостей работы современного сборщика мусора в .NET. Казалось бы, внутреннее устройство сборщика мусора не особенно полезно при разработке бизнес-приложений, однако это совсем не так. Такая сложная система, как сборщик мусора, не обходится без интересных инженерных решений, изучая которые, можно существенно расширить свои знания о платформе .NET и программировании в целом. Кроме того, каждая глава сопровождается полезными практическими выводами, рекомендациями и примерами использования инструментов отладки и профилирования. Книга будет полезна всем, кто интересуется внутренним устройством .NET, а также тем, кто стремится улучшить свои приложения, сделать их оптимальнее. С ростом популярности облачных вычислений вопросы оптимизации (и в конечном счете экономии денег) будут все более актуальны.

Российское сообщество .NET-разработчиков DotNet.Ru с удовольствием трудилось над этой книгой, чтобы достичь наивысшего качества перевода и позволить читателям погрузиться в мир сборки мусора и работы с памятью. Желаем приятного и полезного чтения!

Над переводом работали представители сообщества DotNet.Ru:
Игорь Лабутин
Ирина Ананьева
Максим Шошин
Елизавета Голенок
Евгений Биккинин
Ренат Тазиев
Анатолий Кулаков

DOT
NET
.RU

Каста авторов книг неизменно пользуется особым вниманием со стороны участников на технических конференциях. Доклады этих людей отличаются глубокой проработкой и широкими взглядами. Ведь за плечами у них рукописи, на подготовку которых уходят годы. Это позволяет не только насладиться качественными презентациями, но и провести несколько часов вместе с автором над обсуждением интересных идей.

Типичным представителем прекрасного докладчика, глубокого специалиста, внимательного автора и является Конрад Кокоса. Благодаря невероятной тяге к исследованиям в столь узкой и сложной области мы получили шанс насладиться этим фундаментальным трудом. Десятилетия разработчики воспринимали сборщик мусора как волшебный ящик. Что порождало немало мифов и заблуждений. Конрад стал первым автором, который не только смог понять тонкости работы этого сложнейшего компонента, но и замечательно систематизировал полученные знания, снабдив наработки великолепными схемами.

Эта книга – незаменимое пособие для разработчиков, интересующихся производительностью программ, а также архитектурой сложных, нестандартных систем. Заложенные здесь знания будут еще долгие годы давать пищу для экспериментов и материал для новых докладов.

Анатолий Кулаков,
член программного комитета конференции DotNext

Об авторе

Конрад Кокоса – опытный проектировщик и разработчик программного обеспечения, интересующийся прежде всего технологиями корпорации Майкрософт, но с любопытством поглядывающий и по сторонам. Он программирует уже больше десяти лет, занимаясь решением проблем производительности и архитектурными головоломками в мире .NET, проектирует и повышает быстродействие приложений. Является независимым консультантом, ведет блог на сайте http://tooslowexception.com, выступает с докладами на встречах по интересам и на конференциях, фанатеет от Твиттера (@konradkokosa). Он также отдается своей страсти к преподаванию в области .NET, особенно в части повышения производительности приложений, хорошего стиля кодирования и диагностики. Основатель варшавской группы по производительности веб-приложений. Имеет звание Microsoft MVP в категории «Visual Studio и средства разработки». Сооснователь сайта Dotnetos.org, созданного тремя любителями .NET, организующими туры и конференции, посвященные производительности в .NET.

О технических рецензентах

Дамьен Фоггон – разработчик, писатель и технический рецензент, работающий в области передовых технологий, внес вклад более чем в 50 книг по .NET, C#, Visual Basic и ASP.NET. Сооснователь базирующейся в Ньюкасле группы пользователей NEBytes (адрес в интернете http://www.nebytes.net), сертифицированный профессионал Майкрософт во многих номинациях, начиная с .NET 2.0. Ведет блог по адресу http://blog.fasm.co.uk.

Маони Стивенс – архитектор и главный разработчик сборщика мусора в .NET, работает в Майкрософт. Ведет блог по адресу https://blogs.msdn.microsoft.com/maoni/.

Благодарности

Во-первых, хочу очень, очень сильно поблагодарить свою жену. Без ее поддержки эта книга никогда не появилась бы на свет. Начиная работу над книгой, я даже представить не мог, каким количеством совместно проведенных часов придется пожертвовать. Спасибо тебе за терпение, поддержку и ободрение, которые ты дарила мне на протяжении всего этого времени!

Во-вторых, я хочу выразить благодарность Маони Стивенс за развернутые, точные и бесценные замечания к первым вариантам рукописи. Без тени сомнения могу утверждать, что благодаря ей книга стала лучше. А то, что ведущий разработчик сборки мусора в .NET помогала при написании этой книги, – само по себе награда для меня! Большое спасибо и другим членам команды .NET, принимавшим участие в рецензировании некоторых частей книги, привлечь которых удалось при помощи Маони. Перечисляю их в соответствии с количеством вложенного труда: Стивен Тоуб (Stephen Toub), Джаред Парсонс (Jared Parsons), Ли Калвер (Lee Culver), Джош Фри (Josh Free), Омар Тофик (Omar Tawfik). Спасибо также Марку Пробсту (Mark Probst) из компании Xamarin, который отредактировал замечания об исполняющей среде Mono. Особая благодарность Патрику Дассуду (Patrick Dussud), «отцу .NET GC», за то, что он нашел время отрецензировать исторический обзор создания CLR.

В-третьих, я признателен Дамьену Фоггону, техническому редактору от издательства Apress, который вложил столько труда в редактирование всех глав. Его бесценный опыт авторской и издательской деятельности помог сделать изложение более понятным и последовательным. Не раз и не два я поражался точности комментариев и предложений Дамьена!

Очевидно, что я благодарен всему коллективу издательства Apress, без которого книга вообще не вышла бы в свет. Отдельное спасибо Лауре Берендсон (редактор-консультант), Нэнси Чен (редактор-координатор) и Джоан Маррей (старший редактор), которые поддерживали меня и терпели бесконечные переносы сроков. В течение какого-то периода даже само упоминание даты сдачи окончательного варианта в наших разговорах было под запретом! Я также благодарен Гвенан Спиринг, с которой начинал работать над книгой, но не смог закончить, потому что она ушла из Apress.

Я очень благодарен сообществу .NET в Польше и во всем мире за идеи, которые черпал из многочисленных презентаций, статей и постов, за ободрение и поддержку и бесконечные вопросы о том, как продвигается книга. Особенно я признателен следующим лицам (перечислены в алфавитном порядке): Мачей Анисерович (Maciej Aniserowicz), Аркадиуш Бенедикт (Arkadiusz Benedykt), Себастьян Гебский (Sebastian Gębski), Михал Гжегоржевский (Michał Grzegorzewski), Якуб Гутковский (Jakub Gutkowski), Павел Климчик (Paweł Klimczyk), Шимон Кулец (Szymon Kulec), Павел Лукашик (Paweł Łukasik), Алисия Мусяу (Alicja Musiał), Лукаш Ольбромский (Łukasz Olbromski), Лукаш Пыржик (Łukasz Pyrzyk), Бартек Сокыл (Bartek Sokył), Себастьян Солница (Sebastian Solnica), Павел Срочиньский (Paweł Sroczyński), Ярек Стадницкий (Jarek Stadnicki), Пётр Стапп (Piotr Stapp),

Михал Сливонь (Michał Śliwoń), Шимон Варда (Szymon Warda) и Артур Винченчак (Artur Wincenciak), все обладатели звания MVP, и многие другие. Искренне прошу прощения у тех, кого не упомянул, огромное спасибо всем, кто в этом нуждается. Перечислить всех просто невозможно. Все вы вдохновляли и поддерживали меня.

Хочу также поблагодарить всех опытных авторов, которые нашли время, чтобы поделиться советом о том, как писать книги, в том числе Тэда Ньюэрда (Ted Neward) (http://blogs.tedneward.com/) и Джона Скита (Jon Skeet) (https://codeblog.jonskeet.uk), хотя готов побиться об заклад, что они сами этих разговоров не помнят! Анджей Кшивда (Andrzej Krzywda) (http://andrzejonsoftware.blogspot.com) и Гынваэль Кольдвинд (Gynvael Coldwind) (https://gynvael.coldwind.pl) также дали мне много советов по написанию и публикации книги.

Далее я благодарю создателей всех тех замечательных инструментов и библиотек, которыми пользовался при написании этой книги: Андрея Щелкина, автора SharpLab (https://sharplab.io), Андрея Акиньшина, автора BenchmarkDotNet (https://benchmarkdotnet.org), и Адама Ситника, отвечающего за ее сопровождение, Сергея Теплякова, автора ObjectLayoutInspector (https://github.com/SergeyTeplyakov/ObjectLayoutInspector), 0xd4d, анонимного создателя dnSpy (https://github.com/0xd4d/dnSpy), Сашу Голдштейна, автора множества полезных инструментов (https://github.com/goldshtn), а также создателей таких великолепных программ, как PerfView и WinDbg (и их расширений, относящихся к .NET).

Я очень признателен своему бывшему работодателю Банку Миллениум, который оказывал мне помощь и поддержку, когда я только приступал к написанию книги. Наши пути разошлись, но я всегда буду помнить, что именно там я начал писать, вести блог и выступать с докладами. Большое спасибо всем моим тогдашним коллегам за ободрение и вопросы «как продвигается книга?» – это здорово мотивирует.

Еще раз спасибо всем анонимным пользователям Твиттера, которые откликались на мои обзоры книг и подсказывали, что интересно, полезно и ценно для нашей семьи .NET, а что – не очень.

И наконец, общее спасибо всей моей семье и друзьям, которым я не мог уделять достаточно внимания, пока был занят книгой.

Предисловие

Когда я вошла в команду разработчиков общеязыковой среды выполнения CLR (исполняющей среды .NET) – тому уже больше десяти лет, – я даже не думала, что компонент под названием «сборщик мусора» (Garbage Collector – GC) станет темой, над которой я буду размышлять большую часть времени, когда не сплю. Среди тех, с кем я работала, был и Патрик Дассуд – архитектор и разработчик CLR GC с момента его создания. Понаблюдав за моей работой на протяжении нескольких месяцев, он передал мне факел, и я стала вторым лицом, отвечающим только за разработку GC для CLR.

Так началось мое путешествие в мир GC. Вскоре я открыла для себя очарование мира сборки мусора – я была поражена сложностью и обширностью возникающих задач и полюбила находить для них эффективные решения. Поскольку количество различных сценариев использования CLR росло, как и число пользователей, а память – один из самых важных аспектов производительности, постоянно возникали новые проблемы в части управления памятью. Когда я начинала работать, редкостью было встретить кучу GC размером хотя бы 200 МБ, сегодня и 20 ГБ не считается чем-то из ряда вон выходящим. Некоторые из самых больших рабочих нагрузок в мире работают на CLR. Как лучше управлять памятью в этих условиях – без сомнения, животрепещущая проблема.

В 2015-м мы раскрыли исходный код CoreCLR. Когда об этом было объявлено, сообщество спрашивало, будет ли код GC исключен из репозитория CoreCLR, – справедливый вопрос, поскольку в нашем GC было много инновационных механизмов и политик. Ответом было решительное «нет», в репозитории находится тот самый код, который используется в CLR. Безусловно, это привлекло некоторых любознательных личностей. Год спустя я с восторгом узнала, что один из наших пользователей планирует написать книгу, посвященную исключительно нашему GC. Когда технологический евангелист из нашего польского отделения спросил, смогу ли я отредактировать книгу Конрада, конечно, я согласилась!

Получая главы от Конрада, я поняла, что он изучил наш код GC со всей тщательностью. Я была поражена детальностью изложения. Конечно, вы можете самостоятельно собрать CoreCLR и пройтись по коду GC в пошаговом режиме. Но эта книга определенно упростит вам задачу. А поскольку важной частью читательской аудитории являются пользователи GC, Конрад включил много материала, который поможет лучше понять поведение GC и способы кодирования, повышающие эффективность его использования. Кроме того, в начале книги имеется основополагающая информация о памяти, а ближе к концу – обсуждение характера использования памяти в различных библиотеках. Я полагаю, что Конрад нашел идеальный баланс между введением в GC, описанием его внутренних механизмов и применения.

Если вы пользуетесь .NET и думаете об эффективном использовании памяти или просто интересуетесь тем, как устроен .NET GC, то эта книга для вас. Надеюсь, что вы получите от чтения не меньшее удовольствие, чем я от редактирования.

Маони Стивенс,
июль 2018

Введение

В информатике память была всегда – перфокарты, магнитные ленты и, наконец, современные высокотехнологичные микросхемы динамических ОЗУ, DRAM. И так будет всегда – быть может, в форме голографических чипов из научно-фантастических романов или еще более удивительных вещей, которые мы даже представить себе сейчас не можем. И конечно, тому есть основательные причины. Хорошо известно определение компьютерных программ как объединения алгоритмов и структур данных. Мне эта формулировка очень нравится. Наверное, все хотя бы раз слышали о книге Никласа Вирта «Алгоритмы + структуры данных = программы» (Prentice Hall, 1976), где она впервые была введена в обращение.

С момента оформления программной инженерии как дисциплины вопрос управления памятью считался приоритетным. Уже конструкторы самых первых вычислительных машин вынуждены были задумываться о памяти для алгоритмов (программного кода) и структур данных (программных данных). Всегда уделялось большое внимание тому, как эти данные загружаются и где сохраняются для дальнейшего использования.

В этом отношении программная инженерия и управление памятью всегда были тесно связаны – так же, как программная инженерия и алгоритмы. И я полагаю, так всегда и будет. Память – ограниченный ресурс, и всегда таковым останется. Поэтому в той или иной степени память будет занимать умы будущих разработчиков. Если ресурс ограничен, то всегда возможны ошибки или неправильное использование, приводящие к его истощению. Память – не исключение из этого правила.

Но при всем при том в управлении памятью есть один постоянно меняющийся аспект – объем. Первые разработчики, а правильнее было бы называть их инженерами, знали о каждом бите в своих программах. В то время в их распоряжении было всего несколько килобайтов памяти. Каждое десятилетие эта величина росла, и сегодня мы живем в эпоху гигабайтов, а в дверь уже стучатся терабайты и петабайты. Вместе с увеличением объема уменьшается время доступа, что позволяет обработать все эти данные за разумное время. Но хотя можно сказать, что память стала быстрой, непритязательные алгоритмы управления памятью, которые пытаются обработать гигабайты данных безо всяких оптимизаций и сложных настроек, обречены на провал. Связано это прежде всего с тем, что время доступа к памяти снижается медленнее, чем вычислительная мощность процессоров, которые с этой памятью работают. Необходимо принимать специальные меры, чтобы доступ к памяти не стал узким местом, ограничивающим мощность современных процессоров.

Это не только придает управлению памятью первостепенную важность, но и делает его по-настоящему чарующей частью информатики. А автоматическое управление памятью – вещь еще более интересная. Оно отнюдь не сводится к пожеланию «освободим-ка неиспользуемые объекты». Что, как и когда – эти простые аспекты управления памятью делают его непрерывным процессом совершенствования старых и изобретения новых алгоритмов. Бесчисленные научные

статьи и диссертации посвящены вопросу о том, как организовать автоматическое управление памятью оптимальным образом. На таких мероприятиях, как Международный симпозиум по управлению памятью (International Symposium on Memory Management – ISMM), подводится итог достижениям в этой области за год: сборке мусора, динамическому выделению, взаимодействиям с исполняющей средой, компиляторами и операционными системами. А затем академические исследования превращаются в коммерческие и открытые продукты, которыми мы пользуемся каждый день.

.NET – идеальный пример управляемой среды, в которой все эти сложности глубоко скрыты и представлены разработчикам в виде приятной и готовой к использованию платформы. Действительно, мы пользуемся всем этим, не подозревая о глубинной сложности, что само по себе является крупным достижением .NET. Однако чем сильнее наша программа нуждается в высокой производительности, тем меньше шансов обойтись без знания о том, как и почему все работает внутри .NET. А что касается меня лично, так мне просто интересно, как устроены вещи, которыми мы пользуемся ежедневно!

Я написал эту книгу такой, какой хотел бы прочитать ее много лет назад, когда только начинал путешествие в мир производительности и диагностики .NET. То есть она начинается не с типичного введения в организацию кучи или стека и не с описания нескольких поколений. Вместо этого я решил начать с самых основ управления памятью. Иными словами, я старался писать так, чтобы вы почувствовали, сколь интересна эта тема, а не ограничиваться констатациями типа «вот это сборщик мусора .NET, и делает он то-то и то-то». Информация не только о том, что, но и о том, как, а главное – почему, поможет вам по-настоящему понять, что происходит за кулисами управления памятью в .NET. А тогда все, что вы будете читать на эту тему в будущем, станет яснее. Я пытался вооружить вас знаниями не только о .NET, особенно в первых двух главах. Это способствует более глубокому проникновению в тему, что часто оказывается полезно в применении к другим задачам программной инженерии (благодаря лучшему пониманию алгоритмов, структур данных и просто добротной инженерной работы).

Я хотел написать книгу, так чтобы ее было приятно читать любому разработчику для .NET. Каким бы ни был ваш предшествующий опыт, что-то интересное для себя вы здесь найдете. Мы начинаем с азов, но младшие программисты довольно быстро получат возможность глубже заглянуть в устройство .NET. Более опытным программистам будут интересны различные детали реализации. И кроме того, каждый, вне зависимости от опытности, получит пользу от изучения практических примеров кода и диагностики проблем.

Таким образом, знания, полученные из этой книги, должны помочь вам писать более качественный код, в котором связанные с памятью средства используются не слепо, а с полным пониманием дела. Это также повысит производительность и масштабируемость приложений – чем сильнее код ориентирован на правильную работу с памятью, тем меньше шансов на возникновение узких мест и неоптимальное использование ресурсов. Я надеюсь, что прочтение книги оправдает подзаголовок «Написание более качественного, производительного и масштабируемого кода».

Я также надеюсь, что все это сделает книгу более общей и долговечной, чем простое описание текущего состояния .NET Framework и его внутреннего устрой-

ства. Как бы ни развивалась платформа .NET в будущем, я верю, что большая часть изложенного в этой книге материала, наверное, сохранит актуальность еще долго. Даже если какие-то детали реализации изменятся, почерпнутые из этой книги знания позволят вам без труда понять их. Просто потому, что основополагающие принципы меняются не так быстро. Желаю вам приятного путешествия по огромному и увлекательному миру автоматического управления памятью!

Вместе с тем я хотел бы подчеркнуть, что некоторые вещи представлены в книге недостаточно полно. Тема управления памятью, хотя и кажется очень узкой и специальной, на удивление широка. И хотя я затронул много вопросов, иногда они из-за недостатка места изложены не так подробно, как мне хотелось бы. Но даже при таких ограничениях книга заняла больше 1000 страниц! К числу опущенных тем относятся, например, исчерпывающие ссылки на другие управляемые среды (Java, Python, Ruby и т. п.). Приношу также извинения поклонникам F# за недостаточное количество ссылок на материалы по этому языку. Просто мне не хватило страниц для серьезного описания, а публиковать что-то менее полное не хотелось. Я хотел бы также уделить больше внимания среде Linux, но на момент написания книги эта тема была еще совсем новой, а инструментов не хватало, поэтому я привел лишь краткие предложения в главе 3 (и по той же причине полностью проигнорировал мир macOs). Понятно, что я вынужден был опустить большую часть вопросов производительности в .NET, не связанных напрямую с памятью, например многопоточность.

И еще – хотя я приложил все усилия, чтобы показать практические применения обсуждаемой теории и технических приемов, не всегда возможно сделать это без примеров. Практических применений попросту слишком много. Поэтому я ожидаю, что читатель будет работать с книгой внимательно, всесторонне осмысливать каждую тему и применять полученные знания в повседневной работе. Поймите, как что-то устроено, – и тогда сможете это использовать!

Особенно это относится к так называемым сценариям. Прошу иметь в виду, что все включенные в книгу сценарии приведены в иллюстративных целях. Их код сведен к абсолютному минимуму, чтобы было проще продемонстрировать глубинную причину какой-то одной проблемы. У наблюдаемого неправильного поведения могут быть и другие причины (например, есть много способов заметить утечку управляемой памяти). Сценарии были подготовлены так, чтобы помочь в иллюстрации подобных проблем на одном примере, поскольку, очевидно, невозможно рассмотреть все мыслимые причины в одной книге. К тому же в реальной ситуации расследование может быть осложнено кучей отвлекающих данных и ложными путями поиска. Зачастую не существует единственного способа решения описанных задач, и тем не менее есть много способов найти истинную причину проблемы в процессе анализа. Поэтому поиск причин ошибки становится гибридом чисто инженерной задачи с искусством, подкрепленным интуицией. Обратите внимание, что сценарии иногда ссылаются друг на друга, чтобы не повторять снова и снова одни и те же шаги, рисунки и описания.

Я специально воздерживался от упоминания различных случаев и источников проблем, характерных для конкретных технологий. Они просто … ну, слишком характерны для конкретных технологий. Если бы я писал эту книгу 10 лет назад, то, наверное, вынужден был бы перечислить типичные сценарии утечки памяти в ASP.NET WebForms и WinForms. Несколько лет назад? ASP.NET MVC, WPF, WCF,

WF… А теперь? ASP.NET Core, EF Core, Azure Functions, что еще? Надеюсь, вы поняли, о чем я. Такие знания слишком быстро устаревают. Книга, под завязку набитая примерами утечек памяти в WCF, сегодня вряд ли кому-то будет интересна. Я очень люблю слова: «Дай человеку рыбу – и он будет сыт один день. Дай человеку удочку – и он будет сыт всю жизнь». Поэтому все знания, изложенные в этой книге, все сценарии служат одной цели – научить человека ловить рыбу. Все проблемы, вне зависимости от технологии, можно диагностировать единообразно, если вы обладаете знаниями и понимаете, как их применять.

Все это также предъявляет большие требования к читателю, поскольку подчас текст изобилует деталями и, быть может, немного перегружен информацией. Но, несмотря ни на что, я советую читать медленно и вдумчиво, не поддаваясь искушению что-то просмотреть по диагонали. Так, чтобы извлечь из книги максимум пользы, следует внимательно изучать код и рисунки (а не просто бросить на них мимолетный взгляд, считая, что все очевидно и можно без ущерба пропустить).

Мы живем в замечательное время, когда исходный код среды выполнения CoreCLR открыт. Это дает качественно новые возможности для изучения и понимания CLR. Не осталось места ни догадкам, ни тайнам. Все есть в коде, все можно прочитать и понять. Поэтому мои исследования внутренних механизмов управления памятью основаны на коде GC из CoreCLR (общем с .NET Framework). Я потратил бесчисленные дни и недели, изучая огромный объем отлично сделанной инженерной работы. Я нахожу ее великолепной и полагаю, что найдутся и другие люди, которые захотят изучить знаменитый файл gc.cpp, насчитывающий десятки тысяч строк кода. Но кривая обучения будет очень крутой. Чтобы помочь вам, я часто оставляю подсказки – с чего начать изучение кода CoreCLR, относящегося к описываемым темам. Ничто не мешает вам еще глубже изучить вопрос, начав с предложенных мной мест в gc.cpp!

Прочитав эту книгу, вы научитесь:

- писать для .NET код с учетом моментов, относящихся к производительности и памяти. Все примеры в книге написаны на C#, но, разобравшись с ними и овладев описанным инструментарием, вы сможете применить полученные знания также к коду на F# и VB.NET;
- диагностировать типичные проблемы, связанные с управлением памятью в .NET. Поскольку большая часть методов основана на данных ETW/LTTng и расширении SOS, они равно применимы к Windows и Linux (правда, в Windows набор инструментов гораздо богаче);
- понимать, как CLR работает в части управления памятью. Я уделил много внимания объяснению не только того, как все устроено, но и почему устроено именно так;
- читать и понимать массу интересных обсуждений, посвященных C# и исполняющей среде CLR на GitHub, и даже принимать в них участие;
- читать код GC в CoreCLR (особенно файл gc.cpp), понимая достаточно, чтобы заниматься дальнейшими исследованиями;
- читать и понимать публикации о сборке мусора и управлении памятью в других средах, например Java, Python и Go.

Теперь конкретно о содержании книги. Глава 1 представляет собой очень общее теоретическое введение в управление памятью, почти без ссылок на особенности .NET. Глава 2 – общее введение в управление памятью на уровне аппаратных

средств и операционной системы. Обе главы можно рассматривать как важное, но все-таки факультативное введение. Они дают полезный, более широкий взгляд на тему и пригодятся в остальной книге. Хотя я, конечно, настоятельно рекомендую прочитать их, можете обойтись без этого, если очень спешите или интересуетесь только сугубо практическими, относящимися к .NET вопросами. Замечание для хорошо осведомленных читателей – даже если вы думаете, что темы, изложенные в первых двух главах, вам хорошо знакомы, пожалуйста, прочитайте их. Я старался включить не только очевидную информацию, но и кое-что такое, что может показаться вам интересным.

Глава 3 целиком посвящена измерениям и разнообразным инструментам (некоторые из которых используются в книге очень часто). Текст состоит в основном из перечня инструментов и информации о том, как с ними работать. Если вас больше интересует теоретическая часть книги, можете бегло пролистать эту главу. С другой стороны, если вы планируете интенсивно использовать полученные знания для диагностики проблем, то, наверное, будете часто возвращаться к ней.

В главе 4 мы впервые начнем предметный разговор о .NET, хотя пока еще общего характера – с целью понять некоторые относящиеся к теме внутренние механизмы, в т. ч. систему типов (включая различия между значимыми и ссылочными типами), интернирование строк и статические данные. Если вы очень торопитесь, то имеет смысл начать чтение отсюда. В главе 5 рассматривается первый по-настоящему относящийся к памяти вопрос – как организована память в приложениях для .NET, включая понятия кучи малых и больших объектов, а также сегментов. В главе 6 обсуждение внутренних механизмов памяти продолжится, речь пойдет исключительно о выделении памяти. Удивительно, что такая длинная глава может быть посвящена столь простому, с теоретической точки зрения, вопросу. Важной и большой частью главы является описание различных источников выделения – а точнее того, как их избегать.

Главы с 7 по 10 содержат описание основных механизмов работы GC в .NET, с практическими примерами и выводами, вытекающими из полученных знаний. Чтобы не вываливать на бедного читателя всю информацию одновременно, в этих главах описывается самый простой вариант GC – неконкурентный, для рабочей станции (Workstation Non-Concurrent). А вот глава 11 посвящена описанию всех остальных вариантов с подробными соображениями о том, какой выбрать. Глава 12 завершает часть книги, посвященную GC, описанием трех важных механизмов: финализации, уничтожаемых объектов и слабых ссылок.

Последние три главы составляют «продвинутую» часть книги в том смысле, что в них обсуждается, как работают механизмы, выходящие за пределы базового управления памятью в .NET. Так, в главе 13 рассматривается вопрос об управляемых указателях и далее о структурах (включая недавно добавленные ссылочные структуры). В главе 14 много внимания уделено типам и техническим приемам, которые в последнее время приобретают все более широкую популярность, в частности `Span<T>` и `Memory<T>`. Имеется также раздел, посвященный не столь известной теме проектирования, ориентированного на данные, и несколько слов сказано об ожидаемых новшествах в C# (ссылочные типы, допускающие и не допускающие null, и конвейеры (pipelines)). В главе 15, последней, описываются различные способы управления и мониторинга работы GC из кода программы, включая API класса GC, размещение CLR и библиотеку ClrMD.

Большая часть приведенных в книге листингов доступна в сопроводительном репозитории на GitHub по адресу https://github.com/Apress/pro-.net-memory. Он организован по главам, и для большинства глав включено два решения: одно для проведенных тестов, а второе содержит прочие листинги. Обратите внимание, что хотя для включенных в книгу проектов имеются листинги, часто доступный вам код не ограничивается ими. Если вы хотите поэкспериментировать с конкретным листингом, проще всего найти его по номеру. Но я также рекомендую познакомиться с проектами по различным темам для лучшего понимания предмета.

Не так уж много есть важных соглашений, о которых я хотел бы здесь упомянуть. Самое существенное – различать две главные концепции, встречающиеся далее в этой книге:

- сборка мусора (GC) – понятный в общих чертах процесс освобождения более не нужной памяти;
- сборщик мусора (GC) – конкретный механизм реализации сборки мусора, очевидно, в контексте .NET[1].

Эта книга в значительной мере автономная, ссылок на другие материалы и книги в ней немного. Но понятно, что в мире существует великое множество знаний, и мне бы надо было очень часто ссылаться на различные источники. Вместо этого я перечислю книги, которые, на мой взгляд, могут служить дополнительными источниками информации:

- Sasha Goldshtein, Dima Zurbalev, Ido Flatow «Pro .NET Performance» (Apress, 2012);
- Jeffrey Richter «CLR via C#» (Microsoft Press, 2012);
- Ben Watson «Writing High-Performance .NET Code» (Ben Watson, 2014);
- Mario Hewardt «Advanced .NET Debugging» (Addison-Wesley Professional, 2009);
- Serge Lidin «.NET IL Assembler» (Microsoft Press, 2012);
- David Stutz «Shared Source CLI Essentials» (O'Reilly Media, 2003);
- «Book Of The Runtime», открытая документация, разрабатываемая параллельно самой исполняющей среде, доступна по адресу https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md.

Огромный объем информации доступен также в различных блогах и статьях. Но я не буду загромождать их перечнем страницы этой книги, а переадресую на замечательный репозиторий https://github.com/adamsitnik/awesome-dot-net-performance, поддерживаемый Адамом Ситником.

[1] В оригинале употребляются термины GC и «the GC». Передать это различие в русском языке невозможно, но обычно из контекста понятно, о чем идет речь. – *Прим. перев.*

От издательства

Отзывы и пожелания

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

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

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

Список опечаток

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

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

Скачивание исходного кода

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Нарушение авторских прав

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

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

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

Глава 1

Основные понятия

Начнем с простого, но очень важного вопроса. Когда следует задумываться об управлении памятью в .NET, если оно полностью автоматизировано? И надо ли задумываться вообще? Вы, наверное, понимаете, что раз уж я написал такую книгу, то настоятельно рекомендую помнить о памяти в любой ситуации. Это просто вопрос нашего профессионализма. Следствие того, как мы строим свою работу. Пытаемся ли мы сделать наилучшим образом или просто сделать? Если мы думаем о качестве своей работы, то недостаточно, чтобы произведение нашего труда просто работало. Важно еще, и как оно работает. Оптимально ли оно использует процессор и память? Удобно ли оно для сопровождения и тестирования, открыто ли для расширения и закрыто для модификации? Отвечает ли наш код принципам SOLID? Полагаю, что такие вопросы отличают начинающего от опытного программиста. Первый думает в основном о том, чтобы код работал, и не печется о вышеупомянутых нефункциональных аспектах своей деятельности. У второго достаточно «ментальной вычислительной мощности», чтобы задумываться о качестве работы. Думается мне, что все хотели бы быть такими. Но, конечно, это не тривиально. Писать элегантный код, который не содержит ошибок и удовлетворяет всем нефункциональным требованиям, действительно трудно.

Но верно ли, что только из стремления к мастерству стоит приобретать глубокие знания об управлении памятью в .NET? Повреждения памяти, ведущие к исключению `AccessViolationException`, случаются крайне редко[1]. Неконтролируемый рост потребления памяти – вроде бы тоже нечастое событие. Так есть ли повод для беспокойства? Поскольку среда выполнения .NET тщательно реализована Майкрософт, то, к счастью, нам не приходится много думать о памяти. С другой стороны, при анализе проблем с производительностью крупных приложений на базе .NET проблемы потребления памяти всегда стояли на одном из первых мест. Станет ли проблемой утечка памяти после нескольких дней непрерывной работы? В интернете можно найти забавный мем об утечке памяти в программном обеспечении некой боевой ракеты, которую не стали устранять, потому что ракета раньше поразит цель, чем кончится память. Является ли наша система такой ракетой одноразового использования? Можем ли мы сказать, чревато автоматизированное управление памятью в нашем приложении серьезными накладными

---

[1] `AccessViolationException` и другие повреждения памяти часто возбуждаются механизмом автоматического управления памятью не потому, что он является причиной, а потому, что это самый крупный компонент среды, относящийся к памяти. Поэтому именно у него наибольшие шансы обнаружить противоречивое состояние памяти.

расходами или нет? Быть может, мы могли бы обойтись всего двумя серверами вместо десяти? И кстати, память не бесплатна даже во времена бессерверных облачных вычислений. Один из примеров – служба Azure Functions, которая тарифицируется на основе показателя «гигабайт-секунды» (GB-s). Он вычисляется путем умножения среднего объема памяти в гигабайтах на время в секундах, необходимое для выполнения конкретной функции. Потребление памяти самым непосредственным образом отражается на сумме счета.

Так или иначе, мы приходим к выводу, что понятия не имеем, с чего начинать поиск причины и что измерять. И вот тогда-то приходит осознание того, что было бы полезно разбираться во внутренних механизмах работы приложения и исполняющей его среды.

Чтобы глубже понять, как устроено управление памятью в .NET, лучше начать с азов. Не имеет значения, новичок вы или крутейший программист. Я рекомендую вместе со мной познакомиться с теоретическим введением, составляющим содержание этой главы. Тогда у нас будет общий уровень знаний и понимания, необходимый для чтения книги. Чтобы не скатываться в сухую теорию, я иногда буду давать отсылки к конкретным технологиям. Заодно у нас будет шанс кое-что узнать об эволюции программного обеспечения. Этот экскурс хорошо ложится на развитие концепций, относящихся к управлению памятью. Мы также отметим несколько любопытных фактов, которые, надеюсь, будут вам интересны. Знание истории неизменно является одним из лучших способов шире взглянуть на вопрос.

Но не пугайтесь – это не книга об истории. Я не буду приводить биографии всех ученых, занимавшихся разработкой алгоритмов сборки мусора, начиная с 1950 года. Знания древней истории тоже не понадобятся. Тем не менее я льщу себя надеждой, что вам будет интересно узнать, как эта дисциплина развивалась и где мы сейчас находимся. Это позволит нам сравнить принятый в .NET подход со многими другими языками и исполняющими средами, о которых вы, вероятно, наслышаны.

Терминология, относящаяся к памяти

В самом начале будет полезно привести ряд очень важных определений, без которых трудно представить себе обсуждение вопроса о памяти.

- *Бит* – самая маленькая единица информации в компьютерных технологиях. Представляет два возможных состояния, которые обычно интерпретируются как числовые значения 0 и 1 или логические значения true и false. В главе 2 мы вкратце опишем, как в современных компьютерах хранятся одиночные биты. Для представления больших чисел необходимо использовать комбинации битов, кодирующие двоичное значение числа, как будет описано ниже. Если требуется выразить размер данных в битах, употребляется буква b.
- *Двоичное число* – целое число, представленное в виде последовательности битов. Каждый последующий бит описывает вклад соответствующей степени 2 в сумму данного значения. Например, для представления числа 5 можно использовать три последовательных бита со значениями 1, 0 и 1,

поскольку 1×1 + 0×2 + 1×4 равно 5. С помощью *n* бит можно представить натуральные числа до 2^*n* – 1 включительно. Часто один дополнительный бит используют для представления знака, чтобы различать положительные и отрицательные значения. Есть также другие, более сложные способы двоичного кодирования числовых значений, особенно чисел с плавающей точкой.

- *Двоичный код* – последовательность битов может представлять не только числовые значения, но и другие данные, например символы текста. Каждому символу сопоставляется своя последовательность битов. Самой простой кодировкой, бывшей самой популярной на протяжении многих лет, является ASCII, которая использовала 7-битовый двоичный код для представления букв и других символов. Есть и другие важные двоичные коды, например *коды операций*, кодирующие команды, выполняемые компьютером.
- *Байт* – исторически так называлась последовательность битов для двоичного кодирования одного символа текста. Чаще всего встречаются 8-битовые байты, хотя размер зависит от архитектуры компьютера и может отличаться для разных архитектур. Из-за такой неоднозначности существует более точный термин – *октет*, означающий блок данных размером точно 8 бит. Но все же 8-битовый байт – это стандарт де-факто, и в таком качестве он используется для задания размера данных. В настоящее время практически невозможно встретить архитектуру, в которой размер байта был бы другим. Если требуется выразить размер данных в байтах, употребляется буква В.

При задании размера данных мы используем префиксы, определяющие порядок величины. Тут постоянно возникает путаница, поэтому пояснения будут нелишними. Такие повсеместно распространенные префиксы, как кило-, мега- и гига-, обозначают степень 1000. *Кило-* – это 1000 (обозначается буквой k), *мега-* – 1 миллион (буква М) и т. д. Но есть и другой, не менее популярный подход – выражать порядок величины степенями 1024. В таком случае употребляют префиксы *киби-* – 1024 (обозначается Ki), *меби-* – 1024*1024 (обозначается Mi), *гиби-* (Gi) – 1024*1024*1024 и т. д. Это вносит неразбериху. Говоря «1 гигабайт», человек может иметь в виду 1 миллиард байт (1 ГБ) или 1024^3 байт (1 ГиБ) – в зависимости от контекста. На практике редко кому интересна точная интерпретация этих префиксов. Емкость модулей памяти для компьютеров практически всегда выражается в гигабайтах (ГБ), хотя на самом деле имеются в виду гибибайты (ГиБ), а в спецификации емкости жестких дисков дело обстоит ровно наоборот. Даже в стандарте JEDEC 100B.01 «Термины, определения и буквенные символы, относящиеся к микрокомпьютерам, микропроцессорам и интегральным схемам памяти» общепринятые буквы K, M и G определяются как умножение на 1024, и их употребление явно не осуждается. В таких ситуациях нам остается только прибегнуть к здравому смыслу и интерпретировать префиксы в зависимости от контекста.

Мы все уже привыкли к употреблению таких терминов, как RAM (ОЗУ[1]) или постоянная память, для обозначения памяти, установленной в компьютере. Даже

---

[1] На самом деле RAM (Random Access Memory) переводится как «запоминающее устройство с произвольной выборкой (ЗУПВ)», но эта аббревиатура уже давно уступила месту ОЗУ (оперативное запоминающее устройство), хотя это и не совсем правильно. – *Прим. перев.*

умные часы теперь оснащаются ОЗУ емкостью 8 ГиБ. Легко забыть, что в первых компьютерах такой роскоши не было. Можно даже сказать, что они вообще не оснащались памятью. Беглый обзор истории развития компьютеров позволит по-другому взглянуть на саму память. Начнем с самого начала.

Следует иметь в виду, что вопрос о том, какое устройство можно назвать «самым первым компьютером», весьма спорный. И не менее трудно назвать имя единственного «изобретателя компьютера». Проблема в определении того, что такое «компьютер». Поэтому не будем вдаваться в бесконечные споры о том, что и кто был первым, а просто посмотрим, что предлагали программистам прежние машины, хотя до появления самого слова *программист* должно было пройти еще много лет. Поначалу они назывались *кодировщиками*, или *операторами*.

Следует подчеркнуть, что вычислительные машины, которые можно было бы назвать первыми компьютерами, были не электронными, а электромеханическими. Поэтому работали они очень медленно и, несмотря на внушительный размер, предлагали совсем немного. Первый такой программируемый электромеханический компьютер был спроектирован в Германии Конрадом Цузе и назван Z3. Он весил тонну! Операция сложения занимала одну секунду, а операция умножения – целых три секунды! Машина состояла из 2000 электромеханических реле и содержала арифметическое устройство, способное только складывать, вычитать, умножать, делить и извлекать квадратный корень. Арифметическое устройство включало два регистра памяти размером 22 бита, используемых для вычислений. Предлагалось также 64 ячейки памяти общего назначения тоже длиной 22 бита. Сегодня мы бы сказали, что машина была оснащена 176 байтами внутренней памяти для данных!

Данные вводились с помощью специальной клавиатуры, а программа считывалась в процессе вычислений с перфорированной целлулоидной пленки. Возможность хранить программу во внутренней памяти компьютера была реализована спустя несколько лет, мы еще дойдем до этого, но уже Цузе эта идея была известна. Однако в данной книге нам важнее вопрос о доступе к памяти в Z3. Для программирования Z3 в нашем распоряжении было всего девять команд! Одна из них позволяла загрузить значение из одной из 64 ячеек памяти в регистр арифметического устройства, другая – сохранить значения обратно в ячейку. И это все, что можно назвать «управлением памятью» в этом первом компьютере. Хотя Z3 во многих отношениях опередил свое время, по политическим причинам и из-за Второй мировой войны его влияние на развитие компьютеров оказалось пренебрежимо малым. Цузе разрабатывал свою линейку компьютеров еще много лет после войны, а последняя версия под номером Z22 была построена в 1955 году.

Во время войны и спустя короткое время после нее основными центрами развития информатики были США и Великобритания. Одним из первых компьютеров, построенных в США, был Harvard Mark I, созданный совместно компанией IBM и Гарвардским университетом. Он получил название «автоматический вычислитель, управляемый последовательностями» (Automatic Sequence Controlled Calculator). Как и Z3, он был электромеханическим. Размеры его были огромны – 17 м в длину, более 2,5 м в высоту и чуть меньше метра в глубину. А весил он 5 т! Его называют самой большой вычислительной машиной всех времен. Строили его несколько лет, а первые программы были выполнены в конце войны, в 1944 году. Он обслуживал интересы ВМС, а также Джона фон Неймана, когда тот рабо-

тал над первой атомной бомбой в рамках Манхэттенского проекта. Несмотря на свои размеры, он предлагал только 72 ячейки памяти для чисел с 23 десятичными разрядами со знаком. Ячейка называлась *аккумулятором* и представляла собой место в памяти, где хранились промежуточные результаты арифметических и логических вычислений. В современных терминах мы бы сказали, что эта 5-тонная машина предоставляла доступ к 72 ячейкам памяти длиной 78 бит (для представления 23-разрядного десятичного числа со знаком нужно 78 бит), т. е. имела 702 байта памяти! Программы представляли собой последовательности математических вычислений над этими 72 ячейками памяти. То были языки программирования первого поколения (1GL), или машинные языки, когда программы хранились на перфоленте, которая физически подавалась машине по мере необходимости, или набирались с помощью переключателей на передней панели. Машина могла выполнять только три операции сложения или вычитания в секунду. Операция умножения занимала 20 секунд, а вычисление sin(*x*) – минуту! Как и в Z3, никакого управления памятью не было – можно было только читать или записывать одно значение в вышеупомянутые ячейки памяти.

Для нас интересно, что с этого компьютера пошел термин «гарвардская архитектура» (см. рис. 1.1), согласно которой память для программ и данных была физически разделена. Данные обрабатываются некоторым электронным или электромеханическим устройством (центральным процессором). Такое устройство часто отвечает также за управление устройствами ввода-вывода: устройствами считывания перфокарт, клавиатурами или устройствами отображения. Хотя в компьютерах Z3 и Mark I эта архитектура применялась из-за ее простоты, она еще не забыта и в наши дни. В главе 2 мы увидим, что практически во всех современных компьютерах используется *модифицированная гарвардская архитектура*. И мы даже увидим, как она влияет на программы, которые мы пишем каждый день.

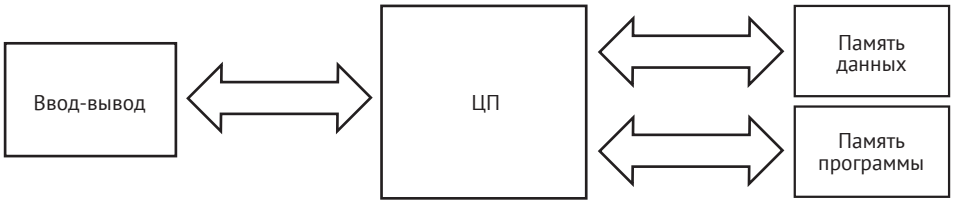

Рис. 1.1 ❖ Гарвардская архитектура

Гораздо более широко известный компьютер ЭНИАК, законченный в 1946 году, был уже полностью электронным устройством на электровакуумных лампах. Скорость выполнения математических операций была в несколько тысяч раз выше, чем у Mark I. Но с точки зрения памяти он выглядел столь же непривлекательно. Предлагалось только двадцать 10-разрядных аккумуляторов со знаком, а внутренней памяти для хранения программ не было. Если говорить по-простому, то во время Второй мировой войны важно было как можно быстрее построить машину для военных целей, а на всякие изыски внимания не обращали.

Но такие ученые, как Конрад Цузе, Алан Тьюринг и Джон фон Нейман, исследовали идею использования внутренней памяти компьютера для хранения про-

граммы вместе с данными. Это позволило бы существенно упростить программирование (а особенно перепрограммирование) по сравнению с кодированием с помощью перфокарт или механических переключателей. В 1945 году Джон фон Нейман опубликовал оказавшую большое влияние статью «Первый проект отчета о EDVAC» (First Draft of a Report on the EDVAC), в которой описал архитектуру, получившую название *архитектура фон Неймана*. Следует отметить, что заслуга принадлежит не только фон Нейману, поскольку он опирался на труды других ученых своего времени.

Архитектура фон Неймана, показанная на рис. 1.2, является упрощенной гарвардской архитектурой, где для хранения программы и данных используется одно запоминающее устройство. Безусловно, она покажется вам похожей на архитектуру современного компьютера – и не без причины. На верхнем уровне именно так сегодня конструируются компьютеры, только гарвардская архитектура и архитектура фон Неймана слились в модифицированную гарвардскую архитектуру.

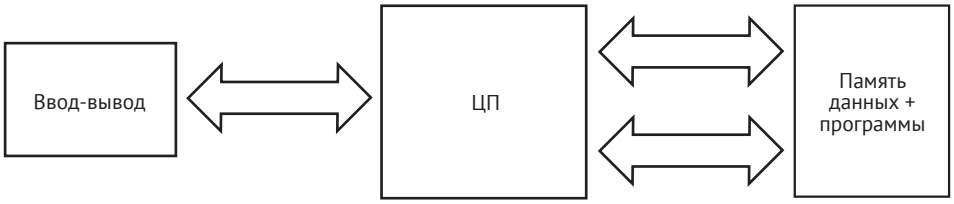

Рис. 1.2 ❖ Архитектура фон Неймана

Манчестерская малая экспериментальная машина (Small-Scale Experimental Machine – SSEM, прозванная также «Baby»), построенная в 1948 году, и Кембриджская EDSAC, построенная в 1949 году, стали первыми в мире компьютерами с хранением данных и команд программы в одном месте, т. е. организованными в соответствии с архитектурой фон Неймана. «Baby» был гораздо более современным и инновационным, поскольку в нем впервые было применено запоминающее устройство нового типа – трубки Вильямса, основанные на электронно-лучевых трубках (ЭЛТ). Трубки Вильямса можно считать первыми запоминающими устройствами с произвольным доступом (ЗУПВ, англ. RAM). В машине SSEM было 32 ячейки памяти по 32 бита каждая. Таким образом, можно сказать, что первый компьютер с ЗУПВ располагал аж 128 байтами памяти! Какое же путешествие нам предстоит – от 128 байтов в 1949 году до 16 гигабайтов в типичном компьютере 2018 года. Тем не менее трубки Вильямса стали стандартом на рубеже 1940-х и 1950-х годов, когда было построено много других компьютеров.

И теперь мы подошли к моменту, когда можем описать все основные понятия компьютерной архитектуры. Они изображены на рис. 1.3.

- *Память*, или *запоминающее устройство*, отвечает за хранение данных и самой программы. Техническая реализация памяти со временем сильно менялась – сначала были перфокарты, потом магнитные ленты и электронно-лучевые трубки, а сейчас транзисторы. Запоминающие устройства можно далее подразделить на две категории:
  - *запоминающее устройство с произвольным доступом* (*ЗУПВ*, англ. *RAM*) – время доступа к данным не зависит от их положения в памя-

ти. Как мы увидим в главе 2, современные запоминающие устройства удовлетворяют этому требованию лишь приближенно, в силу технических причин;
  - *неоднородная память* – противоположность ЗУПВ, время доступа к памяти зависит от положения на физическом носителе. Очевидно, к этой категории относятся перфокарты, магнитные ленты, классические жесткие диски, CD- и DVD-диски и прочие устройства, нуждающиеся в позиционировании (например, повороте) для приведения в нужное для доступа положение.
- *Адрес* представляет конкретное место в памяти. Обычно выражается в байтах, потому что байт – минимальная адресуемая единица на многих платформах.
- *Арифметико-логическое устройство* (*АЛУ*) отвечает за выполнение таких операций, как сложение и вычитание. Это основа компьютера, именно здесь выполняется основная работа. Современные компьютеры включают несколько АЛУ, что дает возможность распараллелить вычисления.
- *Устройство управления* (*УУ*) декодирует команды программы (коды операций), прочитанные из памяти. Исходя из внутреннего описания команды, понимает, какую операцию – арифметическую или логическую – нужно выполнить и над какими данными.
- *Регистр* – часть памяти, к которой АЛУ и УУ (общее название – *исполнительные устройства*) могут обратиться очень быстро. Обычно являются частью АЛУ или УУ. Вышеупомянутые аккумуляторы – упрощенные регистры специального назначения. Время доступа к регистрам очень мало, никакие данные не могут располагаться ближе к исполнительным устройствам, чем регистры.
- *Слово* – базовая единица данных фиксированного размера в конструкции конкретного компьютера. Встречается в описании различных конструктивных элементов, например: размер большинства регистров, максимальный адрес или длина максимального блока данных, передаваемого за одну операцию. Его величина чаще всего выражается в битах и называется *размер слова* или *длина слова*. Современные компьютеры по большей части 32- или 64-разрядные, длина слова составляет соответственно 32 или 64 бит, и таков же размер регистров и т. п.

Архитектура фон Неймана, воплощенная в машинах SSEM и EDSAC, привела к появлению термина *компьютер с хранимой программой*, который сегодня кажется очевидным, хотя в начале компьютерной эры это было далеко не так. При такой конструкции код подлежащей выполнению программы хранится в памяти, и к нему можно обращаться, как к обычным данным, в т. ч. модифицировать его и замещать кодом новой программы.

В устройстве управления имеется дополнительный регистр – *указатель команд* (instruction pointer – IP), или *счетчик команд* (program counter – PC), который указывает на текущую выполняемую команду. Нормальное выполнение программы сводится просто к увеличению адреса, хранящегося в счетчике команд, так чтобы он указывал на следующую команду. А для организации цикла или перехода нужно записать в счетчик команд другой адрес, описывающий, куда программа должна перейти.

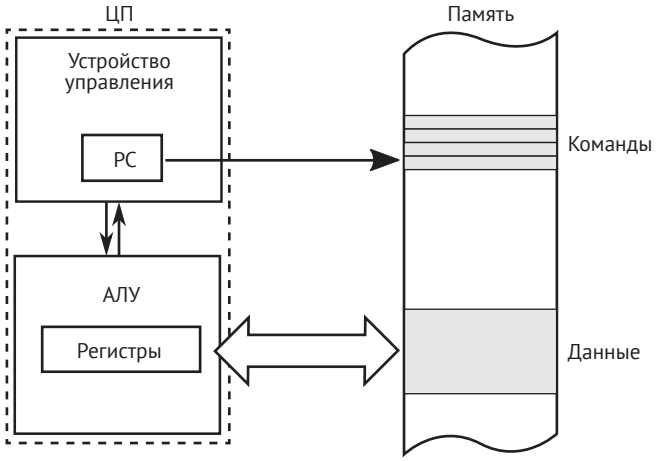

Рис. 1.3 ❖ Компьютер с хранимой программой – память + указатель команд

Первые компьютеры программировались в двоичном коде, который непосредственно описывал выполняемые команды. Но с ростом сложности программ такое решение стало чересчур обременительным. Был создан новый язык программирования (второго поколения – 2GL), позволявший описывать код более простыми средствами – в виде так называемого *ассемблерного кода*. Это текстовое и очень краткое описание команд, выполняемых процессором. Тем не менее этот язык оказался гораздо удобнее прямого двоичного кодирования. Затем были разработаны языки еще более высокого уровня (3GL), в частности хорошо известные C, C ++ и Pascal.

Нам интересно, что программы на любом из этих языков необходимо было преобразовать из текстовой формы в двоичную, а затем поместить в память компьютера. Процесс такого преобразования называется *компиляцией*, а выполняющая его программа – *компилятором*. В случае ассемблерного кода говорят об *ассемблировании* и соответственно об *ассемблере*. В конечном итоге получается программа в двоичном формате, которую можно выполнить позже, – последовательность кодов операций и их аргументов (операндов).

Вооружившись этими базовыми знаниями, мы можем начать путешествие в мир управления памятью.

Статическое выделение

Большинство первых языков программирования допускали только *статическое выделение памяти* – объем и точное расположение памяти должны быть известны на этапе компиляции, еще до начала выполнения программы. Если размер фиксирован и заранее известен, то управление памятью тривиально. Все основные «древние» языки программирования, начиная с машинного или ассемблерного кода, и до первых версий языков FORTRAN и ALGOL располагали лишь такими ограниченными возможностями. У этого способа много недостатков. Статическое выделение ведет к неэффективному использованию памяти – если мы заранее не

знаем, сколько данных предстоит обработать, то откуда же знать, сколько выделить памяти? В результате программы получаются недостаточно гибкими. В общем случае для обработки большего объема данных программу придется перекомпилировать.

В первых компьютерах выделение памяти было статическим, потому что все используемые области памяти (аккумулятор, регистры и ячейки ЗУПВ-памяти) задавались в процессе кодирования программы. Поэтому объявленные «переменные» существовали на протяжении всего времени работы программы. Сегодня статическое выделение в этом смысле используется при создании статических глобальных переменных и подобных им данных, которые хранятся в специальном сегменте данных. Позже мы увидим, где именно они хранятся в программах для .NET.

Регистровая машина

До сих пор мы видели примеры машин, в которых для вычислений в арифметико-логическом устройстве (АЛУ) использовались регистры (в частном случае аккумуляторы). Машины с такой конструкцией называются *регистровыми*, поскольку при выполнении программы на таком компьютере мы фактически производим вычисления с регистрами. Чтобы выполнить сложение, деление или еще какую-то операцию, нужно сначала загрузить данные из памяти в подходящие регистры. Затем выполняются команды, чтобы вызвать соответствующие операции над данными, и напоследок результат записывается из регистра в память.

Пусть требуется написать программу для вычисления выражения `s=x+(2*y)+z` на компьютере с двумя регистрами, `A` и `B`. Предположим также, что `s`, `x`, `y` и `z` – адреса в памяти, по которым хранятся некоторые значения. Еще предположим, что имеется некоторый низкоуровневый псевдоассемблер, в котором определены команды `Load`, `Add`, `Multiply` и т. п. Для такой гипотетической машины можно написать следующую программу.

Листинг 1.1 ❖ Псевдокод программы, реализующей вычисление `s=x+(2*y)+z` на простой регистровой машине с двумя регистрами. В комментариях показано состояние регистров после выполнения каждой команды

```
Load      A, y          // A = y
Multiply  A, 2          // A = A * 2 = 2 * y
Load      B, x          // B = x
Add       A, B          // A = A + B = x + 2 * y
Load      B, z          // B = z
Add       A, B          // A = A + B = x + 2 * y + z
Store     s, A          // s = A
```

Если этот код напоминает x86 или еще какой-то известный вам язык ассемблера, то это не случайно! Дело в том, что большинство современных компьютеров – сложные регистровые машины. В частности, таковыми являются все процессоры компаний Intel и AMD, которые используются в наших компьютерах. При написании ассемблерного кода для архитектур x86 или x64 мы работаем с регистрами общего назначения: `eax`, `ebx`, `ecx` и т. д. Существует, конечно, много других команд, другие специальные регистры и прочее. Но идея сохраняется.

ПРИМЕЧАНИЕ Можно ли представить себе машину с набором команд, которая позволяла бы выполнять команды прямо над операндами в памяти без загрузки в регистры? На нашем языке псевдоассемблера она выглядела бы гораздо более краткой и высокоуровневой, поскольку отпадает нужда в дополнительных командах загрузки из памяти в регистры и сохранения регистров в памяти:

```
Multiply  s, y, 2    // s = 2 * y
Add       s, x       // s = s + x = 2 * y + x
Add       s, z       // s = s + z = 2 * y + x + z
```

Да, такие машины существовали, например IBM System/360, но сегодня я не знаю ни об одном компьютере подобного рода, используемом для решения производственных задач.

Стек

Концептуально стеком называется структура данных, работающая по принципу «последним пришел – первым ушел» (last in, first out – LIFO). Она допускает две основные операции: добавление данных на вершину стека (*push – заталкивание*) и возврат данных, находящихся на вершине (*pop – выталкивание*). См. рис. 1.4.

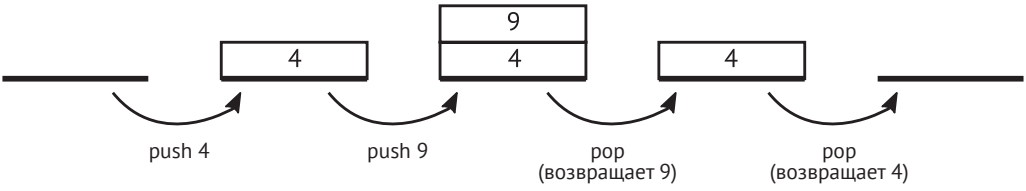

Рис. 1.4 ❖ Операции заталкивания в стек и выталкивания из стека. Рисунок концептуальный и не имеет отношения ни к какой конкретной модели памяти и ее реализации

С самого начала стек был тесно связан с программированием компьютеров, в основном из-за концепции подпрограммы. На современной платформе .NET понятия «стек» и «стек вызовов» используются повсеместно, поэтому посмотрим, с чего все начиналось. Первоначальная семантика стека как структуры данных никуда не делась (например, в .NET есть реализующая ее коллекция `Stack<T>`), но давайте поговорим о том, как она эволюционировала в более общее понятие, связанное с организацией памяти компьютера.

Самые первые компьютеры, о которых мы говорили выше, допускали только последовательное выполнение программы, когда команды считывались одна за другой с перфокарт или перфоленты. Но, конечно, идея о том, чтобы сделать некоторые части программы (*подпрограммы*) повторно используемыми, была очень соблазнительной. Для того чтобы можно было вызывать часть программы, код должен быть адресуемым, поскольку нужно же как-то указать, какую именно часть мы хотим вызвать. Впервые подход к этой проблеме был предложен знаменитой Грейс Хоппер в системе A-0, считающейся первым компилятором. Она записывала набор различных программ на ленту и сопоставляла каждой порядковый номер, который позволял компьютеру найти ее. Таким образом, «программа» состояла из последовательности номеров – индексов программ – и их параметров.

Хотя это действительно вызов подпрограмм, но, очевидно, крайне ограниченный. Программа могла вызывать подпрограммы только поочередно, вложенные вызовы не допускались.

Для вложенных вызовов нужен несколько более сложный подход, потому что компьютер должен где-то запоминать место, с которого следует продолжить вычисление (адрес возврата) после выполнения указанной подпрограммы. Сохранение адреса возврата в одном из аккумуляторов впервые было реализовано Дэвидом Уилером (David Wheeler) в машине EDSAC (этот метод получил название «переход Уилера»). Но его упрощенный подход не допускал *рекурсивных* вызовов, т. е. вызовов подпрограммы из нее же.

Первое упоминание о стеке в том виде, в котором мы знаем его сегодня в контексте компьютерной архитектуры, встречается в отчете Алана Тьюринга об Автоматической вычислительной машине (Automatic Computer Engine – ACE), написанном в начале 1940-х годов. Там описывается концепция машины по типу фон Неймана, т. е. по сути дела компьютера с хранимой программой. Помимо массы других деталей реализации, были описаны две команды – BURY и UNBURY, – работающие с оперативной памятью и сумматорами.

- При вызове подпрограммы (BURY) адрес текущей выполняемой команды плюс единица, указывающий на следующую команду (адрес возврата), сохраняется в памяти. А значение в другой рабочей области памяти, играющее роль указателя стека, увеличивается на 1.
- При возврате из подпрограммы (UNBURY) выполняется противоположное действие.

Это было первой реализацией стека в терминах LIFO-списка адресов возврата из подпрограмм. Именно это решение до сих пор применяется в современных компьютерах; оно, конечно, значительно эволюционировало, но идея сохранилась.

Стек – очень важная часть управления памятью, потому что в программах для .NET данные в нем размещаются сплошь и рядом. Изучим внимательнее стек и его применение для вызова функций. В качестве примера возьмем программу в листинге 1.2, написанную на псевдокоде, напоминающем язык C. В ней имеются два вызова функций: `main` вызывает функцию `fun1` (передавая два аргумента `a` и `b`), в которой объявлены две локальные переменные `x` и `y`. Затем `fun1` в какой-то момент вызывает функцию `fun2` (передавая один аргумент `n`), в которой объявлена одна локальная переменная `z`.

Листинг 1.2 ❖ Псевдокод программы, вызывающей одну функцию из другой

```
void main()
{
   ...
   fun1(2, 3);
   ...
}

int fun1(int a, int b)
{
   int x, y;
   ...
```

```
   fun2(a+b);
}

int fun2(int n)
{
   int z;
   ...
}
```

Изобразим непрерывную область памяти, предназначенную для хранения стека, так что адреса ячеек увеличиваются снизу вверх (левая часть рис. 1.5a), а также другую область памяти, в которой размещена программа (правая часть рис. 1.5a), организованную аналогично. Поскольку код функций не обязан располагаться в смежных участках, блоки, соответствующие `main`, `fun1` и `fun2`, отделены друг от друга. Выполнение программы из листинга 1.2 можно описать следующим образом:

1. Непосредственно перед вызовом `fun1` из `main` (рис. 1.5a). Поскольку программа работает, какой-то стек уже создан (серая часть области стека на рис. 1.5a). В указателе стека (SP) хранится адрес текущей границы стека. Счетчик программы (PC) указывает куда-то внутрь функции `main` (мы обозначили этот адрес `A1`), прямо перед командой вызова `fun1`.

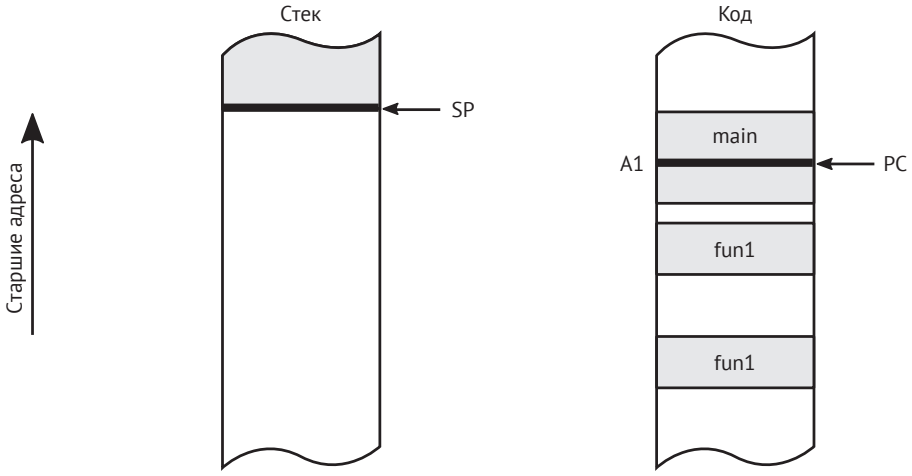

Рис. 1.5a ❖ Стек и память программы – в момент перед вызовом функции `fun1` в листинге 1.2

2. После вызова `fun1` из `main` (рис. 1.5b). Стек растет за счет того, что SP сдвигается, чтобы вместить необходимую информацию. В дополнительной области размещаются:
   - аргументы – все аргументы функции можно сохранить в стеке. В нашем примере туда помещаются аргументы `a` и `b`;
   - адрес возврата – чтобы иметь возможность продолжить выполнение `main` после завершения `fun1`, в стек помещается адрес команды, следующей за вызовом функции. Мы обозначили его `A1+1` (указатель на команду, следующую за командой по адресу `A1`);

– локальные переменные – место для всех локальных переменных также выделяется в стеке. В нашем примере это переменные x и y.

Такая структура, создаваемая в стеке при вызове подпрограммы, называется *записью активации*. В типичной реализации указатель стека уменьшается на соответствующую величину и указывает на область, где может начаться следующая запись активации. Именно поэтому часто говорят, что стек растет вниз.

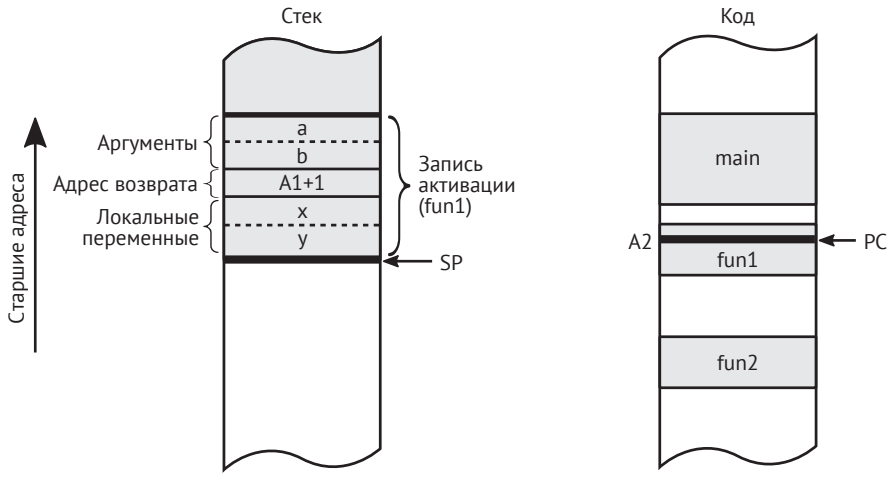

Рис. 1.5b ❖ Стек и память программы – в момент после вызова функции `fun1` в листинге 1.2

3. После вызова `fun2` из `fun1` (рис. 1.5c). Повторяется та же схема создания записи активации. На этот раз она содержит область памяти для аргумента `n`, адреса возврата `A2+1` и локальной переменной `z`.

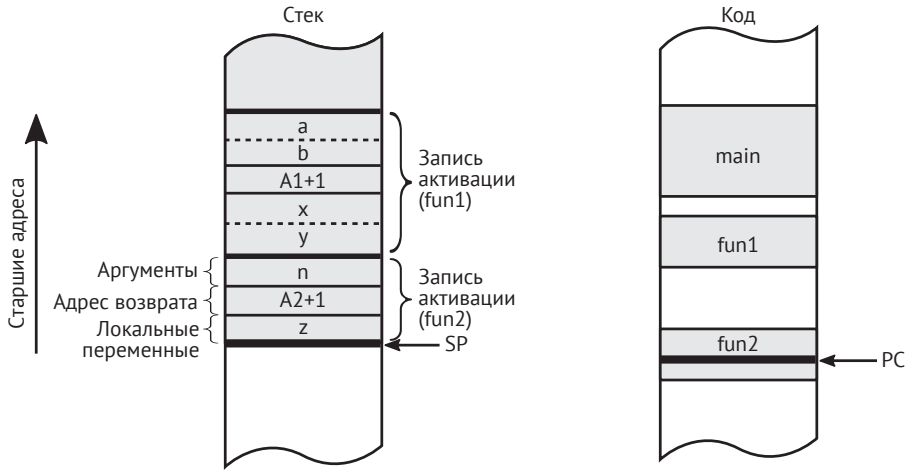

Рис. 1.5c ❖ Стек и память программы – в момент после вызова функции `fun2` из `fun1`

Запись активации называют также более общим термином *кадр стека*, которым обозначают произвольные структурированные данные, сохраненные в стеке для определенных целей.

Как видим, вложенные вызовы подпрограмм просто повторяют схему создания одной записи активации на вызов. Чем больше уровень вложенности, тем больше будет в стеке записей активации. Конечно, бесконечная вложенность невозможна, т. к. это потребовало бы памяти для бесконечного числа записей активации[1]. Если вам когда-нибудь доводилось столкнуться с исключением `StackOverflowException`, то это как раз тот случай – вы сделали так много вложенных вызовов, что память, отведенная для стека, закончилась.

Имейте в виду, что представленный здесь механизм чисто демонстрационный и очень общий. Конкретная реализация может зависеть от архитектуры и операционной системы. В последующих главах мы подробно рассмотрим, как записи активации и стек вообще используются в .NET.

По завершении подпрограммы ее запись активации отбрасывается, для чего достаточно увеличить указатель стека на размер текущей записи активации, а сохраненный адрес возврата копируется в PC, в результате чего выполнение вызывающей функции продолжается. Иными словами, содержимое кадра стека (локальные переменные, параметры) уже не нужно, поэтому увеличения указателя стека достаточно для «освобождения» использованной памяти. Эта область будет попросту перезаписана при следующем использовании стека (рис. 1.6).

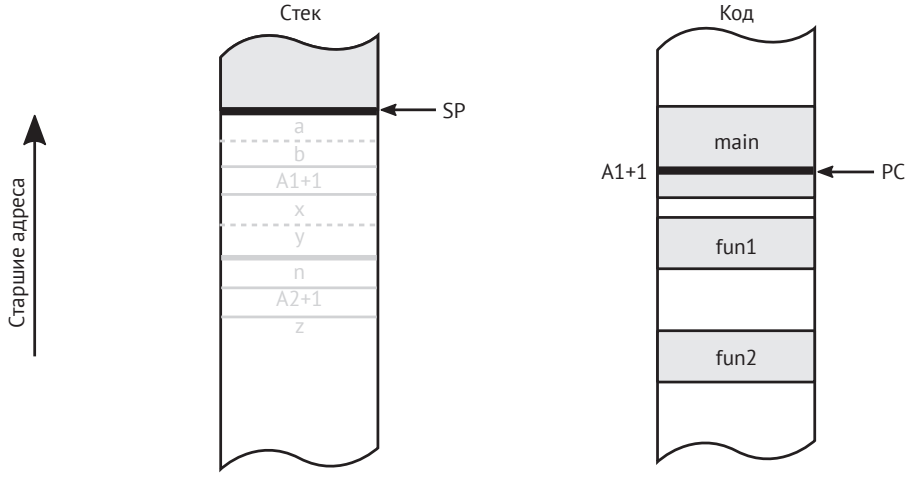

Рис. 1.6 ❖ Стек и память программы – после возврата из функции `fun1` обе записи активации отбрасываются

Что касается реализации: SP и PC обычно хранятся в специальных регистрах. Пока что размеры самого адреса, областей памяти и регистров нам не очень важны.

---

[1] Есть одно интересное исключение – хвостовые вызовы, которое мы за недостатком места здесь не описываем.

В современных компьютерах стек поддержан как на уровне оборудования (специальные регистры для указателей стека), так и на программном уровне (реализованная операционной системой абстракция потока с областью памяти, используемой в роли стека).

Кстати говоря, с точки зрения аппаратной архитектуры, возможно много разных реализаций стека. Стек можно хранить в выделенном блоке памяти внутри ЦП или в выделенной микросхеме. Можно также использовать для этой цели часть общей памяти компьютера. Последний вариант применяется в большинстве современных архитектур – под стек отводится участок памяти процесса фиксированного размера. Возможны даже архитектуры с несколькими стеками. Например, стек адресов возврата может быть отделен от стека параметров и локальных переменных. Это может дать выигрыш в производительности, поскольку доступ к двум стекам может производиться одновременно, что, в свою очередь, открывает возможность для дополнительной оптимизации конвейера ЦП и других низкоуровневых механизмов. Но так или иначе, в современных персональных компьютерах стек – просто часть оперативной памяти.

FORTRAN можно считать первым высокоуровневым языком программирования общего назначения, получившим широкое распространение. Но начиная с 1954 года, когда он был определен, допускалось только статическое выделение памяти. Размер любого массива должен был быть известен на этапе компиляции, а выделение памяти производилось лишь в стеке. ALGOL стал еще одним очень важным языком, он положил начало великому множеству других языков (в частности, C/C++, Pascal, Basic, а через Simula и Smalltalk – всем современным объектно-ориентированным языкам типа Python или Ruby). В ALGOL 60 было только выделение памяти в стеке – наряду с динамическими массивами (размер которых задавался в переменной). Алан Перлис, выдающийся член группы, создавшей ALGOL, писал:

> *Algol 60 невозможно было бы разумно реализовать без понятия стека. Хотя стеки существовали и раньше, лишь после появления Algol 60 они стали занимать центральное место в конструкции процессоров.*

Языки на основе ALGOL и FORTRAN использовались в основном научным сообществом, но одновременно развивалось другое направление, ориентированное на языки программирования для бизнеса: A-0, FLOW-MATIC, COMTRANS и, наконец, широко известный COBOL (Common Business Language). Всем им недоставало явного управления памятью, а работали они в основном с примитивными типами данных: числами и строками.

Стековая машина

Прежде чем двигаться дальше, задержимся еще немного на теме, связанной со стеком, – *стековых машинах*. В отличие от регистровой машины, в стековой машине все команды оперируют специальным *стеком выражений* (или *стеком вычислений*). Имейте в виду, что этот стек не обязан совпадать со стеком, о котором мы говорили до сих пор. Поэтому в такой машине вполне могли бы сосуществовать стек общего назначения и дополнительный *стек выражений*. Регистров может вообще не быть. По умолчанию в такой машине команды получают аргументы

с вершины стека выражений – столько, сколько им необходимо. Результат также записывается в вершину стека. В таком случае говорят о *чисто стековой машине* в противоположность нечистым реализациям, в которых операциям разрешено получать значения не только с вершины стека, но и с более глубоких уровней.

Как именно операция обращается к стеку выражений? Например, гипотетическая команда Multiply (без аргументов) извлекает два значения с вершины стека выражений, перемножает их и помещает результат обратно в стек выражений (рис. 1.7).

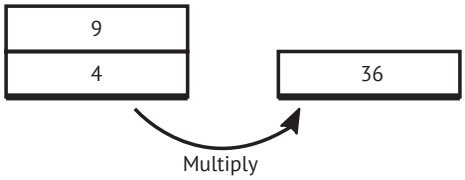

Рис. 1.7 ❖ Гипотетическая команда Multiply в стековой машине – извлекает два элемента из стека и помещает назад результат их умножения

Вернемся к выражению s=x+(2*y)+z из примера для регистровой машины и перепишем его для стековой машины.

Листинг 1.3 ❖ Псевдокод вычисления выражения s=x+(2*y)+z стековой машиной. В комментариях показано состояние стека вычислений

```
            // пустой стек
Push 2      // [2] - в стеке один элемент со значением 2
Push y      // [2][y] - в стеке два элемента: 2 и y
Multiply    // [2*y]
Push x      // [2*y][x]
Add         // [2*y+x]
Push z      // [2*y+x][z]
Add         // [2*y+x+z]
Pop l       // [] (с побочным эффектом - записью значения в l)
```

Эта идея приводит к очень ясному коду. Опишем основные преимущества.

- Не возникает вопроса, как и где хранить временные значения: в регистрах, в стеке или в основной памяти. Концептуально это проще, чем пытаться найти оптимальное решение о месте размещения. А значит, и реализация упрощается.
- Коды операций можно сделать короче, поскольку многие команды вообще не имеют операндов или имеют всего один операнд. Это позволяет более эффективно кодировать команды, а значит, двоичный код будет занимать меньше места в памяти, пусть даже общее число команд больше, чем в регистровой машине (из-за большего числа команд загрузки и сохранения).

На заре развития компьютеров, когда память была очень дорогой и ограниченной, это было существенное преимущество. Оно остается важным и сегодня, когда речь идет о скачиваемом коде для смартфонов или веб-приложений. Кроме того, плотное двоичное кодирование команд улучшает использование кеша процессора.

Но, несмотря на все преимущества, идея стековой машины редко воплощалась в оборудовании. Заметное исключение – машины компании Burroughs, например B5000, в которых имелась аппаратная реализация стека. Сегодня, наверное, нет ни одной широко распространенной машины, которую можно было бы назвать стековой, если не считать сопроцессора x87 для вычислений с плавающей точкой (часть совместимых с x86 процессоров), который проектировался как стековая машина и ради обратной совместимости до сих пор так и программируется.

Тогда зачем же вообще упоминать эти машины? Поскольку такая архитектура очень удобна для проектирования платформенно-независимых виртуальных машин или исполняющих движков. Виртуальная машина Java компании Sun и среда выполнения .NET – идеальные примеры стековых машин. Они исполняются такими хорошо известными регистровыми машинами, как архитектуры x86 или ARM, но это не отменяет тот факт, что они реализуют логику стековой машины. Мы убедимся в этом, когда в главе 4 будем описывать промежуточный язык .NET – Intermediate Language (IL). Почему среда выполнения .NET и JVM (виртуальная машина Java) спроектированы таким образом? Как всегда, причин несколько – инженерного и исторического характера. Код для стековой машины более высокого уровня и лучше абстрагирует особенности реального оборудования. И среду выполнения Майкрософт, и Sun JVM можно было написать как регистровые машины, но как решить, сколько регистров необходимо? Если все они виртуальные, то оптимальный ответ – бесконечно много. Тогда необходимо как-то управлять ими и организовать повторное использование. И как бы выглядела оптимальная абстрактная регистровая машина?

Даже если оставить такие проблемы в стороне и поручить кому-то еще (в данном случае среде выполнения Java или .NET) платформенно-зависимую оптимизацию, то регистровые или стековые механизмы вычислений все равно пришлось бы отобразить на конкретную регистровую архитектуру. Но стековые машины концептуально проще. Виртуальная стековая машина (не исполняемая реальной аппаратной стековой машиной) может обеспечить приемлемую платформенную независимость, порождая при этом высокопроизводительный код. А в совокупности с вышеупомянутой повышенной плотностью кода это дает хорошую платформу, которая может работать на самых разных устройствах. Наверное, именно по этой причине Sun решила пойти по данному пути, когда проектировала Java для небольших устройств, например телевизионных приставок. Майкрософт при проектировании .NET также выбрала этот путь. Концепция стековой машины элегантна, проста и работает. Поэтому реализация виртуальной машины – более приятная инженерная задача!

С другой стороны, регистровые виртуальные машины ближе к конструкции реального оборудования, на котором работают. Это расширяет спектр возможных оптимизаций. Сторонники данного подхода говорят, что так можно добиться гораздо более высокой производительности, особенно для интерпретирующих сред выполнения. У интерпретатора гораздо меньше времени на то, чтобы заниматься сложными оптимизациями, поэтому чем ближе интерпретируемый код к машинному, тем лучше. Кроме того, оперирование самым часто используемым набором регистров улучшает локальность ссылок в кеше[1].

---

[1] Примечание: о важности характера доступа к памяти с точки зрения использования кеша мы будем говорить в главе 2.

Как всегда, принимая решение, приходится идти на компромиссы. Спор между сторонниками обоих подходов долгий и неразрешенный. Но факт остается фактом – среда выполнения .NET реализована в виде стековой машины, хотя и не совсем чистой (мы отметим это в главе 4). Мы также увидим, как стек вычислений отображается на реальное оборудование, состоящее из регистров и памяти.

> **ПРИМЕЧАНИЕ** Все ли виртуальные машины и среды выполнения являются стековыми машинами? Вовсе нет! Например, Dalvik, виртуальная машина в Google Android до версии 4.4, – регистровая реализация JVM. Это был интерпретатор промежуточного «байт-кода Dalvik». Позже JIT (компиляция «на лету», объясняется в главе 4) была включена в преемник Dalvik – среду выполнения Android (ART). Есть и другие примеры: BEAM – виртуальная машина для Erlang/Elixir, Chakra – среда выполнения JavaScript в IE9, Parrot (виртуальная машина в Perl) и Lua VM (виртуальная машина для языка Lua). Никто не скажет, что такие машины не пользуются популярностью.

Указатель

До сих пор мы рассматривали только два вида памяти: статически выделенную и выделенную в стеке (как часть кадра стека). Концепция *указателя* очень общая, она появилась уже в самом начале компьютерной эры – вспомните, например, уже упоминавшиеся указатель команд (счетчик программы) и указатель стека. Специальные регистры, служащие для адресации памяти, например *индексные регистры*, тоже можно считать указателями[1].

В 1965 году компания IBM предложила язык PL/I, предназначенный как для научных, так и для коммерческих приложений. Хотя его цели так и не были в полной меры достигнуты, он является важной исторической вехой, поскольку в нем впервые была введена концепция указателей и выделения памяти. На самом деле Гарольд Лоусон, один из разработчиков языка PL/I, в 2000 году был удостоен премии IEEE «за изобретение переменной указателя и включение этой концепции в язык PL/I, что впервые дало возможность гибко реализовать связные списки в высокоуровневом языке общего назначения». Именно эта задача и послужила стимулом для изобретения указателей – обработка списков и других более-менее сложных структур данных. Концепция указателя затем была использована при разработке языка C, преемника языка B (и его предшественников BCPL и CPL). Лишь в версии FORTRAN 90, последовавшей за FORTRAN 77 и утвержденной в 1991 году, появились динамическое выделение памяти (посредством подпрограмм allocate и deallocate), атрибут `POINTER`, присваивание указателю и выражение `NULLIFY`.

Указатели – это переменные, в которых хранится адрес местоположения в памяти. Проще говоря, указатель позволяет ссылаться на другие места в памяти по адресу. Размер указателя связан с вышеупомянутой длиной слова и зависит от архитектуры компьютера. Поэтому в настоящее время мы работаем с 32- или 64-разрядными указателями. Поскольку это всего лишь небольшой участок па-

---

[1] Если говорить об адресации памяти, то важным усовершенствованием был индексный регистр, впервые появившийся в машине Manchester Mark 1, пришедшей на смену Baby. Индексный регистр позволяет адресовать память косвенно путем сложения его с другим регистром. Поэтому для действий с непрерывными участками памяти, например массивами, требуется меньше команд.

мяти, его можно поместить в стек (например, в качестве локальной переменной или аргумента функции) или в регистр ЦП. На рис. 1.8 изображена типичная ситуация, когда одна из локальных переменных (хранящихся в записи активации функции) – указатель на другую область памяти по адресу `Addr`.

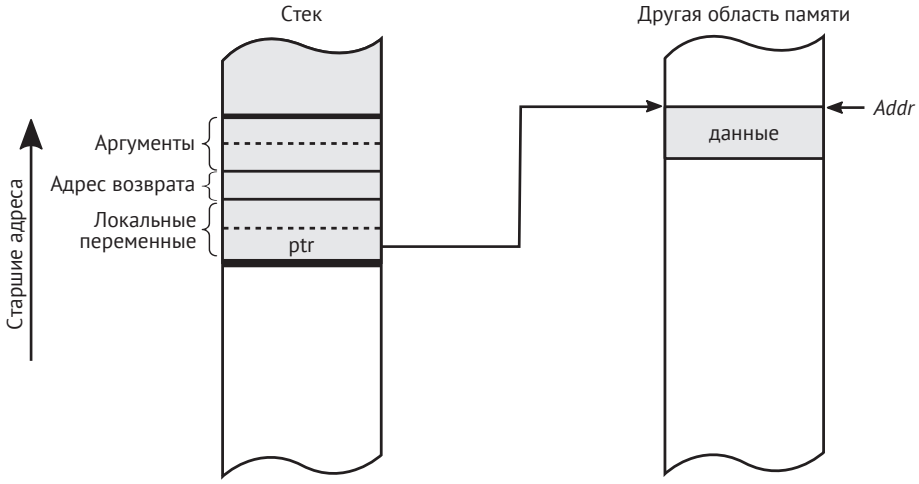

Рис. 1.8 ❖ Локальная переменная функции, являющаяся указателем `ptr` на область памяти по адресу `Addr`

Простая идея указателей позволяет создавать сложные структуры данных, в т. ч. связные списки и деревья, поскольку структуры данных в памяти могут ссылаться друг на друга, образуя еще более сложные структуры (рис. 1.9).

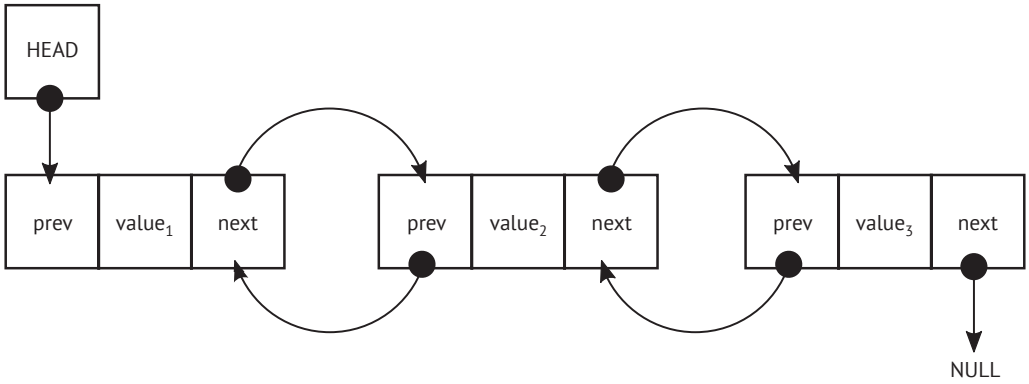

Рис. 1.9 ❖ Указатели, используемые для построения двусвязного списка, в котором каждый элемент указывает на предыдущий и последующий элементы

Кроме того, для указателей можно определить так называемую *адресную арифметику*. Например, оператор инкремента увеличивает значение указателя на длину объекта, на который он указывает, а не на один байт, как можно было бы ожидать.

В языках высокого уровня, в частности Java и C#, указатели нередко недоступны или их использование должно быть разрешено явно, в результате чего код перестает быть безопасным. Почему это так, станет ясно, когда мы будем говорить о ручном управлении памятью ниже в этой главе.

Куча

Вот, наконец, мы и добрались до самого важного понятия в контексте управления памятью в .NET. *Куча* (менее известно название *свободное хранилище*) – это область памяти, используемая для динамически выделенных объектов. Термин «свободное хранилище» лучше, потому что не предполагает никакой внутренней структуры, а только назначение. На самом деле можно с полным правом задать вопрос, что общего между структурой данных *куча* (heap) и собственно кучей. Ответ – ничего. Если стек имеет четкую организацию (структура данных с дисциплиной обслуживания LIFO), то куча больше похожа на черный ящик, у которого можно попросить память, не интересуясь, откуда он ее берет. Поэтому «пул» или «свободное хранилище» лучше отражали бы ее назначение. Возможно, название «куча» происходит от традиционного в английском языке значения «хаотическое нагромождение» – как противоположности упорядоченному стеку. Исторически выделение памяти из кучи было введено в языке ALGOL 68, который так и не получил широкого признания. Однако само название, вероятно, идет оттуда. Но факт остается фактом – точно о происхождении термина теперь никто не знает.

Куча – это механизм, позволяющий получить непрерывный блок памяти указанного размера. Эта операция называется *динамическим выделением* (или *распределением*) *памяти*, поскольку ни размер, ни фактическое местоположение области памяти неизвестно на этапе компиляции. Так как местоположение заранее неизвестно, обращаться к динамически выделенной памяти можно только по указателю. Следовательно, понятия указателя и кучи неразрывно связаны.

Адрес, возвращенный некоторой функцией «дай мне X байт памяти», должен быть запомнен в указателе для последующего обращения к выделенному блоку памяти. Его можно сохранить в стеке (рис. 1.10), в самой куче или где-то еще:

```
PTR ptr = allocate(10);
```

Операция, обратная выделению, называется *освобождением*, она возвращает указанный блок памяти в пул для будущего использования. Как именно из кучи выделяется область указанного размера – деталь реализации. Существует много таких «распределителей», и о некоторых мы скоро узнаем.

Если выделяется и освобождается много блоков, то можно столкнуться с ситуацией, когда блока памяти нужного размера нет, хотя всего в куче памяти достаточно. Это называется *фрагментацией* кучи и может стать причиной крайне неэффективного использования памяти. На рис. 1.11 показано, как выглядит отсутствие непрерывного блока памяти для объекта X. Распределители применяют различные стратегии оптимального управления пространством, чтобы избежать фрагментации (или воспользоваться ей во благо).

Отметим также, что в одном процессе может быть как одна, так и несколько куч – это тоже деталь реализации (мы вернемся к этому вопросу, когда будем глубже обсуждать .NET).

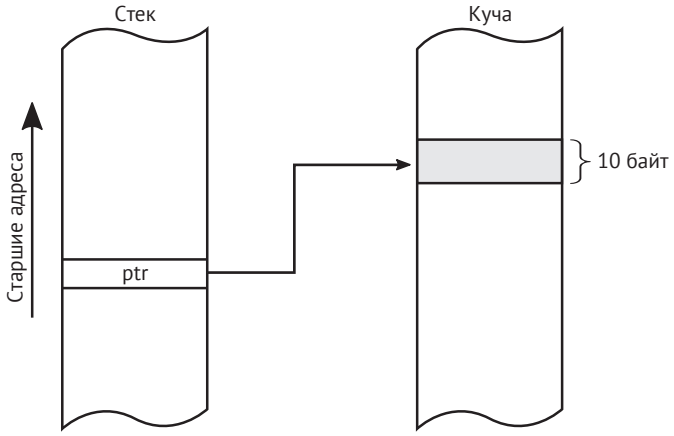

Рис. 1.10 ❖ В стеке хранится указатель `ptr` на область в куче размером 10 байт

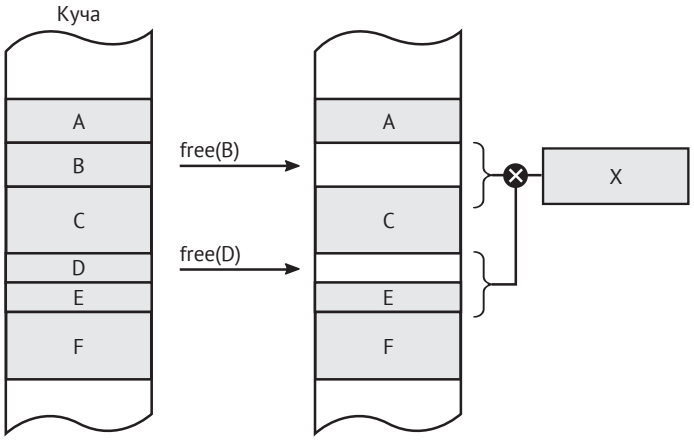

Рис. 1.11 ❖ Фрагментация – после удаления объектов B и D для нового объекта X не хватает памяти, хотя всего памяти достаточно

В табл. 1.1 приведена краткая сводка различий между стеком и кучей.

Помимо этих различий, стек и куча чаще всего располагаются в противоположных концах адресного пространства процесса. Мы вернемся к детальному рассмотрению размещения стека и кучи в адресном пространстве процесса, когда будем обсуждать низкоуровневое управление памятью в главе 2. Однако помните, что это всего лишь деталь реализации. Благодаря абстракциям значимых и ссылочных типов (см. главу 4) нам безразлично, где именно они создаются. Теперь перейдем к сравнению ручного и автоматического управления памятью. В «Аннотированном руководстве по языку C++» Эллис и Страуструп пишут:

> *Программисты на C считают, что управление памятью – слишком важная вещь, чтобы оставлять ее компьютеру. Программисты на Lisp считают, что управление памятью – слишком важная вещь, чтобы оставлять ее пользователю.*

Таблица 1.1. Сравнение стека и кучи

Свойство	Стек	Куча
Время жизни	Область видимости локальных переменных (помещаются при входе, извлекаются при выходе)	Явное (в результате выделения и, возможно, освобождения)
Область видимости	Локальная (поточно[1])	Глобальная (любой, кто имеет доступ к указателю)
Доступ	Локальная переменная, аргументы функции	Указатель
Время доступа	Быстро (с большой вероятностью находится в кеше процессора)	Медленнее (может даже временно выгружаться на диск)
Выделение	Перемещение указателя стека	Возможны различные стратегии
Время выделения	Очень быстро (сдвиг указателя стека вниз)	Медленнее (зависит от стратегии выделения)
Освобождение	Перемещение указателя стека	Возможны различные стратегии
Использование	Параметры подпрограмм, локальные переменные, записи активации, данные небольшого известного на этапе компиляции размера (массивы)	Все, что угодно
Емкость	Ограничена (обычно несколько мегабайтов на поток)	Ограничена только размером жесткого диска
Переменный размер	Нет	Да[2]
Фрагментация	Нет	Вероятна
Основные угрозы	Переполнение стека	Утечка памяти (забыли освободить выделенную память), фрагментация

Ручное управление памятью

Все, о чем мы говорили до сих пор, можно назвать «ручным управлением памятью». Это означает, в частности, что разработчик отвечает за явное выделение памяти и должен освободить ее, когда она больше не нужна. Это самая настоящая ручная работа. Как ручная передача в большинстве европейских автомобилей. Я сам из Европы, и мы привыкли переключать передачи вручную. Мы должны подумать, пора ли уже переключать или надо подождать несколько секунд, пока двигатель наберет нужные обороты. Тут есть одно большое преимущество – мы полностью контролируем машину. Мы сами отвечаем за то, оптимально работает двигатель или нет. А поскольку люди гораздо лучше адаптируются к изменяющимся условиям, хороший водитель справляется лучше, чем автоматическая коробка. Но, конечно, есть и большой недостаток. Вместо того чтобы сосредоточиться на главной цели – как добраться из точки A в точку B, мы должны еще думать о переключении передач – сотни, тысяч раз в течение долгой поездки. Это

[1] Это не совсем так, потому что указатель на переменную в стеке можно передать в другой поток. Однако это, безусловно, аномальное использование.

[2] В силу динамической природы кучи существуют функции, позволяющие изменить размер указанного блока памяти (перераспределить память).

и время отнимает, и утомляет. Я знаю людей, которым это по душе и которые считают, что отдавать управление автоматической коробке было бы скучно. Я даже могу согласиться с ними. Но все равно мне нравится эта автомобильная метафора в применении к управлению памятью.

Явное выделение и освобождение памяти сродни ручной передаче. Вместо того чтобы сосредоточиться на главной цели – а это, наверное, какая-то бизнес-задача, решаемая нашей программой, – мы должны думать также о том, как управлять памятью. Это отвлекает наше ценное внимание от главной цели. Вместо того чтобы размышлять об алгоритмах, бизнес-логике и предметной области, мы вынуждены думать о том, когда и сколько памяти нам нужно. И надолго ли? И кто будет отвечать за ее освобождение? Это что, бизнес-логика? Нет, конечно. Но вопрос о том, хорошо это или плохо, – совсем другая история.

Хорошо известный язык C был создан Дэннисом Ричи в начале 1970-х годов и стал одним из самых распространенных в мире языков программирования. История того, как C эволюционировал от ALGOL через промежуточные языки CPL, BCPL и B, интересна сама по себе, но в нашем контексте важно, что наряду с Pascal (прямой потомок ALGOL) они были в свое время двумя самыми популярными языками, допускающими явное управление памятью. Что касается C, то я могу без сомнения сказать, что компилятор для него имеется на любой когда-либо существовавшей аппаратной платформе. Не буду удивлен, если и на борту кораблей пришельцев тоже есть компилятор C (возможно, применяемый для реализации стека TCP/IP – еще один пример повсеместно распространенного стандарта). Влияние этого языка на другие языки программирования огромно, переоценить его невозможно. Давайте сделаем короткую паузу и взглянем на него в контексте управления памятью. Это позволит сформулировать некоторые характеристики ручного управления.

Рассмотрим простую программу, написанную на C.

Листинг 1.4 ❖ Пример программы на C, демонстрирующий ручное управление памятью

```
#include <stdio.h>

void printReport(int* data)
{
   printf("Отчет: %d\n", *data);
}

int main(void) {
   int *ptr;
   ptr = (int*)malloc(sizeof(int));
   if (ptr == 0)
   {
      printf("ОШИБКА: недостаточно памяти\n");
      return 1;
   }
   *ptr = 25;
   printReport(ptr);
   free(ptr);
   ptr = NULL;
   return 0;
}
```

Конечно, пример несколько утрированный, но он позволяет продемонстрировать суть проблемы. Легко заметить, что у этой тривиальной программы одна простая цель: напечатать «отчет». Для простоты отчет состоит из единственного целого числа, но можно представить себе сложную структуру, содержащую указатели на другие структуры данных. Эту простую задачу затмевает «церемониальный код», который не занимается ничем, кроме работы с памятью. Это и есть ручное управление памятью во всей красе.

Помимо написания бизнес-логики, разработчик в этом примере должен:

- выделить нужное количество памяти для данных с помощью функции `malloc`;
- привести общий указатель (типа `void*`) к указателю на нужный тип (`int*`), чтобы показать, что он указывает на числовое значение (в данном случае типа `int`);
- запомнить указатель на выделенную область памяти в локальной переменной `ptr`;
- проверить, было ли выделение памяти успешным (в случае неудачи возвращенный адрес будет равен 0);
- разыменовать указатель (получить доступ к памяти по этому адресу), чтобы сохранить данные (числовое значение 25);
- передать указатель в функцию `printReport`, которая разыменует его для собственных целей;
- освободить выделенную память, когда необходимость в ней отпадет, вызвав функцию `free`;
- не забыть присвоить указателю специальное значение `NULL` (так мы говорим, что указатель ни на что не указывает, в действительности это значение соответствует 0[1]).

Как видим, управляя памятью вручную, мы многое должны держать в голове. А если любое из описанных выше действий выполнено неправильно или пропущено, то могут воспоследовать серьезные проблемы. Посмотрим, какие именно.

- Мы должны точно знать, сколько памяти необходимо. В нашем примере все просто – `sizeof(int)` байт, но что, если структура гораздо более сложная, да еще вложенная? Легко представить ситуацию, когда из-за мелкой ошибки в вычислениях размера мы выделим меньше памяти, чем необходимо. Впоследствии, когда мы захотим записать в такую область памяти или прочитать из нее, дело, вероятно, закончится ошибкой *Segmentation Fault* (ошибка сегментации) вследствие попытки доступа к памяти, которую мы не выделяли или выделили для других целей. С другой стороны, из-за такой же ошибки можно выделить слишком много памяти, что ведет к ее неэффективному использованию.
- Приведение типов всегда чревато ошибками и может стать причиной трудно диагностируемых дефектов, если случайно допустить несоответствие типов. Это означает, что мы пытаемся интерпретировать указатель одного типа как указатель совершенно другого типа, что может привести к ошибке доступа к памяти.

[1] Детали реализации значения NULL в .NET будут объяснены в главе 10.

- Запомнить адрес нетрудно. Но что, если мы забудем это сделать? Мы выделили память, но не можем ее освободить – адреса-то нет! Это прямой путь к утечке памяти, поскольку размер неосвобождаемой памяти будет только расти. А может быть и так, что указатель хранится в некой сложной структуре, а не просто в локальной переменной. Что, если мы забудем указатель на сложный граф объектов, потому что освободили содержащую его структуру?
- Одна проверка успешности выделения запрошенной памяти не обременительна. Но делать это сотни раз в каждой функции, безусловно, надоедает. И возможно, мы решим опустить эти проверки, что может привести к неопределенному поведению во многих местах приложения, поскольку мы пытаемся обратиться к памяти, которая не была выделена.
- Разыменование указателей всегда опасно. Никто не знает, куда они указывают. Находится ли по этому адресу действительный объект или он уже освобожден? Да и правилен ли сам указатель? Действительно ли он указывает на адрес в доступной пользователю памяти? Полный контроль над указателями в языках, подобных C, обязательно сопряжен с такими трудностями. А также влечет за собой серьезные проблемы с безопасностью – только на программиста возлагается ответственность не раскрывать данные за пределами области, доступной в соответствии с текущей моделью и типом памяти.
- Передача указателя между функциями и потоками лишь умножает вышеуказанные проблемы в многопоточной среде.
- Мы должны помнить о необходимости освободить выделенную память. Если опустить этот шаг, то будет иметь место утечка памяти. В таком простом примере, как выше, забыть о вызове функции `free`, конечно, трудно. Но в более сложном коде, когда владелец структуры данных не столь очевиден и указатели на структуру передаются туда-сюда, ситуация становится куда более сложной. И есть еще одна опасность – никто не помешает нам освободить память, которая уже была освобождена ранее. А это тоже неопределенное поведение и вероятная причина ошибки сегментации.
- И наконец, мы должны обнулить наш указатель (присвоить ему значение NULL, 0 и т. п.). В противном случае останется *висячий указатель*, что рано или поздно приведет к ошибке сегментации или иному неопределенному поведению, поскольку кто-нибудь попробует разыменовать его, считая, что он все еще представляет действительные данные.

Как видим, с точки зрения разработчика, явное выделение и освобождение памяти могут стать очень обременительным делом. Это очень мощный механизм, у которого, безусловно, есть свои применения. Если требуется максимальная производительность и разработчик на 100 % уверен, что контролирует происходящее, то такой подход может быть полезен.

Но «кому многое дано, с того много и спросится», меч этот обоюдоострый. Поэтому по мере развития программной инженерии языки становились все более и более изощренными, стремясь избавить разработчика от груза этих проблем.

Язык C++, прямой преемник C, в этом отношении не сильно изменился. Однако стоит уделить некоторое время C++ , поскольку он очень популярен и в нем впервые появились другие широко распространенные идеи. Как все мы знаем, в этом

языке управление памятью осуществляется вручную. Прямая трансляция предыдущего примера на C++ могла бы выглядеть так.

Листинг 1.5 ❖ Пример программы на C++, демонстрирующий ручное управление памятью

```
#include <iostream>

void printReport(int* data)
{
   std::cout << "Отчет: " << *data << "\n";
}

int main()
{
   try
   {
      int* ptr;
      ptr = new int();
      *ptr = 25;
      printReport(ptr);
      delete ptr;
      ptr = 0;
      return 0;
   }
   catch (std::bad_alloc& ba)
   {
      std::cout << "ОШИБКА: недостаточно памяти\n";
      return 1;
   }
}
```

В контексте нашего рассмотрения мы можем отметить значительные улучшения.

- Оператор `new` сам выделит достаточно памяти, поскольку благодаря поддержке со стороны компилятора (которому известен размер типа) знает, сколько памяти необходимо.
- Не нужно приводить полученный указатель к требуемому типу. Это снимает некоторые из упомянутых выше опасений по поводу типобезопасности.
- Обработка ошибок также стала лучше: мы не обязаны вручную проверять успешность выделения, потому что в случае проблемы будет возбуждено исключение.

И тем не менее в этом примере много церемониального кода. И появилась еще одна причина для беспокойства. Что, если функция `printReport()` возбудит исключение? Без надлежащей обработки ошибок оператор `delete` не будет выполнен – налицо утечка памяти. Исправить этот код легко, но в более сложных приложениях, где владелец данных (кто и на каком уровне должен освобождать память) не очевиден, ситуация может оказаться не столь тривиальной.

Все описанные в этой главе проблемы становятся еще серьезнее в многопоточной среде, когда указатели могут разделяться между несколькими единицами выполнения. В этих случаях необходима тщательная синхронизация, чтобы не допустить смешения некорректных данных. Например, что, если один поток про-

веряет действительность указателя (отличие от NULL), а сразу после этого другой поток освобождает память, на которую тот указывает? Это может привести к нерегулярным ошибкам, которые очень трудно искать. В мире явного управления памятью разработчик сам должен обеспечить подходящий механизм синхронизации, предотвращающий такие ситуации.

В примере из листинга 1.5 сознательно не используются современные паттерны работы с памятью в C++. Следовало бы использовать какой-то вариант идиомы RAII (получение ресурса есть инициализация), когда ресурс (например, память) представляется локальной переменной, принадлежащей типу, который обеспечивает тот или иной вид владения памятью. Подобный пример приведен ниже в листинге 1.10. И хотя, как мы увидим, такие паттерны помогают решить часть проблем, они принципиально ничего не меняют в общих рассуждениях о ручном и автоматическом управлении памятью.

Автоматическое управление памятью

Чтобы преодолеть проблемы ручного управления памятью и предоставить программисту более приятный способ работы с ней, были предложены различные подходы к автоматическому управлению памятью. Интересно, что уже второй старейший высокоуровневый язык программирования, LISP, предложенный в 1958 году (всего через несколько лет после FORTRAN), мог многое предложить в этой области, поскольку в преимущественно функциональном языке, опирающемся на обработку списков, вручную управлять памятью было бы крайне неудобно. В функциональной парадигме программирования программу рассматривают как вычисление комбинированных функций и всячески избегают модификации данных и побочных эффектов. А выделение и освобождение памяти – без сомнения, операции, изменяющие данные, и с очевидными побочными эффектами. Такая работа с памятью в функциональном коде испортила бы его императивными запахами, тогда как LISP проектировался как в высшей степени декларативный язык. Создатель языка LISP говорил: «если бы мы были вынуждены явно стирать списки, то все стало бы безумно уродливым». Поэтому нужно было разработать что-то более изощренное. В самых первых версиях LISP существовала встроенная функция eralist (стереть список), но после реализации автоматического управления памятью она была исключена.

Вообще, LISP был в высшей степени новаторским языком, благодаря его дизайну в информатику вошло много важных идей, одной из которых и было автоматическое управление памятью. Кстати, Джон Маккарти, автор термина «искусственный интеллект» и изобретатель LISP, является также отцом первых алгоритмов сборки мусора. Многие высказанные тогда идеи по-прежнему живут и используются в современных языках. Можно с уверенностью сказать, что автоматическое управление памятью зародилось в LISP'е. В первой статье, написанной Маккарти в 1958 году, был изложен алгоритм пометки и очистки (Mark and Sweep), который мы подробно изучим в последующих главах, потому что он все еще используется в среде .NET и многих других местах.

Благодаря своей выразительности и лаконичности LISP позволяет представить наш пример в очень простой форме.

Листинг 1.6 ❖ Пример программы на LISP, демонстрирующий автоматизированное управление памятью

```
(defun printReport(data)
   (write-line (format nil "Report: ~a" data))
)

(prog
   ((ptr 25))
   (printReport ptr)
)
```

Благодаря автоматическому управлению памятью все лишние детали ушли из кода, и стало отчетливо видно высокоуровневое описание цели программы – напечатать «отчет».

Приведем интересную историю из статьи Джона Маккарти о дизайне LISP, «Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I». Он кратко описал этот механизм, но назвал его просто «reclamation» (восстановление, рекультивация, регенерация). Позже, в аннотации к этой части он писал:

> *Мы уже тогда называли этот процесс «сборкой мусора», но, думаю, я побоялся использовать данный термин в статье, опасаясь, что дамы из редакторского отдела Исследовательской лаборатории по электронике не пропустят ее.*

Но, если не считать названия, идея уже присутствовала в статье и только ждала реализации. В настоящее время термины автоматическое управление памятью и *сборка мусора* употребляются как синонимы. Мы можем определить это как механизм, снимающий с программиста ответственность за ручное управление памятью и гарантирующий, что созданные объекты автоматически уничтожаются (с возвратом памяти системе), когда необходимость в них отпадает.

Один из главных посылов этой книги – тот факт, что даже полностью автоматическое управление памятью не избавляет от всех проблем. В качестве неформального подтверждения стоит привести забавную историю, относящуюся к первой реализации сборки мусора в LISP. В своей книге «История языков программирования» Маккарти вспоминает, что в ходе первой публичной демонстрации LISP на одном из симпозиумов по промышленной связи в MIT из-за мелкого недосмотра Flexowriter (тогдашняя электрическая пишущая машинка) начал печатать одну за другой страницы, содержащие сообщение об ошибке, которое начиналось словами[1]:

> *THE GARBAGE COLLECTOR HAS BEEN CALLED. SOME INTERESTING STATISTICS ARE AS FOLLOWS*

Презентацию пришлось отменить, а в аудитории стоял хохот. Никто, кроме Джона, не знал, что причиной было неправильное использование сборщика мусора. И хотя это была не алгоритмическая, а человеческая ошибка, можно все-таки сказать, что сборщики мусора причиняли неприятности с самого начала!

---

[1] Вызван сборщик мусора. Далее следует представляющая интерес статистика. – *Прим. перев.*

Распределитель, модификатор и сборщик

Модификатор и другие термины, с которыми мы познакомимся в этой главе, являются важными терминами в академических исследованиях по автоматическому управлению памятью. Благодаря строгим определениям мы сможем недвусмысленно распознать их в научных и технических статьях. Например, можно говорить о «накладных расходах модификатора» в конкретном алгоритме. При рассмотрении различных алгоритмов сборки мусора часто обсуждают влияние сборщика на модификатор и наоборот. Познакомимся с этими терминами поближе.

Модификатор

Среди немногих основных понятий, связанных с управлением памятью, самым главным и весьма важным является абстракция *модификатора* (Mutator). В простейшем варианте модификатор можно определить как сущность, отвечающую за выполнение кода приложения. Название связано с тем, что модификатор модифицирует (изменяет) состояние памяти – объекты выделяются или модифицируются, а ссылки между ними изменяются. Иными словами, модификатор стоит за всеми изменениями в приложении, имеющими отношение к памяти. Это название (в числе прочих) предложил Эдсгер Дейкстра в статье 1978 года «On-the-Fly Garbage Collection: An Exercise in Cooperation», где можно найти подробное изложение вопроса. Попутно отметим, что предложение Дейкстра, высказанное в этой довольно старой статье, все еще применяется, например, в языке Go, созданном в 2015 году, и дает неплохие результаты.

Мне нравится абстракция модификатора, поскольку она позволяет элегантно и четко классифицировать сущности в конкретном фреймворке или среде выполнения. Модификатором можно назвать все, что имеет возможность модифицировать память – путем изменения существующих объектов или создания новых. Хотя это не совсем строго, мы можем расширить данное определение, включив в него все, что может читать память (поскольку чтение – важнейшая для выполнения программы операция). Это приводит к важному наблюдению. Полнофункциональный модификатор должен предоставлять работающему приложению три вида операций:

- `New(amount)` – выделить память указанного размера, которая затем будет использована для создания нового объекта. Заметим, что на этом уровне абстракции мы не рассматриваем информацию о типе объекта, которая может предоставляться или не предоставляться средой выполнения. Мы просто выделяем запрошенное количество памяти;
- `Write(address, value)` – записать указанное значение по указанному адресу. Здесь мы также абстрагируемся от того, куда записываем: в поле объекта (в объектно-ориентированном программировании), в глобальную переменную или еще куда-то;
- `Read(address)` – прочитать значение, хранящееся по указанному адресу.

В простейшем мире, где нет никаких алгоритмов сборки мусора, эти три операции реализуются тривиально (псевдокод на C-подобном языке приведен в листинге 1.7).

Листинг 1.7 ❖ Реализация трех главных методов модификатора без автоматического управления памятью

```
Mutator.New(amount)
{
   return Allocator.Allocate(amount);
}

Mutator.Write(address, value)
{
   *address = value;
}

Mutator.Read(address) : value
{
   return *address;
}
```

Но в мире автоматической сборки мусора эти три операции – места, где модификатор кооперируется со сборщиком мусора (*сборщиком*) и механизмом выделения памяти (*распределителем*). Как выглядит эта кооперация и насколько она усложняет показанные выше реализации – один из самых важных вопросов проектирования. Самое распространенное улучшение, которое встретится в этой книге, – это добавление *барьера*, будь то *барьер чтения* или *барьер записи*. Это способ включить дополнительную операцию до или после определенных операций. Барьеры позволяют синхронизироваться (прямо или косвенно, синхронно или асинхронно) с механизмом сборки мусора, чтобы сообщить информацию о выполнении программы и использовании памяти. Методы в листинге 1.7 – это точки, к которым может подключиться любой сборщик мусора. Мы опишем наиболее распространенные вариации в последующих главах, когда будем рассматривать различные алгоритмы сборки мусора.

В повседневной жизни самая часто встречающаяся реализация абстракции модификатора – всем известный *поток*. Он точно отвечает определению: это программная единица, которая выполняет код и изменяет объекты и графы ссылок между ними. Нам это интуитивно понятно, потому что в подавляющем большинстве популярных сред выполнения используется именно эта реализация. В числе прочего функционала поток посредством некоего дополнительного слоя взаимодействует с операционной системой, делая возможными операции `New`, `Write` и `Read`.

Модификаторы необязательно реализовывать как потоки операционной системы. Популярный пример – экосистема языка Erlang с его процессами; они управляются как сверхлегкие сопрограммы, живущие в самой среде выполнения. Их можно считать так называемыми «зелеными потоками», но в терминологии виртуальной машины Erlang лучше называть их «зелеными процессами», поскольку среда выполнения обеспечивает гораздо более сильное разделение, чем между потокоподобными сущностями. Таким образом, управление этими сущностями осуществляется на уровне среды выполнения, а не операционной системы. Другая распространенная реализация модификатора основана на *волокнах* (*fibers*), облегченных единицах выполнения, реализованных и в Linux, и в Windows.

Распределитель

Модификатор должен использовать операцию `New`, которую мы обсудили в предыдущем разделе. Если задуматься о ее внутреннем устройстве, то рано или поздно мы придем еще к одному важному понятию – *распределителю*. Эта сущность отвечает за динамическое выделение и освобождение памяти. Мы уже отмечали, что в древних языках типа ALGOL и FORTRAN распределителя не было, как не было и никакого динамического выделения памяти.

У распределителя две основные операции:

- `Allocator.Allocate(amount)` – выделить указанное количество памяти. Очевидно, сюда можно добавить методы выделения памяти для объекта указанного типа, если распределителю доступна информация о типе. Как мы видели, эта операция используется внутри `Mutator.New`;
- `Allocator.Deallocate(address)` – освободить память по указанному адресу, сделав ее доступной для выделения в будущем. Отметим, что в случае автоматического управления памятью этот метод внутренний, недоступный модификатору (а потому пользовательский код не может вызывать его явно).

Идея может показаться совсем простой, если не сказать тривиальной. Но, как мы увидим, на поверку она совсем не так примитивна. При проектировании распределителя надо учесть массу различных аспектов. И как всегда, повсюду компромиссы, в основном между производительностью, сложностью реализации (а значит, удобством сопровождения) и прочим. Мы подробно рассмотрим два самых популярных вида распределителей: *последовательный* и *список свободных*. Но, поскольку это деталь реализации, уместнее будет рассказать об этом в конкретном контексте .NET в главе 4.

Сборщик

Поскольку мы определили модификатор как сущность, отвечающую за выполнение кода приложения, то можем по аналогии определить сборщик как сущность, которая выполняет код сборки мусора (автоматического освобождения памяти). Иными словами, мы можем рассматривать сборщик либо как программный код, либо как исполняющий его поток, либо как то и другое сразу. Все зависит от контекста.

Откуда сборщик знает, какие объекты больше не нужны и могут быть освобождены? Это неразрешимая задача, потому что для ее решения пришлось бы заглянуть в будущее – собирается ли кто-нибудь использовать объект? Все зависит от кода, который еще только будет выполнен, а это, в свою очередь, может зависеть от таких неизвестных факторов, как действия пользователя, внешние данные и т. д. Идеальный сборщик должен был бы знать *жизнеспособность* объекта – живыми называются объекты, которые понадобятся в будущем. Противоположное понятие – *мертвые* (или *мусорные*) *объекты*, т. е. такие, которые уже не понадобятся и могут быть уничтожены. Таким образом, понятно, что сборщик обычно называют *сборщиком мусора* (Garbage Collector), или кратко – GC.

Отметим интересное следствие кооперации модификатора, распределителя и сборщика. Вспомните, что метод `Allocator.Deallocate` не доступен открыто, т. е. модификатор не может явно освободить полученную память. Модификатор может только запрашивать все новую и новую память, как будто она поступает

из неисчерпаемого источника. По существу, это означает, что механизм сборки мусора моделирует компьютер с бесконечной памятью. Как эта модель работает и насколько она эффективна – деталь реализации.

Можно представить себе специальный сборщик мусора, который вообще не освобождает выделенную память. Он называется *нулевым*, или *пустым*, *сборщиком мусора*. Он правильно работал бы только на компьютерах с бесконечным объемом памяти, каковых, к сожалению, пока не существует. Но и пустой сборщик мусора не совсем бесполезен. Его можно, например, использовать в программах, работающих очень недолго, когда неограниченный рост памяти приемлем. Быть может, такие сборщики мусора станут популярными в мире бессерверных одиночных короткоживущих функций. Пример пустого сборщика мусора для .NET приведен в главе 15.

Поскольку узнать жизнеспособность объекта невозможно[1], сборщик опирается на менее строгое свойство объекта – *достижимость* хотя бы одним модификатором. Объект называется *достижимым*, если существует последовательность ссылок (начинающаяся в памяти, доступной хотя бы одному модификатору) между объектами, которая приводит к данному объекту (рис. 1.12). Очевидно, что достижимость не означает, что объект жизнеспособен, но это наилучшее возможное приближение. Если объект не достижим ни одним модификатором, то его невозможно использовать, т. е. он мертв и может быть безопасно очищен. Обратное не всегда верно. Достижимый объект может вечно оставаться достижимым (его удерживает какой-то сложный граф ссылок), но выполнение устроено так, что к нему никогда не будет обращений, поэтому он все равно что мертв. На самом деле именно в пограничной области между жизнеспособностью и достижимостью происходит большинство утечек управляемой памяти.

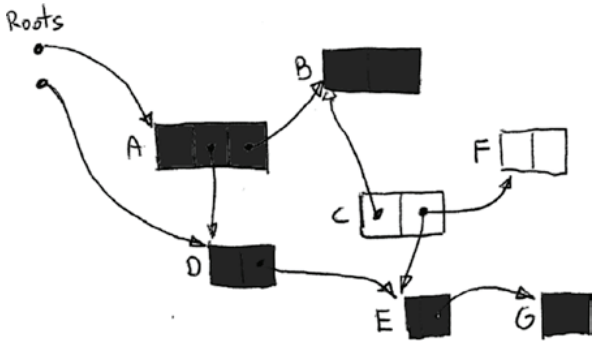

Рис. 1.12 ❖ Достижимость – объекты C и F недостижимы, потому что к ним нет пути от корней (адресов в памяти модификатора)

Когда говорят о достижимости, начальные точки в области памяти модификатора называются *корнями*. Что они собой представляют в действительности, зависит от реализации модификатора. Но в наиболее распространенных случаях,

[1] В главе 4 мы будем обсуждать анализ локальности – метод определения истинной жизнеспособности указателей, по крайней мере в некоторых специальных случаях.

когда модификатор – это просто поток (представленный потоком операционной системы), корнями могут быть:

- локальные переменные и аргументы подпрограмм, размещенные в стеке или в регистрах;
- статически выделенные объекты (например, глобальные переменные), размещенные в куче;
- другие внутренние структуры данных, хранящиеся в самом сборщике.

Зная о трех основных структурных элементах – модификаторе, распределителе и сборщике, – мы можем перейти к знакомству с разнообразными подходами к автоматическому управлению памятью. И хотя есть искушение представить полный список с детальным описанием каждого подхода, книга слишком мала для этого. Вместо этого мы изучим некоторые из наиболее популярных подходов, встречающихся в современных языках.

Подсчет ссылок

Один из двух самых популярных методов автоматического управления памятью называется *подсчетом ссылок*. Идея очень проста. Нужно подсчитывать количество ссылок на каждый объект. В каждом объекте хранится свой *счетчик ссылок*. Когда объект присваивается переменной или полю, количество ссылок на него увеличивается. Одновременно счетчик ссылок в объекте, на который раньше указывала эта переменная, уменьшается.

Жизнеспособность объекта при таком подходе оценивается количеством ссылающихся на него объектов. Если счетчик ссылок обращается в ноль, то на объект никто не ссылается, поэтому его можно освободить. Но что, если счетчик не равен нулю? Это ничего не говорит о жизнеспособности, а означает лишь, что кто-то хранит ссылку на объект, хотя, быть может, и не использует его. Поэтому подсчет ссылок – еще один не особенно строгий способ судить о гипотетической жизнеспособности объекта.

Вернемся к нашему тривиальному примеру модификатора в листинге 1.7. В случае подсчета ссылок его можно было бы написать, как показано ниже.

Листинг 1.8 ❖ Псевдокод, описывающий простой алгоритм подсчета ссылок

```
Mutator.New(amount)
{
   obj = Allocator.Allocate(amount);
   obj.counter = 0;
   return obj;
}

Mutator.Write(address, value)
{
   if (address != NULL)
   ReferenceCountingCollector.DecreaseCounter(address);
   *address = value;
   if (value != NULL)
      value.counter++;
}
```

```
ReferenceCountingCollector.DecreaseCounter(address)
{
   *address.counter--;
   if (*address.counter == 0)
   Allocator.Deallocate(address)
}
```

Поведение подсчета ссылок иллюстрируется простой программой, показанной на рис. 1.13 и в листинге 1.9. Три строки кода переписаны в терминах методов модификатора, чтобы показать, как изменяются ссылки.

Листинг 1.9 ❖ Псевдокод, иллюстрирующий подсчет ссылок

```
o1 = new SomeObject();
o2 = new SomeObject();
o2 = o1;

// транслируется в:

addr1 = Mutator.New(SizeOf(SomeObject))  // addr1.counter = 0
Mutator.Write(&o1, addr1)                // addr1.counter = 1
addr2 = Mutator.New(SizeOf(SomeObject))  // addr2.counter = 0
Mutator.Write(&o2, addr2)                // addr2.counter = 1
Mutator.Write(&o2, &o1)                  // addr1.counter = 0; addr2.counter = 2
```

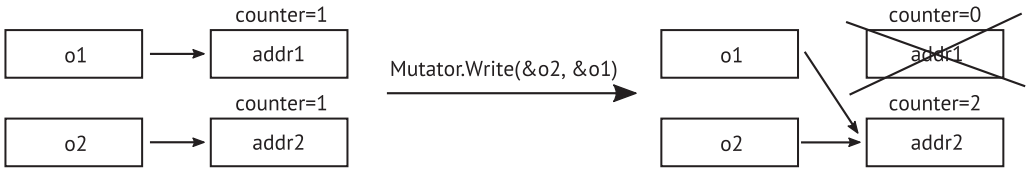

Рис. 1.13 ❖ Иллюстрация подсчета ссылок программе из листинга 1.9

Как видно из листинга 1.9, в операцию `Mutator.Write` добавлены большие накладные расходы. Она должна проверять и изменять счетчик и выполнять освобождение, если тот обращается в ноль. В многозадачной среде (когда параллельно работает несколько модификаторов) эта задача существенно усложняется. Чтобы операции были потокобезопасными, необходимо включить синхронизацию со свойственными ей накладными расходами. Операция `Mutator.Write` встречается очень часто (при любом присваивании), поэтому любые накладные расходы в ней негативно сказываются на времени выполнения программы в целом. К тому же, с точки зрения реализации, не вполне понятно, где хранить счетчики ссылок на объект. Это может быть какая-то выделенная область памяти или некий заголовок, хранящийся настолько близко к объекту, насколько возможно. Но в любом случае каждое присваивание порождает лишнюю запись в память, что крайне нежелательно. Помимо всего прочего, это может привести к неэффективному использованию кеша ЦП, но об этой теме мы будем говорить в следующей главе.

Возвращаясь к вышеупомянутому свойству достижимости, можно сказать, что подсчет ссылок аппроксимирует жизнеспособность по локальным ссылкам, но не пытается отслеживать глобальное состояние графа ссылающихся друг на друга

объектов. В частности, если не принять дополнительных мер, то этот алгоритм будет введен в заблуждение циклическими ссылками, каковые встречаются в таких популярных структурах данных, как двусвязные списки (рис. 1.14). В таком случае счетчик ссылок никогда не обратится в ноль, потому что элементы списка со значениями `value1` и `value2` указывают друг на друга.

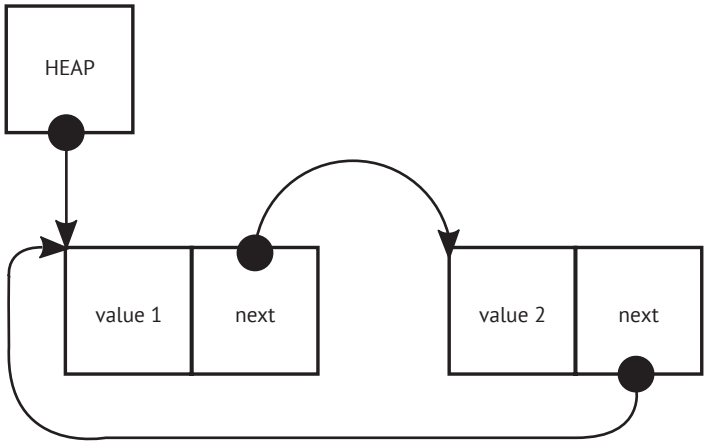

Рис. 1.14 ❖ Проблема подсчета ссылок в случае с циклическими ссылками

Однако создание циклических ссылок можно затруднить на уровне языка, что обнадеживает. В таком случае алгоритм подсчета ссылок можно использовать, не заботясь об утечках памяти, вызванных этой проблемой.

Большим преимуществом и причиной популярности подсчета ссылок является тот факт, что он не требует поддержки со стороны среды выполнения. Его можно реализовать во внешней библиотеке в виде дополнительного механизма для некоторых специальных типов. Следовательно, изначальные методы `Mutator.New` и `Mutator.Write` можно не трогать, а просто ввести высокоуровневые аналоги, например классы с подходящим образом перегруженными операторами и конструкторами. Именно так обстоит дело в большинстве популярных реализаций C++.

Были введены так называемые *умные (интеллектуальные) указатели*, более сложно управляющие жизнью объектов, на которые они указывают. С точки зрения реализации, умные указатели в C++ – это просто шаблоны классов, которые ведут себя как обычные указатели благодаря перегрузке операторов. В C++ есть два вида таких указателей:

- `unique_ptr` реализует семантику монопольного владения (указатель является единственным владельцем объекта, который уничтожается, как только `unique_ptr` покидает область видимости или ему присваивается другой объект);
- `shared_ptr` реализует семантику подсчета ссылок.

С помощью умных указателей код на C++ из листинга 1.5 можно переписать следующим образом:

Листинг 1.10 ❖ Программа на C++, демонстрирующая автоматизированное управление памятью с помощью умных указателей

```
#include <iostream>
#include <memory>

void printReport(std::shared_ptr<int> data)
{
   std::cout << "Отчет: " << *data << "\n";
}

int main()
{
   try
   {
      std::shared_ptr<int> ptr(new int());
      *ptr = 25;
      printReport(ptr);
      return 0;
   }
   catch (std::bad_alloc& ba)
   {
      std::cout << "ОШИБКА: недостаточно памяти\n";
      return 1;
   }
}
```

Если бы мы вызвали метод `data.use_count()` из функции `printReport`, то он вернул бы значение 2, потому что внутри этой функции два разных разделяемых указателя (shared pointers) указывают на один и тот же объект. С другой стороны, после выхода из блока `try` счетчик ссылок будет равен 0, т. к. не осталось ни одного умного указателя на наш объект.

> Отметим, что код в листинге 1.10 не отвечает хорошему стилю программирования на C++. Если требуется только прочитать данные, то лучше передавать умный указатель на них по константной ссылке (`const&`), а не по значению, но тогда счетчик ссылок не увеличился бы, так что пример был бы бесполезен с педагогической точки зрения.

Этот код представляет собой значительное усовершенствование, поскольку:

- нам не нужно вручную уничтожать объект оператором `delete`;
- упрощается обработка исключений, т. к. если функция `printReport()` возбуждает исключение, то умный указатель выйдет из области видимости блока `try` (и всех вложенных в него), поэтому будет автоматически уничтожен. Это следствие вышеупомянутой идиомы *RAII* (*получение ресурса есть инициализация*) – время жизни объекта зависит от области видимости указателя на представляющую его переменную.

Разделяемые и уникальные указатели можно использовать также в полях классов, что делает их весьма мощным и полезным средством.

Проблема в том, что в C++ умные указатели реализованы на уровне библиотеки, а не самого языка. В других библиотеках имеются свои реализации, и иногда взаимодействие между ними затруднительно. В библиотеке Qt имеется класс `QtSharedPointer`, в wxWidgets – `wxSharedPtr<T>` и т. д. Без поддержки со стороны языка

и компилятора это попросту неизбежно. Именно поэтому автоматическое управление памятью так важно в компонентной среде[1] программирования, каковой является .NET. При проектировании .NET передача ответственности за управление памятью от разработчика самой среде выполнения была одним из принципиальных решений. Единый механизм создания, управления и освобождения объектов означает, что все компоненты будут использовать его одним и тем же способом и между компонентами не будет никакой другой связи, кроме самой среды выполнения.

Раз уж речь зашла о C++, интересно отметить, что Бьёрн допустил в стандарте более продвинутый сборщик мусора – он не запрещен, но пока не реализован. Более того, благодаря гибкости C++ сборку мусора можно реализовать в виде дополнительной библиотеки – Memory Pool System или сборщика Бема–Демерса–Вейзера. Мы поговорим об этом чуть ниже.

В других языках умные указатели (включающие подсчет ссылок) могут быть встроены непосредственно, именно так обстоит дело в Rust – современном низкоуровневом языке программирования, созданном в корпорации Mozilla. Он гарантирует безопасность данных на уровне компиляции за счет встраивания концепции умного указателя (на самом деле есть несколько их разновидностей) в сам язык. В языке реализована строгая семантика владения и принцип RAII, что позволяет во время компиляции проверять отсутствие таких нарушений, как разыменование висячего указателя. Другое известное применение подсчета ссылок – встроенный механизм автоматического подсчета ссылок в языке Swift.

Ниже перечислены достоинства и недостатки подсчета ссылок.

Достоинства:

- детерминированный момент освобождения – мы знаем, что освобождение произойдет, когда счетчик ссылок на объект обратится в ноль. Поэтому память будет возвращена системе, как только в ней отпадет необходимость;
- меньшее потребление памяти – поскольку память освобождается сразу, как только объект перестает использоваться, устраняется перерасход памяти из-за того, что она остается занятой в ожидании сборки мусора;
- можно реализовать без поддержки со стороны среды выполнения.

Недостатки:

- наивная реализация типа указанной в листинге 1.8 приводит к очень большим накладным расходам модификатора;
- многопоточные операции со счетчиками ссылок требуют продуманной синхронизации, которая может повлечь дополнительные издержки;
- без дополнительных мер невозможно освободить объекты, связанные циклическими ссылками.

Существуют усовершенствования наивных алгоритмов подсчета ссылок, например *отложенный подсчет ссылок* и *объединенный подсчет ссылок*, которые решают некоторые из этих проблем ценой утраты тех или иных преимуществ (в основном жертвуя немедленным возвратом памяти). Но их описание выходит за рамки этой книги.

---

[1] Когда имеются небольшие взаимозаменяемые зависимости.

Отслеживающий сборщик

Определить достижимость объекта трудно, потому что это глобальный атрибут объекта (он зависит от графа объектов всей программы), а простой явный вызов освобождения объекта – вещь очень локальная. В этом локальном контексте мы ничего не знаем о глобальном – есть ли еще какие-нибудь объекты, пользующиеся данным объектом? Подсчет ссылок – попытка решить эту проблему, снабдив локальный контекст дополнительной информацией – количеством ссылок на объект. Но, очевидно, это может приводить к проблемам при наличии циклических ссылок и, как мы уже видели, имеет ряд других недостатков.

Отслеживающий сборщик мусора опирается на знание глобального контекста времени жизни объекта и может принять более обоснованное решение о том, пора ли уже удалить объект (вернуть память). На самом деле этот подход настолько популярен, что, говоря о сборщике мусора, почти наверняка имеют в виду именно отслеживающий сборщик. Он встречается в .NET, различных реализациях JVM и других средах выполнения.

Основная идея отслеживающего сборщика мусора заключается в том, чтобы определить истинную достижимость объекта, начав с корней модификатора и рекурсивно обойдя весь граф объектов, созданных программой. Очевидно, что это нетривиальная задача, т. к. память процесса может занимать несколько гигабайтов, а проследить все ссылки между объектами при таком объеме данных трудно, особенно если принять во внимание, что модификаторы все время работают и ссылки изменяются. Работа типичного отслеживающего сборщика мусора состоит из двух шагов:

- *пометка* – на этом шаге сборщик определяет достижимость объектов в памяти и решает, какие из них можно убрать в мусор;
- *сборка* – на этом шаге сборщик возвращает системе память, занятую недостижимыми объектами.

Этот простой двухшаговый алгоритм можно расширить, именно так и сделано в .NET, где алгоритм можно описать как пометка–планирование–очистка–уплотнение. Как это устроено, мы подробно обсудим в следующих главах. А пока рассмотрим шаги пометки и сборки в общем виде, поскольку даже это позволяет поднять интересные вопросы.

Этап пометки

На этапе пометки сборщик определяет, какие из находящихся в памяти объектов следует убрать, для чего вычисляет их достижимость. Начав с корней модификатора, сборщик обходит весь граф объектов и помечает те, которые посетил. Объекты, которые остались непомеченными в конце этого этапа, недостижимы. Пометка устраняет проблему циклических ссылок. Если во время обхода графа мы наткнемся на ранее посещенный объект, то просто продолжим обход, как ни в чем не бывало, поскольку этот объект уже помечен.

На рис. 1.15 показано несколько начальных шагов этого алгоритма. Начав с корней, мы движемся внутрь графа объектов по ссылкам между объектами. Как производится обход – в глубину или в ширину, – деталь реализации; на рис. 1.15 показан обход в глубину. Объект может находиться в одном из трех состояний:

- еще не посещен, изображен белым квадратиком;
- помещен в очередь подлежащих посещению, изображен серым квадратиком;
- уже посещен (помечен как достижимый), изображен черным квадратиком.

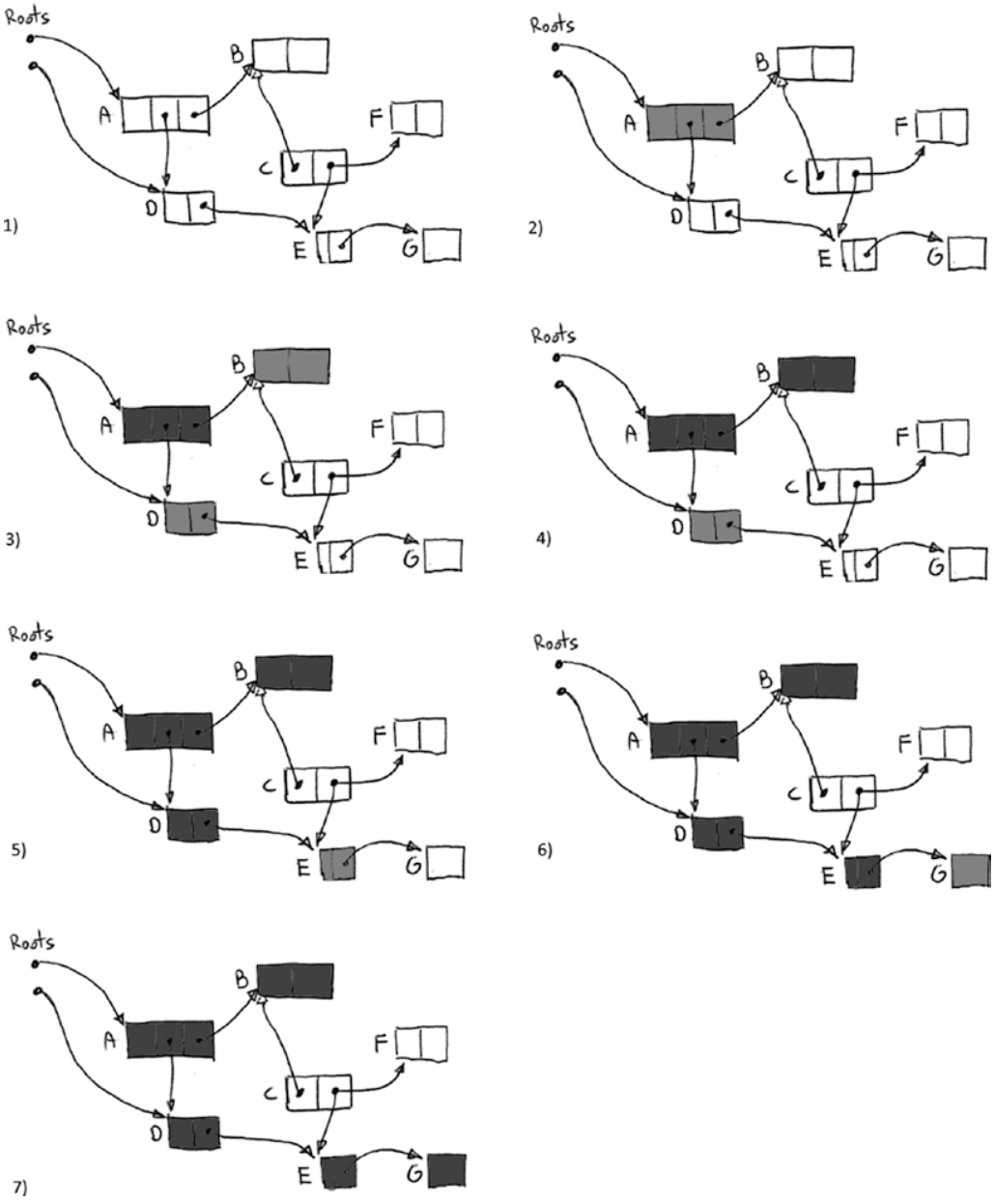

Рис. 1.15 ❖ Несколько начальных шагов фазы пометки

Показанные на рис. 1.15 начальные шаги можно описать словами следующим образом (каждый шаг соответствует одной части рисунка):

1. В начальный момент ни один объект еще не посещен.
2. Объект A запланирован к посещению как первый корень.
3. Поскольку из объекта A ведут указатели (хранящиеся в его полях) на объекты B и D, то они тоже ставятся в очередь на посещение. Сам объект A на этом шаге помечается как достижимый.
4. Посещается следующий объект из очереди ожидающих посещения – объект B. Поскольку из него не выходит ни одной ссылки, он просто помечается как достижимый.
5. Посещается следующий объект из очереди ожидающих посещения – объект D. Он содержит одну ссылку на объект E, который помещается в очередь. Сам объект D помечается как достижимый.
6. Объект G, на который ведет ссылка из объекта E, помещается в очередь на посещение. Сам объект E помечается как достижимый.
7. Посещается последний объект из очереди ожидающих посещения – G. Из него не выходит ссылок, поэтому он просто помечается как достижимый. Больше объектов, требующих посещения, не осталось, поэтому мы выяснили, что объекты C и F недостижимы (мертвы).

Понятно, что обходить такой граф во время нормальной работы модификатора трудно, т. к. он постоянно изменяется как результат работы программы – создаются новые объекты и переменные, полям объектов присваиваются значения и т. д. Поэтому в некоторых реализациях сборщика мусора все модификаторы просто приостанавливаются на время работы фазы пометки. Это позволяет обойти граф безопасно и согласованно. Конечно, как только потоки возобновят выполнение, знания сборщика о графе объектов тут же устаревают. Но это не касается недостижимых объектов – коль скоро объект один раз оказался недостижимым, он уже никогда не станет достижимым. Однако существует немало реализаций сборщика мусора, в которых этап пометки конкурентный, то есть может выполняться одновременно с кодом модификатора. Так устроены популярные алгоритмы CMS в JVM (Concurrent Mark Sweep – конкурентная пометка и очистка), G1 в JVM и сборка мусора в самой среде .NET. Как именно работает конкурентная пометка в .NET, будет подробно описано в главе 11.

На этапе пометки есть одна неочевидная проблема. Для отслеживания достижимости сборщик должен знать о корнях и о том, где в памяти находятся ссылки на другие объекты. Это тривиально, если среда выполнения поддерживает такую информацию. Но есть и обходные пути решения этой проблемы.

Консервативный сборщик мусора

Этот тип сборщика можно считать решением на крайний случай. Его стоит использовать, когда ни среда выполнения, ни компилятор не поддерживают сборку напрямую, т. е. не предоставляют точную информацию о типе (размещение объекта в памяти), и сборщик не получает никакой поддержки от модификатора при работе с указателями. Когда так называемый *консервативный сборщик* хочет узнать, какие объекты достижимы, он просматривает весь стек, области статических данных и регистры. Поскольку без помощи извне он не знает, что является указателем, а что нет, он просто пытается угадать. Для этого сборщик проверяет

несколько условий (какие именно, зависит от реализации), но самое главное из них такое: верно ли, что данное слово, интерпретируемое как адрес (указатель), соответствует действительной, управляемой распределителем области кучи? Если да, то сборщик на всякий случай, *консервативно* (отсюда и название), предполагает, что это в самом деле указатель. И обращается с ним как со ссылкой, по которой нужно проследовать в процессе описанного выше обхода графа на этапе пометки.

Конечно, догадка сборщика может оказаться неверной, поскольку случайные комбинации битов могут выглядеть как указатель на действительный адрес. Тогда мусорная память не будет возвращена системе. Это не слишком частая проблема, потому что числовые значения в памяти обычно малы (счетчики, финансовые данные, индексы), поэтому неприятности могут возникнуть при работе с плотными двоичными данными: растровыми изображениями, числами с плавающей точкой или блоками IP-адресов[1]. Существуют хитроумные улучшения алгоритма, помогающие преодолеть эту проблему, но здесь мы их рассматривать не будем. Кроме того, консервативный подход означает, что объекты нельзя перемещать в памяти. В самом деле, это потребовало бы изменения указателей на перемещенный объект, что, очевидно, невозможно, коль скоро мы не уверены, что нечто, похожее на указатель, действительно таковым является.

Так кому же может понадобиться такой сборщик? Его главное достоинство – возможность работать без поддержки со стороны среды выполнения. По сути дела, он просто просматривает память, так что в поддержке среды выполнения (прослеживании ссылок) не нуждается. Поэтому он удобен, например, при разработке новой среды выполнения, когда полная информация о типах для GC еще отсутствует. Это позволяет заниматься другими частями системы, не останавливая процесс разработки. Майкрософт применяла такую тактику при разработке некоторых версий своей среды выполнения[2].

Однако консервативный сборщик нуждается в поддержке распределителя, чтобы решить проблемы, связанные с неизвестным размещением объекта в памяти. Например, можно организовать выделение памяти таким образом, что объекты одинакового размера будут сгруппированы в сегменты. Консервативный просмотр таких участков возможен, потому что границы объекта определяются путем простого умножения на размер объекта в сегменте.

Во многих языках распределитель можно заменять на уровне языка (библиотеки), что способствует популярности реализации консервативной сборки мусора в виде библиотеки. Одна из самых распространенных реализаций для C и C++, безразличная к API, – *сборщик мусора Бема–Демерса–Вейзера* (или просто *GC Бема*).

Он использовался, например, в Mono (реализация CLR с открытым исходным кодом) вплоть до версии 2.8 (2010 год), а затем был заменен сборщиком мусора *SGen*, в котором реализован смешанный подход – стек и регистры по-прежнему

---

[1] В сборщике мусора Бема и в других консервативных GC разрешается при выделении блока памяти задавать специальный флаг (в случае сборщика Бема – `GC_MALLOC_ATOMIC`), который говорит сборщику, что этот блок не содержит указателей и просматривать его не нужно. В таком блоке можно хранить плотные двоичные данные, например растровые изображения.

[2] Интересно, что .NET содержит реализацию консервативного сборщика мусора, хотя по умолчанию она не включена.

просматриваются консервативно, а просмотр кучи поддерживается информацией о типе во время выполнения.

Коротко перечислим основные свойства консервативного сборщика мусора.

Достоинства:

- подходит для сред, изначально не поддерживающих сборку мусора – например, на ранних этапах разработки среды выполнения и в неуправляемых языках.

Недостатки:

- неточность – все, что случайно выглядит как действительный указатель, препятствует освобождению памяти, хотя это нечастая ситуация, которую можно предотвратить улучшением алгоритма и заданием дополнительных флагов;
- при наивном подходе объекты нельзя перемещать (уплотнять), поскольку сборщик не уверен, что в действительности является указателем (и не может взять и изменить значение только на основании догадки).

Точный сборщик мусора

В так называемом точном сборщике мусора все значительно проще, потому что компилятор и (или) среда выполнения предоставляют сборщику полную информацию о размещении объекта в памяти. Среда может также поддерживать обход стека (перечисление всех корневых объектов в стеке). В таком случае не остается места догадкам. Начав с точно известных корней, сборщик просто просматривает память объект за объектом. Зная, где в памяти находится указатель на начало объекта (или располагая указателем внутрь объекта плюс информацией о том, как такую ссылку интерпретировать), сборщик может просто рекурсивно обойти граф объекта, следуя по таким ссылкам.

В .NET используется точный сборщик мусора, поэтому в следующих главах мы гораздо подробнее узнаем, как он устроен. Собственно, главы с 7 по 10 целиком посвящены этому вопросу.

Этап сборки

После того как отслеживающий сборщик мусора нашел достижимые объекты, он может вернуть системе память, занятую всеми остальными, мертвыми объектами. Этап сборки можно спроектировать по-разному, поскольку у него много аспектов. В одном коротком разделе невозможно описать все возможные комбинации и варианты, но можно и нужно выделить два главных подхода, лежащих в основе различных реализаций.

Очистка

В этом подходе мертвые объекты просто помечаются как свободное пространство, которое можно повторно использовать позже. Эта операция может быть очень быстрой, поскольку (в простейшей реализации) нужно изменить всего один бит в блоке памяти. Такая ситуация показана на рис. 1.16, где неиспользуемые объекты C и F (см. рис. 1.15) становятся доступной памятью после их пометки свободными.

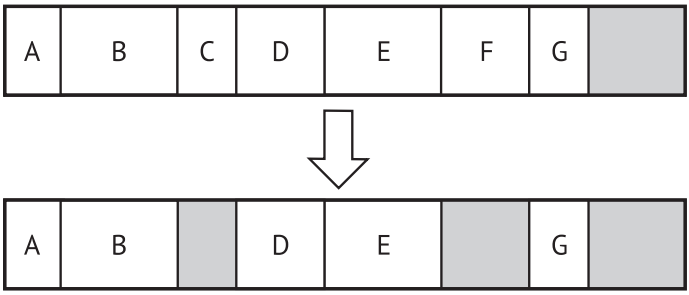

Рис. 1.16 ❖ Сборка очисткой – наивная реализация

Впоследствии в наивной реализации выделения вся память просматривается в поисках свободного участка, размер которого не меньше размера создаваемого объекта.

Но в нетривиальной реализации необходима структура данных, в которой хранится информация о свободных блоках памяти, чтобы их можно было находить быстрее. Обычно для этого используется *список свободных блоков* (рис. 1.17). Более того, эти списки свободных блоков должны быть достаточно умными, чтобы объединять соседние блоки памяти. Возможна и дальнейшая оптимизация – вести несколько списков свободных блоков для блоков памяти разных размеров. Существуют также различные способы просмотра таких списков. Два самых популярных: *лучшее соответствие* и *первое соответствие*. Если используется метод первого соответствия, то просмотр прекращается, как только найден свободный блок подходящего размера. Метод лучшего соответствия подразумевает полный просмотр всего списка свободных блоков с целью найти лучший блок для запрошенного размера. Первый метод быстрее, но может увеличивать фрагментацию, со вторым все наоборот.

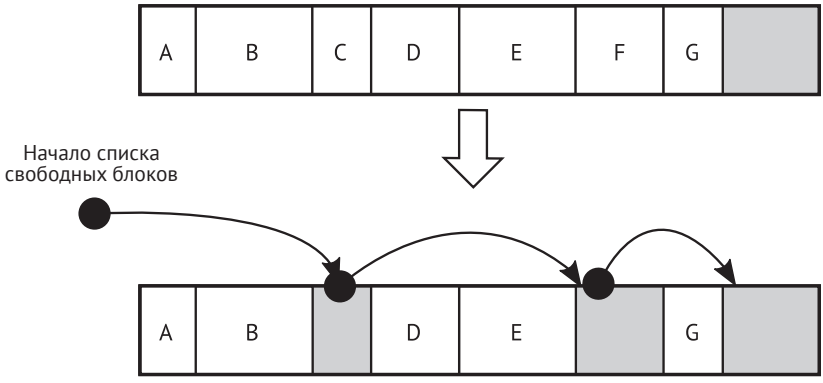

Рис. 1.17 ❖ Сборка очисткой – реализация списка свободных блоков

Хотя подход на основе очистки довольно быстрый, у него есть крупный недостаток – в конечном итоге он приводит к более или менее серьезной фрагментации. По мере создания и уничтожения объектов в куче образуется все больше

мелких или больших свободных участков. Это может привести к ситуации, когда для создания нового объекта памяти достаточно, но вся она распределена между мелкими блоками, а единого сплошного участка нужного размера нет. Мы видели такую ситуацию на рис. 1.11, когда описывали выделение из кучи вообще.

Уплотнение

При таком подходе фрагментация устраняется ценой снижения производительности, потому что требуется перемещать объекты в памяти. Объекты перемещаются таким образом, чтобы убрать пропуски, оставшиеся на месте удаленных объектов. И тут можно, в свою очередь, выделить два разных подхода.

Более простое, с точки зрения реализации, *уплотнение копированием* заключается в том, что все живые (достижимые) объекты копируются в другую область памяти при каждой сборке мусора (рис. 1.18). Уплотнение сводится к простой последовательности операций копирования одного объекта за другим, причем мертвые объекты пропускаются. Очевидно, что это вызывает интенсивное использование памяти (memory traffic), поскольку все живые объекты постоянно перемещаются туда-сюда. К тому же предъявляются повышенные требования к объему памяти, т. к. на время перемещения нужно в два раза больше памяти, чем при нормальной работе.

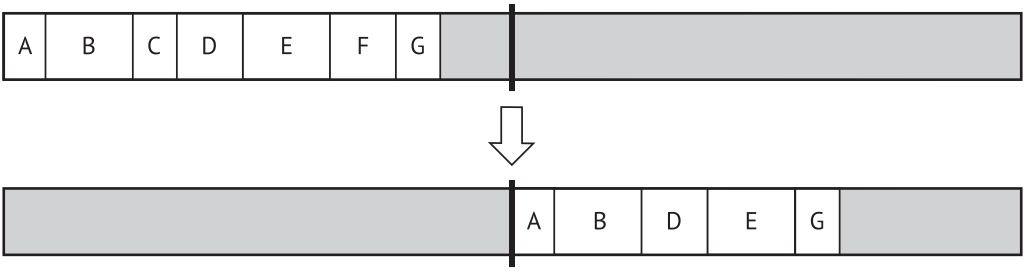

Рис. 1.18 ❖ Сборка уплотнением – реализация путем копирования

Может сложиться впечатление, что из-за этих недостатков алгоритм не имеет практической ценности. Однако его можно использовать эффективно. Нужно только применять его к некоторым небольшим областям памяти, а не ко всей памяти процесса в целом. Именно так делается в некоторых реализациях JVM.

Есть и более сложный алгоритм – *уплотнение на месте*. Объекты сдвигаются друг к другу, так чтобы устранить промежутки между ними (рис. 1.19). Это интуитивно самое естественное решение, так мы поступаем, когда двигаем детали в конструкторе Lego. С точки зрения реализации, это нетривиально, но возможно. Главный вопрос – как перемещать объекты, не затирая один другим и не прибегая к временному буферу?

Как мы увидим в главе 9, в .NET применяется именно этот подход с очень интересной структурой данных для оптимизации, там мы и дадим ответ на этот вопрос.

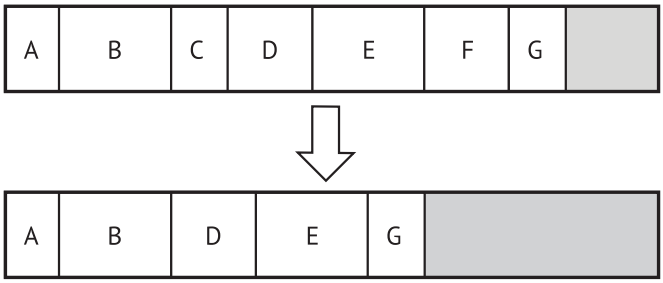

Рис. 1.19 ❖ Сборка уплотнением – реализация на месте

Сравнение сборщиков мусора

Можно задать вопрос: какой сборщик мусора лучше? HotSpot Java 1.8 или .NET 4.6? А быть может, лучше сборщики мусора в Python или Ruby? И что вообще означает «лучше» для GC? Первое и самое важное правило сравнения алгоритмов сборки мусора – всякое сравнение с самого начала необъективно. Дело в том, что GC очень трудно выделить и сравнивать сами по себе. Они настолько переплетены со средой выполнения, что тестировать их по отдельности практически невозможно. Поэтому провести по-настоящему честное сравнение затруднительно. Если бы мы захотели сравнить производительность разных GC, то стали бы измерять такие показатели, как пропускная способность, задержка, время приостановки (в главе 3 мы объясним, что все это такое). Но все они берутся в контексте среды выполнения в целом, а не только GC. Механизмы фреймворка или среды выполнения (к примеру, паттерны выделения, внутренний пул объектов, дополнительная компиляция и прочие скрытые вещи) могут привносить такие накладные расходы, что вклад GC в общую производительность окажется пренебрежимо мал. Кроме того, для любого GC существуют многочисленные настройки, позволяющие ему показывать лучшую производительность на определенных рабочих нагрузках. Одни можно оптимизировать для уменьшения времени реакции в интерактивной среде, другие – для обработки очень больших наборов данных. Третьи могут динамически подстраивать свои характеристики под текущую рабочую нагрузку. Более того, разные GC могут вести себя по-разному в зависимости от конфигурации оборудования (оптимизированы для конкретной архитектуры процессора, количества процессорных ядер или архитектуры памяти).

Конечно, можно сравнивать GC по использованным алгоритмам и предоставляемой функциональности. Есть и еще много способов классификации сборщиков мусора. Как мы уже видели, CG может быть консервативным (Mono до версии 2.8), или точным (.NET), или даже смешанным (Mono 2.8+). В одних реализована сборка очисткой, в других – уплотнением, а в третьих – обоими методами. Еще одно важное различие – как GC делит память на области. В главе 5 мы увидим, как можно разбить кучу на меньшие части. Подсчет ссылок может использоваться иногда или вообще не использоваться. Как реализован распределитель? Является ли GC параллельным или конкурентным (глава 11)? При таком изобилии функциональных различий очень трудно сказать, какая комбинация «лучше». Остановимся на том, что идеального во всех смыслах решения не существует.

Приведем краткий перечень достоинств и недостатков отслеживающего сборщика мусора.

Достоинства:

- полная прозрачность с точки зрения разработчика – память просто представляется бесконечной, и не нужно думать об освобождении памяти, занятой уже не используемыми объектами;

- отсутствуют проблемы из-за циклических ссылок;
- не сильно обременяет модификаторы дополнительными накладными расходами.

Недостатки:

- более сложная реализация;
- недетерминированное освобождение объектов – они освобождаются спустя некоторое время, после того как стали недостижимыми;
- вся работа прекращается (stop the world), когда выполняется этап пометки, – но только в случае неконкурентной реализации;
- повышенный расход памяти – поскольку объекты освобождаются не сразу, потребление памяти может возрастать (мусор продолжает занимать память в течение некоторого времени).

Отслеживающий GC популярен в различных средах выполнения в основном благодаря первому преимуществу.

Немного истории

Заложив основательный теоретический фундамент, бросим беглый взгляд на историю автоматического управления памятью в контексте различных языков программирования.

LISP – один из самых старых до сих пор живущих языков, за время существования которого сменилось множество диалектов, наиболее популярные из которых – Common LISP и Scheme. Но сейчас, без сомнения, шире всего распространен диалект Clojure, который компилируется, среди прочих, для виртуальной машины Java, общеязыковой среды выполнения (.NET) и JavaScript. Это делает его очень гибким и мощным, и, конечно, в этом воплощении LISP встречается со сборкой мусора.

Но не только функциональные языки типа LISP могли похвастаться автоматизированным управлением памятью во времена своего расцвета. Никакой исторический обзор не будет полон без упоминания еще одного языка, оказавшего чрезвычайно сильное влияние, – Simula. Его называют первым по-настоящему объектно-ориентированным языком, в нем впервые появились понятия объектов и классов, наследования, полиморфизма и другие столпы ООП. Все языки, начиная со Smalltalk, а затем через C++ к Java и C#, Python и Ruby, так или иначе черпали идеи из Simula. Что важно, в Simula 67 появилось автоматическое управление памятью, которое поначалу было реализовано с помощью подсчета ссылок и отслеживающего сборщика мусора, а с развитием языка было заменено уплотняющим сборщиком мусора по образцу применяемого в LISP. Благодаря языку Smalltalk сборка мусора приобрела популярность у проектировщиков языков. Возрастание сложности программного обеспечения подталкивало проектировщиков к введению различных по степени изощренности способов освободить программиста от управления памятью.

Популярность веба и начало эры интернета в 1990-х годах заставили индустрию разработки ПО обратиться к программированию на более высоком уровне. Времена беспрекословного доминирования C и C++ миновали. Их способность к управлению системами на низком уровне не имела никакой ценности в контексте веб-программирования и бурного роста серверных приложений. Вместе

с ошеломляюще быстрым развитием интернета возрастала и сложность веб-приложений, и потребность быстрее разрабатывать код.

Невозможно изложить историю автоматического управления памятью, не упомянув языка и платформы Java. Этот язык был задуман компанией Sun как «улучшенный C++», а сборка мусора стала одним из первых и самых фундаментальных требований к новой платформе. Начиная с 1990-х годов, когда проект стартовал под видом внутреннего языка Oak, в нем уже присутствовал механизм пометки и очистки. Первая общедоступная версия Java 1.0a была анонсирована в 1994 году. Взрывной рост популярности Java стал причиной широкой осведомленности о механизмах сборки мусора. Начиная с того времени автоматическое управление памятью стало почти само собой разумеющимся для всех проектировщиков языков высокого уровня.

Уже после рождения Java на свет появились еще два широко распространенных языка: Python и Ruby. Оба были оснащены автоматическим управлением памятью по описанным выше причинам. В Python вплоть до версии 2.0 существовал только подсчет ссылок, но затем были добавлены более сложные способы обращения с циклическими ссылками. Ruby предлагает более простой механизм, основанный на пометке и очистке.

В нашем историческом обзоре нельзя обойти вниманием язык JavaScript, появившийся в те же годы, что и Java. И хотя созвучие с названием Java было больше маркетинговым трюком, чем реальным сходством, JavaScript тоже был задуман как скриптовый язык высокого уровня. В нем не было места ручному управлению памятью. Его целью было манипулирование HTML-содержимым на высоком уровне, не задумываясь о таких вещах, как использование памяти. За это отвечала среда выполнения JavaScript. По мере появления сценариев, в которых программы на JavaScript работали длительное время – одностраничные приложения и серверные программы на node.js, – важность автоматической сборки мусора в движках JavaScript только возрастала. Например, в очень популярном движке V8, который применяется в node.js, используется подход на основе пометки, очистки и уплотнения с дополнительными оптимизациями.

Таким образом, можно отметить, что хотя языки с автоматическим управлением памятью существуют уже 50 лет, настоящий взлет их популярности пришелся на 1990-е годы. И это то место, где можно перейти к истории самой важной и интересной для нас среды – .NET Framework.

Но еще важнее, что в то время Майкрософт уже разработала собственную реализацию JavaScript под названием JScript. JScript играет важную роль в нашей истории, потому что заложил фундамент для решений, использованных при создании .NET. Конечно, больше всего нас интересует вопрос об управлении памятью. На самом деле все началось с языка JScript, написанного четырьмя людьми за несколько выходных. Одним из них был Патрик Дассуд, которого мы можем без всяких сомнений назвать отцом сборщика мусора в .NET. Он написал простой консервативный GC в качестве доказательства концепции.

Прежде чем приступить к работе над CLR, Патрик Дассуд работал над JVM. Да-да, в какой-то момент Майкрософт серьезно подумывала о собственной реализации JVM, вместо того чтобы заняться тем, что мы теперь знаем под названием «среда выполнения .NET». Итак, находясь под воздействием идей JVM и имея в своем багаже реализацию JScript, он написал еще один, по-прежнему консервативный

сборщик мусора. Но команда, которая впоследствии сформировала костяк CLR, быстро поняла, что JVM привносит неудобные ограничения. Во-первых, от новой среды ожидалась всесторонняя поддержка технологии COM и неуправляемого кода. Одной из целей было добиться того, чтобы перекомпиляция программы на C++ с новым флагом /CLR позволила бы запустить ее в новой среде. Кроме того, стандартизация вызывала сложности, и они просто испугались возможных ограничений. В какой-то момент они даже рассматривали возможность выпустить среду выполнения C++ с расширением для сборки мусора.

Позже, посоветовавшись со своим другом (Дэвидом Муном из компании Symbolics, занимавшейся сборщиками мусора на основе поколений), Патрик принял решение написать «лучший из возможных GC» с нуля и реализовал прототип на Common LISP. Почему был выбран этот язык? Потому что он работал на нем много лет и хорошо знал его. Кроме того, у него был опыт работы с лучшими для того времени средствами отладки для LISP. Имея версию на LISP, он затем написал конвертор для транспиляции[1] кода на C++. Так и появился экспериментальный сборщик мусора для экспериментальной же версии JVM. Когда началась работа над CLR, часть этого экспериментального кода была переписана с нуля на C++ и вошла в проект. Так что слухи о том, что код GC для CLR был полностью конвертирован с LISP, – не более чем легенда.

Познакомившись с теоретическими основами и историей, мы можем сформулировать первое из многих правил, которые встретятся в этой книге.

Резюме

В этой главе мы рассмотрели очень широкий круг вопросов. Им вполне можно было бы посвятить несколько книг. Начав с таких базовых понятий, как биты и байты, мы перешли к основным типам компьютерных архитектур – гарвардской и фон Неймана. Мы узнали об основах конструирования компьютеров – регистрах, адресах и словах. Рассмотрев понятия статического и динамического выделения памяти, указателя, стека и кучи, мы добрались до самого главного – идеи автоматического управления памятью, которое еще называют сборкой мусора. Попутно мы поговорили о неудобствах ручного управления памятью и о причинах, по которым его хотелось бы автоматизировать. Фундаментальные для .NET-концепции, в частности отслеживающий сборщик мусора и его этапы – пометка, очистка и уплотнение, мы обсудили лишь в общих чертах. Более подробно они будут рассмотрены в соответствующих главах книги. Мы завершили главу кратким экскурсом в историю и обзором объемлющего контекста, что позволило нам взглянуть на проблему шире.

Полученные сейчас знания дадут нам возможность лучше понять последующие главы. Переходя от главы к главе, мы будем все ближе подбираться к вопросам практической реализации в среде .NET. Но без понимания более широкого контекста, представленного в этой главе, книга была бы неполной. А теперь приглашаю вас к главе 2, где мы перейдем от теоретических основ к конструкции низкоуровневых элементов компьютера и памяти.

---

[1] Транспиляцией называется компиляция исходного кода с одного языка на другой.

Правило 1: учиться, учиться и учиться

Применимость. Самая широкая.

Обоснование. Это самое общее правило в книге, оно применимо в гораздо более широком контексте, чем одно лишь управление памятью. И означает оно, что мы всегда должны быть нацелены на приобретение новых знаний, чтобы оставаться профессионалом. Знание не приходит само по себе. Его нужно заработать. Это утомительный процесс, требующий много времени и труда. Поэтому мы должны постоянно мотивировать себя. Разве эта очевидная истина заслуживает отдельного правила? Я думаю, что да. В повседневной жизни об этом легко забыть. Нам кажется, что решаемые каждый день задачи могут чему-то научить. Ну да, в какой-то степени. Но очевидно, чтобы выйти из зоны комфорта, нужно сделать несколько шагов. Сознательно. А это значит – достать книжку, посмотреть учебное пособие в вебе, прочитать статью. Возможностей много, перечислять их все не имеет смысла. Но это положение настолько фундаментально, что должно занять место в списке правил каждого профессионала. Если мои слова вас не убедили, поинтересуйтесь концепцией Ремесла программиста и манифестом, опубликованным по адресу http://manifesto.softwarecraftsmanship.org. Я также большой поклонник склонности к технике, которую провозглашает гонщица Джеки Стюарт:

> *Чтобы стать водителем гоночного автомобиля, не нужно быть инженером, но нужно иметь склонность к технике.*

Позже эта идея была перенесена в мир ИТ Мартином Томпсоном. Что это означает? Очевидно, чтобы быть гонщиком, необязательно быть механиком. Но без глубокого понимания того, как работает автомобиль, каким правилам механики подчиняется, как работает двигатель, какие силы на него действуют, очень трудно стать хорошим гонщиком. Чтобы гармонично слиться с машиной, нужно «почувствовать» ее. Нужно испытывать склонность к технике. И у нас, программистов, то же самое. Конечно, мы можем просто поразмышлять о таких фреймворках, как .NET или JVM, и на этом остановиться. Но тогда мы уподобимся водителям, выезжающим только по выходным, которые видят в машине лишь руль да педали.

Как применять. Для такого общего правила вряд ли удастся назвать один простой подход. Можете читать книги о том, как работает компьютер или ваш любимый фреймворк. Можете прибегнуть к услугам многочисленных обучающих сервисов. Можете посещать конференции или собрания местных групп пользователей. Можете вести блог и писать на эти темы, поскольку нет лучшего способа научиться, чем начать преподавать. Возможностей много, я даже не буду пытаться перечислить все. Просто помните о девизе «учиться, учиться и учиться» и старайтесь следовать ему в собственной жизни!