Мюррэй Хилл,
Бьярн Страуструп

Язык C++

Содержание

Предисловие

Язык формирует наш способ мышления
и определяет, о чем мы можем мыслить.

Б.Л.Ворф

C++ - универсальный язык программирования, задуманный так, чтобы сделать программирование более приятным для серьезного программиста. За исключением второстепенных деталей C++ является надмножеством языка программирования C. Помимо возможностей, которые дает C, C++ предоставляет гибкие и эффективные средства определения новых типов. Используя определения новых типов, точно отвечающих концепциям приложения, программист может разделять разрабатываемую программу на легко поддающиеся контролю части.

Такой метод построения программ часто называют абстракцией данных. Информация о типах содержится в некоторых объектах типов, определенных пользователем. Такие объекты просты и надежны в использовании в тех ситуациях, когда их тип нельзя установить на стадии компиляции. Программирование с применением таких объектов часто называют объектно-ориентированным. При правильном использовании этот метод дает более короткие, проще понимаемые и легче контролируемые программы.

Ключевым понятием C++ является класс. Класс - это тип, определяемый пользователем. Классы обеспечивают скрытие данных, гарантированную инициализацию данных, неявное преобразование типов

для типов, определенных пользователем, динамическое задание типа, контролируемое пользователем управление памятью и механизмы перегрузки операций. C++ предоставляет гораздо лучшие, чем в C, средства выражнения модульности программы и проверки типов. В языке есть также усовершенствования, не связанные непосредственно с классами, включающие в себя символические константы, inline-подстановку функций, параметры функции по умолчанию, перегруженные имена функций, операции управления свободной памятью и ссылочный тип. В C++ сохранены возможности языка C по работе с основными объектами аппаратного обеспечения (биты, байты, слова, адреса и т.п.). Это позволяет весьма эффективно реализовывать типы, определяемые пользователем.

C++ и его стандартные библиотеки спроектированы так, чтобы обеспечивать переносимость. Имеющаяся на текущий момент реализация языка будет идти в большинстве систем, поддерживающих C. Из C++ программ можно использовать C библиотеки, и с C++ можно использовать большую часть инструментальных средств, поддерживающих программирование на C.

Эта книга предназначена главным образом для того, чтобы помочь серьезным программистам изучить язык и применять его в нетривиальных проектах. В ней дано полное описание C++, много примеров и еще больше фрагментов программ.

Благодарности

C++ никогда бы не созрел без постоянного использования, предложений и конструктивной критики со стороны многих друзей и коллег. Том Карджил, Джим Коплин, Сту Фельдман, Сэнди Фрэзер, Стив Джонсон, Брайэн Керниган, Барт Локанти, Дуг МакИлрой, Дэннис Риччи, Лэрри Рослер, Джерри Шварц и Джон Шопиро подали важные для развития языка идеи. Дэйв Пресотто написал текущую реализацию библиотеки потоков ввода/вывода.

Кроме того, в развитие C++ внесли свой вклад сотни людей, которые присылали мне предложения по усовершенствованию, описания трудностей, с которыми они сталкивались, и ошибки компилятора. Здесь я могу упомянуть лишь немногих из них: Гэри Бишоп, Эндрю Хьюм, Том Карцес, Виктор Миленкович, Роб Мюррэй, Леони Росс, Брайэн Шмальт и Гарри Уокер.

В издании этой книги мне помогли многие люди, в частности, Джон Бентли, Лаура Ивс, Брайэн Керниган, Тэд Ковальски, Стив Махани, Джон Шопиро и участники семинара по C++, который проводился в Bell Labs, Колумбия, Огайо, 26-27 июня 1985 года.

Мюррэй Хилл, Нью Джерси Бьярн Страуструп

Заметки для читателя

"О многом,"-молвил Морж,-"Пришла
пора поговорить."

Л. Кэррол

В этой главе содержится обзор книги, список библиографических ссылок и некоторые замечания по C++ вспомогательного характера. Замечания касаются истории C++, идей, оказавших влияние на разработку C++, и мыслей по поводу программирования на C++. Эта глава не является введением: замечания не обязательны для понимания последующих глав, и некоторые из них предполагают знание C++.

Структура этой книги

Глава 1 - это короткое турне по основным особенностям C++, предназначенное для того, чтобы дать читателю почувствовать язык. Программисты на C первую половину главы могут прочитать очень

быстро; она охватывает главным образом черты, общие для C и C++. Во второй главе описаны средства определения новых типов в C++; начинающие могут отложить более подробное изучение этого до того, как прочтут Главы 2, 3 и 4.

В Главах 2, 3 и 4 описываются средства C++, не включенные в определение новых типов: основные типы, выражения и структуры управления в C++ программах. Другими словами, в них описывается подмножество C++, которое по существу является языком C. Рассмотрение в них проводится гораздо подробнее, но полную информацию можно найти только в справочном руководстве.

В Главах 5, 6 и 7 описываются средства C++ по описанию новых типов, особенности языка, не имеющие эквивалента в C. В Главе 5 приводится понятие базового класса, и показывается, как можно инициализировать объекты типа, определенного пользователем, обращаться к ним и, наконец, убирать их. В Главе 6 объясняется, как для определенного пользователем типа определять унарные и бинарные операции, как задавать преобразования между типами, определенными пользователем, и как как задавать то, каким образом должно обрабатываться каждое создание, уничтожение и копирование значения определенного пользователем типа. Глава 7 описывает концепцию производных классов, которая позволяет программисту строить более сложные классы из более простых, обеспечивать альтернативные интерфейсы класса и работать с объектами безопасным и не требующим беспокоиться о типе способом в тех ситуациях, когда типы объектов не могут быть известны на стадии компиляции.

В Главе 8 представлены классы ostream и istream, предоставляемые стандартной библиотекой для осуществления ввода-вывода. Эта глава имеет двоякую цель: в ней представлены полезные средства, что одновременно является реальным примером использования C++.

И, наконец, в книгу включено справочное руководство по C++.

Ссылки на различные части этой книги даются в форме #2.3.4 (Глава 2 подраздел 3.4). Глава с - это справочное руководство; например, #c.8.5.5.

Замечания по реализации

Во время написания этой книги все реализации C++ использовали версии единственного интерфейсного компилятора#. Он используется на многих архитектурах, включая действующие версии системы операционной системы UNIX на AT&T 3B, DEC VAX, IBM 370 и Motorolla 68000. Фрагменты программ, которые приводятся в этой книге, взяты непосредственно из исходных файлов, которые компилировались на 3B в UNIX System V версии 2 [15], VAX11/750 под 8-ой Редакцией UNIX [16] и CCI Power 6/32 под BSD4.2 UNIX [17]. Язык, описанный в этой книге, - это "чистый C++", но имеющиеся на текущий момент компиляторы реализуют большое число "анахронизмов" (описанных в #c.15.3), которые должны способствовать переходу от C к C++.

Упражнения

Упражнения находятся в конце глав. Все упражнения главным образом типа напишите-программу. Для решения всегда пишите такую прогармму, которая будет компилироваться и работать по меньшей мере на нескольких тестовых случаях. Упражнения различаются в основном по сложности, поэтому они помечены оценкой степени сложности. Шкала экспоненциальная, так что если на упражение (*1) вам потребовалось пять минут, то упражнение (*2) вам может потребоваться час, а на (*3) - день. Время, которое требуется на то, чтобы написать и оттестировать программу, зависит больше от опыта читателя, нежели от самого упражения. Упражнение (*1) может отнять день, если для того, чтобы запустить ее, читателю сначала придется знакомиться с новой вычислительной системой. С другой стороны, тот, у кого под рукой окажется нужный набор программ, может сделать упражнение (*5) за час. В качестве источника упражнений к Главам 2-4 можно использовать любую книгу по C. У Ахо и др. [1] приведено большое

количество общих структур данных и алгоритмов втерминах абстрактных типов данных. Эту книгу также может служить источником упражнений к Главам 5-7. Однако языку, который в этой книге использовался, недостает как функций членов, так и производных классов. Поэтому определенные пользователем типы часто можно выражать в C++ более элегантно.

Замечания по проекту языка

Существенным критерием при разработке языка была простота; там, где возникал выбор между упрощением руководства по языку и другой документации и упрощением компилятора, выбиралось первое. Огромное значение также придавалось совместимости с C; это помешало удалить синтаксис C.

В C++ нет типов данных высокого уровня и нет первичных операций высокого уровня. В нем нет, например, матричного типа с операцией обращения или типа строка с операцией конкатенации. Если пользователю понадобятся подобные типы, их можно определить в самом

C++ можно купить в AT&T, Software Sales and Marketing, PO Box 25000, Greensboro, NC 27420, USA (телефон 800-828-UNIX) или в ваших местных организациях, осуществляющих продажу Системы UNIX. (прим. автора)

языке. По сути дела, основное, чем занимается программирование на C++, - это определение универсальных и специально-прикладных типов. Хорошо разработанный тип, определенный пользователем, отличается от встроенного типа только способом определения, но не способом использования.

Исключались те черты, которые могли бы повлечь дополнительные расходы памяти или времени выполнения. Например, мысли о том, чтобы сделать необходимым хранение в каждом объекте "хозяйственной" информации, были отвергнуты; если пользователь описывает структуру, состоящую из двух 16-битовых величин, то структура поместится в 32-битовый регистр.

C++ проектировался для использования в довольно традиционной среде компиляции и выполнения, среде программирования на C в системе UNIX. Средства обработки особых ситуаций и параллельного программирования, требующие нетривиальной загрузки и поддержки в процессе выполнения, не были включены в C++. Вследствие этого реализация C++ очень легко переносима. Однако есть полные основания использовать C++ в среде, где имеется гораздо более существенная поддержка. Такие средства, как динамическая загрузка, пошаговая трансляция и база данных определений типов могут с пользой применяться без воздействия на язык.

Типы и средства скрытия данных в C++ опираются на проводимый во время компиляции анализ программ с целью предотвращения случайного искажения данных. Они не обеспечивают секретности или защиты от умышленного нарушения правил. Однако эти средства можно использовать без ограничений, что не приводит к дополнительным расходам времени на выполнение или пространства памяти.

Исторические замечания

Безусловно, C++ восходит главным образом к C [7]. C сохранено как подможество, поэтому сделанного в C акцента на средствах низкого уровня достаточно, чтобы справляться с самыми насущными задачами системного программирования. C, в свою очередь, многим обязано своему предшественнику BCPL [9]; на самом деле, комментарии // (заново) введены в C++ из BCPL. Если вы знаете BCPL, то вы заметите, что в C++ по-прежнему нет VALOF блока. Еще одним источником вдохновения послужил язык Simula67 [2,3]; из него была позаимствована концепция класса (вместе с производными классами и функциями членами). Это было сделано, чтобы способствовать

модульности через использование виртуальных функций. Возможности C++ по перегрузке операций и свобода в расположении описаний везде, где может встречаться оператор, похожи на Алгол68 [14].

Название C++ - изобретение совсем недавнее (лета 1983его). Более ранние версии языка использовались начиная с 1980ого и были известны как "C с Классами". Первоначально язык был придуман потому, что автор хотел написать модели, управляемые прерываниями, для чего был бы идеален Simula67, если не принимать во внимание эффективность. "C с Классами" использовался для крупных проектов моделирования, в которых строго тестировались возможности написания программ, требующих минимального (только) пространства памяти и времени на выполнение. В "C с Классами" не хватало перегрузки операций, ссылок, виртуальных функций и многих деталей. C++ был впервые введен за пределами исследовательской группы автора в июле 1983его; однако тогда многие особенности C++ были еще не придуманы.

Название C++ выдумал Рик Масситти. Название указывает на эволюционную природу перехода к нему от C. "++" - это операция приращения в C. Чуть более короткое имя C+ является синтаксической ошибкой; кроме того, оно уже было использовано как совсем другого языка. Знатоки семантики C находят, что C++ хуже, чем ++C. Названия D язык не получил, поскольку он является расширением C и в нем не делается попыток исцеляться от проблем путем выбрасывания различных особенностей. Еще одну интерпретацию названия C++ можно найти в приложении к Оруэллу [8].

Изначально C++ был разработан, чтобы автору и его друзьям не приходилось программировать на ассемблере, C или других современных языках высокого уровня. Основным его предназначением было сделать написание хороших программ более простым и приятным для отдельного программиста. Плана разработки C++ на бумаге никогда не было; проект, документация и реализация двигались одновременно. Разумется, внешний интерфейс C++ был написан на C++. Никогда не существовало "Проекта C++" и "Комитета по разработке C++". Поэтому C++ развивался и продолжает развиваться во всех направлениях чтобы справляться со сложностями, с которыми сталкиваются пользователи, а также в процессе дискуссий автора с его друзьями и коллегами.

В качестве базового языка для C++ был выбран C, потому что он (1) многоцелевой, лаконичный и относительно низкого уровня; (2) отвечает большинству задач системного программирования; (3) идет везде и на всем; и (4) пригоден в среде программирования UNIX. В C есть свои сложности, но в наспех спроектированном языке тоже были бы свои, а сложности C нам известны. Самое главное, работа с C позволила "C с Классами" быть полезным (правда, неудобным) инструментом в ходе первых месяцев раздумий о добавлении к C Simula-образных классов.

C++ стал использоваться шире, и по мере того, как возможности, предоставляемые им помимо возможностей C, становились все более существенными, вновь и вновь поднимался вопрос о том, сохранять ли совместимость с C. Ясно, что отказавшись от определенной части наследия C можно было бы избежать ряда проблем (см., например, Сэти [12]). Это не было сделано, потому что (1) есть миллионы строк на C, которые могли бы принести пользу в C++ при условии, что их не нужно было бы полностью переписывать с C на C++; (2) есть сотни тысяч строк библиотечных функций и сервисных программ, написанных на C, которые можно было бы использовать из или на C++ при условии, что C++ полностью совместим с C по загрузке и синтаксически очень похож на C; (3) есть десятки тысяч программистов, которые знают C, и которым, поэтому, нужно только научиться использовать новые особенности C++, а не заново изучать его основы; и (4), поскольку C++ и C будут использоваться на одних и тех же системах одними и теми же людьми, отличия должны быть либо очень большими, либо очень маленькими, чтобы свести к минимуму ошибки и недоразумения. Позднее была проведена проверка определения C++, чтобы удостовериться в том, что любая конструкциия, допустимая и в C и в C++, действительно означает в обоих языках одно и то же.

Язык C сам эволюционировал за последние несколько лет, частично

под влиянием развития C++ (см. Ростлер [11]). Предварительный грубый ANSI стандарт C [10] содержит синтаксис описаний функций, заимствованный из "C с Классами". Заимствование идей идет в обе стороны; например, указатель void* был придуман для ANSI C и

впервые реализован в C++. Когда ANSI стандарт разовьется несколько дальше, придет время пересмотреть C++, чтобы удалить необоснованную несовместимость. Будет, например, модернизирован препроцессор (#с.11), и нужно будет, вероятно, отрегулировать правила осуществления плавающей арифметики. Это не должно оказаться болезненным, и C и ANSI C очень близки к тому, чтобы стать подмножествами C++ (см. #с.11).

Эффективность и структура

C++ был развит из языка программирования C и за очень немногими исключениями сохраняет C как подмножество. Базовый язык, C подмножество C++, спроектирован так, что имеется очень близкое соответствие между его типами, операциями и операторами и компьютерными объектами, с которыми непосредственно приходится иметь дело: числами, символами и адресами. За исключением операций свободной памяти new и delete, отдельные выражения и операторы C++ обычно не нуждаются в скрытой поддержке во время выполнения или подпрограммах.

В C++ используются те же последовательности вызова и возврата из функций, что и в C. В тех случаях, когда даже этот довольно эффективный механизм является слишком дорогим, C++ функция может быть подставлена inline, удовлетворяя, таким образом, соглашению о записи функций без дополнительных расходов времени выполнения.

Одним из первоначальных предназначений C было применение его вместо программирования на ассемблере в самых насущных задачах системного программирования. Когда проектировался C++, были приняты меры, чтобы не ставить под угрозу успехи в этой области. Различие между C и C++ состоит в первую очередь в степени внимания, уделяемого типам и структурам. C выразителен и снисходителен. C++ еще более выразителен, но чтобы достичь этой выразительности, программист должен уделить больше внимания типам объектов. Когда известны типы объектов, компилятор может правильно обрабатывать выражения, тогда как в противном случае программисту пришлось бы задавать действия с мучительными подробностями. Знание типов объектов также позволяет компилятору обнаруживать ошибки, которые в противном случае остались бы до тестирования. Заметьте, что использование системы типов для того, чтобы получить проверку параметров функций, защитить данные от случайного искажения, задать новые операции и т.д., само по себе не увеличивает расходов по времени выполнения и памяти.

Особое внимание, уделенное при разработке C++ структуре, отразилось на возрастании масштаба программ, написанных со времени разработки C. Маленькую программу (меньше 1000 строк) вы можете заставить работать с помощью грубой силы, даже нарушая все правила хорошего стиля. Для программ больших размеров это не совсем так. Если программа в 10 000 строк имеет плохую структуру, то вы обнаружите, что новые ошибки появляются так же быстро, как удаляются старые. C++ был разработан так, чтобы дать возможность разумным образом структурировать большие программы таким образом, чтобы для одного человека не было непомерным справляться с программами в 25 000 строк. Существуют программы гораздо больших размеров, однако те, которые работают, в целом, как оказывается, состоят из большого числа почти независимых частей, каждая из которых намного ниже указанных пределов. Естественно, сложность написания и поддержки программы зависит от сложности разработки, а не просто от числа строк текста программы, так что точные цифры, с помощью которых были выражены предыдущие соображения, не следует воспринимать слишком серьезно.

Не каждая часть программы, однако, может быть хорошо структурирована, независима от аппаратного обеспечения, легко читаема и т.п. C++ обладает возможностями, предназначенные для того, чтобы непосредственно и эффективно работать с аппаратными средствами, не беспокоясь о безопасности или простоте понимания. Он также имеет возможности, позволяющие скрывать такие программы за элегантными и надежными интерфейсами.

В этой книге особый акцент делается на методах создания универсальных средств, полезных типов, библиотек и т.д. Эти средства пригодятся как тем программистам, которые пишут небольшие программы, так и тем, которые пишут большие. Кроме того, поскольку все нетривиальные программы состоят из большого числа полунезависимых частей, методы написания таких частей пригодятся и системным, и прикладным программистам.

У кого-то может появиться подозрение, что спецификация программы с помощью более подробной системы типов приведет к увеличению исходных текстов программы. В C++ это не так; C++ программа, описывающая типы параметров функций, использующая классы и т.д., обычно немного короче эквивалентной C программы, в которой эти средства не используются.

Философские замечания

Язык программирования служит двум связанным между собой целям: он дает программисту аппарат для задания действий, которые должны быть выполнены, и формирует концепции, которыми пользуется программист, размышляя о том, что делать. Первой цели идеально отвечает язык, который настолько "близок к машине", что всеми основными машинными аспектами можно легко и просто оперировать достаточно очевидным для программиста образом. С таким умыслом первоначально задумывался C. Второй цели идеально отвечает язык, который настолько "близок к решаемой задаче", чтобы концепции ее решения можно было выражать прямо и коротко. С таким умыслом предварительно задумывались средства, добавленные к C для создания C++.

Связь между языком, на котором мы думаем/программируем, и задачами и решениями, которые мы можем представлять в своем воображении, очень близка. По этой причине ограничивать свойства языка только целями исключения ошибок программиста в лучшем случае опасно. Как и в случае с естественными языками, есть огромная польза быть по крайней мере двуязычным. Язык предоставляет программисту набор концептуальных инструментов; если они не отвечают задаче, то их просто игнорируют. Например, серьезные ограничения концепции указателя заставляют программиста применять вектора и целую арифметику, чтобы реализовать структуры, указатели и т.п. Хорошее проектирование и отсутствие ошибок не может гарантироваться чисто за счет языковых средств.

Система типов должна быть особенно полезна в нетривиальных задачах. Действительно, концепция классов в C++ показала себя мощным концептуальным средством.

Размышления о программировании на C++

В идеальном случае подход к разработке программы делится на три части: вначале получить ясное понимание задачи, потом выделить ключевые идеи, входящие в ее решение, и наконец выразить решение в виде программы. Однако подробности задачи и идеи решения часто становятся ясны только в результате попытки выразить их в виде программы - именно в этом случае имеет значение выбор языка программирования.

В большинстве разработок имеются понятия, которые трудно представить в программе в виде одного из основных типов или как функцию без ассоциированных с ней статических данных. Если имеется подобное понятие, опишите класс, представляющий его в программе. Класс - это тип; это значит, что он задает поведение объектов его

класса: как они создаются, как может осуществляться работа с ними, и как они уничтожаются. Класс также задает способ представления объектов; но на ранних стадиях разработки программы это не является (не должно являться) главной заботой. Ключом к написанию хорошей программы является разработка таких классов, чтобы каждый из них представлял одно основное понятие. Обычно это означает, что программист должен сосредоточиться на вопросах: Как создаются объекты этого класса? Могут ли эти объекты копироваться и/или уничтожаться? Какие действия можно производить над этими объектами? Если на такие вопросы нет удовлетворительных ответов, то во-первых, скорее всего, понятие не было "ясно", и может быть неплохо еще немного подумать над задачей и предлагаемым решением вместо того, чтобы сразу начинать "программировать вокруг" сложностей.

Проще всего иметь дело с такими понятиями, которые имеют традиционную математическую форму: числа всех видов, множества, геометрические фигуры и т.п. На самом деле, следовало бы иметь стандартные библиотеки классов, представляющих такие понятия, но к моменту написания это не имело места. C++ еще молод, и его библиотеки не развились еще до той же степени, что и сам язык.

Понятие не существует в пустоте, всегда есть группы связанных между собой понятий. Организовать в программе взаимоотношения между классами, то есть определить точную взаимосвязь между различными понятиями, часто труднее, чем сначала спланировать отдельные классы. Лучше, чтобы не получилось неразберихи, когда каждый класс (понятие) зависит от всех остальных. Рассмотрим два класса, A и B. Взаимосвязи вроде "A вызывает функции из B", "A создает объекты B" и "A имеет члены B" редко вызывают большие сложности, а взаимосвязь вроде "A использует данные из B" обычно можно исключить (просто не используйте открытые данные-члены). Неприятными, как правило, являются взаимосвязи, которые по своей природе имеют вид "A есть B и ...".

Одним из наиболее мощных интеллектуальных средств, позволяющих справляться со сложностью, является иерархическое упорядочение, то есть организация связанных между собой понятий в древовидную структуру с самым общим понятием в корне. В C++ такие структуры представляются производными классами. Часто можно организавать программу как множество деревьев (лес?). То есть, программист задает набор базовых классов, каждый из которых имеет свое собственное множество производных классов. Для определения набора действий для самой общей интерпретации понятия (базового класса) часто можно использовать виртуальные функции (#7.2.8). Интерпретацию этих действий можно, в случае необходимости, усовершенствовать для отдельных специальных классов (производных классов).

Естественно, такая организация имеет свои ограничения. В частности, множество понятий иногда лучше организуется в виде ациклического графа, в котором понятие может непосредственно зависеть от более чем одного другого понятия; например, "A есть B и C и ...". В C++ нет непосредственной поддержки этого, но подобные связи можно представить, немного потеряв в элегантности и сделав малость дополнительной работы (#7.2.5).

Иногда для организации понятий некоторой программы оказывается непригоден даже ациклический граф; некоторые понятия оказываются взаимозависимыми по своей природе. Если множество взаимозависимых классов настолько мало, что его легко себе представить, то циклические зависимости не должны вызвать сложностей. Для представления множеств взаимозависимых классов с C++ можно использовать идею friend классов (#5.4.1).

Если вы можете организовать понятия программы только в виде общего графа (не дерева или ациклического направленного графа), и если вы не можете локализовать взаимные зависимости, то вы, по всей видимости, попали в затруднительное положение, из которого вас не выручит ни один язык программирования. Если вы не можете представить какой-либо просто формулируемой зависимости между основными понятиями, то скорее всего справиться с программой не удастся.

Напомню, что большую часть программирования можно легко и

очевидно выполнять, используя только простые типы, структуры данных, обычные функции и небольшое число классов из стандартной библиотеки. Весь аппарат, входящий в определение новых типов, не следует использовать за исключением тех случаев, когда он действительно нужен.

Вопрос "Как пишут хорошие программы на C++" очень похож на вопрос "Как пишут хорошую английскую прозу?" Есть два вида ответов: "Знайте, что вы хотите сказать" и "Практикуйтесь. Подражайте хорошему языку." Оба совета оказываются подходящими к C++ в той же мере, сколь и для английского - и им столь же трудно следовать.

Правила Правой Руки (*)

Здесь приводится набор правил, которых вам хорошо бы придерживаться изучая C++. Когда вы станете более опытны, вы можете превратить их в то, что будет подходить для вашего рода деятельности и вашего стиля программирования. Они умышленно сделаны очень простыми, поэтому подробности в них опущены. Не воспринимайте их чересчур буквально. Написание хороших программ требует ума, вкуса и терпения. Вы не собираетесь как следует понять это с самого начала; поэкспериментируйте!

[1] Когда вы программируете, вы создаете конкретное представление идей вашего решения некоторой задачи. Пусть структура отражает эти идеи настолько явно, насколько это возможно:
  - [a] Если вы считате "это" отдельным понятием, сделайте его классом.
  - [b] Если вы считате "это" отдельным объектом, сделайте его объектом некоторого класса.
  - [c] Если два класса имеют общим нечто существенное, сделайте его базовым классом. Почти все классы в вашей программе будут иметь нечто общее; заведите (почти) универсальный базовый класс, и разработайте его наиболее тщательно.

[2] Когда вы определяете класс, который не реализует некоторый математический объект, вроде матрицы или комплексного числа, или тип низкого уровня, вроде связанного списка, то:
  - [a] Не используйте глобальные данные.
  - [b] Не используйте глобальные функции (не члены).
  - [c] Не используйте открытые данные-члены.
  - [d] Не используйте друзей, кроме как чтобы избежать [a], [b] или [c].
  - [e] Не обращайтесь к данным-членам или другим объектам непосредственно.
  - [f] Не помещайте в класс "поле типа"; используйте виртуальные функции.
  - [g] Не используйте inline-функции, кроме как средство существенной оптимизации.

(*) Некоторые легко запоминаемые эмпирические правила, "Правила-помошники." (прим. перев.)

Замечания для программистов на C

Чем лучше кто-нибудь знает C, тем труднее окажется избежать писания на C++ в стиле C, теряя, тем самым, некоторые возможные выгоды C++. Поэтому проглядите, пожалуйста, раздел "Отличия от C" в справочном руководстве (#c.15). Там указывается на области, в которых C++ позволяет делать что-то лучше, чем C. Макросы (#define) в C++ почти никогда не бывают необходимы; чтобы определять провозглашаемые константы, используйте const (#2.4.6) или enum (#2.4.7), и inline (#1.12) - чтобы избежать лишних расходов на

вызов функции. Старайтесь описывать все функции и типы всех параметров - есть очень мало веских причин этого не делать. Аналогично, практически нет причин описывать локальную переменную не инициализируя ее, поскольку описание может появляться везде, где может стоять оператор, - не описывайте переменную, пока она вам не нужна. Не используйте malloc() - операция new (#3.2.6) делает ту же работу лучше. Многие объединения не нуждаются в имени - используйте безымянные объединения (#2.5.2).

Глава 1

Турне по C++

Единственный способ изучать новый язык программирования - писать на нем программы.
- Брайэн Керниган

Эта глава представляет собой краткий обзор основных черт языка программирования C++. Сначала приводится программа на C++, затем показано, как ее откомпилировать и запустить, и как такая программа может выводить выходные данные и считывать входные. В первой трети этой главы после введения описаны наиболее обычные черты C++: основные типы, описания, выражения, операторы, функции и структура программы. Оставшаяся часть главы посвящена возможностям C++ по определению новых типов, скрытию данных, операциям, определяемым пользователем, и иерархии определяемых пользователем типов.

1.1 Введение

Это турне проведет вас через ряд программ и частей программ на C++. К концу у вас должно сложиться общее представление об основных особенностях C++, и будет достаточно информации, чтобы писать простые программы. Для точного и полного объяснения понятий, затронутых даже в самом маленьком законченном примере, потребовалось бы несколько страниц определений. Чтобы не превращать эту главу в описание или в обсуждение общих понятий, примеры снабжены только самыми короткими определениями используемых терминов. Термины рассматриваются позже, когда будет больше примеров, способствующих обсуждению.

1.1.1 Вывод

Прежде всего, давайте напишем программу, выводящую строку выдачи:

```
#include <iostream.h>

main()
{
    cout << "Hello, world\n";
}
```

Строка #include сообщает компилятору, чтобы он включил стандартные возможности потока ввода и вывода, находящиеся в файле stream.h. Без этих описаний выражеине cout << "Hello, world\n" не имело бы смысла. Операция << ("поместить в"*) пишет свой первый аргумент во второй (в данном случае, строку "Hello, world\n" в

---

* Программирующим на C << известно как операция сдвига влево для целых. Такое использование << не утеряно; просто в дальнейшем << было определено для случая, когда его левый операнд является потоком вывода. Как это делается, описано в #1.8. (прим. автора)

стандартный поток вывода cout). Строка - это последовательность символов, заключенная в двойные кавычки. В строке символ обратной косой \, за которым следует другой символ, обозначает один специальный символ; в данном случае, \n является символом новой строки. Таким образом выводимые символы состоят из Hello, world и перевода строки.

Остальная часть программы

```
main() { ... }
```

определяет функцию, названную main. Каждая программа должна содержать функцию с именем main, и работа программы начинается с выполнения этой функции.

1.1.2 Компиляция

Откуда появились выходной поток cout и код, реализующий операцию вывода <<? Для получения выполняемого кода написанная на C++ программа должна быть скомпилирована; по своей сути процесс компляции такой же, как и для C, и в нем участвует большая часть входящих в последний программ. Производится чтение и анализ текста программы, и если не обнаружены ошибки, то генерируется код. Затем программа проверяется на наличие имен и операций, которые использовались, но не были определены (в нашем случае это cout и <<). Если это возможно, то программа делается полной посредством дополнения недостающих определений из библиотеки (есть стандартные библиотеки, и пользователи могут создавать свои собственные). В нашем случае cout и << были описаны в stream.h, то есть, были указаны их типы, но не было дано никаких подробностей относительно

их реализации. В стандартной библиотеке содержится спецификация пространства и инициализирующий код для cout и <<. На самом деле, в этой библиотеке содержится и много других вещей, часть из которых описана в stream.h, однако к скомпилированной версии добавляется только подмножество библиотеки, необходимое для того, чтобы сделать нашу программу полной.

Команда компиляции в C++ обычно называется CC. Она используется так же, как команда cc для программ на C; подробности вы можете найти в вашем руководстве. Предположим, что программа с "Hello, world" хранится в файле с именем

hello.c, тогда вы можете ее скомпилировать и запустить примерно так ($ - системное приглашение):

```
$ CC hello.c
$ a.out
Hello,world
$
```

a.out - это принимаемое по умолчанию имя исполняемого результата компиляции. Если вы хотите назвать свою программу, вы можете сделать это с помощью опции -o:

```
$ CC hello.c -o hello
$ hello
Hello,world
$
```

1.1.3 Ввод

Следующая (довольно многословная) программа предлагает вам ввести число дюймов. После того, как вы это сделаете, она напечатает соответствующее число сантиметров.

```
#include <iostream.h>


main()
{
    int inch = 0;         // inch - дюйм
    cout << "inches";
    cin >> inch;
    cout << inch;
    cout << " in = ";
    cout << inch*2.54;
    cout << " cm\n";
}
```

Первая строка функции main() описывает целую переменную inch. Ее значение считывается с помощью операции >> ("взять из") над стандартным потоком ввода cin. Описания cin и >>, конечно же, находятся в . После ее запуска ваш терминал может выглядеть примерно так:

```
$ a.out
inches=12
12 in = 30.48 cm
$
```

В этом примере на каждую команду вывода приходится один оператор; это слишком длинно. Операцию вывода << можно применять к ее собственному результату, так что последние четыре команды вывода можно было записать одним оператором:

```
cout << inch << " in = " << inch*2.54 << " cm\n";
```

В последующих разделах ввод и вывод будут описаны гораздо более подробно. Вся эта глава фактически может рассматриваться как объяснение того, как можно написать предыдущие программы на языке, который не обеспечивает операции ввода-вывода. На самом деле, приведенные выше программы написаны на C++, "расширенном" операциями ввода-вывода посредством использования библиотек и включения файлов с помощью #include. Другими словами, язык C++ в том виде, в котором он описан в справочном руководстве, не определяет средств ввода-вывода; вместо этого исключительно с помощью средств, доступных любому программисту, определены операции << и >>.

1.2 Комментарии

Часто бывает полезно вставлять в программу текст, который предназначается в качестве комментария только для читающего программу человека и игнорируется компилятором в программе. В C++ это можно сделать одним из двух способов.

Символы /* начинают комментарий, заканчивающийся символами */. Вся эта последовательность символов эквивалентна символу пропуска (например, символу пробела). Это наиболее полезно для многострочных комментариев и изъятия частей программы при редактировании, однако следует помнить, что комментарии /* */ не могут быть вложенными.

Символы // начинают комментарий, который заканчивается в конце строки, на которой они появились. Опять, вся последовательность символов эквивалентна пропуску. Этот способ наиболее полезен для коротких комментариев. Символы // можно использовать для того, чтобы закомментировать символы /* или */, а символами /* можно закомментировать //.

1.3 Типы и Описания

Каждое имя и каждое выражение имеет тип, определяющий операции, которые могут над ними производиться. Например, описание

```
int inch;
```

определяет, что inch имеет тип int, то есть, inch является целой переменной.

Описание - это оператор, который вводит имя в программе. Описание задает тип этого имени. Тип определяет правильное использование имени или выражения. Для целых определены такие операции, как +, -, * и /. После того, как включен файл stream.h, объект типа int может также быть вторым операндом <<, когда первый операнд ostream.

Тип объекта определяет не только то, какие операции могут к нему применяться, но и смысл этих операций. Например, оператор

```
cout << inch << " in = " << inch*2.54 << " cm\n";
```

правильно обрабатывает четыре входных значения различным образом. Строки печатаются буквально, тогда как целое inch и значение с плавающей точкой inch*2.54 преобразуются из их внутреннего представлениия в подходящее для человеческого глаза символьное представление.

В C++ есть несколько основных типов и несколько способов создавать новые. Простейшие виды типов C++ описываются в следующих разделах, а более интересные оставлены на потом.

1.3.1 Основные Типы

Основные типы, наиболее непосредственно отвечающие средствам аппаратного обеспечения, такие:

```
char  short  int  long  float  double
```

Первые четыре типа используются для представления целых, последние два - для представления чисел с плавающей точкой. Переменная типа char имеет размер, естественный для хранения символа на данной машине (обычно, байт), а переменная типа int имеет размер, соответствующий целой арифметике на данной машине (обычно, слово). Диапазон целых чисел, которые могут быть представлены типом, зависит от его размера. В C++ размеры измеряются в единицах размера данных типа char, поэтому char по определению имеет размер единица. Соотношение между основными типами можно записать так:

```
1 = sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(float) <= sizeof(double)
```

В целом, предполагать что-либо еще относительно основных типов неразумно. В частности, то, что целое достаточно для хранения указателя, верно не для всех машин.

К основному типу можно применять прилагательное const. Это дает тип, имеющий те же свойства, что и исходный тип, за исключением того, что значение переменных типа const не может изменяться после инициалиизации.

```
const float pi = 3.14;
const char plus = '+';
```

Символ, заключенный в одинарные кавычки, является символьной константой. Заметьте, что часто константа, определенная таким образом, не занимает память; просто там, где требуется, ее значение может использоваться непосредственно. Константа должна инициализироваться при описании. Для переменных инициализация

необязательна, но настоятельно рекомендуется. Оснований для введения локальнной переменной без ее инициализации очень немного.

К любой комбинации этих типов могут применяться арифметические операции:

```
+    (плюс, унарный и бинарный)
-    (минус, унарный и бинарный)
*    (умножение)
/    (деление)
```

А также операции сравнения:

```
==   (равно)
!=   (не равно)
<    (меньше)
>    (больше)
<=   (меньше или равно)
>=   (больше или равно)
```

Заметьте, что целое деление дает целый результат: 7/2 есть 3. Над целыми может выполняться операция % получения остатка: 7%2 равно 1.

При присваивании и арифметических операциях C++ выполняет все осмысленнные преобразования между основными типами, чтобы их можно было сочетать без ограничений:

```
double d = 1;
int i = 1;
d = d + i;
i = d + i;
```

1.3.2 Производные Типы

Вот операции, создающие из основных типов новые типы:

```
*        указатель на
*const   константный указатель на
&        ссылка на
[]       вектор*
()       функция, возвращающая
```

Например:

```
char* p        // указатель на символ
char *const q  // константный указатель на символ
char v[10]     // вектор из 10 символов
```

Все вектора в качестве нижней границы индекса имеют ноль, поэтому в v десять элементов:v[0] ... v[9]. Функции объясняются в #1.5, ссылки в #1.9. Переменная указатель может содержать адрес объекта соответствующего типа:

```
char c;
// ...
p = &c; // p указывает на c
```

Унарное & является операцией взятия адреса.

1.4 Выражения и Операторы

В C++ имеется богатый набор операций, с помощью которых в выражениях образуются новые значения и изменяются значения

---

* одномерный массив. Это принятый термин (например, вектора прерываний), и мы сочли, что стандартный перевод его как "массив"

затуманит изложение. (прим. перев.)

переменных. Поток управления в программе задается с помощью операторов , а описания используются для введения в программе имен переменных, констант и т.д. Заметьте, что описания являются операторами, поэтому они свободно могут сочетаться с другими операторами.

1.4.1 Выражения

В C++ имеется большое число операций, и они будут объясняться там, где (и если) это потребуется. Следует учесть, что операции

```
~     (дополнение)
&     (И)
^     (исключающее ИЛИ)
|     (включающее ИЛИ)
<<    (логический сдвиг влево)
>>    (логический сдвиг вправо)
```

применяются к целым, и что нет отдельного типа данных для логических действий.

Смысл операции зависит от числа операндов; унарное & является операцией взятия адреса, а бинарное & - это операция логического И. Смысл операции зависит также от типа ее операндов: + в выражении a+b означает сложение с плавающей точкой, если операнды имеют тип float, но целое сложение, если они типа int. В #1.8 объясняется, как можно определить операцию для типа, определяемого пользователем, без потери ее значения, предопределенного для основных и производных типов.

В C++ есть операция присваивания =, а не оператор присваивания, как в некоторых языках. Таким образом, присваивание может встречаться в неожиданном контексте; например, x=sqrt(a=3*x). Это бывает полезно. a=b=c означает присвоение с объекту b, а затем объекту a. Другим свойством операции присваивания является то, что она может совмещаться с большинством бинарных операций. Например, x[i+3]*=4 означает x[i+3]=x[i+3]*4, за исключением того факта, что выражение x[i+3] вычисляется только один раз. Это дает привлекательную степень эффективности без необходимости обращения к оптимизирующим компиляторам. К тому же это более кратко.

В большинстве программ на C++ широко применяются указатели. Унарная операция * разыменовывает* указатель, т.е. *p есть объект, на который указывает p. Эта операция также называется косвенной адресацией. Например, если имеется char* p, то *p есть символ, на который указывает p. Часто при работе с указателями бывают полезны операция увеличения ++ и операция уменьшения --. Предположим, p указывает на элемент вектора v, тогда p++ делает p указывающим на следующий элемент.

1.4.2 Операторы Выражения

Самый обычный вид оператора - оператор выражение. Он состоит из выражения, за которым следует точка с запятой. Например:

---

* англ. dereference - получить значение объекта, на который указывает данный указатель. (прим. перев.)

```
a = b*3+c;
cout << "go go go";
lseek(fd,0,2);
```

1.4.3 Пустой оператор

Простейшей формой оператора является пустой оператор:

```
;
```

Он не делает ничего. Однако он может быть полезен в тех случаях, когда синтаксис требует наличие оператора, а вам оператор не нужен.

1.4.4 Блоки

Блок - это возможно пустой список операторов, заключенный в фигурные скобки:

```
{ a=b+2; b++; }
```

Блок позволяет рассматривать несколько операторов как один. Область видимости имени, описанного в блоке, простирается до конца блока. Имя можно сделать невидимым с помощью описаний такого же имени во внутренних блоках.

1.4.5 Операторы if

Программа в следующем примере осуществляет преобразование дюймов в сантиметры и сантиметров в дюймы; предполагается, что вы укажете единицы измерения вводимых данных, добавляя i для дюймов и c для сантиметров:

```
#include

main()
{
    const float fac = 2.54;
    float x, in, cm;
    char ch = 0;

    cout << "введите длину: ";
    cin >> x >> ch;

    if (ch == 'i') {       // inch - дюймы
        in = x;
        cm = x*fac;
    }
    else if (ch == 'c')   //  cm - сантиметры
        in = x/fac;
        cm = x;
    }
    else
        in = cm = 0;

    cout << in << " in = " << cm << " cm\n";
}
```

Заметьте, что условие в операторе if должно быть заключено в круглые скобки.

1.4.6 Операторы switch

Оператор switch производит сопоставление значения с множеством констант. Проверки в предыдущем примере можно записать так:

```
switch (ch) {
case 'i':
    in = x;
```

```
        cm = x*fac;
        break;
    case 'c':
        in = x/fac;
        cm = x;
        break;
    default:
        in = cm = 0;
        break;
    }
```

Операторы break применяются для выхода из оператора switch. Константы в вариантах case должны быть различными, и если проверяемое значение не совпадает ни с одной из констант, выбирается вариант default. Программисту не обязательно предусматривать default.

1.4.7 Оператор while

Рассмотрим копирование строки, когда заданы указатель p на ее первый символ и указатель q на целевую строку. По соглашению строка оканчивается символом с целым значением 0.

```
    while (p != 0) {
        *q = *p;        // скопировать символ
        q = q+1;
        p = p+1;
    }
    *q = 0;             // завершающий символ 0 скопирован не был
```

Следующее после while условие должно быть заключено в круглые скобки. Условие вычисляется, и если его значение не ноль, выполняется непосредственно следующий за ним оператор. Это повторяется до тех пор, пока вычисление условия не даст ноль.

Этот пример слишком пространен. Можно использовать операцию ++ для непосредственного указания увеличения, и проверка упростится:

```
    while (*p)  *q++ = *p++;
    *q = 0;
```

где конструкция *p++ означает: "взять символ, на который указывает p, затем увеличить p."

Пример можно еще упростить, так как указатель p разыменовывается дважды за каждый цикл. Копирование символа можно делать тогда же, когда производится проверка условия:

```
    while (*q++ = *p++) ;
```

Здесь берется символ, на который указывает p, p увеличивается, этот символ копируется туда, куда указывает q, и q увеличивается. Если символ ненулевой, цикл повторяется. Поскольку вся работа выполняется в условии, не требуется ниодного оператора. Чтобы указать на это, используется пустой оператор. C++ (как и C) одновременно любят и ненавидят за возможность такого чрезвычайно краткого ориентированного на выразительность программирования*.

1.4.8 Оператор for

Рассмотрим копирование десяти элементов одного вектора в другой:

```
    for (int i=0; i<10; i++) q[i]=p[i];
```

Это эквивалентно

```
int i = 0;
while (i<10) {
    q[i] = p[i];
    i++;
}
```

---

* в оригинале expression-oriented (expression - выразительность и выражение). (прим. перев.)

но более удобочитаемо, поскольку вся информация, управляющая циклом, локализована. При применении операции ++ к целой переменной к ней просто добавляется единица. Первая часть оператора for не обязательно должна быть описанием, она может быть любым оператором. Например:

```
for (i=0; i<10; i++) q[i]=p[i];
```

тоже эквивалентно предыдущей записи при условии, что i соответствующим образом описано раньше.

1.4.9 Описания

Описание - это оператор, вводящий имя в программе. Оно может также инициализировать объект с этим именем. Выполнение описания означает, что когда поток управления доходит до описания, вычисляется инициализирующее выражение (инициализатор) и производится инициализация. Например:

```
for (int i = 1; i
```

1.5 Функции

Функция - это именованная часть программы, к которой можно обращаться из других частей программы столько раз, сколько потребуется. Рассмотрим программу, печатающую степени числа 2:

```
extern float pow(float, int); //pow() определена в другом месте

main()
{
    for (int i=0; i<10; i++) cout << pow(2,i) << "\n";
}
```

Первая строка функции - описание, указывающее, что pow - функция, получающая параметры типа float и int и возвращающая float. Описание функции используется для того, чтобы сделать определенными обращения к функции в других местах.

При вызове тип каждого параметра функции сопоставляется с ожидаемым типом точно так же, как если бы инициализировалась переменная описанного типа. Это гарантирует надлежащую проверку и преобразование типов. Напрмер, обращение pow(12.3,"abcd") вызовет недовольство компилятора, поскольку "abcd" является строкой, а не int. При вызове pow(2,i) компилятор преобразует 2 к типу float, как того требует функция. Функция pow может быть определена например так:

```
float pow(float x, int n)
{
 if (n < 0) error("извините, отрицателный показатель для pow()");

    switch (n) {
    case 0: return 1;
    case 1: return x;
```

```
    default:    return x*pow(x,n-1);
    }
}
```

Первая часть определения функции задает имя функции, тип возвращаемого ею значения (если таковое имеется) и типы и имена ее параметров (если они есть). Значение возвращается из функции с помощью оператора return.

Разные функции обычно имеют разные имена, но функциям, выполняющим сходные действия над объектами различных типов, иногда лучше дать возможность иметь одинаковые имена. Если типы их параметров различны, то компилятор всегда может различить их и выбрать для вызова нужную функцию. Может, например, иметься одна функция возведения в степень для целых переменных и другая для переменных с плавающей точкой:

```
overload pow;
int pow(int, int);
double pow(double, double);
//...
x=pow(2,10);
y=pow(2.0,10.0);
```

Описание

```
overload pow;
```

сообщает компилятору, что использование имени pow более чем для одной функции является умышленным.

Если функция не возвращает значения, то ее следует описать как void:

```
void swap(int* p, int* q)    // поменять местами
{
    int t = *p;
    *p = *q;
    *q = t;
}
```

1.6 Структура программы

Программа на C++ обычно состоит из большого числа исходных файлов, каждый из которых содержит описания типов, функций, переменных и констант. Чтобы имя можно было использовать в разных исходных файлах для ссылки на один и тот же объект, оно должно быть описано как внешнее. Например:

```
extern double sqrt(double);
extern instream cin;
```

Самый обычный способ обеспечить согласованность исходных файлов - это поместить такие описания в отдельные файлы, называемые заголовочными (или хэдер) файлами, а затем включить, то есть скопировать, эти заголовочные файлы во все файлы, где нужны эти описания. Например, если описание sqrt хранится в заголовочном файле для стандартных математических функций math.h, и вы хотите извлечь квадратный корень из 4, можно написать:

```
#include
//...
x = sqrt(4);
```

Поскольку обычные заголовочные файлы включаются во многие исходные файлы, они не содержат описаний, которые не должны повторяться. Например, тела функций даются только для inline-подставляемых функций (#1.12) и инициализаторы даются только для констант

(#1.3.1). За исключением этих случаев, заголовочный файл является хранилищем информации о типах. Он обеспечивает интерфейс между отдельно компилируемыми частями программы.

В команде включения include имя файла, заключенное в угловые скобки, например , относится к файлу с этим именем в стандартном каталоге (часто это /usr/include/CC); на файлы, находящиеся в каких-либо других местах ссылаются с помощью имен, заключенных в двойные кавычки. Например:

```
#include "math1.h"
#include "/usr/bs/math2.h"
```

включит math1.h из текущего пользовательского каталога, а math2.h из каталога /usr/bs.

Здесь приводится очень маленький законченный пример программы, в котором строка определяется в одном файле, а ее печать производится в другом. Файл header.h определяет необходимые типы:

```
// header.h

extern char* prog_name;
extern void f();
```

В файле main.c находится главная программа:

```
// main.c

#include "header.h"
char* prog_name = "дурацкий, но полный";
main()
{
    f();
}
```

а файл f.c печатает строку:

```
// f.c

#include
#include "header.h"
void f()
{
    cout << prog_name << "\n";
}
```

Скомпилировать и запустить программу вы можете например так:

```
$ CC main.c f.c -o silly
$ silly
дурацкий, но полный
$
```

1.7 Классы

Давайте посмотрим, как мы могли бы определить тип потока вывода ostream. Чтобы упростить задачу, предположим, что для буферизации определен тип streambuf. Тип streambuf на самом деле определен в , где также находится и настоящее определение ostream. Пожалуйста, не испытывайте примеры, определяющие ostream в этом и последующих разделах; пока вы не сможете полностью избежать использования , компилятор будет возражать против переопределений.

Определение типа, определяемого пользователем (который в C++ называется class, т.е. класс), специфицирует данные, необходимые

для представления объекта этого типа, и множество операций для работы с этими объектами. Определение имеет две части: закрытую (private) часть, содержащую информацию, которой может пользоваться только его разработчик, и открытую (public) часть, представляющую интерфейс типа с пользователем:

```
class ostream {
    streambuf* buf;
    int state;
public:
    void put(char*);
    void put(long);
    void put(double);
}
```

Описания после метки public задают интерфейс: пользователь может обращаться только к трем функциям put(). Описания перед меткой public задают представление объекта класса ostream; имена buf и state могут использоваться только функциями put(), описанными в открытой части.

class определяет тип, а не объект данных, поэтому чтобы использовать ostream, мы должны один такой объект описать (так же, как мы описываем переменные типа int):

```
ostream my_out;
```

Считая, что my_out был соответствующим образом проинициализирован (как, объясняется в #1.10), его можно использовать например так:

```
my_out.put("Hello, world\n");
```

С помощью операции точка выбирается член класса для данного объекта этого класса. Здесь для объекта my_out вызывается член функция put().

Функция может определяться так:

```
void ostream::put(char* p)
{
    while (*p) buf.sputc(*p++);
}
```

где sputc() - функция, которая помещает символ в streambuf. Префикс ostream необходим, чтобы отличить put() ostream'а от других функций с именем put().

Для обращения к функции члену должен быть указан объект класса. В функции члене можно ссылаться на этот объект неявно, как это делалось выше в ostream::put(): в каждом вызове buf относится к члену buf объекта, для которого функция вызвана.

Можно также ссылаться на этот объект явно посредством указателя с именем this. В функции члене класса X this неявно описан как X* (указатель на X) и инициализирован указателем на тот объект, для которого эта функция вызвана. Определение ostream::put() можно также записать в виде:

```
void ostream::put(char* p)
{
    while (*p) this->buf.sputc(*p++);
}
```

Операция -> применяется для выбора члена объекта, заданного указателем.

1.8 Перегрузка операций

Настоящий класс ostream определяет операцию <<, чтобы сделать удобным вывод нескольких объектов одним оператором. Давайте посмотрим, как это сделано.

Чтобы определить @, где @ - некоторая операция языка C++, для каждого определяемого пользователем типа вы определяете функцию с именем operator@, которая получает параметры соответствующего типа. Например:

```
class ostream {
    //...
    ostream operator<<(char*);
};

ostream ostream::operator<<(char* p)
{
    while (*p) buf.sputc(*p++);
    return *this;
}
```

определяет операцию << как член класса ostream, поэтому s<
");

а если применить операцию взятия адреса, то вы получите адрес объекта, на который ссылается ссылка:

```
&s1 == &my_out
```

Первая очевидная польза от ссылок состоит в том, чтобы обеспечить передачу адреса объекта, а не самого объекта, в фукнкцию вывода (в некоторых языках это называется передачей параметра по ссылке):

```
ostream& operator<<(ostream& s, complex z) {
    return s << "(" << z.real << "," << z.imag << ")";
}
```

Достаточно интересно, что тело функции осталось без изменений, но если вы будете осуществлять присваивание s, то будете воздействовать на сам объект, а не на его копию. В данном случае то, что возвращается ссылка, также повышает эффективность, поскольку очевидный способ реализации ссылки - это указатель, а передача указателя гораздо дешевле, чем передача большой структуры данных.

Ссылки также существенны для определения потока ввода, поскольку операция ввода получает в качестве операнда переменную для считывания. Если бы ссылки не использовались, то пользователь должен был бы явно передавать указатели в функции ввода.

```
class istream {
    //...
    int state;
public:
    istream& operator>>(char&);
    istream& operator>>(char*);
    istream& operator>>(int&);
    istream& operator>>(long&);
    //...
};
```

Заметьте, что для чтения long и int используются разные функции, тогда как для их печати требовалась только одна. Это вполне обычно, и причина в том, что int может быть преобразовано в long по стандартным правилам неявного преобразования (#c.6.6), избавляя таким образом программиста от беспокойства по поводу написания обеих функций ввода.

1.10 Конструкторы

Определение ostream как класса сделало члены данные закрытыми. Только функция член имеет доступ к закрытым членам, поэтому надо предусмотреть функцию для инициализации. Такая функция называется конструктором и отличается тем, что имеет то же имя, что и ее класс:

```
class ostream {
    //...
    ostream(streambuf*);
    ostream(int size, char* s);
};
```

Здесь задано два конструктора. Один получает вышеупомянутый streambuf для реального вывода, другой получает размер и указатель на символ для форматирования строки. В описании необходимый для конструктора список параметров присоединяется к имени. Теперь вы можете, например, описать такие потоки:

```
ostream my_out(&some_stream_buffer);
char xx[256];
ostream xx_stream(256,xx);
```

Описание my_out не только задает соответствующий объем памяти где-то в другом месте, оно также вызывает конструктор ostream::ostream(streambuf*), чтобы инициализировать его параметром &some_stream_buffer, предположительно указателем на подходящий объект класса streambuf. Описание конструкторов для класса не только дает способ инициализации объектов, но также обеспечивает то, что все объекты этого класса будут проинициализированы. Если для класса были описаны конструкторы, то невозможно описать переменную этого класса так, чтобы конструктор не был вызван. Если класс имеет конструктор, не получающий параметров, то этот конструктор будет вызываться в том случае, если в описании нет ни одного параметра.

1.11 Вектора

Встоенное в C++ понятие вектора было разработано так, чтобы обеспечить максимальную эффективность выполнения при минимальном расходе памяти. Оно также (особенно когда используется совместно с указателями) является весьма универсальным инструментом для построения средств более высокого уровня. Вы могли бы, конечно, возразить, что размер вектора должен задаваться как константа, что нет проверки выхода за границы вектора и т.д. Ответ на подобные возражения таков: "Вы можете запрограммировать это сами." Давайте посмотрим, действительно ли оправдан такой ответ. Другими словами, проверим средства абстракции языка C++, попытавшись реализовать эти возможности для векторных типов, которые мы создадим сами, и посмотрим, какие с этим связаны трудности, каких это требует затрат, и насколько получившиеся векторные типы удобны в обращении.

```
class vector {
    int* v;
    int  sz;
public:
        vector(int);      // конструктор
        ~vector();        // деструктор
    int size() { return sz; }
    void set_size(int);
```

```
    int& operator[](int);
    int& elem(int i) { return v[i]; }
};
```

Функция size возвращает число элементов вектора, таким образом индексы должны лежать в диапазоне 0 ... size()-1. Функция set_size сделана для изменения этого размера, elem обеспечивает доступ к элементам без проверки индекса, а operator[] дает доступ с проверкой границ.

Идея состоит в том, чтобы класс сам был структурой фиксированного размера, управляющей доступом к фактической памяти вектора, которая выделяется конструктором вектора с помощью распределителя свободной памяти new:

```
vector::vector(int s)
{
    if (s<=0) error("плохой размер вектора");
    sz = s;
    v = new int[s];
}
```

Тепрь вы можете описывать вектора типа vector почти столь же элегантно, как и вектора, встроенные в сам язык:

```
vector v1(100);
vector v2(nelem*2-4);
```

Операцию доступа можно определить как

```
int& vector::operator[](int i)
{
    if(i<0 || sz<=i) error("индекс выходит за границы");
    return v[i];
}
```

Операция || (ИЛИИЛИ) - это логическая операция ИЛИ. Ее правый операнд вычисляется только тогда, когда это необходимо, то есть если вычисление левого операнда дало ноль. Возвращение ссылки обеспечивает то, что запись [] может использоваться с любой стороны операции присваивания:

```
v1[x] = v2[y];
```

Функция со странным именем ~vector - это деструктор, то есть функция, описанная для того, чтобы она неявно вызывалась, когда объект класса выходит из области видимости. Деструктор класса C имеет имя ~C. Если его определить как

```
vector::~vector()
{
    delete v;
}
```

то он будет, с помощью операции delete, освобождать пространство, выделенное конструктором, поэтому когда vector выходит из области видимости, все его пространство возвращается обратно в память для дальнейшего использования.

1.12 Inline-подстановка

Если часто повторяется обращение к очень маленькой функции, то вы можете начать беспокоиться о стоимости вызова функции. Обращение к функции члену не дороже обращения к функции не члену с тем же числом параметров (надо помнить, что функция член всегда имеет хотя

бы один параметр), и вызовы в функций в C++ примерно столь же эффективны, сколь и в любом языке. Однако для слишком маленьких функций может встать вопрос о накладных расходах на обращение. В этом случае можно рассмотреть возможность спецификации функции как inline-подставляемой. Если вы поступите таким образом, то компилятор сгенерирует для функции соответствующий код в месте ее вызова. Семантика вызова не изменяется. Если, например, size и elem inline-подставляемые, то

```
vector s(100);
//...
i = s.size();
x = elem(i-1);
```

порождает код, эквивалентный

```
//...
i = 100;
x = s.v[i-1];
```

C++ компилятор обычно достаточно разумен, чтобы генерировать настолько хороший код, насколько вы можете получить в результате прямого макрорасширения. Разумеется, компилятор иногда вынужден использовать временные переменные и другие уловки, чтобы сохранить семантику.

Вы можете указать, что вы хотите, чтобы функция была inline-подставляемой, поставив ключевое слово inline, или, для функции члена, просто включив определение функции в описание класса, как это сделано в предыдущем примере для size() и elem().

При хорошем использовании inline-функции резко повышают скорость выполнения и уменьшают размер объектного кода. Однако, inline-функции запутывают описания и могут замедлить компиляцию, поэтому, если они не необходимы, то их желательно избегать. Чтобы inline-функция давала существенный выигрыш по сравнению с обычной функцией, она должна быть очень маленькой.

1.13 Производные классы

Теперь давайте определим вектор, для которого пользователь может задавать границы изменения индекса.

```
class vec: public vector {
    int low, high;
public:
    vec(int,int);
    int& elem(int);
    int& operator[](int);
};
```

Определение vec как

```
:public vector
```

означает, в первую очередь, что vec это vector. То есть, тип vec имеет (наследует) все свойства типа vector дополнительно к тем, что описаны специально для него. Говорят, что класс vector является базовым классом для vec, а о vec говорится, что он производный от vector.

Класс vec модифицирует класс vector тем, что в нем задается другой конструктор, который требует от пользователя указывать две границы изменения индекса, а не длину, и имеются свои собственные функции доступа elem(int) и operator[](int). Функция elem() класса vec легко выражается через elem() класса vector:

```
int& vec::elem(int i)
{
    return vector::elem(i-low);
}
```

Операция разрешения области видимости :: используется для того, чтобы не было бесконечной рекурсии обращения к vec::elem() из нее самой. с помощью унарной операции :: можно ссылаться на нелокальные

имена. Было бы разумно описать vec::elem() как inline, поскольку, скорее всего, эффективность существенна, но необязательно, неразумно и невозможно написать ее так, чтобы она непосредственно использовала закрытый член v класса vector. Фунции производного класса не имеют специального доступа к закрытым членам его базового класса.

Конструктор можно написать так:

```
vec::vec(int lb, int hb) : (hb-lb+1)
{
    if (hb-lb<0) hb = lb;
    low = lb;
    high = hb;
}
```

Запись : (hb-lb+1) используется для определения списка параметров конструктора базового класса vector::vector(). Этот конструктор вызывается перед телом vec::vec(). Вот небольшой пример, который можно запустить, если скомпилировать его вместе с остальными описаниями vector:

```
#include

void error(char* p)
{
    cerr << p << "n\"; // cerr - выходной поток сообщений об
ошибках
    exit(1);
}

void vector::set_size(int) { /* пустышка */ }

int& vec::operator[](int i)
{
    if (i
```

1.14 Еще об операциях

Другое направление развития - снабдить вектора операциями:

```
class Vec : public vector {
public:
    Vec(int s) : (s) {}
    Vec(Vec&);
    ~Vec() {}
    void operator=(Vec&);
    void operator*=(Vec&);
    void operator*=(int);
    //...
};
```

Обратите внимание на способ определения конструктора производного класса, Vec::Vec(), когда он передает свой параметр конструктору базового класса vector::vector() и больше не делает ничего. Это

полезная парадигма. Операция присваивания перегружена, ее можно определить так:

```
void Vec::operator=(Vec& a)
{
    int s = size();
    if (s!=a.size()) error("плохой размер вектора для =");
    for (int i = 0; i

void error(char* p) {
    cerr << p << "\n";
    exit(1);
}

void vector::set_size(int) { /*...*/ }

int& vec::operator[](int i) { /*...*/ }

main()
{
    Vec a(10);
    Vec b(10);
    for (int i=0; i
```

1.15 Друзья (friend)

Функция operator+() не воздействует непосредственно на представление вектора. Действтельно, она не может этого делать, поскольку не является членом. Однако иногда желательно дать функциям не членам возможность доступа к закрытой части класса. Например, если бы не было функции "доступа без проверки" vector::elem(), вам пришлось бы проверять индекс i на соответствие границам три раза за каждый проход цикла. Здесь мы избежали этой сложности, но она довольно типична, поэтому у класса есть механизм предоставления права доступа к своей закрытой части функциям не членам. Просто в описание класса помещается описание функции, перед которым стоит ключевое слово friend. Например, если имеется

```
class Vec;  // Vec - имя класса

class vector {
    friend Vec operator+(Vec, Vec);
    //...
};
```

То вы можете написать

```
Vec operator+(Vec a, Vec b)
{
    int s = a.size();
    if (s != b.size()) error("плохой размер вектора для +");
    Vec& sum = *new Vec(s);
    int* sp = sum.v;
    int* ap = a.v;
    int* bp = b.v;
    while (s--) *sp++ = *ap++ + *bp++;
    return sum;
}
```

Одним из особенно полезных аспектов механизма friend является то, что функция может быть другом двух и более классов. Чтобы увидеть это, рассмотрим определение vector и matrix, а затем определение функции умножения (см. #c.8.8).

1.16 Обобщенные Вектора

"Пока все хорошо," - можете сказать вы, - "но я хочу, чтобы один из этих векторов был типа matrix, который я только что определил." К сожалению, в C++ не предусмотрены средства для определения класса векторов с типом элемента в качестве параметра. Один из способов - продублировать описание и класса, и его функций членов. Это не идеальный способ, но зачастую вполне приемлемый.

Вы можете воспользоваться препроцессором (#4.7), чтобы механизировать работу. Например, класс vector - упрощенный вариант класса, который можно найти в стандартном заголовочном файле. Вы могли бы написать:

```
#include

declare(vector,int);

main()
{
    vector(int) vv(10);
    vv[2] = 3;
    vv[10] = 4; // ошибка: выход за границы
}
```

Файл vector.h таким образом определяет макросы, чтобы declare(vector,int) после расширения превращался в описание класса vector, очень похожий на тот, который был определен выше, а implement(vector,int) расширялся в определение функций этого класса. Поскольку implement(vector,int) в результате расширения превращается в определение функций, его можно использовать в программе только один раз, в то время как declare(vector,int) должно использоваться по одному разу в каждом файле, работающем с этим типом целых векторов.

```
declare(vector,char);
//...
implement(vector,char);
```

даст вам отдельный тип "вектор символов". Пример реализации обобщенных классов с помощью макросов приведен в #7.3.5.

1.17 Полиморфные Вектора

У вас есть другая возможность - определить ваш векторный и другие вмещающие классы через указатели на объекты некоторого класса:

```
class common {
    //...
};
class vector {
    common** v;
    //...
public:
    cvector(int);
    common*& elem(int);
    common*& operator[](int);
    //...
};
```

Заметьте, что поскольку в таких векторах хранятся указатели, а не сами объекты, объект может быть "в" нескольких таких векторах одновременно. Это очень полезное свойство подобных вмещающих классов, таких, как вектора, связанные списки, множества и т.д. Кроме того, можно присваивать указатель на производный класс указателю на его базовый класс, поэтому можно использовать

приведенный выше cvector для хранения указателей на объекты всех производных от common классов. Например:

```
class apple : public common { /*...*/ }
class orange : public common { /*...*/ }
class apple_vector : public cvector {
public:

cvector fruitbowl(100);
//...
apple aa;
orange oo;
//...
fruitbowl[0] = &aa;
fruitbowl[1] = &oo;
}
```

Однако, точный тип объекта, вошедшего в такой вмещающий класс, больше компилятору не известен. Например, в предыдущем примере вы знаете, что элемент вектора является common, но является он apple или orange? Обычно точный тип должен впоследствие быть восстановлен, чтобы обеспечить правильное использование объекта. Для этого нужно или в какой-то форме хранить информацию о типе в самом объекте, или обеспечить, чтобы во вмещающий класс помещались

только объекты данного типа. Последнее легко достигается с помощью производного класса. Вы можете, например, создать вектор указателей на apple:

```
class apple_vector : public cvector {
public:
    apple*& elem(int i)
        { return (apple*&) cvector::elem(i); }
    //...
};
```

используя запись приведения к типу (тип)выражение, чтобы преобразовать common*& (ссылку на указатель на common), которую возвращает cvector::elem, в apple*&. Такое применение производных классов создает альтернативу обобщенным классам. Писать его немного труднее (если не использовать макросы таким образом, чтобы производные классы фактически реализовывали обобщенные классы; см. #7.3.5), но оно имеет то преимущество, что все производные классы совместно используют единственную копию функции базового класса. В случае обобщенных классов, таких, как vector(type), для каждого нового используемого типа должна создаваться (с помощью implement()) новая копия таких функций. Другой способ, хранение идентификации типа в каждом объекте, приводит нас к стилю программирования, который часто называют объекто-основанным или объектно-ориентированным.

1.18 Виртуальные функции

Предположим, что мы пишем программу для изображения фигур на экране. Общие атрибуты фигуры представлены классом shape, а специальные атрибуты - специальными классами:

```
class shape {
    point center;
    color col;
    //...
public:
    void move(point to) { center=to; draw(); }
```

```
    point where() { return center; }
    virual void draw();
    virtual void rotate(int);
    //...
};
```

Функции, которые можно определить не зная точно определенной фигуры (например, move и where, то есть, "передвинуть" и "где"), можно описать как обычно. Остальные функции описываются как virual, то есть такие, которые должны определяться в производном классе. Например:

```
class circle: public shape {
    int radius;
public:
    void draw();
    void rotatte(int i) {}
    //...
};
```

Теперь, если shape_vec - вектор фигур, то можно написать:

```
for (int i = 0; i
```

Глава 2

Описания и Константы

Совершенство достигается только к моменту краха.
- С.Н. Паркинсон

В этой главе описаны основные типы (char, int, float и т.д.) и основные способы построения из них новых типов (функций, векторов, указателей и т.д.). Имя вводится в программе посредством описаниия, которое задает его тип и, возможно, начальное значение. Даны понятия описания, определения, области видимости имен, времени жизни объектов и типов. Описываются способы записи констант в C++, а также способы определения символических констант. Примеры просто демонстрируют характерные черты языка. Более развернутый и реалистичный пример приводится в следующей главе для знакомства с выраженями и операторами языка C++. Механизмы задания типов, определяемых пользователем, с присоединенными операциями представлены в Главах 4, 5 и 6 и здесь не упоминаются.

2.1 Описания

Прежде чем имя (идентификатор) может быть использовано в C++ программе, он должно быть описано. Это значит, что надо задать его тип, чтобы сообщить компилятору, к какого вида объектам относится имя. Вот несколько примеров, иллюстрирующих разнообразие описаний:

```
char ch;
int count = 1;
char* name = "Bjarne";
struct complex { float re, im; };
complex cvar;
extern complex sqrt(complex);
extern int error_number;
typedef complex point;
float real(complex* p) { return p->re; };
const double pi = 3.1415926535897932385;
struct user;
```

Как можно видеть из этих примеров, описание может делать больше чем просто ассоциировать тип с именем. Большинство описаний являются также определениями; то есть они также определяют для имени сущность, к которой оно относится. Для ch, count и cvar этой сущностью является соответствующий объем памяти, который должен использоваться как переменная - эта память будет выделена. Для real это заданная функция. Для constant pi это значение 3.14159265358979323385. Для complex этой сущностью является новый тип. Для point это тип complex, поэтому point становится синонимом complex. Только описания

```
extern complex sqrt(complex);
extern int error_number;
struct user;
```

не являются одновременно определениями. Это означает, что объект, к которому они относятся, должен быть определен где-то еще. Код (тело) функции sqrt должен задаваться неким другим описанием, память для переменной error_number типа int должна выделяться неким другим описанием, и какое-то другое описание типа user должно определять, что он из себя представляет. В C++ программе всегда должно быть только одно определение каждого имени, но описаний может быть много, и все описания должны согласовываться с типом объекта, к которому они относятся, поэтому в этом фрагменте есть две ошибки:

```
int count;
int count;               // ошибка: переопределение
exnern int error_number;
exnern int error_number; // ошибка: несоответствие типов
```

а в этом - ниодной (об использовании extern см. #4.2):

```
exnern int error_number;
exnern int error_number;
```

Некотрые описания задают "значение" для сущностей, которые они определяют:

```
struct complex { float re, im; };
typedef complex point;
float real(complex* p) { return  p->re };
const double pi = 3.14159265358979323385;
```

Для типов, функций и костант "значение" неизменно; для неконстантных типов данных начальное значение может впоследствие изменяться:

```
int count = 1;
char* name = "Bjarne";
//...
count = 2;
name = "Marian";
```

Из всех определений только

```
char ch;
```

не задает значение. Всякое описание, задающее значение, является определением.

2.1.1 Область Видимости

Описание вводит имя в области видимости; то есть, имя может использоваться только в определенной части программы. Для имени, описанного в функции (такое имя часто называют локальным), эта область видимости простирается от точки описания до конца блока, в

котором появилось описание; для имени не в функции и не в классе (называемого часто глобальным именем) область видимости простирается от точки описания до конца файла, в котором появилось описание. Описание имени в блоке может скрывать (прятать) описание во внутреннем блоке или глобальное имя. Это значит, что можно

переопределять имя внутри блока для ссылки на другой объект. После выхода из блока имя вновь обретает свое прежнее значение. Например:

```
int x;            // глобальное x

f() {
    int x;        // локальное x прячет глобальное x
    x = 1;        // присвоить локальному x
    {
        int x;    // прячет первое локальное x
        x = 2;    // присвоить второму локальному x
    }
    x = 3;        // присвоить первому локальному x
}

int* p = &x;      // взять адрес глобального x
```

Скрытие имен неизбежно при написании больших программ. Однако читающий человек легко может не заметить, что имя скрыто, и некоторые ошибки, возникающие вследствие этого, очень трудно обнаружить, главным образом потому, что они редкие. Значит скрытие имен следует минимизировать. Использование для глобальных переменных имен вроде i или x напрашивается на неприятности.

С помощью применения операции разрешения области видимости :: можно использовать скрытое глобальное имя. Например:

```
int x;

f()
{
    int x = 1;      // скрывает глобальное x
    ::x = 2;        // присваивает глобальному x
}
```

Но возможности использовать скрытое локальное имя нет.

Область видимости имени начинается в точке описания. Это означает, что имя можно использовать даже для задания его собственного значения. Например:

```
int x;

f()
{
    int x = x;    // извращение
}
```

Это не является недопустимым, хотя и бессмысленно, и компилятор предупредит, что x "used before set" ("использовано до того, как задано"), если вы попробуете так сделать. Можно, напротив, не применяя операцию ::, использовать одно имя для ссылки на два различных объекта в блоке. Например:

```
int x;

f()             // извращение
```

```
{
    int y = x;  // глобальное x
    int x = 22;
    y = x;      // локальное x
}
```

Переменная y инициализируется значением глобального x, 11, а затем ему присваивается значение локальной переменной x, 22.

Имена параметров функции считаются описанными в самом внешнем блоке функции, поэтому

```
f(int x)
{
    int x;        // ошибка
}
```

содержит ошибку, так как x определено дважды в одной и той же области видимости.

2.1.2 Объекты и Адреса (Lvalue)

Можно назначать и использовать переменные, не имеющие имен, и можно осуществлять присваивание выражениям странного вида (например, *p[a+10]=7). Следовательно, есть потребность в имени "нечто в памяти". Вот соответствующая цитата из справочного руководства по C++:"Объект есть область памяти; lvalue есть выражение, ссылающееся на объект"(#c.5). Слово "lvalue" первоначально было придумано для значения "нечто, что может стоять в левой части присваивания". Однако не всякий адрес можно использовать в левой части присваивания; бывают адреса, ссылающиеся на константу (см. #2.4).

2.1.3 Время Жизни

Если программист не указал иного, то объект создается, когда встречается его описание, и уничтожается, когда его имя выходит из области видимости, Объекты с глобальными именами создаются и инициализируются один раз (только) и "живут" до завершения программы. Объекты, определенные описанием с ключевым словом static, ведут себя так же. Например*:

---

* Команда #include была выброшена из примеров в этой главе для экономии места. Она необходима в примерах, производящих ввывод, чтобы они были полными. (прим. автора)

```
int a = 1;

void f()
{
    int b = 1;          // инициализируется при каждом вызове f()
    static int c = 1;   // инициализируется только один раз
    cout << " a = " << a++
         << " b = " << b++
         << " c = " << c++ << "\n";
}

main()
{
    while (a < 4) f();
}
```

производит вывод

```
a = 1 b = 1 c = 1
a = 2 b = 1 c = 2
a = 3 b = 1 c = 3
```

Не инициализированная явно статическая (static) переменная неявно инициализируется нулем.

С помощью операций new и delete программист может также создавать объекты, время жизни которых управляется непосредственно; см. #3.2.4.

2.2 Имена

Имя (идентификатор) состоит из последовательности букв и цифр. Первый символ должен быть буквой. Символ подчерка _ считается буквой. C++ не налагает ограничений на число символов в имени, но некоторые части реализации находятся вне ведения автора компилятора (в частности, загрузчик), и они, к сожалению, такие ограничения налагают. Некоторые среды выполнения также делают необходимым расширить или ограничить набор символов, допустимых в идентификаторе; расширения (например, при допущении в именах символа $) порождают непереносимые программы. В качестве имени не могут использоваться ключевые слова C++ (см. #с.2.3). Примеры имен:

```
hello    this_is_a_most_unusially_long_name
DEFINED  foO    bAr     u_name    HorseSense
var0     var1   CLASS   _class    ___
```

Примеры последовательностей символов, которые не могут использоваться как идентификаторы:

```
012      a fool    $sys     class    3var
pay.due  foo~bar   .name    if
```

Буквы в верхнем и нижнем регистрах считаются различными, поэтому Count и count - различные имена, но вводить имена, лишь незначительно отличающиеся друг от друга, нежелательно. Имена, начинающиеся с подчерка, по традиции используются для специальных

средств среды выполнения, поэтому использовать такие имена в прикладных программах нежелательно.

Во время чтения программы компилятор всегда ищет наиболее длинную строку, составляющую имя, поэтому var10 - это одно имя, а не имя var, за которым следует число 10; и elseif - одно имя, а не ключевое слово else, после которого стоит ключевое слово if.

2.3 Типы

Каждое имя (идентификатор) в C++ программе имеет ассоциированный с ним тип. Этот тип определяет, какие операции можно применять к имени (то есть к объекту, на который оно ссылается), и как эти операции интерпретируются. Например:

```
int error number;
float real(complex* p);
```

Поскольку error_number описано как int, его можно присваивать, использовать в арифметических выражениях и т.д. Тогда как функция real может вызываться с адресом complex в качестве параметра. Можно взять адрес любого из них. Некоторые имена, вроде int и complex, являются именами типов. Обычно имя типа используется в описании для спецификации другого имени. Единственные отличные от этого действия над именем типа - это sizeof (для определения количества памяти, которая требуется для хранения объекта типа) и new (для размещения объекта типа в свободной памяти). Например:

```
main()
{
    int* p = new int;
    cout << "sizeof(int) = " << sizeof(int) "\n";
}
```

Имя типа можно также использовать для задания явного преобразования одного типа в другой, например:

```
float f;
char* p;
//...
long ll = long(p);     // преобразует p в long
int i = int(f);        // преобразует f в int
```

2.3.1 Основные Типы

В C++ есть набор основных типов, которые соответствуют наиболее общим основным единицам памяти компьютера и наиболее общим основным способам их использования:

```
char
short int
int
long int
```

для представления целых различных размеров,

```
float
double
```

для представления чисел с плавающей точкой,

```
unsigned char
unsigned short int
unsigned int
unsigned long int
```

для представления беззнаковых целых, логических значений, битовых массивов и т.п. Для большей компактности записи можно опускать int в комбинациях из нескольких слов, что не меняет смысла; так, long означает long int, и unsigned означает unsigned int. В общем, когда в описании опущен тип, он предполагается int. Например:

```
const a = 1;
static x;
```

все определяют объект типа int.

Целый тип char наиболее удобен для хранения и обработки символов на данном компьютере; обычно это 8-битовый байт. Размеры объектов C++ выражаются в единицах размера char, поэтому по определению sizeof(char)==1. В зависимости от аппаратного обеспечения char является знаковым или беззнаковым целым. Тип unsigned char, конечно, всегда беззнаковый, и при его использовании получаются более переносимые программы, но из-за применения его вместо просто char могут возникать значительные потери в эффективности.

Причина того, что предоставляется более чем один целый тип, более чем один беззнаковый тип и более чем один тип с плавающей точкой, в том, чтобы дать возможность программисту воспользоваться характерными особенностями аппаратного обеспечения. На многих машинах между различными разновидностями основных типов существуют значительные различия в потребностях памяти, временах доступа к памяти и временах вычислений. Зная машину обычно легко, например,

выбрать подходящий тип для конкретной переменной. Написать действительно переносимую программу нижнего уровня сложнее. Вот все, что гарантируется относительно размеров основных типов:

```
1==sizeof(char)<=sizeof(short)<= sizeof(int)<=sizeof(long)
sizeof(float)<=sizeof(double)
```

Однако обычно разумно предполагать, что в char могут храниться целые числа в диапазоне 0...127 (в нем всегда могут храниться символы машинного набора символов), что short и int имеют не менее 16 бит, что int имеет размер, соответствующий целой арифметике, и что long имеет по меньшей мере 24 бита. Предполагать что-либо помимо этого рискованно, и даже эти эмпирические правила применимы не везде. Таблицу характеристик аппаратного обеспечения для некоторых машин можно найти в #c.2.6.

Беззнаковые (unsigned) целые типы идеально подходят для применений, в которых память рассматривается как массив битов. Использование unsigned вместо int с тем, чтобы получить еще один бит для представления положительных целых, почти никогда не оказывается хорошей идеей. Попытки гарантировать то, что некоторые значения положительны, посредством описания переменных как

unsigned, обычно срываются из-за правил неявного преобразования. Например:

```
unsigned surprise = -1;
```

допустимо (но компилятор обязательно сделает предупреждение).

2.3.2 Неявное Преобразование Типа

Основные типы можно свободно сочетать в присваиваниях и выражениях. Везде, где это возможно, значения преобразуются так, чтобы информация не терялась. Точные правила можно найти в #c.6.6.

Существуют случаи, в которых информация может теряться или искажаться. Присваивание значения одного типа переменной другого типа, представление которого содержит меньшее число бит, неизбежно является источником неприятностей. Допустим, например, что следующая часть программы выполняется на машине с двоичным дополнительным предсталением целых и 8-битовыми символами:

```
int i1 = 256+255;
char ch = i1        // ch == 255
int i2 = ch;        // i2 == ?
```

В присваивании ch=i1 теряется один бит (самый значимый!), и ch будет содержать двоичный код "все-единицы" (т.е. 8 единиц); при присваивании i2 это никак не может превратиться в 511! Но каким же может быть значение i2? На DEC VAX, где char знаковые, ответ будет -1; на AT&T 3B-20, где char беззнаковые, ответ будет 255. В C++ нет динамического (т.е. действующего во время исполнения) механизма для разрешения такого рода проблемм, а выяснение на стадии компиляции вообще очень сложно, поэтому программист должен быть внимателен.

2.3.3 Производные Типы

Другие типы модно выводить из основных типов (и типов, определенных пользователем) посредством операций описания:

```
*     указатель
&     ссылка
[]    вектор
()    функция
```

и механизма определения структур. Например:

```
int* a;
float v[10];
char* p[20];    // вектор из 20 указателей на символ
void f(int);
struct str { short length; char* p; };
```

Правила построения типов с помощью этих операций подробно объясняются в #с.8.3-4. Основная идея состоит в том, что описание производного типа отражает его использование. Например:

```
int v[10];      // описывает вектор
i = v[3];       // использует элемент вектора

int* p;         // описывает указатель
i = *p;         // использует указываемый объект
```

Вся сложность понимания записи производных типов проистекает из того, что операции * и & префиксные, а операции [] () постфиксные, поэтому для формулировки типов в тех случаях, когда приоритеты операций создают затруднения, надо использовать скобки. Например, поскольку приоритет у [] выше, чем у *, то

```
int* v[10];         // вектор указателей
int (*p)[10];       // указатель на вектор
```

Большинство людей просто помнят, как выглядят наиболее обычные типы.

Описание каждого имени, вводимого в программе, может оказаться утомительным, особенно если их типы одинаковы. Но можно описывать в одном описании несколько имен. В этом случае описание содержит вместо одного имени список имен, разделенных запятыми. Например, два имени можно описать так:

```
int x, y;    // int x; int y;
```

При описании производных типов можно указать, что операции применяются только к отдельным именам (а не ко всем остальным именам в этом описании). Например:

```
int* p, y;        // int* p; int y; НЕ int* y;
int x, *p;        // int x; int* p;
int v[10], *p;    // int v[10]; int* p;
```

Мнение автора таково, что подобные конструкции делают программу менее удобочитаемой, и их следует избегать.

2.3.4 Тип void

Тип void (пустой) синтаксически ведет себя как основной тип. Однако использовать его можно только как часть производного типа, объектов типа void не существует. Он используется для того, чтобы указать, что функция не возвращает значения, или как базовый тип для указателей на объекты неизвестного типа.

```
void f()      // f не возвращает значение
void* pv;     // указатель на объект неизвестного типа
```

Переменной тиа void* можно присваивать указатель любого типа. На первый взгляд это может показаться не особенно полезным, поскольку void* нельзя разыменовать, но именно это ограничение и делает тип void* полезным. Главным образом, он применяется для передачи указателей функциям, которые не позволяют сделать предположение о типе объекта, и для возврата из функций нетипизированных объектов. Чтобы использовать такой объект, необходимо применить явное

преобразование типа. Подобные функции обычно находятся на самом нижнем уровне системы, там, где осуществляется работа с основными аппаратными ресурсами. Например:

```
void* allocate(int size);    // выделить
void deallocate(void*);      // освободить

f() {
    int* pi = (int*)allocate(10*sizeof(int));
    char* pc = (char*)allocate(10);
//...
deallocate(pi);
deallocate(pc);
}
```

2.3.5 Указатели

Для большинства типов T T* является типом указатель на T. То есть, в переменной типа T* может храниться адрес объекта типа T. Для указателей на вектора и указателей на функции вам, к сожалению, придется пользоваться более сложной записью:

```
int* pi;
char** cpp;              // указатель на указатель на char
int (*vp)[10];           // указатель на вектор из 10 int'ов
int (*fp)(char, char*); // указатель на функцию
                         // получающую параметры (char, char*)
                         // и возвращающую int
```

Основная операция над указателем - разыменование, то есть ссылка на объект, на который указывает указатель. Эта операция также называется косвенным обращением. Операция разыменования - это унарное * (префиксное). Например:

```
char c1 = 'a';
char* p = &c1;    // в p хранится адрес c1
char c2 = *p;    // c2 = 'a'
```

Переменная, на которую указывает p,- это c1, а значение, которое хранится в c1, это 'a', поэтому присваиваемое c2 значение *p есть 'a'.

Над указателями можно осуществлять некоторые арифметические действия. Вот, например, функция, подсчитывающая число символов в строке (не считая завершающего 0):

```
int strlen(char* p)
{
    int i = 0;
    while (*p++) i++;
    return i;
}
```

Другой способ найти длину состоит в том, чтобы сначала найти конец строки, а затем вычесть адрес начала строки из адреса ее конца:

```
int strlen(char* p)
{
    char* q = p;
    while (*q++) ;
    return q-p-1;
}
```

Очень полезными могут оказаться указатели на функции; они обсуждаются в #4.6.7.

2.3.6 Вектора

Для типа T T[size] является типом "вектор из size элементов типа T". Элементы индексируются (нумеруются) от 0 до size-1. Например:

```
float v[3];   // вектор из трех float: v[0], v[1], v[2]
int a[2][5];  // два вектора из пяти int
char* vpc;    // вектор из 32 указателей на символ
```

Цикл для печати целых значений букв нижнего регистра можно было бы написать так:

```
extern int strlen(char*);

char alpha[] = "abcdefghijklmnoprstuvwxyz";

main()
{
    int sz = strlen(alpha);

    for (int i=0; i. Функция strlen() использовалась для подсчета числа символов в alpha; вместо этого можно было использовать значение размера alpha (#2.4.4). Если применяется набор символов ASCII, то выдача выглядит так:

```
'a' = 97 = 0141 = 0x61
'b' = 98 = 0142 = 0x62
'c' = 99 = 0143 = 0x63
...
```

Заметим, что задавать размер вектора alpha необязательно; компилятор считает число символов в символьной строке, указанной в качестве инициализатора. Использование строки как инициализатора для вектора символов - удобное, но к сожалению и единственное применение строк. Аналогичное этому присваивание строки вектору отсутствует. Например:

```
char v[9];
v = "строка";          // ошибка
```

ошибочно, поскольку присваивание не определено для векторов.

Конечно, для инициализации символьных массивов подходят не только строки. Для остальных типов нужно применять более сложную запись. Эту запись можно использовать и для символьных векторов. Например:

```
int  v1[] = { 1, 2, 3, 4 };
int  v2[] = { 'a', 'b', 'c', 'd' };

char v3[] = { 1, 2, 3, 4 };
char v4[] = { 'a', 'b', 'c', 'd' };
```

Заметьте, что v4 - вектор из четырех (а не пяти) символов; он не оканчивается нулем, как того требуют соглашение и библиотечные подпрограммы. Обычно применение такой записи ограничивается статическими объектами.

Многомерные массивы представляются как вектора векторов, и применение записи через запятую, как это делается в некоторых других языках, дает ошибку при компиляции, так как запятая (,) является операцией последования (см. #3.2.2). Попробуйте, например, сделать так:
```

```
int bad[5,2];    // ошибка
```

и так:

```
int v[5][2];
int bad = v[4,1];      // ошибка
int good = v[4][1];    // ошибка
```

Описание

```
char v[2][5];
```

описывает вектор из двух элементов, каждый из которых является вектором типа char[5]. В следующем примере первый из этих векторов инициализируется первыми пятью буквами, а второй - первыми пятью цифрами.

```
char v[2][5] = {
    'a', 'b', 'c', 'd', 'e',
    '0', '1', '2', '3', '4'
}

main() {
    for (int i = 0; i<2; i++) {
        for (int j = 0; j<5; j++)
            cout << "v[" << i << "][" << j
                 << "]=" << chr(v[i][j]) << "  ";
        cout << "\n";
    }
}
```

это дает в результате

```
v[0][0]=a v[0][1]=b v[0][2]=c v[0][3]=d v[0][4]=e
v[1][0]=0 v[1][1]=1 v[1][2]=2 v[1][3]=3 v[1][4]=4
```

2.3.7 Указатели и Вектора

Указатели и вектора в C++ связаны очень тесно. Имя вектора можно использовать как указатель на его первый элемент, поэтому пример с алфавитом можно было написать так:

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz";
char* p = alpha;
char ch;

while (ch = *p++)
    cout << chr(ch) << " = " << ch
         << " = 0" << oct(ch) << "\n";
```

Описание p можно было также записать как

```
char* p = &alpha[0];
```

Эта эквивалентность широко используется в вызовах функций, в которых векторный параметр всегда передается как указатель на первый элемент вектора; так, в примере

```
extern int strlen(char*);
char v[] = "Annemarie";
char* p = v;
strlen(p);
strlen(v);
```

функции strlen в обоих вызовах передается одно и то же значение. Вся штука в том, что этого невозможно избежать; то есть не существует способа описать функцию так, чтобы вектор v в вызове функции копировался (#4.6.3).

Результат применения к указателям арифметических операций +, -, ++ или -- зависит от типа объекта, на который они указывают. Когда к указателю p типа T* применяется арифметическая операция,

предполагается, что p указывает на элемент вектора объектов типа T; p+1 означает следующий элемент этого вектора, а p-1 - предыдущий элемент. Отсюда следует, что значение p+1 будет на sizeof(T) больше значения p. Например, выполнение

```
main()
{
    char cv[10];
    int iv[10];

    char* pc = cv;
    int* pi = iv;

    cout << "char* " << long(pc+1)-long(pc) << "\n";
    cout << "int*  " << long(ic+1)-long(ic) << "\n";
}
```

дает

```
char* 1
int*  4
```

поскольку на моей машине каждый символ занимает один байт, а каждое целое занимает четыре байта. Перед вычитанием значения указателей преобразовывались к типу long с помощью явного преобразования типа (#3.2.5). Они преобразовывались к long, а не к "очевидному" int, поскольку есть машины, на которых указатель не влезет в int (то есть, sizeof(int)

2.3.8 Структуры

Вектор есть совокупность элементов одного типа; struct является совокупностью элементов (практически) произвольных типов. Например:

```
struct address {        // почтовый адрес
    char* name;         // имя "Jim Dandy"
    long  number;       // номер дома 61
    char* street;       // улица "South Street"
    char* town;         // город "New Providence"
    char* state[2];     // штат 'N' 'J'
    int   zip;          // индекс 7974
}
```

определяет новый тип, названный address (почтовый адрес), состоящий из пунктов, требующихся для того, чтобы послать кому-нибудь корреспонденцию (вообще говоря, address не является достаточным для работы с полным почтовым адресом, но в качетве примера достаточен). Обратите внимание на точку с запятой в конце; это одно из очень немногих мест в C++, где необходимо ставить точку с запятой после фигурной скобки, поэтому люди склонны забывать об этом.

Переменные типа address могут описываться точно также, как другие переменные, а доступ к отдельным членам получается с помощью операции . (точка). Например:

```
address jd;
jd.name = "Jim Dandy";
jd.number = 61;
```

Запись, которая использовалась для инициализации векторов, можно применять и к переменным структурных типов. Например:

```
address jd = {
    "Jim Dandy",
    61, "South Street",
    "New Providence", {'N','J'}, 7974
};
```

Однако обычно лучше использовать конструктор (#5.2.4). Заметьте, что нельзя было бы инициализировать jd.state строкой "NJ". Строки оканчиваются символом '\0', поэтому в "NJ" три символа, то есть на один больше, чем влезет в jd.state.

К структурным объектам часто обращаются посредством указателей используя операцию ->. Например:

```
void print_addr(address* p)
{
    cout << p->name << "\n"
        << p->number << " " << p->street << "\n"
        << p->town << "\n"
        << chr(p->state[0]) << chr(p->state[1])
        << " " << p->zip << "\n";
}
```

Объекты типа структур можно присваивать, передавать как параметры функции и возвращать из функции в качестве результата. Например:

```
address current;

address set_current(address next)
{
    address prev = current;
    current = next;
    return prev;
}
```

Остальные осмысленные операции, такие как сравнение (== и !=) не определены. Однако пользователь может определить эти операции; см. Главу 6.

Размер объекта структурного типа нельзя вычислить просто как сумму его членов. Причина этого состоит в том, что многие машины требуют, чтобы объекты определенных типов выравнивались в памяти только по некоторым зависящим от архитектуры границам (типичный пример: целое должно быть выравнено по границе слова) или просто гораздо более эффективно обрабатывают такие объекты, если они выравнены в машине. Это приводит к "дырам" в структуре. Например, (на моей машине) sizeof(address) равен 24, а не 22, как можно было ожидать.

Заметьте, что имя типа становится доступным сразу после того, как оно встретилось, а не только после того, как полностью просмотрено все описание. Например:

```
struct link{
    link* previous;
    link* successor;
}
```

Новые объекы структурного типа не могут быть описываться, пока все

описание не просмотрено, поэтому

```
struct no_good {
    no_good member;
};
```

является ошибочным (компилятор не может установить размер no_good). Чтобы дать возможность двум (или более) структурным типам ссылаться друг на друга, можно просто описать имя как имя структурного типа. Например:

```
struct list;        // должна быть определена позднее

struct link {
    link* pre;
    link* suc;
    link* member_of;
};

struct list {
    link* head;
}
```

Без первого описания list описание link вызвало бы к синтаксическую ошибку.

2.3.9 Эквивалентность типов

Два структурных типа являются различными даже когда они имеют одни и те же члены. Например:

```
struct s1 { int a; };
struct s2 { int a; };
```

есть два разных типа, поэтому

```
s1 x;
s2 y = x;    // ошибка: несоответствие типов
```

Структурные типы отличны также от основных типов, поэтому

```
s1 x;
int i = x;    // ошибка: несоответствие типов
```

Однако, существует механизм для описания нового имени для типа без введения нового типа. Описание с префиксом typedef описывает не новую переменную данного типа, а новое имя этого типа. Например:

```
typedef char* Pchar;
Pchar p1, p2;
char* p3 = p1;
```

Это может служить удобной сокращенной записью.

2.3.10 Ссылки

Ссылка является другим именем объекта. Главное применение ссылок состоит в спецификации операций для типов, определяемых пользователем; они обсуждаются в Главе 6. Они могут также быть полезны в качестве параметров функции. Запись x& означает ссылка на x. Например:

```
int i = 1;
```

```
int& r = i;     // r и i теперь ссылаются на один int
int x = r       // x = 1
r = 2;          // i = 2;
```

Ссылка должна быть инициализирована (должно быть что-то, для чего она является именем). Заметьте, что инициализация ссылки есть нечто совершенно отличное от присваивания ей.

Вопреки ожиданиям, ниодна операция на ссылку не действует. Например,

```
int ii = 0;
int& rr = ii;
rr++;         // ii увеличивается на 1
```

допустимо, но rr++ не увеличивает ссылку; вместо этого ++ применяется к int, которым оказывается ii. Следовательно, после инициализации значение ссылки не может быть изменено; она всегда ссылается на объект, который ей было дано обозначать (денотировать)

при инициализации. Чтобы получить указатель на объект, денотируемый ссылкой rr, можно написать &rr.

Очевидным способом реализации ссылки является константный указатель, который разыменовывается при каждом использовании. Это делает инициализацию ссылки тривиальной, когда инициализатор является lvalue (объектом, адрес которого вы можете взять, см. #c.5). Однако инициализатор для &T не обязательно должен быть lvalue, и даже не должен быть типа T. В таких случаях:

[1] Во-первых, если необходимо, применяются преобразование типа (#c.6.6-8,#c.8.5.6);

[2] Затем полученное значение помещается во временную переменную; и

[3] Наконец, ее адрес используется в качестве значения инициализатора.

Рассмотрим описание

```
double& dr = 1;
```

Это интерпретируется так:

```
double* drp;    // ссылка, представленная как указатель
double temp;
temp = double(1);
drp = &temp;
```

Ссылку можно использовать для реализации функции, которая, как предполагается, изменяет значение своего параметра.

```
int x = 1;
void incr(int& aa) { aa++; }
incr(x)                 // x = 2
```

По определению семантика передачи параметра та же, что семантика инициализации, поэтому параметр aa функции incr становится другим именем для x. Однако, чтобы сделать программу читаемой, в большинстве случаев лучше всего избегать функций, которые изменяют значение своих параметров. Часто предпочтительно явно возвращать значение из функции или требовать в качестве параметра указатель:

```
int x = 1;
int next(int p) { return p+1; }
x = next(x);            // x = 2

void inc(int* p) { (*p)++; }
inc(&x);                // x = 3
```

Ссылки также можно применять для определения функций, которые могут использоваться и в левой, и в правой части присваивания. Опять, большая часть наиболее интересных случаев этого обнаруживается в проектировании нетривиальных определяемых пользователем типов. Для примера давайте определим простой ассоциативный массив. Вначале мы определим структуру пары следующим образом:

```
struct pair {
    char* name;
    int val;
};
```

Основная идея состоит в том, что строка имеет ассоциированное с ней целое значение. Легко определить функцию поиска find(), которая поддерживает структуру данных, состоящую из одного pair для каждой отличной отличной от других строки, которая была ей представлена. Для краткости представления используется очень простая (и неэффективная) реализация:

```
const large = 1024;
static pair vec[large+1};

pair* find(char* p)
/*
    поддерживает множество пар "pair":
    ищет p, если находит, возвращает его "pair",
    иначе возвращает неиспользованную "pair"
*/
{
    for (int i=0; vec[i].name; i++)
        if (strcmp(p,vec[i].name)==0) return &vec[i];

    if (i == large) return &vec[large-1];

    return &vec[i];
}
```

Эту функцию может использовать функция value(), реализующая массив целых, индексированый символьными строками (вместо обычного способа):

```
int& value(char* p)
{
    pair* res = find(p);
    if (res->name == 0) {       // до сих пор не встречалось:
      res->name = new char[strlen(p)+1];  // инициализировать
      strcpy(res->name,p);
      res->val = 0;             // начальное значение 0
    }
    return res->val;
}
```

Для данной в качестве параметра строки value() находит целый объект (а не значение соответствующего целого); после чего она возвращает ссылку на него. Ее можно использовать, например, так:

```
const MAX = 256;    // больше самого большого слова

main()
// подсчитывает число вхождений каждого слова во вводе
```

```
{
    char buf[MAX];

    while (cin>>buf) value(buf)++;

    for (int i=0; vec[i].name; i++)
        cout << vec[i].name << ": " << vec [i].val << "\n";
}
```

На каждом проходе цикл считывает одно слово из стандартной строки ввода cin в buf (см. Главу 8), а затем обновляет связанный с ней счетчик спомощью find(). И, наконец, печатается полученная таблица различных слов во введенном тексте, каждое с числом его встречаемости. Например, если вводится

```
aa bb bb aa aa bb aa aa
```

то программа выдаст:

```
aa: 5
bb: 3
```

Легко усовершенствовать это в плане собственного типа ассоциированного массива с помощью класса с перегруженной операцией (#6.7) выбора [].

2.3.11 Регистры

Во многих машинных архитектурах можно обращаться к (небольшим) объектам заметно быстрее, когда они помещены в регистр. В идеальном случае компилятор будет сам определять оптимальную стратегию использования всех регистров, доступных на машине, для которой компилируется программа. Однако это нетривиальная задача, поэтому иногда программисту стоит дать подсказку компилятору. Это делается с помощью описания объекта как register. Например:

```
register int i;
register point cursor;
register char* p;
```

Описание register следует использовать только в тех случаях, когда эффективность действительно важна. Описание каждой переменной как register засорит текст программы и может даже увеличить время выполнения (обычно воспринимаются все инструкции по помещению объекта в регистр или удалению его оттуда).

Невозможно получить адрес имени, описанного как register, регистр не может также быть глобальным.

2.4 Константы

C++ дает возможность записи значений основных типов: символьных констант, целых констант и констант с плавающей точкой. Кроме того, ноль (0) может использоваться как константа любого указательного типа, и символьные строки являются константами типа char[]. Можно также задавать символические константы. Символическая константа - это имя, значение которого не может быть изменено в его области видимости. В C++ имеется три вида символических констант: (1) любому значению любого типа можно дать имя и использовать его как константу, добавив к его описанию ключевое слово const; (2) множество целых констант может быть определено как перечисление; и (3) любое имя вектора или функции является константой.

2.4.1 Целые Константы

Целые константы предстают в четырех обличьях: десятичные, восьмеричные, шестнадцатиричные и символьные константы. Десятичные используются чаще всего и выглядят так, как можно было бы ожидать:

```
0    1234    976    12345678901234567890
```

Десятичная константа имеет тип int, при условии, что она влезает в int, в противном случае ее тип long. Компилятор должен предупреждать о константах, которые слишком длинны для представления в машине.

Константа, которая начинается нулем за которым идет x (0x), является шестнадцатиричным числом (с основанием 16), а константа, которая начинается нулем за которым идет цифра, является восьмеричным числом (с основанием 8). Вот примеры восьмеричных констант:

```
0        02        077        0123
```

их десятичные эквиваленты - это 0, 2, 63, 83. В шестнадцатиричной записи эти константы выглядят так:

```
0x0      0x2       0x3f       0x53
```

Буквы a, b, c, d, e и f, или их эквиваленты в верхнем регистре, используются для представления чисел 10, 11. 12, 13, 14 и 15, соответственно. Восьмеричная и шестнадцатиричная записи наиболее полезны для записи набора битов; применение этих записей для выражения обычных чисел может привести к неожиданностям. Например, на машине, где int представляется как двоичное дополнительное шестнадцатеричное целое, 0xffff является отрицательным десятичным числом -1; если бы для представления целого использовалось большее число битов, то оно было бы числом 65535.

2.4.2 Константы с Плавающей Точкой

Константы с плавающей точкой имеют тип double. Как и в предыдущем случае, компилятор должен предупреждать о константах с плавающей точкой, которые слишком велики, чтобы их можно было представить. Вот некоторые константы с плавающей точкой:

```
1.23    .23      0.23     1.    1.0     1.2e10    1.23e-15
```

Заметьте, что в середине константы с плавающей точкой не может встречаться пробел. Например, 65.43 e-21 является не константой с плавающей точкой, а четыремя отдельными лексическими символами (лексемами):

```
65.43    e    -    21
```

и вызовет синтаксическую ошибку.

Если вы хотите иметь константу с плавающей точкой типа float, вы можете определить ее так (#2.4.6):

```
const float pi = 3.14159265;
```

2.4.3 Символьные Константы

Хотя в C++ и нет отдельного символьного типа данных, точнее, символ может храниться в целом типе, в нем для символов имеется специальная и удобная запись. Символьная константа - это символ, заключенный в одинарные кавычки; например, 'a' или '0'. Такие символьные константы в действительности являются символическими

константами для целого значения символов в наборе символов той машины, на которой будет выполняться программа (который не обязательно совпадает с набором символов, применяемом на том компьютере, где программа компилируется). Поэтому, если вы выполняетесь на машине, использующей набор символов ASCII, то значением '0' будет 48, но если ваша машина использует EBCDIC, то оно будет 240. Употребление символьных констант вместо десятичной записи делает программу более переносимой. Несколько символов также имеют стандартные имена, в которых обратная косая \ используется как escape-символ:

```
'\b'    возрат назад
'\f'    перевод формата
'\n'    новая строка
'\r'    возврат каретки
'\t'    горизонтальная табуляция
'\v'    вертикальная табуляция
'\\'    обратная косая (обратный слэш)
'\''    одинарная кавычка
'\"'    двойная кавычка
'\0'    null, пустой символ, целое значение 0
```

Вопреки их внешнему виду каждое является одним символом. Можно также представлять символ одно-, дву- или трехзначным восьмеричным

числом (символ \, за которым идут восьмеричные цифры), или одно-, дву- или трехзначным шестнадцатиричным числом (\x, за которым идут шестнадцатиричные цифры). Например:

```
'\6'      '\x6'       6       ASCII ack
'\60'     '\x30'      48      ASCII '0'
'\137'    '\x05f'     95      ASCII '_'
```

Это позволяет представлять каждый символ из машинного набора символов, и в частности вставлять такие символы в символьные строки (см. следующий раздел). Применение числовой записи для символов делает программу непереносимой между машинами с различными наборами символов.

2.4.4 Строки

Строковая константа - это последовательность символов, заключенная в двойные кавычки:

```
"это строка"
```

Каждая строковая константа содержит на один символ больше, чем кажется; все они заканчиваются пустым символом '\0' со значением 0. Например:

```
sizeof("asdf")==5;
```

Строка имеет тип "вектор из соответствующего числа символов", поэтому "asdf" имеет тип char[5]. Пустая строка записывается "" (и имеет тип char[1]). Заметьте, что для каждой строки s `strlen(s)==sizeof(s)-1`, поскольку strlen() не учитывает завершающий `0`.

Соглашение о представлении неграфических символов с обратной косой можно использовать также и внутри строки. Это дает возможность представлять в строке двойные кавычки и escape-символ \. Самым обычным символом этого рода является, безусловно, символ новой строки '\n'. Например:

```
  cout << "гудок в конце сообщения\007\n"
```

где 7 - значение ASKII символа bel (звонок).
  В строке невозможно иметь "настоящую" новую строку:

```
  "это не строка,
  а синтаксическая ошибка"
```

Однако в строке может стоять обратная косая, сразу после которой идет новая строка; и то, и другое будет проигнорировано. Например:

```
  cout << "здесь все \
  ok"
```

напечатает

```
  здесь все ok
```

Новая строка, перед которой идет escape (обратная косая), не приводит к появлению в строке новой строки, это просто договоренность о записи.

  В строке можно иметь пустой символ, но большинство программ не будет предполагать, что есть символы после него. Например, строка "asdf\000hjkl" будет рассматриваться стандартными функциями, вроде strcpy() и strlen(), как "asdf".
  Вставляя численную константу в строку с помощью восьмеричной или шестнадцатиричной записи благоразумно всегда использзовать число из трех цифр. Читать запись достаточно трудно и без необходимости беспокоиться о том, является ли символ после константы цифрой или нет. Разберите эти примеры:

```
  char v1[] = "a\x0fah\0129";    // 'a' '\xfa' 'h' '\12' '9'
  char v2[] = "a\xfah\129";      // 'a' '\xfa' 'h' '\12' '9'
  char v3[] = "a\xfad\127";      // 'a' '\xfad' '\127'
```

Имейте в виду, что двузначной шестнадцатиричной записи на машинах с 9-битовым байтом будет недостаточно.

2.4.5 Ноль

  Ноль (0) можно употреблять как константу любого целого, плавающего или указательного типа. Никакой объект не размещается по адресу 0. Тип нуля определяется контекстом. Обычно (но не обязательно) он представляется набором битов все-нули соответствующей длины.

2.4.6 Const

  Ключевое слово const может добавляться к описанию объекта, чтобы сделать этот объект константой, а не переменной. Например:

```
  const int model = 145;
  const int v[] = { 1, 2, 3, 4 };
```

  Поскольку константе ничего нельзя присвоить, она должна быть инициализирована. Описание чего-нибудь как const гарантирует, что его значение не изменится в области видимости:

```
  model = 145;        // ошибка
  model++;            // ошибка
```

  Заметьте, что const изменяет тип, то есть ограничивает способ использования объекта, вместо того, чтобы задавать способ

размещения константы. Поэтому например вполне разумно, а иногда и полезно, описывать функцию как возвращающую const:

```
const char* peek(int i)
{
    return private[i];
}
```

Функцию вроде этой можно было бы использовать для того, чтобы давать кому-нибудь читать строку, которая не может быть затерта или переписана (этим кем-то).

С другой стороны, компилятор может несколькими путями воспользоваться тем, что объект является константой (конечно, в зависимости от того, насколько он сообразителен). Самое очевидное - это то, что для константы не требуется выделять память, поскольку компилятор знает ее значение. Кроме того, инициализатор константы часто (но не всегда) является константным выражением, то есть он может быть вычислен на стадии компиляции. Однако для вектора констант обычно приходится выделять память, поскольку компилятор в общем случае не может вычислить, на какие элементы вектора сделаны ссылки в выражениях. Однако на многих машинах даже в этом случае может достигаться повышение эффективности путем размещения векторов констант в память, доступную только для чтения.

Использование указателя вовлекает два объекта: сам указатель и указываемый объект. Снабжение описания указателя "префиксом" const делает объект, но не сам указатель, константой. Например:

```
const char* pc = "asdf";    // указатель на константу
pc[3] = 'a';                // ошибка
pc = "ghjk";                // ok
```

Чтобы описать сам указатель, а не указываемый объект, как константный, используется операция const*. Например:

```
char *const cp = "asdf";    // константный указатель
cp[3] = 'a';                // ok
cp = "ghjk";                // ошибка
```

Чтобы сделать константами оба объекта, их оба нужно описать const. Например:

```
const char *const cpc = "asdf";    // const указатель на const
cpc[3] = 'a';                      // ошибка
cpc = "ghjk";                      // ошибка
```

Объект, являющийся контстантой при доступе к нему через один указатель, может быть переменной, когда доступ осуществляется другими путями. Это в частноси полезно для параметров функции. Посредством описания параметра указателя как const функции запрещается изменять объект, на который он указывает. Например:

```
char* strcpy(char* p, const char* q); // не может изменить q
```

Указателю на константу можно присваивать адрес переменной, поскольку никакого вреда от этого быть не может. Однако нельзя присвоить адрес константы указателю, на который не было наложено ограничение, поскольку это позволило бы изменить значение объекта. Например:

```
int a = 1;
const c = 2;
const* p1 = &c;    // ok
const* p2 = &a;    // ok
```

```
int* p3 = &c;       // ошибка
*p3 = 7;            // меняет значение с
```

Как обычно, если тип в описании опущен, то он предполагается int.

2.4.7 Перечисления

Есть другой метод определения целых констант, который иногда более удобен, чем применение const. Например:

```
enum { ASM, AUTO, BREAK };
```

определяет три целых константы, называемы перечислителями, и присваивает им значения. Поскольку значения перечислителей по умолчанию присваиваются начиная с 0 в порядке возрастания, это эквивалентно записи:

```
const ASM = 0;
const AUTO = 1;
const BREAK = 2;
```

Перечисление может быть именованным. Например:

```
enum keyword { ASM, AUTO, BREAK };
```

Имя перечисления становится синонимом int, а не новым типом. Описание переменной keyword, а не просто int, может дать как программисту, так и компилятору подсказку о том, что использование преднамеренное. Например:

```
keyword key;

switch (key) {
case ASM:
    // что-то делает
    break;
case BREAK:
    // что-то делает
    break;
}
```

побуждает компилятор выдать предупреждение, поскольку только два значения keyword из трех используются.

Можно также задавать значения перечислителей явно. Например:

```
enum int16 {
    sign=0100000,               // знак
    most_significant=040000,    // самый значимый
    least_significant=1         // наименее значимый
};
```

Такие значения не обязательно должны быть различными, возрастающими или положительными.

2.5 Экономия Пространства

В ходе программирования нетривиальных разработок неизбежно наступает время, когда хочется иметь больше пространства памяти, чем имеется или отпущено. Есть два способа выжать побольше пространства из того, что доступно:

[1] Помещение в байт более одного небольшого объекта; и

[2] Использование одного и того же пространства для хранения разных объектов в разное время.

Первого можно достичь с помощью использования полей, второго - через использование объединений. Эти конструкции описываются в следующих разделах. Поскольку обычное их применение состоит чисто в оптимизации программы, и они в большинстве случаев непереносимы, программисту следует дважды подумать, прежде чем использовать их. Часто лучше изменить способ управления данными; например, больше полагаться на динамически выделяемую память (#3.2.6) и меньше на заранее выделенную статическую память.

2.5.1 Поля

Использование char для представления двоичной переменной, например, переключателя включено/выключено, может показаться экстравагантным, но char является наименьшим объектом, который в C++ может выделяться независимо. Можно, однако, сгруппировать несколько таких крошечных переменных вместе в виде полей struct. Член определяется как поле путем указания после его имени числа битов, которые он занимает. Допустимы неименованные поля; они не влияют на смысл именованных полей, но неким машинно-зависимым образом могут улучшить размещение:

```
struct sreg {
    unsigned enable : 1;
    unsigned page : 3;
    unsigned : 1;        // неиспользуемое
    unsigned mode : 2;
    unsigned : 4:        // неиспользуемое
    unsigned access : 1;
    unsigned length : 1;
    unsigned non_resident : 1;
}
```

Получилось размещение регистра 0 сосояния DEC PDP11/45 (в предположении, что поля в слове размещаются слева направо). Этот пример также иллюстрирует другое основное применение полей: именовать части внешне предписанного размещения. Поле должно быть целого типа и используется как другие целые, за исключением того, что невозможно взять адрес поля. В ядре операционной системы или в отладчике тип sreg можно было бы использовать так:

```
sreg* sr0 = (sreg*)0777572;
//...
if (sr->access) {        // нарушение доступа
    // чистит массив
    sr->access = 0;
}
```

Однако применение полей для упаковки нескольких переменных в один байт не обязательно экономит пространство. Оно экономит пространство, занимаемое данными, но объем кода, необходимого для манипуляции этими переменными, на большинстве машин возрастает. Известны программы, которые значительно сжимались, когда двоичные

переменные преобразовывались из полей бит в символы! Кроме того, доступ к char или int обычно намного быстрее, чем доступ к полю. Поля - это просто удобная и краткая запись для применения логических операций с целью извлечения информации из части слова

или введения информации в нее.

2.5.2 Объединения

Рассмотрим проектирование символьной таблицы, в которой каждый элемент содержит имя и значение, и значение может быть либо строкой, либо целым:

```
struct entry {
    char* name;
    char  type;
    char* string_value;        // используется если type == 's'
    int   int_value;           // используется если type == 'i'
};

void print_entry(entry* p)
{
    switch p->type {
    case 's':
        cout << p->string_value;
        break;
    case 'i':
        cout << p->int_value;
        break;
    default:
        cerr << "испорчен type\n";
        break;
    }
}
```

Поскольку string_value и int_value никогда не могут использоваться одновременно, ясно, что пространство пропадает впустую. Это можно легко исправить, указав, что оба они должны быть членами union (объединения); например, так:

```
struct entry {
    char* name;
    char  type;
    union {
        char* string_value;    // используется если type == 's'
        int   int_value;       // используется если type == 'i'
    };
};
```

Это оставляет всю часть программы, использующую entry, без изменений, но обеспечивает, что при размещении entry string_value и int_value имеют один и тот же адрес. Отсюда следует, что все члены объединения вместе занимают лишь столько памяти, сколько занимает наибольший член.

Использование объединений таким образом, чтобы при чтении значения всегда применялся тот член, с применением которого оно записывалось, совершенно оптимально. Но в больших программах непросто гарантировать, что объединения используются только таким образом, и из-за неправильного использования могут появляться трудно уловимые ошибки. Можно капсулизировать объединение таким образом, чтобы соответствие между полем типа и типами членов было гарантированно правильным (#5.4.6).

Объединения иногда испольуют для "преобразования типов" (это делают главным образом программисты, воспитанные на языках, не обладающих средствами преобразования типов, где жульничество является необходимым). Например, это "преобразует" на VAX'е int в int*, просто предполагая побитовую эквивалентность:

```
struct fudge {
    union {
        int  i;
        int* p;
```

```
    };
};

fudge a;
a.i = 4096;
int* p = a.p;    // плохое использование
```

Но на самом деле это совсем не преобразование: на некоторых машинах int и int* занимают неодинаковое количество памяти, а на других никакое целое не может иметь нечетный адрес. Такое применение объединений непереносимо, а есть явный способ указать преобразование типа (#3.2.5).

Изредка объединения умышленно применяют, чтобы избежать преобразования типов. Можно, например, использовать fudge, чтобы узнать представление указателя 0:

```
fudge.p = 0;
int i = fudge.i;    // i не обязательно должно быть 0
```

Можно также дать объединению имя, то есть сделать его полноправным типом. Например, fudge можно было бы описать так:

```
union fudge {
        int  i;
        int* p;
};
```

и использовать (неправильно) в точности как раньше. Имеются также и оправданные применения именованных объединений; см. #5.4.6.

2.6 Упражнения

1. (*1) Заставьте работать программу с "Hello, world" (1.1.1).
2. (*1) Для каждого описания в #2.1 сделайте следующее: Если описание не является определением, напишите для него определение. Если описание является определением, напишите для него описание, которое при этом не является определением.
3. (*1) Напишите описания для: указателя на символ; вектора из 10 целых; ссылки на вектор из 10 целых; указателя на вектор из символьных строк; указателя на указатель на символ; константного целого; указателя на константное целое; и константного указателя на целое. Каждый из них инициализируйте.
4. (*1.5) Напишите программу, которая печатает размеры основных и указательных типов. Используйте операцию sizeof.
5. (*1.5) Напишите программу, которая печатает буквы 'a'...'z' и цифры '0'...'9' и их числовые значения. Сделайте то же для остальных печатаемых символов. Сделайте то же, но используя шестнадцатиричную запись.
6. (*1) Напечатайте набор битов, которым представляется указатель 0 на вашей системе. Подсказка: #2.5.2.
7. (*1.5) Напишите функцию, печатающую порядок и мантиссу параметра типа double.
8. (*2) Каковы наибольшие и наименьшие значения, на вашей системе, следующих типов: char, short, int, long, float, double, unsigned, char*, int* и void*? Имеются ли дополнительные ограничения на принимаемые ими значения? Может ли, например, int* принимать нечетное значение? Как выравниваются в памяти объекты этих типов? Может ли, например, int иметь нечетный адрес?
9. (*1) Какое самое длинное локальное имя можно использовать в C++ программе в вашей системе? Какое самое длинное внешнее имя можно использовать в C++ программе в вашей системе? Есть ли

какие-нибудь ограничения на символы, которые можно употреблять в имени?

10. (*2) Определите one следующим образом:

```
const one = 1;
```

Попытайтесь поменять значение one на 2. Определите num следующим образом:

```
const num[] = { 1, 2 };
```

Попытайтесь поменять значение num[1] на 2.

11. (*1) Напишите функцию, переставляющую два целых (меняющую значения). Используйте в качесте типа параметра int*. Напишите другую переставляющую функцию, использующую в качесте типа параметра int&.
12. (*1) Каков размер вектора str в следующем примере:

```
char str[] = "a short string";
```

Какова длина строки "a short string"?

13. (*1.5) Определите таблицу названий месяцев года и числа дней в них. Выведите ее. Сделайте это два раза: один раз используя вектор для названий и вектор для числа дней, и один раз используя вектор структур, в каждой из которых хранится название месяца и число дней в нем.
14. (*1) С помощью typedef определите типы: беззнаковый char; константный беззнаковый char; указатель на целое; указатель на указатель на char; указатель на вектора символов; вектор из 7 целых указателей; указатель на вектор из 7 целых указателей; и вектор из 8 векторов из 7 целых указателей.

Глава 3

Выражения и операторы

С другой стороны,
мы не можем игнорировать эффективность
- Джон Бентли

C++ имеет небольшой, но гибкий набор различных видов операторов для контроля потока управления в программе и богатый набор операций для манипуляции данными. С наиболее общепринятыми средствами вас познакомит один законченный пример. После него приводится резюмирующий обзор выражений и с довольно подробно описываются явное описание типа и работа со свободной памятью. Потом представлена краткая сводка операций, а в конце обсуждаются стиль выравнивания* и комментарии.

3.1 Настольный калькулятор

С операторами и выражениями вас познакомит приведенная здесь программа настольного калькулятора, предоставляющего четыре стандартные арифметические опреации над числами с плавающей точкой. Пользователь может также определять переменные. Например, если вводится

```
r=2.5
area=pi*r*r
```

(pi определено заранее), то программа калькулятора напишет:

```
2.5
19.635
```

где 2.5 - результат первой введенной строки, а 19.635 - результат второй.

Калькулятор состоит из четырех основных частей: программы синтаксического разбора (parser'a), функции ввода, таблицы имен и управляющей программы (драйвера). Фактически, это миниатюрный компилятор, в котором программа синтаксического разбора производит синтаксический анализ, функция ввода осуществляет ввод и лексический анализ, в таблице имен хранится долговременная информация, а драйвер распоряжается инициализцией, выводом и обработкой ошибок. Можно было бы многое добавить в этот калькулятор, чтобы сделать его более полезным, но в существующем виде эта программа и так достаточно длинна (200 строк), и большая часть дополнительных возможностей просто увеличит текст программы не давая дополнительного понимания применения C++.

---

* Нам неизвестен русскоязычный термин, эквивалентный английскому indentation. Иногда это называется отступами. (прим. перев.)

3.1.1 Программа синтаксического разбора

Вот грамматика языка, допускаемого калькулятором:

```
program:
    END                     // END - это конец ввода
    expr_list END

expr_list:
    expression PRINT      // PRINT - это или '\n' или ';'
    expression PRINT expr_list

expression:
    expression + term
    expression - term
    term

term:
    term / primary
    term * primary
    primary

primary:
    NUMBER                // число с плавающей точкой в C++
    NAME                  // имя C++ за исключением '_'
    NAME = expression
    - primary
    ( expression )
```

Другими словами, программа есть последовательность строк. Каждая строка состоит из одного или более выражений, разделенных запятой. Основными элементами выражения являются числа, имена и операции *, /, +, - (унарный и бинарный) и =. Имена не обязательно должны описываться до использования.

Используемый метод синтаксического анализа обычно называется рекурсивным спуском; это популярный и простой нисходящий метод. В таком языке, как C++, в котором вызовы функций относительно дешевы, этот метод к тому же и эффективен. Для каждого правила вывода грамматики имеется функция, вызывающая другие функции. Терминальные символы (например, END, NUMBER, + и -) распознаются лексическим анализатором get_token(), а нетерминальные символы распознаются функциями синтаксического анализа expr(), term() и prim(). Как только оба операнда (под)выражения известны, оно вычисляется; в настоящем компиляторе в этой точке производится

генерация кода.

Программа разбора для получения ввода использует функцию get_token(). Значение последнего вызова get_token() находится в переменной curr_tok; curr_tok имеет одно из значений перечисления token_value:

```
enum token_value {
    NAME        NUMBER    END
    PLUS='+'    MINUS='-'    MUL='*'    DIV='/'
    PRINT=';'    ASSIGN='='    LP='('    RP=')'
};
token_value curr_tok;
```

В каждой функции разбора предполагается, что было обращение к get_token(), и в curr_tok находится очередной символ, подлежащий анализу. Это позволяет программе разбора заглядывать на один лексический символ (лексему) вперед и заставляет функцию разбора всегда читать на одну лексему больше, чем используется правилом, для обработки которого она была вызвана. Каждая функция разбора вычисляет "свое" выражение и возвращает значение. Функция expr() обрабатывает сложение и вычитание; она состоит из простого цикла, который ищет термы для сложения или вычитания:

```
double expr()                    // складывает  и вычитает
{
    double left = term();

    for(;;)                      // ``навсегда``
        switch(curr_tok) {
        case PLUS:
            get_token();         // ест '+'
            left += term();
            break;
        case MINUS:
            get_token();         // ест '-'
            left -= term();
            break;
        default:
            return left;
        }
}
```

Фактически сама функция делает не очень много. В манере, достаточно типичной для функций более высокого уровня в больших программах, она вызывает для выполнения работы другие функции. Заметьте, что выражение 2-3+4 вычисляется как (2-3)+4, как указано грамматикой.

Странная запись for(;;) - это стандартный способ задать бесконечный цикл; можно произносить это как "навсегда"*. Это вырожденная форма оператора for; альтернатива - while(1). Выполнение оператора switch повторяется до тех пор, пока не будет найдено ни + ни -, и тогда выполняется оператор return в случае default.

Операции += и -= используются для осуществления сложения и вычитания. Можно было бы не изменяя смысла программы использовать left=left+term() и left=left-term(). Однако left+=term() и left-=term() не только короче, но к тому же явно выражают подразумеваемое действие. Для бинарной операции @ выражение x@=y означает x=x@y за исключением того, что x вычисляется только один раз. Это применимо к бинарным операциям

```
+    -    *    /    %    &    |    ^    <<    >>
```

поэтому возможны следующие операции присваивания:

```
+=    -=    *=    /=    %=    &=    |=    ^=    <<=    >>=
```

---

* игра слов: "for" - "forever" (навсегда). (прим. перев.)

Каждая является отдельной лексемой, поэтому a+ =1 является синтаксической ошибкой из-за пробела между + и =. (% является операцией взятия по модулю; &,| и ^ являются побитовми операциями И, ИЛИ и исключающее ИЛИ; << и >> являются операциями левого и правого сдвига). Функции term() и get_token() должны быть описаны до expr().

Как организовать программу в виде набора файлов, обсуждается в Главе 4. За одним исключением все описания в данной программе настольного калькулятора можно упорядочить так, чтобы все описывалось ровно один раз и до использования. Исключением является expr(), которая обращается к term(), которая обращается к prim(), которая в свою очередь обращается к expr(). Этот круг надо как-то разорвать; описание

```
double expr();    // без этого нельзя
```

перед prim() прекрасно справляется с этим.

Функция term() аналогичным образом обрабатывает умножение и сложение:

```
double term()                    // умножает и складывает
{
    double left = prim();

    for(;;)
        switch(curr_tok)    {
        case MUL:
            get_token();         // ест '*'
            left *= prim();
            break;
        case DIV:
            get_token();         // ест '/'
            double d = prim();
            if (d == 0) return error("деление на 0");
            left /= d;
            break;
        default:
            return left;
        }
}
```

Проверка, которая делается, чтобы удостовериться в том, что нет деления на ноль, необходима, поскольку результат деления на ноль неопределен и как правило является роковым. Функция error(char*) будет описана позже. Переменная d вводится в программе там, где она нужна, и сразу же инициализируется. Во многих языках описание может располагаться только в голове блока. Это ограничение может приводить к довольно скверному искажению стиля программирования и/или излишним ошибкам. Чаще всего неинициализированнные локальные переменные являются просто признаком плохого стиля; исключением являются переменные, подлежащие инициализации посредством ввода, и переменные векторного или структурного типа, кторые нельзя удобно

инициализировать одними присваиваниями*. Заметьте, что = является операцией присваивания, а == операцией сравнения.

Функция prim, обрабатывающая primary, написана в основном в том же духе, не считая того, что немного реальной работы в ней все-таки

выполняется, и нет нужды в цикле, поскольку мы попадаем на более низкий уровень иерархии вызовов:

```
double prim()                 // обрабатывает primary (первичные)
{
    switch (curr_tok) {
    case NUMBER:              // константа с плавающей точкой
        get_token();
        return number_value;
    case NAME:
        if (get_token() == ASSIGN) {
            name* n = insert(name_string);
            get_token();
            n->value = expr();
            return n->value;
        }
        return look(name-string)->value;
    case MINUS:               // унарный минус
        get_token();
        return -prim();
    case LP:
        get_token();
        double e = expr();
        if (curr_tok != RP) return error("должна быть )");
        get_token();
        return e;
    case END:
        return 1;
    default:
        return error("должно быть primary");
    }
}
```

При обнаружении NUMBER (то есть, константы с плавающей точкой), возвращается его значение. Функция ввода get_token() помещает значение в глобальную переменную number_value. Использование в программе глобальных переменных часто указывает на то, что структура не совсем прозрачна, что применялась некоторого рода оптимизация. Здесь дело обстоит именно так. Теоретически лексический символ обычно состоит из двух частей: значения, определяющего вид лексемы (в данной программе token_value), и (если необходимо) значения лексемы. У нас имеется только одна простая переменная curr_tok, поэтому для хранения значения последнего считанного NUMBER понадобилась глобальная переменная number_value. Это работает только потому, что калькулятор при вычислениях использует только одно число перед чтением со входа другого.

Так же, как значение последнего встреченного NUMBER хранится в number_value, в name_string в виде символьной строки хранится представление последнего прочитанного NAME. Перед тем, как что-либо

---

* В языке немного лучше этого с этими исключениями тоже надо бы справляться. (прим. автора)

сделать с именем, калькулятор должен заглянуть вперед, чтобы посмотреть, осуществляется ли присваивание ему, или оно просто используется. В обоих случаях надо справиться в таблице имен. Сама таблица описывается в #3.1.3; здесь надо знать только, что она состоит из элементов вида:

```
srtuct name {
    char* string;
    char* next;
    double value;
}
```

где next используется только функциями, которые поддерживают работу с таблицей:

```
name* look(char*);
name* insert(char*);
```

Обе возвращают указатель на name, соответствующее параметру - символьной строке; look() выражает недовольство, если имя не было определено. Это значит, что в калькуляторе можно использовать имя без предварительного описания, но первый раз оно должно использоваться в левой части присваивания.

3.1.2 Функция ввода

Чтение ввода - часто самая запутанная часть программы. Причина в том, что если программа должна общаться с человеком, то она должна справляться с его причудами, условностями и внешне случайными ошибками. Попытки заставить человека вести себя более удобным для машины образом часто (и справедливо) рассматриваются как оскорбительные. Задача низкоуровненовой программы ввода состоит в том, чтобы читать символы по одному и составлять из них лексические символы более высокого уровня. Далее эти лексемы служат вводом для программ более высокого уровня. У нас ввод низкого уровня осуществляется get_token(). Обнадеживает то, что написание программ ввода низкого уровня не является ежедневной работой; в хорошей системе для этого будут стандартные функции.

Для калькулятора правила ввода сознательно были выбраны такими, чтобы функциям по работе с потоками было неудобно эти правила обрабатывать; незначительные изменения в определении лексем сделали бы get_token() обманчиво простой.

Первая сложность состоит в том, что символ новой строки '\n' является для калькулятора существенным, а функции работы с потоками считают его символом пропуска. То есть, для этих функций '\n' значим только как ограничитель лексемы. Чтобы преодолеть это, надо проверять пропуски (пробел, символы табуляции и т.п.):

```
char ch

do {    // пропускает пропуски за исключением '\n'
    if(!cin.get(ch)) return curr_tok = END;
} while (ch!='\n' && isspace(ch));
```

Вызов cin.get(ch) считывает один символ из стандартного потока ввода в ch. Проверка if(!cin.get(ch)) не проходит в случае, если из cin нельзя считать ниодного символа; в этом случае возвращается END, чтобы завершить сеанс работы калькулятора. Используется операция ! (НЕ), поскольку get() возвращает в случае успеха ненулевое значение.

Функция (inline) isspace() из обеспечивает стандартную проверку на то, является ли символ пропуском (#8.4.1); isspace(c) возвращает ненулевое значение, если с является символом пропуска, и ноль в противном случае. Проверка реализуется в виде поиска в таблице, поэтому использование isspace() намного быстрее, чем проверка на отдельные символы пропуска; это же относится и к функциям isalpha(), isdigit() и isalnum(), которые используются в get_token().

После того, как пустое место пропущено, следущий символ используется для определения того, какого вида какого вида лексема приходит. Давайте сначала рассмотрим некоторые случаи отдельно, прежде чем приводить всю функцию. Ограничители лексем '\n' и ';' обрабатываются так:

```
switch (ch) {
```

```
  case ';':
  case '\n':
      cin >> WS;    // пропустить пропуск
      return curr_tok=PRINT;
```

Пропуск пустого места делать необязательно, но он позволяет избежать повторных обращений к get_token(). WS - это стандартный пропусковый объект, описанный в ; он используется только для сброса пропуска. Ошибка во вводе или конец ввода не будут обнаружены до следующего обращения к get_token(). Обратите внимание на то, как можно использовать несколько меток case (случаев) для одной и той же последовательности операторов, обрабатывающих эти случаи. В обоих случаях возвращается лексема PRINT и помещается в curr_tok.

Числа обрабатыватся так:

```
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
  case '.':
      cin.putback(ch);
      cin >> number_value;
      return curr_tok=NUMBER;
```

Располагать метки случаев case горизонтально, а не вертикально, не очень хорошая мысль, поскольку читать это гораздо труднее, но отводить по одной строке на каждую цифру нудно.

Поскольку операция >> определена также и для чтения констант с плавающей точкой в double, программирование этого не составляет труда: сперва начальный символ (цифра или точка) помещается обратно в cin, а затем можно считывать константу в number_value.

Имя, то есть лексема NAME, определяется как буква, за которой возможно следует несколько букв или цифр:

```
  if (isalpha(ch)) {
      char* p = name_string;
      *p++ = ch;
      while (cin.get(ch) && isalnum(ch)) *p++ = ch;
      cin.putback(ch);
      *p = 0;
      return curr_tok=NAME;
  }
```

Эта часть строит в name_string строку, заканчивающуюся нулем. Функции isalpha() и isalnum() заданы в ; isalnum(c) не ноль, если с буква или цифра, ноль в противном случае.

Вот, наконец, функция ввода полностью:

```
  token_value get_token()
  {
      char ch;

      do {    // пропускает пропуски за исключением '\n'
          if(!cin.get(ch)) return curr_tok = END;
      } while (ch!='\n' && isspace(ch));

      switch (ch) {
      case ';':
      case '\n':
          cin >> WS;    // пропустить пропуск
          return curr_tok=PRINT;
      case '*':
      case '/':
      case '+':
```

```
        case '-':
        case '(':
        case ')':
        case '=':
            return curr_tok=ch;
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
        case '.':
            cin.putback(ch);
            cin >> number_value;
            return curr_tok=NUMBER;
        default:            // NAME, NAME= или ошибка
            if (isalpha(ch)) {
                char* p = name_string;
                *p++ = ch;
                while (cin.get(ch) && isalnum(ch)) *p++ = ch;
                cin.putback(ch);
                *p = 0;
                return curr_tok=NAME;
            }
            error("плохая лексема");
            return curr_tok=PRINT;
        }
    }
```

Поскольку token_value (значение лексемы) операции было определено как целое значение этой операции*, обработка всех операций тривиальна.

3.1.3 Таблица имен

К таблице имен доступ осуществляется с помощью одной функции

```
name* look(char* p, int ins =0);
```

Ее второй параметр указывает, нужно ли сначала поместить строку символов в таблицу. Инициализатор =0 задает параметр, кторый надлежит использовать по умолчанию, когда look() вызывается с одним параметром. Это дает удобство записи, когда look("sqrt2") означает look("sqrt2",0), то есть просмотр, без помещения в таблицу. Чтобы получить такое же удобство записи для помещения в таблицу, определяется вторая функция:

```
inline name* insert(char* s) { return look(s,1);}
```

Как уже отмечалось раньше, элементы этой таблицы имеют тип:

```
srtuct name {
    char* string;
    char* next;
    double value;
}
```

Член next используется только для сцепления вместе имен в таблице.

Сама таблица - это просто вектор указателей на объекты типа name:

```
const TBLSZ = 23;
name* table[TBLSZ];
```

Поскольку все статические объекты инициализируются нулем, это тривиальное описание таблицы table гарантирует также надлежащую инициализацию.

Для нахождения элемента в таблице в look() принимается простой алгоритм хэширования (имена с одним и тем же хэш-кодом зацепляются

вместе):

```
  int ii = 0;        // хэширование
  char* pp = p;
  while (*pp) ii = ii<<1 ^ *pp++;
  if (ii < 0) ii = -ii;
  ii %= TBLSZ;
```

То есть, с помощью исключающего ИЛИ каждый символ во входной строке "добавляется" к ii ("сумме" предыдущих символов). Бит в x^y устанавливается единичным тогда и только тогда, когда соответствующие биты в x и y различны. Перед применением в символе исключающего ИЛИ, ii сдвигается на один бит влево, чтобы не

---

* знака этой операции. (прим. перев.)

использовать в слове только один байт. Это можно было написать и так:

```
  ii <<= 1;
  ii ^= *pp++;
```

Кстати, применение ^ лучше и быстрее, чем +. Сдвиг важен для получения приемлемого хэш-кода в обоих случаях. Операторы

```
  if (ii < 0) ii = -ii;
  ii %= TBLSZ;
```

обеспечивают, что ii будет лежать в диапазоне 0...TBLSZ-1; % - это операция взятия по модулю (еще называемая получением остатка).

Вот функция полностью:

```
  extern int strlen(const char*);
  extern int strcmp(const char*, const char*);
  extern int strcpy(const char*, const char*);

  name* look(char* p, int ins =0)
  {
      int ii = 0;        // хэширование
      char* pp = p;
      while (*pp) ii = ii<<1 ^ *pp++;
      if (ii < 0) ii = -ii;
      ii %= TBLSZ;

      for (name* n=table[ii]; n; n=n->next)    // поиск
          if (strcmp(p,n->string) == 0) return n;

      if (ins == 0) error("имя не найдено");

      name* nn = new name;                     // вставка
      nn->string = new char[strlen(p)+1];
      strcpy(nn->string,p);
      nn->value = 1;
      nn->next = table[ii];
      table[ii] = nn;
      return nn;
  }
```

После вычисления хэш-кода ii имя находится простым просмотром через поля next. Проверка каждого name осуществляется с помощью стандартной функции strcmp(). Если строка найдена, возвращается ее name, иначе добавляется новое name.

Добавление нового name включает в себя создание нового объекта в

свободной памяти с помощью операции new (см. #3.2.6), его инициализацию, и добавление его к списку имен. Последнее осуществляется просто путем помещения нового имени в голову списка, поскольку это можно делать даже не проверяя, имеется список, или нет. Символьную строку для имени тоже нужно сохранить в свободной памяти. Функция strlen() используется для определения того, сколько памяти нужно, new - для выделения этой памяти, и strcpy() - для копирования строки в память.

3.1.4 Обработка ошибок

Поскольку программа так проста, обработка ошибок не составляет большого труда. Функция обработки ошибок просто считает ошибки, пишет сообщение об ошибке и возвращает управление обратно:

```
int no_of_errors;

double error(char* s) {
    cerr << "error: " << s << "\n";
    no_of_errors++;
    return 1;
}
```

Возвращается значение потому, что ошибки обычно встречаются в середине вычисления выражения, и поэтому надо либо полностью прекращать вычисление, либо возвращать значение, которое по всей видимости не должно вызвать последующих ошибок. Для простого калькулятора больше подходит последнее. Если бы get_token() отслеживала номера строк, то error() могла бы сообщать пользователю, где приблизительно обнаружена ошибка. Это наверняка было бы полезно, если бы калькулятор использовался неитерактивно.

Часто бывает так, что после появления ошибки программа должна завершиться, поскольку нет никакого разумного пути продолжить работу. Это можно сделать с помощью вызова exit(), которая очищает все вроде потоков вывода (#8.3.2), а затем завершает программу используя свой параметр в качестве ее возвращаемого значения. Более радикальный способ завершения программы - это вызов abort(), которая обрывает выполнение сразу же или сразу после сохранения где-то информации для отладчика (дамп памяти); о подробностях справьтесь, пожалуйста, в вашем руководстве.

3.1.5 Драйвер

Когда все части программы на месте, нам нужен только драйвер для инициализации и всего того, что связано с запуском. В этом простом примере main() может работать так:

```
int main()
{
    // вставить предопределенные имена:
    insert("pi")->value = 3.1415926535897932385;
    insert("e")->value = 2.7182818284590452354;

    while (cin) {
        get_token();
        if (curr_tok == END) break;
        if (curr_tok == PRINT) continue;
        cout << expr() << "\n";
    }

    return no_of_errors;
}
```

Принято обычно, что main() возвращает ноль при нормальном завершении программы и не ноль в противном случае, поэтому это прекрасно может сделать возвращение числа ошибок. В данном случае оказывается, что инициализация нужна только для введения предопределенных имен в таблицу имен.

Основная работа цикла - читать выражения и писать ответ. Это делает строка:

```
cout << expr() << "\n";
```

Проверка cin на каждом проходе цикла обеспечивает завершение программы в случае, если с потоком ввода что-то не так, а проверка на END обеспечивает корректный выход из цикла, когда get_token() встречает конец файла. Оператор break осуществляет выход из ближайшего содержащего его оператора switch или цикла (то есть, оператора for, оператора while или оператора do). Проверка на PRINT (то есть, на '\n' или ';') освобождает expr() от обязанности обрабатывать пустые выражения. Оператор continue равносилен переходу к самому концу цикла, поэтому в данном случае

```
while (cin) {
    // ...
    if (curr_tok == PRINT) continue;
    cout << expr() << "\n";
}
```

эквивалентно

```
while (cin) {
    // ...
    if (curr_tok == PRINT) goto end_of_loop;
    cout << expr() << "\n";
    end_of_loop
}
```

Более подробно циклы описываются в #c.9.

3.1.6 Параметры командной строки

После того, как программа была написана и оттестирована, я заметил, что часто набирать выражения на клавиатуре в стандартный ввод надоедает, поскольку обычно использование программы состоит в вычислении одного выражения. Если бы можно было представлять это выражение как параметр командной строки, не приходилось бы так много нажимать на клавиши.

Как уже говорилось, программа запускается вызовом main(). Когда это происходит, main() получает два параметра: указывающий число параметров, обычно называемый argc, и вектор параметров, обычно называемый argv. Параметры - это символьные строки, поэтому argv имеет тип char*[argc]. Имя программы (так, как оно стоит в командной строке) передается в качестве argv[0], поэтому argc всегда не меньше единицы. Например, в случае команды

```
dc 150/1.1934
```

параметры имеют значения:

```
argc        2
argv[0]        "dc"
argv[1]        "150/1.1934"
```

Научиться пользоваться параметрами командной строки несложно; сложность состоит в том, как использовать их без

перепрограммирования. В данном случае это оказывается совсем просто, поскольку поток ввода можно связать с символьной строкой, а не с файлом (#8.5). Например, можно заставить cin читать символы из стандартного ввода:

```
int main(int argc, char* argv[])
{
    switch(argc) {
    case 1:        // читать из стандартного ввода
        break;
    case 2:        // читать параметр строку
        cin = *new istream(strlen(argv[1]),argv[1]);
        break;
    default:
        error("слишком много параметров");
        return 1;
    }

    // как раньше
}
```

Программа осталась без изменений, за исключением добавления в main() параметров и использования этих параметров в операторе switch. Можно было бы легко модифицировать main() так, чтобы она получала несколько параметров командной строки, но это оказывается ненужным, особенно потому, что несколько выражений можно передавать как один параметр:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

Здесь кавычки необходимы, поскольку ; является разделителем команд в системе UNIX.

3.2 Краткая сводка операций

Операции C++ подробно и систематически описываются в #c.7; прочитайте, пожалуйста, этот раздел. Здесь же приводится краткая сводка и некоторые примеры. После каждой операции приведено одно или более ее общеупотребительных названий и пример ее использования. В этих примерах имя_класса - это имя класса, член - имя члена, объект - выражение, дающее в результате объект класса, указатель - выражение, дающее в результате указатель, выр - выражение, а lvalue - выражение, денотирующее неконстантный объект. Тип может быть совершенно произвольным именем типа (со *, () и т.п.) только когда он стоит в скобках, во всех остальных случаях существуют ограничения.

Унарные операции и операции присваивания правоассоциативны, все остальные левоассоциативны. Это значит, что a=b=c означает a=(b=c), a+b+c означает (a+b)+c, и *p++ означает *(p++), а не (*p)++.

Сводка Операций (часть 1)

::	разрешение области видимости глобальное	имя_класса :: член :: имя
->	выбор члена	указатель->член
[]	индексация	указатель [выр]
()	вызов функции	выр (список_выр)
()	построение значения	тип (список_выр)
sizeof	размер объекта	sizeof выр

sizeof	размер типа	sizeof (тип)
++	приращение после	lvalue++
++	приращение до	++lvalue
--	уменьшение после	lvalue--
--	уменьшение до	--lvalue
~	дополнение	~ выр
!	не	! выр
-	унарный минус	- выр
+	унарный плюс	+ выр
&	адрес объекта	& lvalue
*	разыменование	* выр
new	создание (размещение)	new тип
delete	уничтожение (освобождение)	delete указатель
delete[]	уничтожение вектора	delete[выр]
указатель		
()	приведение (преобразование типа)	(тип) выр
*	умножение	выр * выр
/	деление	выр / выр
%	взятие по модулю (остаток)	выр % выр
+	сложение (плюс)	выр + выр
-	вычитание (минус)	выр - выр

В каждой отчерченной части находятся операции с одинаковым приоритетом. Операция имеет приоритет больше, чем операции из частей, расположенных ниже. Например: a+b*c означает a+(b*c), так как * имеет приоритет выше, чем +, а a+b-c означает (a+b)-c, поскольку + и - имеют одинаковый приоритет (и поскольку + левоассоциативен).

Сводка Операций (часть 2)

<<	сдвиг влево	lvalue << выр
>>	сдвиг вправо	lvalue >> выр
<	меньше	выр < выр
<=	меньше или равно	выр <= выр
>	больше	выр > выр
>=	больше или равно	выр >= выр
==	равно	выр == выр
!=	не равно	выр != выр
&	побитовое И	выр & выр
^	побитовое исключающее ИЛИ	выр ^ выр
|	побитовое включающее ИЛИ	выр | выр
&&	логическое И	выр && выр

```
-----
||      логическое включающее ИЛИ       выр || выр
-------------------------------------------------------------
-----
? :     арифметический if               выр ? выр : выр
-------------------------------------------------------------
-----
=       простое присваивание            lvalue = выр
*=      умножить и присвоить            lvalue = выр
/=      разделить и присвоить           lvalue /= выр
%=      взять по модулю и присвоить     lvalue %= выр
+=      сложить и присвоить             lvalue += выр
-=      вычесть и присвоить             lvalue -= выр
<<=     сдвинуть влево и присвоить      lvalue <<= выр
>>=     сдвинуть вправо и присвоить     lvalue >>= выр
&=      И и присвоить                   lvalue &= выр
|=      включающее ИЛИ и присвоить      lvalue |= выр
^=      исключающее ИЛИ и присвоить     lvalue ^= выр
-------------------------------------------------------------
-----
,       запятая (последование)          выр , выр
```

3.2.1 Круглые скобки

Скобками синтаксис C++ злоупотребляет; количество способов их использования приводит в замешательство: они применяются для заключения в них параметров в вызовах функций, в них заключается тип в преобразовании типа (приведении к типу), в именах типов для обозначения функций, а также для разрешения конфликтов приоритетов. К счастью, последнее требуется не слишком часто, потому что уровни приоритета и правила ассоциативности определены таким образом, чтобы выражения "работали ожидаемым образом" (то есть, отражали наиболее привычный способ употребления). Например, значение

```
if (i<=0 || max
```

3.2.2 Порядок вычисления

Порядок вычисления подвыражений в выражении неопределен. Например

```
int i = 1;
v[i] = i++;
```

может вычисляться или как v[1]=1, или как v[2]=1. При отсутствии ограничений на порядок вычисления выражения может генерироваться более хороший код. Было бы замечательно, если бы компилятор предупреждал о подобных неоднозначностях, но большинство компиляторов этого не делают.

Относительно операций

```
,    &&    ||
```

гарантируется, что их левый операнд вычисляется раньше, чем правый. Например, b=(a=2,a=1) присвоит b 3. В #3.3.1 приводятся примеры использования && и ||. Заметьте, что операция последования , (запятая) логически отличается от запятой, которая используется для разделения параметров в вызове функции. Рассмотрим

```
f1(v[i],i++);       // два параметра
f2( (v[i],i++) )    // один параметр
```

В вызове f1 два параметра, v[i] и i++, и порядок вычисления выражений-параметров неопределен. Зависимость выражения-параметра

от порядка вычисления - это очень плохой стиль, а также непереносимо. В вызове f2 один параметр, выражение с запятой, которое эквивалентно i++.

С помощью скобок нельзя задать порядок вычисления. Например, a*(b/c) может вычисляться и как (a*b)/c, поскольку * и / имеют одинаковый приоритет. В тех случаях, когда важен порядок вычисления, можно вводить дополнительную переменную, например, (t=b/c,a*t).

3.2.2 Увеличение и уменьшение*

Операция ++ используется для явного выражения приращения вместо его неявного выражения с помощью комбинации сложения и присваивания. По определению ++lvalue означает lvalue+=1, что в свою очередь означает lvalue=lvalue+1 при условии, что lvalue не вызывает никаких побочных эффектов. Выражение, обозначающее (денотирующее) объект, который должен быть увеличен, вычисляется один раз (только). Аналогично, уменьшение выражается операцией --. Операции ++ и -- могут применяться и как префиксные, и как постфиксные. Значением ++x является новое (то есть увеличенное) значение x. Например, y=++x эквивалентно y=(x+=1). Значение x++, напротив, есть старое значение x. Например, y=x++ эквивалентно y=(t=x,x+=1,t), где t - переменная того же типа, что и x.

Операции приращения особенно полезны для увеличения и уменьшения переменных в циклах. Например, оканчивающуюся нулем строку можно копировать так:

```
inline void cpy(char* p, const char* q)
{
    while (*p++ = *q++) ;
}
```

---

* Следовало бы переводить как "инкремент" и "декремент", однако мы следовали терминологии, принятой в переводной литературе по C, поскольку эти операции унаследованы от C. (прим. перев.)

Напомню, что увеличение и уменьшение указателей, так же как сложение и вычитание указателей, осуществляется в терминах элементов вектора, на которые указывает указатель; p++ приводит к тому, что p указывает на следующий элемент. Для указателя p типа T* по определению выполняется следующее:

```
long(p+1) == long(p)+sizeof(T);
```

3.2.4 Побитовые логические операции

Побитовые логические операции

```
&    |    ^    ~    >>    <<
```

применяются к целым, то есть к объектам типа char, short, int, long и их unsigned аналогам, результаты тоже целые.

Одно из стандартных применений побитовых логических операций - реализация маленькиого множества (вектора битов). В этом случае каждый бит беззнакового целого представляет один член множества, а число членов ограничено числом битов. Бинарная операция & интерпретируется как пересечение, | как объединение, а ^ как разность. Для именования членов такого множества можно использовать перечисление. Вот маленький пример, заимствованный из реализации (не пользовательского интерфейса) :

```
enum state_value { _good=0, _eof=1, _fail=2, _bad=4};
                  // хорошо, конец файла, ошибка, плохо
```

Определение _good не является необходимым. Я просто хотел, чтобы состояние, когда все в порядке, имело подходящее имя. Состояние потока можно установить заново следующим образом:

```
cout.state = _good;
```

Например, так можно проверить, не был ли испорчен поток или допущена операционная ошибка:

```
if (cout.state&(_bad|_fail)) // не good
```

Еще одни скобки необходимы, поскольку & имеет более высокий приоритет, чем |.

Функция, достигающая конца ввода, может сообщать об этом так:

```
cin.state |= _eof;
```

Операция |= используется потому, что поток уже может быть испорчен (то есть, state==_bad), поэтому

```
cin.state = _eof;
```

то не использующие

ep,

ига

t и

oro

ния

бъекты различных типов

т подробностей реализации.
Например:

```
  int i = 1;
  char* pc = "asdf";
f";
  int* pi = &i;
```

```
i = (int)pc;
```

тесь: значение pc может измениться

ых машинах
// sizeof(int)

.6 Свободная память

атическим, либо автоматическим

ости, часто полезно

после возврата из

здает операция new, а

Про объекты,

ни в свободной

деревьев или

ьшей структуры

нных, размер которой не может быть известен на стадии компиляции.

ссмотрим, как можно было бы написать компилятор в духе

писанного настольного калькулятора. Функции синтаксического

ализа могут строить древовидное представление выражений, которое

дет использоваться при генерации кода. Например:

```
  struct enode {
      token_value oper;
```

```
alue oper;
      enode*
```

□□□□††††□□□□†††□□□□††††□□□□□□□†††□†□

например так:

```
  void generate(enode* n)
* n)
  {
      switch (n->oper) {
```

```
case PLUS:
```

ет нечто соответствующее

```
n;
      }
  }
```

Объект, созданный с помощью new, существует, пока он не будет

данный с помощью new, существует, пока он не будет

ожен delete, после чего пространство, которое он

a

и

яжение new, нет. Операция delete может

телю, который был возвращен операцей new,

ete к нулю не вызывает никаких действий.

создавать вектора объектов. Например:

```
 {
     char* s = new char[strlen(p)+1];
     strcpy(s,p);
     return s;
return s;
 }
```

во, выделенное

мер выделенного
объекта. Например:

```
  int main(int argc, char* argv[])
])
  {
```

```
if (argc < 2) exit(1);
```

```
[1]);
      delete p;
  }
```

Это приводит к тому, что объект, выделенный стандартной реализацией

ый стандартной реализацией

○,

во).

ть размер вектора в операции уничтожения delete. Например:

```
int main(int argc, char* argv[])
```

```
ain(int argc, char* argv[])
  {
```

```
gc < 2) exit(1);
```

```
trlen(argv[1])+1;
```

```
char* p = save_string(argv[1]);
      delete[size] p;
  }
```

Заданный пользователем размер вектора игнорируется за исключением

пользователем размер вектора игнорируется за исключением

типов, определяемых пользователем (#5.5.5).

мяти реализуются функциями (#с.7.2.3):

```
  void operator new(long);
  void operator delete(void*);
 void operator delete(void*);
```

кт.

ия?

жно
происходить. Запрос вроде

```
char* p = new char[100000000];
```

как правило, приводит к каким-то неприятностям. Когда у new ничего

правило, приводит к каким-то неприятностям. Когда у new ничего

получается, она вызывает функцию, указываемую указаетелем

w_handler (указатели на функции обсуждаются в #4.6.9). Вы можете

ать указатель явно или использовать функцию set_new_handler().

ример:

```
  #include

  void out_of_store()
  {
      cerr << "операция new не прошла: за пределами памяти\n";
я new не прошла: за пределами памяти\n";
      exit(1);
  }
```

```
ef void (*PF)();     // тип указатель на функцию
```

```
_handler(PF);

  main()
  {
      set_new_handler(out_of_store);
      char* p = new char[100000000];
```

```
char* p = new char[100000000];
```

```
ng(p) << "\n";
  }
```

как правило, не будеB□ ?□8□A□0□B□L□ " A□4□5□;□0□=□>□" , 0□

L□

, что вы задали свои собственные operator new() и

ся найти

угими

бразом, использование delete необязательным. Но это, конечно, все-

аки задача не для начинающего.

o возвращает указатель 0, если

чество памяти и не был задан
никакой _new_handler. Например

```
  include
lude

  main()
  {
```

```
char* p = new char[100000000];
```

```
< "сделано, p = " << long(p) << "\n";
  }



выдаст

  сделано, p = 0
```

дение! Заметьте, что тот, кто задает

памяти при

заданных типов, определяемых пользователем; см. #5.5.6).

3.3 Сводка операторов

Сводка операторов

лностью изложены в #с.9,

краткая
сводка и некоторые примеры.

таксис оператора

- - - - - - - - - - - - - - -

```
-
   оператор:
       описание
       {список_операторов opt}
       выражение opt
```

выражение opt

```
if ( выражение ) опреатор
```

оператор

оператор

оператор

```
while (выражение)
```

```
opt ) оператор
```

```
ератор
       default : оператор
       break ;
       continue ;
```

;

```
opt ;
```

```
p ;
        идентификатор : оператор

        список_операторов:
```

```
список_операторов:
        оператор
```

_операторов

т операторов

как выражения.

3.3.1 Проверки

Проверка значения может осуществляться или оператором if, или
ожет осуществляться или оператором if, или
оператором switch:

```
if ( выражение ) оператор
  if ( выражение ) оператор else оператор
```

```
switch ( выражение ) оператор
```

> >=

звращают 0. Не

ЛОЖЬ
определена как 0.

няется в

ненулевое, иначе выполняется второй

он задан). Отсюда следует, что в качестве условия

ое целое выражение. В частности, если a

е, то

```
if (a) // ...
```

эквивалентно

```
if (a != 0) // ...
```

Логические операции
ские операции

```
&&    ||    !
```

виях. Операции && и || не будут

ненужно. Например:

```
if (p && 1count) // ...
```

вначале проверяет, является ли p не нулем, и только если это так,
вляется ли p не нулем, и только если это так,
то проверяет 1count.

Некоторые простые операторы if могут быть с удобством заменены выражениями арифметического if. Например:

```
if (a <= d)
```

```
 b;
  else
      max = a;
```

лучше выражается так:
выражается так:

```
  max = (a<=b) ? b : a;
```

обязательны, но я считаю, что когда они

егче читать.

о по-другому записать в

операторов if. Например:

```
switch (val) {
case 1:
    f();
```

```
break;
```

```
se 2;
      g();
      break;
  default:
      h();
      break;
  }
```

иначе можно было бы записать так:

```
  if (val == 1)
```

```
f();
```

```
se if (val == 2)
      g();
  else
      h();
```

Смысл тот же, однако первый вариант (switch) предпочтительнее,

ко первый вариант (switch) предпочтительнее,

случае явно выражается сущность действия

нт). Поэтому в нетривиальных
случаях оператор switch читается легче.

Заботьтесь о том, что switch должен как-то завершаться, если полнялся следующий case. Например:

```
  switch (val) {    // осторожно
val) {    // осторожно
  case 1:
```

```
cout << "case 1\n";
```

;

```
2\n";
```

```
е найден\n";
  }

при val==1 напечатает
val==1 напечатает
```

```
case 1
```

```
e 2
  default: case не найден
```

к великому изумлению непосвященного. Самый обычный способ завершить

```
соб завершить
```

пример:

```
switch (val) {    // осторожно
```

```
{     // осторожно
 case 0:
```

```
ase 0\n";
  case1:
```

```
1:
```

```
< "case 1\n";ue ;

        return выражение opt ;

        goto идентификатор ;
        идентификатор : оператор

        список_операторов:
            оператор
            оператор список_операторов
```

Заметьте, что описание является оператором, и что нет операторов присваивания и вызова процедуры. Присваивание и вызов функции обрабатываются как выражения.

3.3.1 Проверки

Проверка значения может осуществляться или оператором if, или оператором switch:

```
if ( выражение ) оператор
if ( выражение ) оператор else оператор
switch ( выражение ) оператор
```

В C++ нет отдельного булевского типа. Операции сравнения

```
==    !=    <    <=    >    >=
```

возвращают целое 1, если сравнение истинно, иначе возвращают 0. Не так уж непривычно видеть, что ИСТИНА определена как 1, а ЛОЖЬ определена как 0.

В операторе if первый (или единственный) оператор выполняется в том случае, если выражение ненулевое, иначе выполняется второй оператор (если он задан). Отсюда следует, что в качестве условия может использоваться любое целое выражение. В частности, если a целое, то

```
if (a) // ...
```

эквивалентно

```
if (a != 0) // ...
```

Логические операции

```
&&    ||    !
```

наиболее часто используются в условиях. Операции && и || не будут вычислять второй аргумент, если это ненужно. Например:

```
if (p && 1count) // ...
```

вначале проверяет, является ли p не нулем, и только если это так, то проверяет 1count.

Некоторые простые операторы if могут быть с удобством заменены выражениями арифметического if. Например:

```
if (a <= d)
    max = b;
else
    max = a;
```

лучше выражается так:

```
max = (a<=b) ? b : a;
```

Скобки вокруг условия необязательны, но я считаю, что когда они используются, программу легче читать.

Некоторые простые операторы switch можно по-другому записать в виде набора операторов if. Например:

```
switch (val) {
case 1:
    f();
    break;
case 2;
    g();
    break;
default:
    h();
    break;
}
```

иначе можно было бы записать так:

```
if (val == 1)
    f();
else if (val == 2)
    g();
else
    h();
```

Смысл тот же, однако первый вариант (switch) предпочтительнее, поскольку в этом случае явно выражается сущность действия (сопоставление значения с рядом констант). Поэтому в нетривиальных случаях оператор switch читается легче.

Заботьтесь о том, что switch должен как-то завершаться, если только вы не хотите, чтобы выполнялся следующий case. Например:

```
switch (val) {    // осторожно
case 1:
    cout << "case 1\n";
case 2;
    cout << "case 2\n";
default:
    cout << "default: case не найден\n";
}
```

при val==1 напечатает

```
case 1
case 2
default: case не найден
```

к великому изумлению непосвященного. Самый обычный способ завершить случай - это break, иногда можно даже использовать goto. Например:

```
switch (val) {    // осторожно
case 0:
    cout << "case 0\n";
case1:
case 1:
    cout << "case 1\n";
    return;
case 2;
```

```
        cout << "case 2\n";
        goto case1;
    default:
        cout << "default: case не найден\n";
        return;
    }
```

При обращении к нему с val==2 выдаст

```
    case 2
    case 1
```

Заметьте, что метка case не подходит как метка для употребления в операторе goto:

```
    goto case 1;        // синтаксическая ошибка
```

3.3.2 Goto

C++ снабжен имеющим дурную репутацию оператором goto.

```
    goto идентификатор;
    идентификатор : оператор
```

В общем, в программировании высокого уровня он имеет очень мало применений, но он может быть очень полезен, когда C++ программа генерируется программой, а не пишется непосредственно человеком. Например, операторы goto можно использовать в синтаксическом анализаторе, порождаемом генератором синтаксических анализаторов. Оператор goto может быть также важен в тех редких случаях, когда важна наилучшая эффективность, например, во внутреннем цикле какой-нибудь программы, работающей в реальном времени.

Одно из немногих разумных применений состоит в выходе из вложенного цикла или переключателя (break лишь прекращает выполнение самого внутреннего охватывающего его цикла или переключателя). Например:

```
    for (int i = 0; i
```

3.4 Комментарии и Выравнивание

Продуманное использование комментариев и согласованное использование отступов может сделать чтение и понимание программы намного более приятным. Существует несколько различных стилей согласованного использования отступов. Автор не видит никаких серьезных оснований предпочесть один другому (хотя как и у большинства, у меня есть свои предпочтения). Сказанное относится также и к стилю комментариев.

Неправильное использование комментариев может серьезно повлиять на удобочитаемость программы, Компилятор не понимает содержание комментария, поэтому он никаким способом не может убедиться в том, что комментарий

[1] осмыслен;
[2] описывает программу; и
[3] не устарел.

Непонятные, двусмысленные и просто неправильные комментарии содержатся в большинстве программ. Плохой комментарий может быть хуже, чем никакой.

Если что-то можно сформулировать средствами самого языка, следует это сделать, а не просто отметить в комментарии. Данное замечание относится к комментариям вроде:

```
    // переменная "v" должна быть инициализирована.

    // переменная "v" должна использоваться только функцией "f()".

    // вызвать функцию init() перед вызовом
```

```
// любой другой функции в этом файле.

// вызовите функцию очиститки "cleanup()" в конце вашей
программы.

// не используйте функцию "wierd()".

// функция "f()" получает два параметра.
```

При правильном использовании C++ подобные комментарии как правило становятся ненужными. Чтобы предыдущие комментарии стали излишними, можно, например, использовать правила компоновки (#4.2) и видимость, инициализацию и правила очистки для классов (см. #5.5.2).

Если что-то было ясно сформулировано на языке, второй раз упоминать это в комментарии не следует. Например:

```
a = b+c;     // a становится b+c
count++;     // увеличить счетчик
```

Такие комментарии хуже чем просто излишни, они увеличивают объем текса, который надо прочитать, они часто затуманивают структуру программы, и они могут быть неправильными.

Автор предпочитает:

[1] Комментарий для каждого исходного файла, сообщающий, для чего в целом предназначены находящиеся в нем комментарии, дающий ссылки на справочники и руководства, общие рекомендации по использованию и т.д.;
[2] Комментарий для каждой нетривиальной функции, в котором сформулировано ее назначение, используемый алгоритм (если он неочевиден) и, быть может, что-то о принимаемых в ней предположениях относительно среды выполнения;
[3] Небольшое число комментариев в тех местах, где программа неочевидна и/или непереносима; и
[4] Очень мало что еще.

Например:

```
// tbl.c: Реализация таблицы имен
/*
    Гауссовское исключение с частичным
    См. Ralston: "A first course ..." стр. 411.
*/

// swap() предполагает размещение стека AT&T sB20.

/************************************

    Copyright (c) 1984 AT&T, Inc.
    All rights reserved

************************************/
```

Удачно подобранные и хорошо написанные комментарии - существенная часть программы. Написание хороших комментариев может быть столь же сложным, сколь и написание самой программы.

Заметьте также, что если в функции используются исключительно комментарии //, то любую часть этой функции можно закомментировать с помощью комментариев /* */, и наоборот.

3.5 Упражнения

1. (*1) Перепишите следующий оператор for в виде эквивалентного оператора while:

```
for (i=0; im
*p.m
*a[i]
```

7. (*2) Напишите функции: strlen(), которая возвращает длину строки, strcpy(), которая копирует одну строку в другую, и strcmp(), которая сравнивает две строки. Разберитесь, какие должны быть типы параметров и типы возвращаемых значений, а потом сравните их со стандартными версиями, которые описаны в и в вашем руководстве.
8. (*1) Посмотрите, как ваш компилятор реагирует на ошибки:

```
a := b+1;
if (a = 3) // ...
if (a&077 == 0) // ...
```

   Придумайте ошибки попроще, и посмотрите, как компилятор на них реагирует.
9. (*2) Напишите функцию cat(), получающую два строковых параметра и возвращающую строку, которая является конкатенацией параметров. Используйте new, чтобы найти память для результата. Напишите функцию rev(), которая получает строку и переставляет в ней символы в обратном порядке. То есть, после вызова rev(p) последний символ p становится первым.
10. (*2) Что делает следующая программа?

```
void send(register* to, register* from, register count)
// Полезные комментарии несомненно уничтожены.
{
    register n=(count+7)/8;
    switch (count%8) {
        case 0: do {    *to++ = *from++;
        case 7: do {    *to++ = *from++;
        case 6: do {    *to++ = *from++;
        case 5: do {    *to++ = *from++;
        case 4: do {    *to++ = *from++;
        case 3: do {    *to++ = *from++;
        case 2: do {    *to++ = *from++;
        case 1: do {    *to++ = *from++;
            while (--n>0);
    }
}
```

   Зачем кто-то мог написать нечто похожее?
11. (*2) Напишите функцию atoi(), которая получает строку, содержащую цифры, и возвращает соответствующее int. Например, atoi("123") - это 123. Модифицируйте atoi() так, чтобы помимо обычной десятичной она обрабатывала еще восьмеричную и шестнадцатиричную записи C++. Модифицируйте atoi() так, чтобы обрабатывать запись символьной константы. Напишите функцию itoa(), которая строит представление целого параметра в виде строки.
12. (*2) Перепишите get_token() (#3.1.2), чтобы она за один раз читала строку в буфер, а затем составляла лексемы, читая символы из буфера.
13. (*2) Добавьте в настрольный калькулятор из #3.1 такие функции, как sqrt(), log() и sin(). Подсказка: предопределите имена и вызывайте функции с помощью вектора указателей на функции. Не забывайте проверять параметры в вызове функции.
14. (*3) Дайте пользователю возможность определять функции в настольном калькуляторе. Подсказка: определяйте функции как

последовательность действий, прямо так, как их набрал пользователь. Такую последовательность можно хранить или как символьную строку, или как список лексем. После этого, когда функция вызывается, читайте и выполняйте эти действия. Если вы хотите, чтобы пользовательская функция получала параметры, вы должны придумать форму записи этого.

15. (*1.5) Преобразуйте настольный калькулятор так, чтобы вместо статических переменных name_string и number_value использовалась структура символа symbol:

```
struct symbol {
    token_value tok;
    union {
        double number_value;
        char* name_string;
    };
};
```

16. (*2.5) Напишите программу, которая выбрасывает комментарии из C++ программы. То есть, читает из cin, удаляет // и /* */ комментарии и пишет результат в cout. Не заботьтесь о приятном виде выходного текста (это могло бы быть другим, более сложным упражнением). Не беспокойтесь о правильности программ. Остерегайтесь // и /* и */ внутри комментариев, строк и символьных констант.
17. (*2) Посмотрите какие-нибудь программы, чтобы понять принцип различных стилей комментирования и выравнивания, которые используются на практике.

Глава 4

Функции и Файлы

Итерация свойственна человеку,
рекусия божественна.
- Л. Питер Дойч

Все нетривиальные программы собираются из нескольких раздельно компилируемых единиц (их принято называть просто файлами). В этой главе описано, как раздельно откомпилированные функции могут обращаться друг к другу, как такие функции могут совместно пользоваться данными (разделять данные), и как можно обеспечить согласованность типов, которые используются в разных файлах программы. Функции обсуждаются довольно подробно. Сюда входят передача параметров, параметры по умолчанию, перегрузка имен функций, и, конечно же, описание и определение функций. В конце описываются макросы.

4.1 Введение

Иметь всю программу в одном файле обычно невозможно, поскольку коды стандартных библиотек и операционной системы находятся где-то в другом месте. Кроме того, хранить весь текст пользовательской программы в одном файле как правило непрактично и неудобно. Способ организации программы в файлы может помочь читающему охватить всю структуру программы, а также может дать возможность компилятору реализовать эту структуру. Поскольку единицей компиляции является файл, то во всех случаях, когда в файл вносится изменение (сколь бы мало оно ни было), весь файл нужно компилировать заново. Даже для программы умеренных размеров время, затрачиваемое на перекомпиляцию, можно значительно снизить с помощью разбиения программы на файлы подходящих размеров.

Рассмотрим пример с калькулятором. Он был представлен в виде одного исходного файла. Если вы его набили, то у вас наверняка были небольшие трудности с расположением описаний в правильном порядке, и пришлось использовать по меньшей мере одно "фальшивое" описание, чтобы компилятор смог обработать взаимно рекурсивные функции expr(), term() и prim(). В тексте уже отмечалось, что программа состоит из четырех частей (лексического анализатора, программы синтаксического разбора, таблицы имен и драйвера), но это никак не было отражено в тексте самой программы. По сути дела, калькулятор был написан по-другому. Так это не делается; даже если в этой программе "на выброс" пренебречь всеми соображениями методологии программирования, эксплуатации и эффективности компиляции, автор все равно разобьет эту программу в 200 строк на несколько файлов, чтобы програмировать было приятнее.

Программа, состоящая из нескольких раздельно компилируемых файлов, должна быть согласованной в смысле использования имен и типов, точно так же, как и программа, состоящая из одного исходного файла. В принципе, это может обеспечить и компоновщик*. Компоновщик - это программа, стыкующая отдельно скомпилированные части вместе.

---

* или линкер. (прим. перев.)

Компоновщик часто (путая) называют загрузчиком. В UNIX'е компоновщик называется ld. Однако компоновщики, имеющиеся в большинстве систем, обеспечивают очень слабую поддержку проверки согласованности.

Программист может скомпенсировать недостаток поддержки со стороны компоновщика, предоставив дополнительную информацию о типах (описания). После этого согласованность программы обеспечивается проверкой согласованности описаний, которые находятся в отдельно компилируемых частях. Средства, которые это обеспечивают, в вашей системе будут. C++ разработан так, чтобы способствовать такой явной компоновке**.

4.2 Компоновка

Если не указано иное, то имя, не являющееся локальным для функции или класса, в каждой части программы, компилируемой отдельно, должно относиться к одному и тому же типу, значению, функции или объекту. То есть, в программе может быть только один нелокальный тип, значение, функция или объект с этим именем. Рассмотрим, например, два файла:

```
// file1.c:
    int a = 1;
    int f() { /* что-то делает */ }

// file2.c:
    extern int a;
    int f();
    void g() { a = f(); }
```

a и f(), используемые g() в файле file2.c,- те же, что определены в файле file1.c. Ключевое слово extern (внешнее) указывает, что описание a в file2.c является (только) описанием, а не определением. Если бы a инициализировалось, extern было бы просто проигнорировано, поскольку описание с инициализацией всегда является определением. Объект в программе должен определяться только один раз. Описываться он может много раз, но типы должны точно согласовываться. Например:

```
// file1.c:
    int a = 1;
    int b = 1;
```

```
    extern int c;

// file2.c:
    int a;
    extern double b;
    extern int c;
```

** C разработан так, чтобы в большинстве случаев позволять осуществлять неявную компоновку. Применение C, однако, возросло неимоверно, поэтому случаи, когда можно использовать неявную линковку, сейчас составляют незначительное меньшинство. (прим. автора)

Здесь три ошибки: a определено дважды (int a; является определением, которое означает int a=0;), b описано дважды с разными типами, а c описано дважды, но не определено. Эти виды ошибок (ошибки компоновки) не могут быть обнаружены компилятором, который за один раз видит только один файл. Компоновщик, однако, их обнаруживает.

Следующая программа не является C++ программой (хотя C программой является):

```
// file1.c:
    int a;
    int f() { return a; }

// file2.c:
    int a;
    int g() { return f(); }
```

Во-первых, file2.c не C++, потому что f() не была описана, и поэтому компилятор будет недоволен. Во-вторых, (когда file2.c фиксирован) программа не будет скомпонована, поскольку a определно дважды.

Имя можно сделать локальным в файле, описав его static. Например:

```
// file1.c:
    static int a = 6;
    static int f() { /* ... */ }

// file2.c:
    static int a = 7;
    static int f() { /* ... */ }
```

Поскольку каждое a и f описано как static, получающаяся в результате программа является правильной. В каждом файле своя a и своя f().

Когда переменные и функции явно описаны как static, часть программы легче понять (вам не надо никуда больше заглядывать). Использование static для функций может, помимо этого, выгодно влиять на расходы по вызову функции, поскольку дает оптимизирующему компилятору более простую работу.

Рассмотрим два файла:

```
// file1.c:
    const int a = 6;
    inline int f() { /* ... */ }
    struct s { int a,b; }

// file1.c:
    const int a = 7;
    inline int f() { /* ... */ }
    struct s { int a,b; }
```

Раз правило "ровно одно определение" применяется к константам, inline-функциям и определениям функций так же, как оно применяется к функциям и переменным, то file1.c и file2.c не могут быть частями одной C++ программы. Но если это так, то как же два файла могут использовать одни и те же типы и константы? Коротко, ответ таков:

типы, костанты и т.п. могут определяться столько раз, сколько нужно, при условии, что они определяются одинаково. Полный ответ несколько более сложен (это объясняется в следующем разделе).

4.3 Заголовочные Файлы

Типы во всех описаниях одного и того же объекта должны быть согласованными. Один из способов это достичь мог бы состоять в обеспечении средств проверки типов в компоновщике, но большинство компоновщиков - образца 1950-х, и их нельзя изменить по практическим соображениям*. Другой подход состоит в обеспечении того, что исходный текст, как он передается на рассмотрение компилятору, или согласован, или содержит информацию, которая позволяет компилятору обнаружить несогласованности. Один несовершенный, но простой способ достичь согласованности состоит во включении заголовочных файлов, содержащих интерфейсную информацию, в исходные файлы, в которых содержится исполняемый код и/или определения данных.

Механизм включения с помощью #include - это чрезвычайно простое средство обработки текста для сборки кусков исходной программы в одну единицу (файл) для ее компиляции. Директива

```
#include "to_be_included"
```

замещает строку, в которой встретилось #include, содержимым файла "to_be_included". Его содержимым должен быть исходный текст на C++, поскольку дальше его будет читать компилятор. Часто включение обрабатывается отдельной программой, называемой C препроцессором, которую CC вызывает для преобразования исходного файла, который дал программист, в файл без директив включения перед тем, как начать собственно компиляцию. В другом варианте эти директивы обрабатывает интерфейсная система компилятора по мере того, как они встречаются в исходном тексте. Если программист хочет посмотреть на результат директив включения, можно воспользоваться командой

```
CC -E file.c
```

для препроцессирования файла file.c точно также, как это сделала бы CC перед запуском собственно компилятора. Для включения файлов из стандартной директории включения вместо кавычек используются угловые скобки < и >. Например:

```
#include       // из стандартной директории включения
#define "myheader.h"     // из текущей директории
```

Использование <> имеет то преимущество, что в программу фактическое имя директории включения не встраивается (как правило, сначала просматривается /usr/include/CC, а потом usr/include). К сожалению, пробелы в директиве include существенны:

```
#include < stream.h >    // не найдет
```

---

* Легко изменить один компоновщик, но сделав это и написав программу, которая зависит от усовершенствований, как вы будете переносить эту программу в другое место? (прим. автора)

Может показаться, что перекомпилировать файл заново каждый раз, когда он куда-либо включается, расточительно, но время компиляции такого файла обычно слабо отличается от времени, которое необходимо для чтения его некоторой заранее откомпилированной формы. Причина в том, что текст программы является довольно компактным представлением программы, и в том, что включаемые файлы обычно содержат только описания и не содержат программ, требующих от компилятора значительного анализа.

Следующее эмпирическое правило относительно того, что следует, а что не следует помещать в заголовочные файлы, является не требованием языка, а просто предложением по разумному использованию аппарата #include.

В заголовочном файле могут содержаться:

Определения типов	`struct point { int x, y; }`
Описания функций	`extern int strlen(const char*);`
Определения inline-функций	`inline char get() { return *p++; }`
Описания данных	`extern int a;`
Определения констант	`const float pi = 3.141593`
Перечисления	`enum bool { false, true };`
Директивы include	`#include`
Определения макросов	`#define Case break;case`
Комментарии	`/* проверка на конец файла */`

но никогда

Определения обычных функций	`char get() { return *p++; }`
Определения данных	`int a;`
Определения сложных константных объектов	`const tbl[] = { /* ... */ }`

В системе UNIX принято, что заголовочные файлы имеют суффикс (расширение) .h. Файлы, содержащие определение данных или функций, должны иметь суффикс .c. Такие файлы часто называют, соответственно, ".h файлы" и ".c файлы". В #4.7 описываются макросы. Следует заметить, что в C++ макросы гораздо менее полезны, чем в C, поскольку C++ имеет такие языковые конструкции, как const для определения констант и inline для исключения расходов на вызов функции.

Причина того, почему в заголовочных файлах допускается определение простых констант, но не допускается определение сложных константных объектов, прагматическая. В принципе, сложность тут только в том, чтобы сделать допустимым дублирование определений переменных (даже определения функций можно было бы дублировать). Однако для компоновщиков старого образца слишком трудно проверять тождественность нетривиальных констант и убирать ненужные повторы. Кроме того, простые случаи гораздо более обиходны и потому более важны для генерации хорошего кода.

4.3.1 Один Заголовочный Файл

Проще всего решить проблему разбиения программы на несколько файлов поместив функции и определения данных в подходящее число исходных файлов и описав типы, необходимые для их взаимодействия, в одном заголовочном файле, который включается во все остальные

файлы. Для программы калькулятора можно использовать четыре .c файла: lex.c, syn.c, table.c и main.c, и заголовочный файл dc.h, содержащий описания всех имен, которые используются более чем в одном .c файле:

```
// dc.h: общие описания для калькулятора
```

```
enum token_value {
    NAME,    NUMBER,    END,
    PLUS='+',    MINUS='-',    MUL='*',    DIV='/',
    PRINT=';',    ASSIGN='=',    LP='(',    RP=')'
};

extern int no_of_errors;
extern double error(char* s);
extern token_value get_token();
extern token_value curr_tok;
extern double number_value;
extern char name_string[256];

extern double expr();
extern double term();
extern double prim();

struct name {
    char* string;
    name* next;
    double value;
};

extern name* look(char* p, int ins = 0);
inline name* insert(char* s) { return look(s,1); }
```

Если опустить фактический код, то lex.c будет выглядеть примерно так:

```
// lex.c: ввод и лексический анализ

#include "dc.h"
#include

token_value curr_tok;
double number_value;
char name_string[256];

token_value get_token() { /* ... */ }
```

Заметьте, что такое использование заголовочных файлов гарантирует, что каждое описание в заголовочном файле объекта, определенного пользователем, будет в какой-то момент включено в файл, где он определяется. Например, при компиляции lex.c компилятору будет передано:

```
extern token_value get_token();
// ...
token_value get_token() { /* ... */ }
```

Это обеспечивает то, что компилятор обнаружит любую несогласованность в типах, указанных для имени. Например, если бы get_token() была описана как возвращающая token_value, но при этом определена как возвращающая int, компиляция lex.c не прошла бы из-за ошибки несоответствия типов.

Файл syn.c будет выглядеть примерно так:

```
// syn.c: синтаксический анализ и вычисление

#include "dc.h"

double prim() { /* ... */ }
double term() { /* ... */ }
double expr() { /* ... */ }
```

Файл table.c будет выглядеть примерно так:

```
// table.c: таблица имен и просмотр

#include "dc.h"

extern char* strcmp(const char*, const char*);
extern char* strcpy(char*, const char*);
extern int strlen(const char*);

const TBLSZ = 23;
name* table[TBLSZ];

name* look(char* p; int ins) { /* ... */ }
```

Заметьте, что table.c сам описывает стандартные функции для работы со строками, поэтому никакой проверки согласованности этих описаний нет. Почти всегда лучше включать заголовочный файл, чем описывать имя в .c файле как extern. При этом может включаться "слишком много", но это обычно не оказывает серьезного влияния на время, необходимое для компиляции, и как правило экономит время программиста. В качестве примера этого, обратите внимание на то, как strlen() заново описывается в main() (ниже). Это лишние нажатия клавиш и возможный источник неприятностей, поскольку компилятор не может проверить согласованность этих двух определений. На самом деле, этой сложности можно было бы избежать, будь все описания extern помещены в dc.h, как и предлагалось сделать. Эта "небрежность" сохранена в программе, поскольку это очень типично для C программ, очень соблазнительно для программиста, и чаще приводит, чем не приводит, к ошибкам, которые трудно обнаружить, и к программам, с которыми тяжело работать. Вас предупредили!

И main.c, наконец, выглядит так:

```
// main.c: инициализация, главный цикл и обработка ошибок

#include "dc.h"

int no_of_errors;

double error(char* s) { /* ... */ }

extern int strlen(const char*);

main(int argc, char* argv[]) { /* ... */ }
```

Важный случай, когда размер заголовочных файлов становится серьезной помехой. Набор заголовочных файлов и библиотеку можно использовать для расширения языка множеством обще- и специально-прикладных типов (см. Главы 5-8). В таких случаях не принято осуществлять чтение тысяч строк заголовочных файлов в начале каждой компиляции. Содержание этих файлов обычно "заморожено" и изменяется очень нечасто. Наиболее полезным может оказаться метод затравки компилятора содержанием этих заголовочных фалов. По сути, создается язык специльного назначения со своим собственным компилятором. Никакого стандартного метода создания такого компилятора с затравкой не принято.

4.3.2 Множественные Заголовочные Файлы

Стиль разбиения программы с одним заголовочным файлом наиболее пригоден в тех случаях, когда программа невелика и ее части не предполагается использовать отдельно. Поэтому то, что невозможно установить, какие описания зачем помещены в заголовочный файл,

несущественно. Помочь могут комментарии. Другой способ - сделать так, чтобы каждая часть программы имела свой заголовочный файл, в котором определяются предоставляемые этой частью средства. Тогда каждый .c файл имеет соответствующий .h файл, и каждый .c файл включает свой собственный (специфицирующий то, что в нем задается) .h файл и, возможно, некоторые другие .h файлы (специфицирущие то, что ему нужно).

Рассматривая организацию калькулятора, мы замечаем, что error() используется почти каждой функцией программы, а сама использует только . Это обычная для функции ошибок ситуация, поэтому error() следует отделить от main():

```
// error.h: обработка ошибок

extern int no_errors;

extern double error(char* s);

// error.c

#include
#include "error.h"

int no_of_errors;

double error(char* s) { /* ... */ }
```

При таком стиле использования заголовочных файлов .h файл и связанный с ним .c файл можно рассматривать как модуль, в котором .h файл задает интерфейс, а .c файл задает реализацию.

Таблица символов не зависит от остальной части калькулятора за исключением использования функции ошибок. Это можно сделать явным:

```
// table.h: описания таблицы имен

struct name {
    char* string;
    name* next;
    double value;
};

extern name* look(char* p, int ins = 0);
inline name* insert(char* s) { return look(s,1); }

// table.c: определения таблицы имен

#include "error.h"
#include
#include "table.h"

const TBLSZ = 23;
name* table[TBLSZ];

name* look(char* p; int ins) { /* ... */ }
```

Заметьте, что описания функций работы со строками теперь включаются из . Это исключает еще один возможный источник ошибок.

```
// lex.h: описания для ввода и лексического анализа

enum token_value {
```

```
    NAME,        NUMBER,        END,
    PLUS='+',    MINUS='-',     MUL='*',    DIV='/',
    PRINT=';',   ASSIGN='=',    LP='(',     RP=')'
};

extern token_value curr_tok;
extern double number_value;
extern char name_string[256];

extern token_value get_token();
```

Этот интерфейс лексического анализатора достаточно беспорядочен. Недостаток в надлежащем типе лексемы обнаруживает себя в необходимости давать пользователю get_token() фактические лексические буферы number_value и name_string.

```
// lex.c: определения для ввода и лексического анализа

#include
#include
#include "error.h"
#include "lex.h"

token_value curr_tok;
double number_value;
char name_string[256];

token_value get_token() { /* ... */ }
```

Интефейс синтаксического анализатора совершенно прозрачен:

```
// syn.c: описания для синтаксического анализа и вычисления

extern double expr();
extern double term();
extern double prim();
```

```
// syn.c: определения для синтаксического анализа и вычисления

#include "error.h"
#include "lex.h"
#include "syn.h"

double prim() { /* ... */ }
double term() { /* ... */ }
double expr() { /* ... */ }
```

Главная программа, как всегда, тривиальна:

```
// main.c: главная программа

#include
#include "error.h"
#include "lex.h"
#include "syn.h"
#include "table.h"
#include

main(int argc, char* argv[]) { /* ... */ }
```

Сколько заголовочных файлов использовать в программе, зависит от многих факторов. Многие из этих факторов сильнее связаны с тем, как ваша система работает с заголовочными файлами, нежели с C++. Например, если в вашем редакторе нет средств, позволяющих

одновременно видеть несколько файлов, использование большого числа файлов становится менее привлекательным. Аналогично, если открывание и чтение 10 файлов по 50 строк в каждом требует заметно больше времени, чем чтение одного файла в 500 строк, вы можете дважды подумать, прежде чем использовать в небольшом проекте стиль множественных заголовочных файлов. Слово предостережения: набор из десяти заголовочных файлов плюс стандартные заголовочные файлы обычно легче поддаются управлению. С другой стороны, если вы разбили описания в большой программе на логически минимальные по размеру заголовочные файлы (помещая каждое описание структуры в свой отдельный файл и т.д.), у вас легко может получиться неразбериха из сотен файлов.

4.3.3 Скрытие Данных

Используя заголовочные файлы пользователь может определять явный интерфейс, чтобы обеспечить согласованное использование типов в программе. С другой стороны, пользователь может обойти интерфейс, задаваемый заголовочным файлом, вводя в .c файлы описания extern.

Заметьте, что такой стиль компоновки не рекомендуется:

```
// file1.c:                 // "extern" не используется
    int a = 7;
    const c = 8;
    void f(long) { /* ... */ }

// file2.c:                 // "extern" в .c файле
    extern int a;
    extern const c;
    extern f(int);
    int g() { return f(a+c); }
```

Поскольку описания extern в file2.c не включаются вместе с определениями в файле file1.c, компилятор не может проверить согласованность этой программы. Следовательно, если только загрузчик не окажется гораздо сообразительнее среднего, две ошибки в этой программе останутся, и их придется искать программисту.

Пользователь может защитить файл от такой недисциплинированной компоновки, описав имена, которые не предназначены для общего пользования, как static, чтобы их областью видимости был файл, и они были скрыты от остальных частей программы. Например:

```
// table.c: определения таблицы имен

#include "error.h"
#include
#include "table.h"

const TBLSZ = 23;
static name* table[TBLSZ];

name* look(char* p; int ins) { /* ... */ }
```

Это гарантирует, что любой доступ к table действительно будет осуществляться именно через look(). "Прятать" константу TBLSZ не обязательно.

4.4 Файлы как Модули

В предыдущем разделе .c и .h файлы вместе определяли часть программы. Файл .h является интерфейсом, который используют другие части программы; .c файл задает реализацию. Такой объект часто называют модулем. Доступными делаются только те имена, которые необходимо знать пользователю, остальные скрыты. Это качество часто называют скрытием данных, хотя данные - лишь часть того, что может быть скрыто. Модули такого вида обеспечивают большую гибкость.

Например, реализация может состоять из одного или более .c файлов, и в виде .h файлов может быть предоставлено несколько интерфейсов. Информация, которую пользователю знать не обязательно, искусно скрыта в .c файлах. Если важно, что пользователь не должен точно знать, что содержится в .c файлах, не надо делать их доступными в исходом виде. Достаточно эквивалентных им выходных файлов компилятора (.o файлов).

Иногда возникает сложность, состоящая в том, что подобная гибкость достигается без формальной структуры. Сам язык не распознает такой модуль как объект, и у компилятора нет возможности отличить .h файлы, определяющие имена, которые должны использовать другие модули (экспортируемые), от .h файлов, которые описывают имена из других модулей (импортируемые).

В других случаях может возникнуть та проблема, что модуль определяет множество объектов, а не новый тип. Например, модуль table определяет одну таблицу, и если вам нужно две таблицы, то нет простого способа задать вторую таблицу с помощью понятия модуля. Решение этой проблемы приводится в Главе 5.

Каждый статически размещенный объект по умолчанию инициализируется нулем, программист может задать другие (константные) значения. Это только самый примитивный вид инициализации. К счастью, с помощью классов можно задать код, который выполняется для инициализации перед тем, как модуль каким-либо образом используется, и/или код, который запускается для очистки после последнего использования модуля; см. #5.5.2.

4.5 Как Создать Библиотеку

Фразы типа "помещен в библиотеку" и "ищется в какой-то библиотеке" используются часто (и в этой книге, и в других), но что это означает для C++ программы? К сожалению, ответ зависит от того, какая операционная система используется; в этом разделе объясняется, как создать библиотеку в 8-ой версии системы UNIX. Другие системы предоставляют аналогичные возможности.

Библиотека в своей основе является множеством .o файлов, полученных в результате компиляции соответствующего множества .c файлов. Обычно имеется один или более .h файлов, в которых содержатся описания для использования этих .o файлов. В качестве примера рассмотрим случай, когда нам надо задать (обычным способом) набор математических функций для некоторого неопределенного множества пользователей. Заголовочный файл мог бы выглядеть примерно так:

```
extern double sqrt(double);        // подмножество
extern double sin(double);
extern double cos(double);
extern double exp(double);
extern double log(double);
```

а определения этих функций хранились бы, соответственно, в файлах sqrt.c, sin.c, cos.c, exp.c и log.c.

Библиотеку с именем math.h можно создать, например, так:

```
$ CC -c sqrt.c sin.c cos.c exp.c log.c
$ ar cr math.a sqrt.o sin.o cos.o exp.o log.o
$ ranlib math.a
```

Вначале исходные файлы компилируются в эквивалентные им объектные файлы. Затем используется команда ar, чтобы создать архив с именем math.a. И, наконец, этот архив индексируется для ускорения доступа. Если в вашей системе нет команды runlib, значит она вам, вероятно, не понадобится. Подробности посмотрите, пожалуйста, в вашем руководстве в разделе под заголовком ar. Использовать библиотеку можно, например, так:

```
$ CC myprog.c math.a
```

Теперь разберемся, в чем же преимущества использования math.a перед просто непосредственным использованием .o файлов? Например:

```
$ CC myprog.c sqrt.o sin.o cos.o exp.o log.o
```

Для большинства программ определить правильный набор .o файлов, несомненно, непросто. В приведенном выше примере они включались все, но если функции в myprog.c вызывают только функции sqrt() и cos(), то кажется, что будет достаточно

```
$ CC myprog.c sqrt.o cos.o
```

Но это не так, поскольку cos.c использует sin.c.

Компоновщик, вызываемый командой CC для обработки .a файла (в данном случае, файла math.a) знает, как из того множества, которое

использовалось для создания .a файла, извлечь только необходимые .o файлы.

Другими словами, используя библиотеку можно включать много определений с помощью одного имени (включения определений функций и переменных, используемых внутренними функциями, никогда не видны пользователю), и, кроме того, обеспечить, что в результате в программу будет включено минимальное количество определений.

4.6 Функции

Обычный способ сделать что-либо в C++ программе - это вызвать функцию, которая это делает. Определение функции является способом задать то, как должно делаться некоторое действие. Фукнция не может быть вызвана, пока она не описана.

4.6.1 Описания Функций

Описание функции задает имя функции, тип возвращаемого функцией значения (если таковое есть) и число и типы параметров, которые должны быть в вызове функции. Например:

```
extern double sqrt(double);
extern elem* next_elem();
extern char* strcpy(char* to, const char* from);
extern void exit(int);
```

Семантика передачи параметров идентична семантике инициализации. Проверяются типы параметров, и когда нужно производится неявное преобразование типа. Например, если были заданы предыдущие определения, то

```
double sr2 = sqrt(2);
```

будет правильно обращаться к функции sqrt() со значением с плавающей точкой 2.0. Значение такой проверки типа и преобразования типа огромно.

Описание функции может содержать имена параметров. Это может помочь читателю, но компилятор эти имена просто игнорирует.

4.6.2 Определения Функций

Каждая функция, вызываемая в программе, должна быть где-то определена (только один раз). Определение функции - это описание функции, в котором приводится тело функции. Например:

```
extern void swap(int*, int*);    // описание

void swap(int*, int*)            // определение
{
    int t = *p;
    *p =*q;
    *q = t;
}
```

Чтобы избежать расходов на вызов функции, функцию можно описать как inline (#1.12), а чтобы обеспечить более быстрый доступ к параметрам, их можно описать как register (#2.3.11). Оба средства могут использоваться неправильно, и их следует избегать везде где есть какие-либо сомнения в их полезности.

4.6.3 Передача Параметров

Когда вызывается функция, дополнительно выделяется память под ее формальные параметры, и каждый формальный параметр инициализируется соответствующим ему фактическим параметром. Семантика передачи параметров идентична семантике инициализации. В частности, тип фактического параметра сопоставляется с типом формального параметра, и выполняются все стандартные и определеннные пользователем преобразования типов. Есть особые правила для передачи векторов (#4.6.5), средство передавать параметр без проверки (#4.6.8) и средство для задания параметров по умолчанию (#4.6.6). Рассмотрим

```
void f(int val, int& ref)
{
    val++;
    ref++;
}
```

Когда вызывается f(), val++ увеличивает локальную копию первого фактического параметра, тогда как ref++ увеличивает второй фактический параметр. Например:

```
int i = 1;
int j = 1;
f(i,j);
```

увеличивает j, но не i. Первый параметр, i, передается по значению, второй параметр, j, передается по ссылке. Как уже отмечалось в #2.3.10, использование функций, которые изменяют переданные по ссылке параметры, могут сделать программу трудно читаемой, и их следует избегать (но см. #6.5 и #8.4). Однако передача большого объекта по ссылке может быть гораздо эффективнее, чем передача его по значению. В этом случае параметр можно описать как const, чтобы указать, что ссылка применяется по соображениям эффективности, а также чтобы не позволить вызываемой функции изменять значение объекта:

```
void f(const large& arg)
{
    // значение "arg" не может быть изменено
}
```

Аналогично, описание параметра указателя как const сообщает читателю, что значение объекта, указываемого указателем, функцией не изменяется. Например:

```
extern int strlen(const char*);        // из
extern char* strcpy(char* to, const char* from);
extern int strcmp(const char*, const char*);
```

Важность такой практики растет с размером программы.

Заметьте, что семантика передачи параметров отлична от семантики присваивания. Это важно для const параметров, ссылочных параметров и параметров некоторых типов, определяемых пользователем (#6.6).

4.6.4 Возврат Значения

Из функции, которая не описана как void, можно (и должно) возвращать значение. Возвращаемое значение задается опреатором return. Например:

```
int fac(int n) {return (n>1) ? n*fac(n-1) : 1; }
```

В функции может быть больше одного оператора return:

```
int fac(int n)
{
    if (n > 1)
        return n*fac(n-1);
    else
        return 1;
}
```

Как и семантика передачи параметров, семантика возврата функцией значения идентична семантике инициализации. Возвращаемое значение рассматривается как инициализатор переменной возвращаемого типа. Тип возвращаемого выражения проверяется на согласованность с возвращаемым типом и выполняются все стандартные и определеннные пользователем преобразования типов. Например:

```
double f()
{
    // ...
    return 1;    // неявно преобразуется к double(1)
}
```

Каждый раз, когда вызывается функция, создается новая копия ее параметров и автоматических переменных. После возврата из функции память используется заново, поэтому возвращать указатель на локальную переменную неразумно. Содержание указываемого места изменится непредсказуемо:

```
int* f() {
    int local = 1;
    // ...
    return &local;            // так не делайте
}
```

Эта ошибка менее обычна, чем эквивалентная ошибка при использовании ссылок:

```
int& f() {
    int local = 1;
    // ...
    return local;            // так не делайте
}
```

К счастью, о таких возвращаемых значениях предупреждает компилятор. Вот другой пример:

```
int& f() { return 1;}        // так не делайте
```

4.6.5 Векторные Параметры

Если в качестве параметра функции используется вектор, то передается указатель на его первый элемент. Например:

```
int strlen(const char*);

void f()
{
    char v[] = "a vector"
    strlen(v);
    strlen("Nicholas");
};
```

Иначе говоря, при передаче как параметр параметр типа T[] преобразуется к T*. Следовательно, присваивание элементу векторного параметра изменяет значение элемента вектора, который является параметром. Другими словами, вектор отличается от всех остальных типов тем, что вектор не передается (и не может передаваться) по значению.

Размер вектора недоступен вызываемой функции. Это может быть неудобно, но эту сложность можно обойти несколькими способами. Строки оканчиваются нулем, поэтому их размер можно легко вычислить. Для других векторов можно передавать второй параметр, который задает размер, или определить тип, содержащий указатель и индикатор длины, и передавать его вместо просто вектора (см. также #1.11). Например:

```
void compute1(int* vec_ptr, int vec_size);    // один способ

struct vec {                                  // другой способ
    int* ptr;
    int size;
};

void compute2(vec v);
```

С многомерными массивами все хитрее, но часто можно вместо них использовать векторы указателей, которые не требуют специального рассмотрения. Например:

```
char* day[] = {
    "mon", "tue", "wed", "thu", "fri", "sat", "sun"
};
```

С другой стороны, рассмотрим определение функции, которая работает с двумерными матрицами. Если размерность известна на стадии компиляции, то никаких проблем нет:

```
void print_m34(int m[3][4])
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<4; j++)
            cout << " " << m[i][j];
        cout << "\n";
    }
}
```

Матрица, конечно, все равно передается как указатель, а размерности используются просто для удобства записи.

Первая размерность массива не имеет отношения к задаче отыскания

положения элемента (#2.3.6). Поэтому ее можно передавать как параметр:

```
void print_mi4(int m[][4], int dim1)
{
    for (int i = 0; i
```

4.6.6 Параметры по Умолчанию

Часто в самом общем случае функции требуется больше параметров, чем в самом простом и более употребительном случае. Например, в библиотеке потоков есть функция hex(), порождающая строку с шестнадцатиричным представлением целого. Второй параметр используется для задания числа символов для представления первого параметра. Если число символов слишком мало для представления целого, происходит усечение, если оно слишком велико, то строка дополняется пробелами. Часто программист не заботится о числе символов, необходимых для представления целого, поскольку символов достаточно. Поэтому для нуля в качестве второго параметра определено значение "использовать столько символов, сколько нужно". Чтобы избежать засорения программы вызовами вроде hex(i,0), функция описывается так:

```
extern char* hex(long, int =0);
```

Инициализатор второго параметра является параметром по умолчанию. То есть, если в вызове дан только один параметр, в качестве второго используется параметр по умолчанию. Например:

```
cout << "**" << hex(31) << hex(32,3) << "**";
```

интерпретируется как

```
cout << "**" << hex(31,0) << hex(32,3) << "**";
```

и напечатает:

```
** 1f 20**
```

Параметр по умолчанию проходит проверку типа во время описания функции и вычисляется во время ее вызова. Задавать параметр по умолчанию возможно только для последних параметров, поэтому

```
int f(int, int =0, char* =0);    // ok
int g(int =0, int =0, char*);    // ошибка
int f(int =0, int, char* =0);    // ошибка
```

Заметьте, что в этом контексте пробел между * и = является существенным (*= является операцией присваивания):

```
int nasty(char*=0);                // синтаксическая ошибка
```

4.6.7 Перегрузка Имен Функций

Как правило, давать разным функциям разные имена - мысль хорошая, но когда некоторые функции выполняют одинаковую работу над объектами разных типов, может быть более удобно дать им одно и то же имя. Использование одного имени для различных действий над различными типами называется перегрузкой (overloading). Метод уже используется для основных операций C++: у сложения существует только одно имя, +, но его можно применять для сложения значений целых, плавающих и указательных типов. Эта идея легко расширяется на обработку операций, определенных пользователем, то есть, функций. Чтобы уберечь программиста от случайного повторного использования имени, имя может использоваться более чем для одной

функции только если оно сперва описано как перегруженное. Например:

```
overload print;
void print(int);
void print(char*);
```

Что касается компилятора, единственное общее, что имеют функции с одинаковым именем, это имя. Предположительно, они в каком-то смысле похожи, но в этом язык ни стесняет программиста, ни помогает ему. Таким образом, перегруженные имена функций - это главным образом удобство записи. Это удобство значительно в случае функций с общепринятыми именами вроде sqrt, print и open. Когда имя семантически значимо, как это имеет место для операций вроде +, * и << (#6.2) и в случае конструкторов (#5.2.4 и #6.3.1), это удобство становится существенным. Когда вызывается перегруженная f(), компилятор должен понять, к какой из функций с именем f следует обратиться. Это делается путем сравнения типов фактических параметров с типами формальных параметров всех функций с именем f. Поиск функции, которую надо вызвать, осуществляется за три отдельных шага:

[1] Искать функцию соответствующую точно, и использовать ее, если она найдена;

[2] Искать соответствующую функцию используя встроенные преобразования и использовать любую найденную функцию; и

[3] Искать соответствующую функцию используя преобразования, оперделенные пользователем (#6.3), и если множество преобразований единственно, использовать найденную функцию.

Например:

```
overload print(double), print(int);

void f();
{
    print(1);
    print(1.0);
}
```

Правило точного соответствия гарантирует, что f напечатает 1 как целое и 1.0 как число с плавающей точкой. Ноль, char или short точно соответствуют параметру int. Аналогично, float точно соответствует double.

К параметрам функций с перегруженными именами стандартные C++ правила преобразования (#c.6.6) применяются не польностью. Преобразования, могущие уничтожить информацию, не выполняются. Остаются int в long, int в double, ноль в long, ноль в double и преобразования указателей: ноль в указатель, ноль в void*, и указатель на производный класс в указатель на базовый класс (#7.2.4).

Вот пример, в котором преобразование необходимо:

```
overload print(double), print(long);

void f(int a);
{
    print(a);
}
```

Здесь a может быть напечатано или как double, или как long. Неоднозначность разрешается явным преобразованием типа (или `print(long(a))` или `print(double(a))`).

При этих правилах можно гарантировать, что когда эффективность или точность вычислений для используемых типов существенно различаются, будет использоваться простейший алгоритм (функция). Например:

```
overload pow;
int pow(int, int);
double pow(double, double);      // из
complex pow(double, complex);    // из
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);
```

Процесс поиска подходящей функции игнорирует unsigned и const.

4.6.8 Незаданное Число Параметров

Для некоторых функций невозможно задать число и тип всех параметров, которые можно ожидать в вызове. Такую функцию описывают завершая список описаний параметров многоточием (...), что означает "и может быть, еще какие-то параметры". Например:

```
int printf(char* ...);
```

Это задает, что в вызове printf должен быть по меньшей мере один параметр, char*, а остальные могут быть, а могут и не быть. Например:

```
printf("Hello, world\n");
printf("Мое имя %s %s\n", first_name, second_name);
printf("%d + %d = %d\n",2,3,5);
```

Такая функция полагается на информацию, которая недоступна компилятору при интерпретации ее списка параметров. В случае printf() первым параметром является строка формата, содержащая специальные последовательности символов, позволяющие printf() правильно обрабатывать остальные параметры. %s означает "жди параметра char*", а %d означает "жди параметра int". Однако, компилятор этого не знает, поэтому он не может убедиться в том, что ожидаемые параметры имеют соответствующий тип. Например:

```
printf("Мое имя %s %s\n",2);
```

откомпилируется и в лучшем случае приведет к какой-нибудь странного вида выдаче.

Очевидно, если параметр не был описан, то у компилятора нет информации, необходимой для выполнения над ним проверки типа и преобразования типа. В этом случае char или short передаются как int, а float передается как double. Это не обязательно то, чего ждет пользователь.

Чрезмерное использование многоточий, вроде wild(...), полностью выключает проверку типов параметров, оставляя программиста открытым перед множеством неприятностей, которые хорошо знакомы программистам на C. В хорошо продуманной программе требуется самое большее несколько функций, для которых типы параметров не определены полностью. Для того, чтобы позаботиться о проверке типов, можно использовать перегруженные функции и функции с параметрами по умолчанию в большинстве тех случаев, когда иначе пришлось бы оставить типы параметров незаданными. Многоточие необходимо только если изменяются и число параметров, и тип параметров. Наиболее обычное применеие многоточия в задании интерфейса с функциями C библиотек, которые были определены в то время, когда альтернативы не было:

```
extern int fprintf(FILE*, char* ...);    // из
extern int execl(char* ...);             // из
extern int  abort(...);                  // из
```

Стандартный набор макросов, имеющийся для доступа к неспецифицированнным параметрам в таких функциях, можно найти в . Разберем случай написания функции ошибок, которая получает один целый параметр, указывающий серьезность ошибки, после которого идет произвольное число строк. Идея состоит в том, чтобы составлять сообщение об ошибке с помощью передачи каждого слова как отдельного строкового параметра:

```
void error(int ...);

main(int argc, char* argv[])
{
    switch(argc) {
    case 1:
        error(0,argv[0],0);
        break;
    case 2:
        error(0,argv[0],argv[1],0);
    default:
        error(1,argv[0],"с",dec(argc-1),"параметрами",0);
    }
}
```

Функцию ошибок можно определить так:

```
#include

void error(int n ...)
/*
    "n" с последующим списком char*, оканчивающихся нулем
*/
{
    va_list ap;
    va_start(ap,n);           // раскрутка arg

    for (;;) {
    char* p = va_arg(ap,char*);
    if(p == 0) break;
    cerr << p << " ";
    }

    va_end(ap);               // очистка arg

    cerr << "\n";
    if (n) exit(n);
}
```

Первый из va_list определяется и инициализируется вызовом va_start(). Макрос va_start получает имя va_list'a и имя последнего формального параметра как параметры. Макрос va_arg используется для выбора неименованных параметров по порядку. При каждом обращении программист должен задать тип; va_arg() предполагает, что был передан фактический параметр, но обычно способа убедиться в этом нет. Перед возвратом из функции, в которой был использован va_start(), должен быть вызван va_end(). Причина в том, что va_start() может изменить стек так, что нельзя будет успешно осуществить возврат; va_end() аннулирует все эти изменения.

4.6.9 Указатель на Функцию

С функцией можно делать только две вещи: вызывать ее и брать ее адрес. Указатель, полученный взятием адреса функции, можно затем использовать для вызова этой функции. Например:

```
void error(char* p) { /* ... */ }

void (*efct)(char*);                // указатель на функцию

void f()
{
    efct = &error;                  // efct указывает на error
    (*efct)("error");               // вызов error через efct
}
```

Чтобы вызвать функцию через указатель, например, efct, надо сначала этот указатель разыменовать, *efct. Поскольку операция вызова функции () имеет более высокий приоритет, чем операция разыменования *, то нельзя писать просто *efct("error"). Это означает *efct("error"), а это ошибка в типе. То же относится и к синтаксису описаний (см. также #7.3.4).

Заметьте, что у указателей на функции типы параметров описываются точно также, как и в самих функциях. В присваиваниях указателя должно соблюдаться точное соответствие полного типа функции. Например:

```
void (*pf)(char*);          // указатель на void(char*)
void f1(char*);             // void(char*)
int  f2(char*);             // int(char*)
void f3(int*);              // void(int*)

void f()
{
    pf = &f1;               // ok
    pf = &f2;               // ошибка: не подходит возвращаемый тип
    pf = &f3;               // ошибка: не подходит тип параметра

    (*pf)("asdf");          // ok
    (*pf)(1);               // ошибка: не подходит тип параметра

    int i = (*pf)("qwer"); // ошибка: void присваивается int'у
}
```

Правила передачи параметров для непосредственных вызовов функции и для вызовов функции через указатель одни и те же.

Часто, чтобы избежать использования какого-либо неочевидного синтаксиса, бывает удобно определить имя типа указатель-на-функцию. Например:

```
typedef int (*SIG_TYP)();        // из
typedef void (*SIG_ARG_TYP);
SIG_TYP signal(int,SIG_ARG_TYP);
```

Бывает часто полезен вектор указателей на фукнцию. Например, система меню для моего редактора с мышью* реализована с помощью векторов указателей на функции для представления действий. Подробно эту систему здесь описать не получится, но вот общая идея:

---

* Мышь - это указывающее устройство по крайней мере с одной кнопкой. Моя мышь красная, круглая и с тремя кнопками. (прим. автора)

```
typedef void (*PF)();

PF edit_ops[] = { // операции редактирования
    cut, paste, snarf, search
};
```

```
PF file_ops[] = { // управление файлом
    open, reshape, close, write
};
```

Затем определяем и инициализируем указатели, определяющие действия, выбранные в меню, которое связано с кнопками (button) мыши:

```
PF* button2 = edit_ops;
PF* button3 = file_ops;
```

В полной реализации для опреледения каждого пункта меню требуется больше информации. Например, где-то должна храниться строка, задающая текст, который высвечивается. При использовании системы значение кнопок мыши часто меняется в зависимости от ситуации. Эти изменения осуществляются (частично) посредством смены значений указателей кнопок. Когда пользователь выбирает пункт меню, например пункт 3 для кнопки 2, выполняется связанное с ним действие:

```
(button2[3])();
```

Один из способов оценить огромную мощь указателей на функции - это попробовать написать такую систему не используя их. Меню можно менять в ходе использования программы, внося новые функции в таблицу действий. Во время выполнения можно также легко сконструировать новое меню.

Указатели на функции можно использовать для задания полиморфных подпрограмм, то есть подпрограмм, которые могут применяться к объектам многих различных типов:

```
typedef int (*CFT)(char*,char*);

int sort(char* base, unsigned n, int sz, CFT cmp)
/*
    Сортирует "n" элементов вектора "base"
    в возрастающем порядке
    с помощью функции сравнения, указываемой "cmp".
    Размер элементов "sz".

    Очень неэффективный алгоритм: пузырьковая сортировка
*/
{
    for (int i=0; iname, Puser(q)->name);
}

int cmp2(char*p, char* q)        // Сравнивает числа dept
{
    return Puser(p)->dept-Puser(q)->dept;
}
```

Эта программа сортирует и печатает:

```
main ()
{
    sort((char*)heads,6,sizeof(user),cmp1);
    print_id(heads,6);        // в алфавитном порядке
    cout << "\n";
    sort((char*)heads,6,sizeof(user),cmp2);
    print_id(heads,6);        // по порядку подразделений
}
```

Можно взять адрес inline-функции, как, впрочем, и адрес перегруженной функции(#с.8.9).

4.7 Макросы

Макросы* определяются в #с.11. В С они очень важны, но в С++ применяются гораздо меньше. Первое правило относительно них такое: не используйте их, если вы не обязаны это делать. Как было замечено, почти каждый макрос проявляет свой изъян или в языке, или в программе. Если вы хотите использовать макросы, прочитайте, пожалуйста, вначале очень внимательно руководство по вашей реализации С препроцессора.

Простой макрос определяется так:

```
#define name rest of line
```

Когда name встречается как лексема, оно заменяется на rest of line. Например:

```
named = name
```

после расширения даст:

```
named = rest of line
```

Можно также определить макрос с параметрами. Например:

```
#define mac(a,b) argument1: a argument2: b
```

При использовании mac должно даваться две строки параметра. После расширения mac() они заменяют a и b. Например:

```
expanded = mac(foo bar, yuk yuk)
```

после расширения даст

```
expanded = argument1: foo bar argument2: yuk yuk
```

Макросы обрабатывают строки и о синтаксисе С++ знают очень мало, а о типах С++ или областях видимости - ничего. Компилятор видит только расширенную форму макроса, поэтому ошибка в макросе диагностируется когда макрос расширен, а не когда он определен. В результате этого возникают непонятные сообщения об ошибках.

Вот такими макросы могут быть вполне:

```
#define Case break;case
#define nl <<"\n"
#define forever for(;;)
#define MIN(a,b) (((a)<(b))?(a):(b))
```

Вот совершенно ненужные макросы:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

А вот примеры опасных макросов:

---

* часто называемые также макроопределениями. (прим. перев.)

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
#define DISP = 4
```

Чтобы увидеть, чем они опасны, попробуйте провести расширения в

следующем примере:

```
int xx = 0;                 // глобальный счетчик

void f() {
    int xx = 0;             // локальная переменная
    xx = SQUARE(xx+2);      // xx = xx+2*xx+2
    INCR_xx;                // увеличивает локальный xx
    if (a-DISP==b) {        // a-= 4==b
        // ...
    }
}
```

Если вы внуждены использовать макрос, при ссылке на глобальные имена используйте операцию разрешения области видимости :: (#2.1.1) и заключайте вхождения имени параметра макроса в скобки везде, где это возможно (см. MIN выше).

Обратите внимание на различие результатов расширения этих двух макросов:

```
#define m1(a) something(a)    // глубокомысленный комментарий
#define m2(a) something(a)    /* глубокомысленный комментарий */
```

например,

```
int a = m1(1)+2;
int b = m2(1)+2;
```

расширяется в

```
int a = something(1)    // глубокомысленный комментарий+2;
int b = something(1)    /* глубокомысленный комментарий */+2;
```

С помошью макросов вы можете разработать свой собственный язык. Скорее всего, для всех остальных он будет непостижим. Кроме того, С препроцессор - очень простой макропроцессор. Когда вы попытаетесь сделать что-либо нетривиальное, вы, вероятно, обнаружите, что сделать это либо невозможно, либо чрезвычайно трудно (но см. #7.3.5).

4.8 Упражнения

1. (*1) Напишите следующие описания: функция, получающая параметр типа указатель на символ и ссылку на целое и не возвращающая значения; указатель на такую фунцию; функция, получающая такой указатель в качестве параметра; и функция, возвращающая такой указатель. Напишите определение функции, которая получает такой указатель как параметр и возвращает свой параметр как возвращаемое значение. Подсказка: используйте typedef.
2. (*1) Что это значит? Для чего это может использоваться?

```
typedef int (rifii&) (int, int);
```

3. (*1.5) Напишите программу вроде "Hello, world", которая получает имя как параметр командной строки и печатает "Hello, имя". Модифицируйте эту программу так, чтобы она получала получала любое количество имен и говорила hello каждому из них.
4. (*1.5) Напишите программу, которая читает произвольное число файлов, имена которых задаются как аргументы командной стоки, и пишет их один за другим в cout. Поскольку эта программа при выдаче конкатенирует свои параметры, вы можете назвать ее cat (кошка).
5. (*2) Преобразуйте небольшую С программу в С++. Измените

заголовочные файлы так, чтобы описывать все вызываемые функции и описывать тип каждого параметра. Замените, где возможно, директивы #define на enum и const или inline. Уберите из .c файлов описания extern и преобразуйте определения функций к синтаксису C++. Замените вызовы malloc() и free() на new и delete. Уберите необязательные приведения типа.

6. (*2) Реализуйте sort() (#4.6.7) используя эффективный алгоритм сортировки.
7. (*2) Посмотрите на определение struct tnode в с.#8.5. Напишите функцию для введения новых слов в дерево узлов tnode. Напишите функцию для вывода дерева узлов tnode. Напишите функцию для вывода дерева узлов tnode со словами в алфавитном порядке. Модифицируйте tnode так, чтобы в нем хранился (только) указатель на слово произвольной длины, помещенное с помощью new в свободную память. Модифицируйте функции для использования нового определения tnode.
8. (*2) Напишите "модуль", реализующий стек. Файл .h должен описывать функции push(), pop() и любые другие удобные функции (только). Файл .c определяет функции и данные, необходимые для хранения стека.
9. (*2) Узнайте, какие у вас есть стандартные заголовочные файлы. Составьте список файлов, находящихся в /usr/include и /usr/include/CC (или там, где хранятся стандртные заголовочные файлы в вашей системе). Прочитайте все, что покажется интересным.
10. (*2) Напишите функцию для обращения двумерного мосссива.
11. (*2) Напишите шифрующую программу, которая читает из cin и пишет в cout закодированные символы. Вы можете воспользоваться следующей простой схемой шифровки: Зашифрованная форма символа **c** - это c^key[i], где key (ключ) - строка, которая передается как параметр командной строки. Программа использует символы из key циклически, пока не будет считан весь ввод. Перекодирование зашифрованного текста с той же строкой key дает исходный текст. Если не передается никакого ключа (или передается пустая строка), то никакого кодирования не делается.
12. (*3) Напишите программу, которая поможет расшифровывать тексты, зашифрованные описанным выше способом, не зная ключа. Подсказка: David Kahn: The Code-Breakers, Macmillan, 1967, New York, pp 207-213.
13. (*3) Напишите функцию error, которая получает форматную строку в стиле printf, которая содержит директивы %s, %c и %d, и произвольное количество параметров. Не используйте printf().

Если вы не знаете значения %s и т.д., посмотрите #8.2.4. Используйте .

14. (*1) Как вы будете выбирать имя для указателя на тип функции, определенный с помощью typedef?
15. (*2) Посмотрите какие-нибудь программы, чтобы создать представление о разнообразии стилей и имен, использующихся на практике. Как используются буквы в верхнем регистре? Как используется подчерк? Где используются короткие имена вроде x и y?
16. (*1) Что неправильно в следующих макроопределениях?

```
#define PI = 3.141593
#define MAX(a,b) a>b?a:b
#define fac(a) (a)*fac((a)-1)
```

17. (*3) Напишите макропроцесор, который определяет и расширяет простые макросы (как C препроцессор). Читайте из cin и пишите в cout. Сначала не пытайтесть обрабатывать макросы с параметрами. Подсказка: В настольном калькуляторе (#3.1) есть таблица имен и лексический анализатор, которые вы можете

модифицировать.

Глава 5

Классы

Эти типы не "абстрактны",
они столь же реальны, как int и float.
- Дуг МакИлрой

В этой главе описываются возможности определения новых типов в C++, для которых доступ к данным ограничен заданным множеством функций доступа. Объясняются способы защиты структуры данных, ее инициализации, доступа к ней и, наконец, ее уничтожения. Примеры содержат простые классы для работы с таблицей имен, манипуляции стеком, работу с множеством и реализацию дискриминирующего (то есть, "надежного") объединения. Две следующие главы дополнят описание возможностей определения новых типов в C++ и познакомят читателя еще с некоторыми интересными примерами.

5.1 Знакомство и краткий обзор

Предназначение понятия класса, которому посвящены эта и две последующие главы, состоит в том, чтобы предоставить программисту инструмент для создания новых типов, столь же удобных в обращении сколь и встроенные типы. В идеале тип, определяемый пользователем, способом использования не должен отличаться от встроенных типов, только способом создания.

Тип есть конкретное представление некоторой концепции (понятия). Например, имеющийся в C++ тип float с его операциями +, -, * и т.д. обеспечивает ограниченную, но конкретную версию математического понятия действительного числа. Новый тип создается для того, чтобы дать специальное и конкретное определение понятия, которому ничто прямо и очевидно среди встроенных типов не отвечает. Например, в программе, которая работает с телефоном, можно было бы создать тип trunk_module (элемент линии), а в программе обработки текстов - тип list_of_paragraphs (список параграфов). Как правило, программу, в которой создаются типы, хорошо отвечающие понятиям приложения, понять легче, чем программу, в которой это не делается. Хорошо выбранные типы, определяемые пользователем, делают программу более четкой и короткой. Это также позволяет компилятору обнаруживать недопустимые использования объектов, которые в противном случае останутся необнаруженными до тестирования программы.

В определении нового типа основная идея - отделить несущественные подробности реализации (например, формат данных, которые используются для хранения объекта типа) от тех качеств, которые существенны для его правильного использования (например, полный список функций, которые имеют доступ к данным). Такое разделение можно описать так, что работа со структурой данных и внутренними административными подпрограммами осуществляется через специальный интерфейс (каналируется).

Эта глава состоит из четырех практически отдельных частей:

#5.2 Классы и Члены. Этот раздел знакомит с основным понятием типа, определяемого пользователем, который называется класс (class). Доступ к объектам класса может ограничиваться набором функций, которые описаны как часть этого класса. Такие функции называются функциями членами. Объекты класса создаются и инициализируются функциями членами, специально для этой цели описанными. Эти функции называются конструкторами. Функция член может быть специальным образом описана для "очистки" каждого классового объекта при его уничтожении. Такая фнукция называется деструктором.

#5.3 Интерфейсы и Реализации. В этом разделе приводится два

примера того, как класс проектируется, реализуется и используется.

#5.4 Друзья и Объединения. В этом разделе приводится много дополнительных подробностей, касающихся классов. В нем показано, как предоставить доступ к закрытой части класса функции, которая не является членом этого класса. Такая функция называется друг (friend). В этом разделе показано также, как определить дискриминирующее объединение.

#5.5 Конструкторы и Деструкторы. Объект может создаваться как автоматический, статический или как объект в свободной памяти. Объект может также быть членом некоторой совокупности (типа вектора или класса), которая в свою очередь может размещаться одним из этих трех способов. Довольно подробно объсняется использование конструкторов и деструкторов.

5.2 Классы и Члены

Класс - это определяемый пользователем тип. Этот раздел знакомит с основными средствами определения класса, создания объекта класса, работы с такими объектами и, наконец, уничтожения таких объектов после использования.

5.2.1 Функции Члены

Рассмотрим реализацию понятия даты с использованием struct для того, чтобы определить представление даты date и множества функций для работы с переменными этого типа:

```
struct date { int month, day, year; };
    // дата:     месяц, день, год  }
date today;
void set_date(date*, int, int, int);
void next_date(date*);
void print_date(date*);
// ...
```

Никакой явной связи между функциями и типом данных нет. Такую связь можно установить, описав функции как члены:

```
struct date {
    int month, day, year;

void set(int, int, int);
void get(int*, int*, int*);
void next();
void print();
};
```

Функции, описанные таким образом, называются функциями членами и могут вызываться только для специальной переменной соответствующего типа с использованием стандартного синтаксиса для доступа к членам структуры. Например:

```
date today;          // сегодня
date my_burthday;    // мой день рождения

void f()
{
    my_burthday.set(30,12,1950);
    today.set(18,1,1985);

    my_burthday.print();
    today.next();
}
```

Поскольку разные структуры могут иметь функции члены с одинаковыми именами, при определении функции члена необходимо указывать имя структуры:

```
void date::next()
{
    if ( ++day > 28 ) {
        // делает сложную часть работы
    }
}
```

В функции члене имена членов могут использоваться без явной ссылки на объект. В этом случае имя относится к члену того объекта, для которого функция была вызвана.

5.2.2 Классы

Описание date в предыдущем подразделе дает множество функций для работы с date, но не указывает, что эти функции должны быть единственными для доступа к объектам типа date. Это ограничение можно наложить используя вместо struct class:

```
class date {
    int month, day, year;
public:
    void set(int, int, int);
    void get(int*, int*, int*);
    void next();
    void print();
};
```

Метка public делит тело класса на две части. Имена в первой, закрытой части, могут использоваться только функциями членами. Вторая, открытая часть, составляет интерфейс к объекту класса. Struct - это просто class, у которого все члены общие, поэтому функции члены определяются и используются точно так же, как в предыдущем случае. Например:

```
void date::ptinr()        // печатает в записи, принятой в США
{
    cout << month << "/" << day << "/" year;
}
```

Однако функции не члены отгорожены от использования закрытых членов класса date. Например:

```
void backdate()
{
    today.day--;        // ошибка
}
```

В том, что доступ к структуре данных ограничен явно описанным списком функций, есть несколько преимуществ. Любая ошибка, которая приводит к тому, что дата принимает недопустимое значение (например, Декабрь 36, 1985) должна быть вызвана кодом функции члена, поэтому первая стадия отладки, локализация, выполняется еще до того, как программа будет запущена. Это частный случай общего утверждения, что любое изменение в поведении типа date может и должно вызываться изменениями в его членах. Другое преимущество - это то, что потенциальному пользователю такого типа нужно будет только узнать определение функций членов, чтобы научиться им пользоваться.

Защита закрытых данных связана с ограничением использования имен

членов класса. Это можно обойти с помощью манипуляции адресами, но это уже, конечно, жульничество.

5.2.3 Ссылки на Себя

В функции члене на члены объекта, для которого она была вызвана, можно ссылаться непосредственно. Например:

```
class x {
    int m;
public:
    int readm() { return m; }
};

x aa;
x bb;

void f()
{
    int a = aa.readm();
    int b = bb.readm();
    // ...
}
```

В первом вызове члена member() m относится к aa.m, а во втором - к bb.m.

Указатель на объект, для которого вызвана функция член, явяется скрытым параметром функции. На этот неявный параметр можно ссылаться явно как на this. В каждой функции класса x указатель this неявно описан как

```
x* this;
```

и инициализирован так, что он указывает на объект, для которого была вызвана функция член. this не может быть описан явно, так как это ключевое слово. Класс x можно эквивалентным образом описать так:

```
class x {
    int m;
public:
    int readm() { return this->m; }
};
```

При ссылке на члены использование this излишне. Главным образом this используется при написании функций членов, которые манипулируют непосредственно указателями. Типичный пример этого - функция, вставляющая звено в дважды связанный список:

```
class dlink {
    dlink* pre;    // предшествующий
    dlink* suc;    // следующий
public:
    void append(dlink*);
    // ...
};

void dlink::append(dlink* p)
{
    p->suc = suc;    // то есть, p->suc = this->suc
    p->pre = this;   // явное использование this
    suc->pre = p;    // то есть, this->suc->pre = p
```

```
    suc = p;          // то есть, this->suc = p
}

dlink* list_head;

void f(dlink*a, dlink *b)
{
    // ...
    list_head->append(a);
    list_head->append(b);
}
```

Цепочки такой общей природы являются основой для списковых классов, которые описываются в Главе 7. Чтобы присоединить звено к списку необходимо обновить объекты, на которые указывают указатели this, pre и suc (текущий, предыдущий и последующий). Все они типа dlink, поэтому функция член dlink::append() имеет к ним доступ. Единицей защиты в C++ является class, а не отдельный объект класса.

5.2.4 Инициализация

Использование для обеспечения инициализации объекта класса функций вроде set_date() (установить дату) неэлегантно и чревато ошибками. Поскольку нигде не утверждается, что объект должен быть инициализирован, то программист может забыть это сделать, или (что приводит, как правило, к столь же разрушительным последствиям) сделать это дважды. Есть более хороший подход: дать возможность программисту описать функцию, явно предназначенную для инициализации объектов. Поскольку такая функция конструирует значения данного типа, она называется конструктором. Конструктор распознается по тому, что имеет то же имя, что и сам класс. Например:

```
class date {
    // ...
    date(int, int, int);
};
```

Когда класс имеет конструктор, все объекты этого класса будут инициализироваться. Если для конструктора нужны параметры, они должны даваться:

```
date today = date(23,6,1983);
date xmas(25,12,0);         // сокращенная форма
                          // (xmas - рождество)
date my_burthday;          // недопустимо, опущена инициализация
```

Часто бывает хорошо обеспечить несколько способов инициализации объекта класса. Это можно сделать, задав несколько конструкторов. Например:

```
class date {
    int month, day, year;
public:
    // ...
    date(int, int, int);    // день месяц год
    date(char*);            // дата в строковом представлении
    date(int);                // день, месяц и год сегодняшние
    date();                    // дата по умолчанию: сегодня
};
```

Конструкторы подчиняются тем же правилам относительно типов параметров, что и перегруженные функции (#4.6.7). Если контрукторы существенно различаются по типам своих парметров, то компилятор при

каждом использовании может выбрать правильный:

```
date today(4);
date july4("Июль 4, 1983");
date guy("5 Ноя");
date now;                    // инициализируется по умолчанию
```

Заметьте, что функции члены могут быть перегружены без явного использования ключевого слова overload. Поскольку полный список функций членов находится в описании класса и как правило короткий, то нет никакой серьезной причины требовать использования слова overload для предотвращения случайного повторнго использования имени.

Размножение конструкторов в примере с date типично. При разработке класса всегда есть соблазн обеспечить "все", поскольку кажется проще обеспечить какое-нибудь средство просто на случай, что оно кому-то понадобится или потому, что оно изящно выглядит, чем решить, что же нужно на самом деле. Последнее требует больших размышлений, но обычно приводит к программам, которые меньше по размеру и более понятны. Один из способов сократить число родственных функций - использовать параметры по умолчанию. В случае date для каждого параметра можно задать значение по умолчанию, интерпретируемое как "по умолчанию принимать: today" (сегодня).

```
class date {
    int month, day, year;
public:
    // ...
    date(int d =0, int m =0, int y =0);
    date(char*);            // дата в строковом представлении
};

date::date(int d, int m, int y)
{
    day = d ? d : today.day;
    month = m ? m : today.month;
    year = y ? y : today.year;
    // проверка, что дата допустимая
    // ...
}
```

Когда используется значение параметра, указывающее "брать по умолчанию", выбранное значение должно лежать вне множества возможных значений параметра. Для дня day и месяца mounth ясно, что это так, но для года year выбор нуля неочевиден. К счастью, в европейском календаре нет нулевого года . Сразу после 1 г. до н.э. (year==-1) идет 1 г. н.э. (year==1), но для реальной программы это может оказаться слишком тонко.

Объект класса без конструкторов можно инициализировать путем присваивания ему другого объекта этого класса. Это можно делать и тогда, когда конструкторы описаны. Например:

```
date  d = today;    // инициализация посредством присваивания
```

По существу, имеется конструктор по умолчанию, определенный как побитовая копия объекта того же класса. Если для класса X такой конструктор по умолчанию нежелателен, его можно переопределить конструктором с именем X(X&). Это будет обсуждаться в #6.6.

5.2.5 Очистка

Определяемый пользователем тип чаще имеет, чем не имеет, конструктор, который обеспечивает надлежащую инициализацию. Для многих типов также требуется обратное действие, деструктор, чтобы

обеспечить соответствующую очистку объектов этого типа. Имя деструктора для класса X есть ~X() ("дополнение конструктора"). В частности, многие типы используют некоторый объем памяти из свободной памяти (см. #3.2.6), который выделяется конструктором и освобождается деструктором. Вот, например, традиционный стековый тип, из которого для краткости полностью выброшена обработка ошибок:

```
class char_stack {
    int size;
    char* top;
    char* s;
public:
    char_stack(int sz) { top=s=new char[size=sz]; }
    ~char_stack()      { delete s; }    // деструктор
    void push(char c)  { *top++ = c; }
    char pop()         { return *--top;}
}
```

Когда char_stack выходит из области видимости, вызывается деструктор:

```
void f()
{
    char_stack s1(100);
    char_stack s2(200);
    s1.push('a');
    s2.push(s1.pop());
    char ch = s2.pop();
    cout << chr(ch) << "\n";
}
```

Когда вызывается f(), конструктор char_stack вызывается для s1, чтобы выделить вектор из 100 символов, и для s2, чтобы выделить вектор из 200 символов. При возврате из f() эти два вектора будут освобождены.

5.2.6 Inline

При программировании с использованием классов очень часто используется много маленьких функций. По сути, везде, где в программе традиционной структуры стояло бы просто какое-нибудь обычное использование структуры данных, дается функция. То, что было соглашением, стало стандартом, который распознает компилятор. Это может страшно понизить эффективность, потому что стоимость вызова функции (хотя и вовсе не высокая по сравнению с другими языками) все равно намного выше, чем пара ссылок по памяти, необходимая для тела функции.

Чтобы справиться с этой проблемой, был разработан аппарат inline-функций. Функция член, определенная (а не просто описанная) в описании класса, считается inline. Это значит, например, что в функциях, которые используют приведенные выше char_stack, нет никаких вызовов функций кроме тех, которые используются для реализации операций вывода! Другими словами, нет никаких затрат времени выполнения, которые стоит принимать во внимание при разработке класса. Любое, даже самое маленькое действие, можно задать эффективно. Это увтерждение снимает аргумент, который чаще всего приводят чаще всего в пользу открытых членов данных.

Функцию член можно также описать как inline вне описания класса. Например:

```
char char_stack {
```

```
    int size;
    char* top;
    char* s;
public:
    char pop();
    // ...
};

inline char char_stack::pop()
{
    return *--top;
}
```

5.3 Интерфейсы и Реализации

Что представляет собой хороший класс? Нечто, имеющее небольшое и хорошо определенное множество действий. Нечто, что можно рассматривать как "черный ящик", которым манипулируют только посредством этого множества действий. Нечто, чье фактическое представление можно любым мыслимым способом изменить, не повлияв на способ использования множества действий. Нечто, чего можно хотеть иметь больше одного.

Для всех видов контейнеров существуют очевидные примеры: таблицы, множества, списки, вектора, словари и т.д. Такой класс имеет операцию "вставить", обычно он также имеет операции для проверки того, был ли вставлен данный элемент. В нем могут быть действия для осуществления проверки всех элементов в определенном порядке, и кроме всего прочего, в нем может иметься операция для удаления элемента. Обычно контейнерные (то есть, вмещающие) классы имеют конструкторы и деструкторы.

Скрытие данных и продуманный интерфейс может дать концепция модуля (см. например #4.4: файлы как модули). Класс, однако, является типом. Чтобы использовать его, необходимо создать объекты этого класса, и таких объектов можно создавать столько, сколько нужно. Модуль же сам является объектом. Чтобы использовать его, его надо только инициализировать, и таких объектов ровно один.

5.3.1 Альтернативные Реализации

Пока описание открытой части класса и описание функций членов остаются неизменными, реализацию класса можно модифицировать не влияя на ее пользователей. Как пример этого рассмотрим таблицу имен, которая использовалась в настольном калькуляторе в Главе 3. Это таблица имен:

```
struct name {
    char* string;
    char* next;
    double value;
};
```

Вот вариант класса table:

```
// файл table.h

class table {
    name* tbl;
public:
    table() { tbl = 0; }

name* look(char*, int = 0);
name* insert(char* s) { return look(s,1); }
};
```

Эта таблица отличается от той, которая определена в Главе 3 тем, что это настоящий тип. Можно описать более чем одну table, можно иметь указатель на table и т.д. Например:

```
#include "table.h"

table globals;
table keywords;
table* locals;

main() {
    locals = new table;
    // ...
}
```

Вот реализация table::look(), которая использует линейный поиск в связанном списке имен name в таблице:

```
#include

name* table::look(char* p, int ins)
{
    for (name* n = tbl; n; n=n->next)
        if (strcmp(p,n->string) == 0) return n;

    if (ins == 0) error("имя не найдено");

    name* nn = new name;
    nn->string = new char[strlen(p)+1];
    strcpy(nn->string,p);
    nn->value = 1;
    nn->next = tbl;
    tbl = nn;
    return nn;
}
```

Теперь рассмотрим класс table, усовершенствованный таким образом, чтобы использовать хэшированный просмотр, как это делалось в примере с настольным калькулятором. Сделать это труднее из-за того ограничения, что уже написанные программы, в которых использовалась только что определенная версия класса table, должны оставаться верными без изменений:

```
class table {
    name** tbl;
    int size;
public:
    table(int sz = 15);
    ~table();

name* look(char*, int = 0);
name* insert(char* s) { return look(s,1); }
};
```

В структуру данных и конструктор внесены изменения, отражающие необходимость того, что при использовании хэширования таблица должна иметь определенный размер. Задание конструктора с параметром по умолчанию обеспечивает, что старая программа, в которой не указывался размер таблицы, останется правильной. Параметры по умолчанию очень полезны в ситуации, когда нужно изменить класс не повлияв на старые программы. Теперь конструктор и деструктор создают и уничтожают хэш-таблицы:

```
table::table(int sz)
```

```
{
    if (sz < 0) error("отрицательный размер таблицы");
    tbl = new name*[size=sz];
    for (int i = 0; inext) {
            delete n->string;
            delete n;
        }
        delete tbl;
}
```

Описав деструктор для класса name можно получить более простой и ясный вариант table::~table(). Функция просмотра практически идентична той, которая использовалась в примере настольного калькулятора (#3.1.3):

```
#include

name* table::look(char* p, int ins)
{
    int ii = 0;
    char* pp = p;
    while (*pp) ii = ii<<1 ^ *pp++;
    if (ii < 0) ii = -ii;
    ii %= size;

    for (name* n=tbl[ii]; n; n=n->next)
        if (strcmp(p,n->string) == 0) return n;

    if (ins == 0) error("имя не найдено");

    name* nn = new name;
    nn->string = new char[strlen(p)+1];
    strcpy(nn->string,p);
    nn->value = 1;
    nn->next = tbl[ii];
    tbl[ii] = nn;
    return nn;

}
```

Очевидно, что функции члены класса должны заново компилироваться всегда, когда вносится какое-либо изменение в описание класса. В идеале такое изменение никак не должно отражаться на пользователях класса. К сожалению, это не так. Для размещения переменной классового типа компилятор должен знать размер объекта класса. Если размер этих объектов меняется, то файлы, в которых класс используется, нужно компилировать заново. Можно написать такую программу (и она уже написана), которая определяет множество (минимальное) файлов, которое необходимо компилировать заново после изменения описания класса, но пока что широкого распространения она не получила.

Почему, можете вы спросить, C++ разработан так, что после изменения закрытой части необходима новая компиляция пользователей класса? И действительно, почему вообще закрытая часть должна быть представлена в описании класса? Другими словами, раз пользователям класса не разрешается обращаться к закрытым членам, почему их описания должны приводиться в заголовочных файлах, которые, как предполагается, пользователь читает? Ответ - эффективность. Во многих системах и процесс компиляции, и последовательность операций, реализующих вызов функции, проще, когда размер автоматических объектов (объектов в стеке) известен во время компиляции.

Этой сложности можно избежать, представив каждый объект класса как указатель на "настоящий" объект. Так как все эти указатели

будут иметь одинаковый размер, а размещение "настоящих" объектов можно определить в файле, где доступна закрытая часть, то это может решить проблему. Однако решение подразумевает дополнительные ссылки по памяти при обращении к членам класса, а также, что еще хуже, каждый вызов функции с автоматическим объектом класса включает по меньшей мере один вызов программ выделения и освобождения свободной памяти. Это сделало бы также невозможным реализацию inline-функций

членов, которые обращаются к данным закрытой части. Более того, такое изменение сделает невозможным совместную компоновку C и C++ программ (поскольку C компилятор обрабатывает struct не так, как это будет делать C++ компилятор). Для C++ это было сочтено неприемлемым.

5.3.2 Законченный Класс

Программирование без скрытия данных (с применением структур) требует меньшей продуманности, чем программирование со скрытием данных (с использованием классов). Структуру можно определить не слишком задумываясь о том, как ее предполагается использовать. А когда определяется класс, все внимание сосредотачивается на обеспечении нового типа полным множеством операций; это важное смещение акцента. Время, потраченное на разработку нового типа, обычно многократно окупается при разработке и тестировании программы.

Вот пример законченного типа intset, который реализует понятие "множество целых":

```
class intset {
    int cursize, maxsize;
    int *x;
public:
    intset(int m, int n);    // самое большее, m int'ов в 1..n
    ~intset();

    int member(int t);       // является ли t элементом?
    void insert(int t);      // добавить "t" в множество

    void iterate(int& i)     { i = 0; }
    int ok(int& i)           { return i

void error(char* s)
{
    cerr << "set: " << s << "\n";
    exit(1);
}
```

Класс intset используется в main(), которая предполагает два целых параметра. Первый параметр задает число случайных чисел, которые нужно сгенерировать. Второй параметр указывает диапазон, в котором должны лежать случайные целые:

```
main(int argc, char* argv[])
{
    if (argc != 3) error("ожидается два параметра");
    int count = 0;
    int m = atoi(argv[1]);        // число элементов множества
    int n = atoi(argv[2]);        // в диапазоне 1..n
    intset s(m,n);

    while (count maxsize) error("слищком много элементов");
    int i = cursize-1;
```

```
    x[i] = t;

    while (i>0 && x[i-1]>x[i]) {
        int t = x[i];               // переставить x[i] и [i-1]
        x[i] = x[i-1];
        x[i-1] = t;
        i--;
    }
}
```

Для нахождения членов используется просто двоичный поиск:

```
int intset::member(int t)       // двоичный поиск
{
    int l = 0;
    int u = cursize-1;

    while (l <= u) {
        int m = (l+u)/2;
        if (t < x[m])
            u = m-1;
        else if (t > x[m])
            l = m+1;
        else
            return 1;           // найдено
    }
    return 0;                   // не найдено
}
```

И, наконец, нам нужно обеспечить множество операций, чтобы пользователь мог осуществлять цикл по множеству в некотором порядке, поскольку представление intset от пользователя скрыто. Множество внутренней упорядоченности не имеет, поэтому мы не можем просто дать возможность обращаться к вектору (завтра я, наверное, реализую intset по-другому, в виде связанного списка).

Дается три функции: iterate() для инициализации итерации, ok() для проверки, есть ли следующий элемент, и next() для того, чтобы взять следующий элемент:

```
class intset {
    // ...
    void iterate(int& i)        { i = 0; }
    int ok(int& i)              { return iiterate(var);
    while (set->ok(var)) cout << set->next(var) << "\n";
}
```

Другой способ задать итератор приводится в #6.8.

Глава 6

Перегрузка Операций

Здесь водятся Драконы!
- старинная карта

В этой главе описывается аппарат, предоставляемый в C++ для перегрузки операций. Программист может определять смысл операций при их применении к объектам определенного класса. Кроме арифметических, можно определять еще и логические операции, операции сравнения, вызова () и индексирования [], а также можно переопределять присваивание и инициализацию. Можно определить явное и неявное преобразование между определяемыми пользователем и основными типами. Показано, как определить класс, объект которого не может быть никак иначе скопирован или уничтожен кроме как специальными определенными пользователем функциями.

6.1 Введение

Часто программы работают с объектами, которые фвляются конкретными представлениями абстрактных понятий. Например, тип данных int в C++ вместе с операциями +, -, *, / и т.д. предоставляет реализацию (ограниченную) математического понятия целых чисел. Такие понятия обычно включают в себя множество операций, которые кратко, удобно и привычно представляют основные действия над объектами. К сожалению, язык программирования может непосредственно поддерживать лишь очень малое число таких понятий. Например, такие понятия, как комплексная арифметика, матричная алгебра, логические сигналы и строки не получили прямой поддержки в C++. Классы дают средство спецификации в C++ представления неэлементарных объектов вместе с множеством действий, которые могут над этими объектами выполняться. Иногда определение того, как действуют операции на объекты классов, позволяет программисту обеспечить более общепринятую и удобную запись для манипуляции объектами классов, чем та, которую можно достичь используя лишь основную функциональную запись. Например:

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }
    friend complex operator+(complex, complex);
    friend complex operator*(complex, complex);
};
```

определяет простую реализацию понятия комплексного числа, в которой число представляется парой чисел с плавающей точкой двойной точности, работа с которыми осуществляется посредством операций + и * (и только). Программист задает смысл операций + и * с помощью определения функций с именами operator+ и operator*. Если, например, даны b и c типа complex, то b+c означает (по определению) operator+(b,c). Теперь есть возможность приблизить общепринятую интерпретацию комплексных выражений. Например:

```
void f()
{
    complex a = complex(1, 3.1);
    complex b = complex(1.2, 2);
    complex c = b;

    a = b+c;
    b = b+c*a;
    c = a*b+complex(1,2);
}
```

Выполняются обычные правила приоритетов, поэтому второй оператор означает b=b+(c*a), а не b=(b+c)*a.

6.2 Функции Операции

Можно описывать функции, определяющие значения следующих операций:

```
+   -   *   /   %   ^   &   |   ~   !
=   <   >   +=  -=  *=  /=  %=  ^=  &=
|=  <<  >>  >>= <<= ==  !=  <=  >=  &&
||  ++  --  []  ()  new delete
```

Последние четыре - это индексирование (#6.7), вызов функции (#6.8), выделение свободной памяти и освобождение свободной памяти (#3.2.6). Изменить приоритеты перецисленных операций невозможно,

как невозможно изменить и синтаксис выражений. Нельзя, например, определить унарную операцию % или бинарную !. Невозможно определить новые лексические символы операций, но в тех случаях, когда множество операций недостаточно, вы можете исользовать запись вызова функции. Используйте например, не **, а pow(). Эти ограничения могут показаться драконовскими, но более гибкие правила могут очень легко привести к неоднозначностям. Например, на первый взгляд определение операции **, охначающей возведение в степень, может показаться очевидной и простой задачей, но подумайте еще раз. Должна ли ** связываться влево (как в Фортране) или впрво (как в Алголе)? Выражение a**p должно интерпретироваться как a*(*p) или как (a)**(p)?

Имя функции операции есть ключевое слово operator (то есть, операция), за которым следует сама операция, например, operator<<. Функция операция описывается и может вызываться так же, как любая другая функция. Использование операции - это лишь сокращенная запись явного вызова функции операции. Например:

```
void f(complex a, complex b)
{
    complex c = a + b;          // сокращенная запись
    complex d = operator+(a,b); // явный вызов
}
```

При наличии предыдущего описания complex оба инициализатора являются синонимиами.

6.2.1 Бинарные и Унарные Онерации

Бинарная операция может быть определена или как функция член, получающая один параметр, или как функция друг, получающая два параметра. Таким образом, для любой бинарной операции @ aa@bb может интерпретироваться или как aa.operator@(bb), или как operator@(aa,bb). Если определены обе, то aa@bb является ошибкой. Унарная операция, префиксная или постфиксная, может быть определена или как функция член, не получающая параметров, или как функция друг, получающая один параметр. Таким образом, для любой унарной операции @ aa@ или @aa может интерпретироваться или как aa.operator@(), или как operator@(aa). Если определена и то, и другое, то и aa@ и @aa являются ошибками. Рассмотрим следующие примеры:

```
class X {
// друзья

    friend X operator-(X);      // унарный минус
    friend X operator-(X,X);    // бинарный минус
    friend X operator-();       // ошибка: нет операндов
    friend X operator-(X,X,X);  // ошибка: тернарная

// члены (с неявным первым параметром: this)

    X* operator&(); // унарное & (взятие адреса)
    X operator&(X); // бинарное & (операция И)
    X operator&(X,X);   // ошибка: тернарное

};
```

Когда операции ++ и -- перегружены, префиксное использование и постфиксное различить невозможно.

6.2.2 Предопределенные Значения Операций

Относительно смысла операций, определяемых пользователем, не

делается никаких предположений. В частности, поскольку не предполагается, что перегруженное = реализует присваивание ее первому операнду, не делается никакой проверки, чтобы удостовериться, является ли этот операнд lvalue (#c.6).

Значения некоторых встроенный операций определены как равносильные определенным комбинациям другий операций над теми же аргументами. Например, если a является int, то ++a означает a+=1, что в свою очередь означает a=a+1. Такие соотношения для определенных пользователем операций не выполняются, если только не случилось так, что пользователь сам определил их таким образом. Например, определение operator+=() для типа complex не может быть выведено из определений complex::operator+() и complex::operator=().

По историческому совпадению операции = и & имеют предопределенный смысл для объектов классов. Никакого элегантного сполоба "неопределить" эти две операции не существует. Их можно, однако, сделать недееспособными для класса X. Можно, например, описать X::operator&(), не задав ее определения. Если где-либо будет браться адрес объекта класса X, то компоновщик обнаружит отсутствие определения*. Или, другой способ, можно определить X::operator&() так, чтобы вызывала ошибку во время выполнения.

6.2.3 Операции и Определяемые Пользователем Типы

Функция операция должна или быть членом, или получать в качестве параметра по меньшей мере один объект класса (функциям, которые переопределяют операции new и delete, это делать необязательно). Это правило гарантирует, что пользователь не может изменить смысл никакого выражения, не включающего в себя определенного пользователем типа. В частности, невозможно определить функцию, которая действует исключительно на указтели.

Функция операция, первым параметром которой предполагается осповной тип, не может быть функцией членом. Рассмотрим, например, сложение комплексной переменной aa с целым 2: aa+2, при подходящим образом описанной функции члене, может быть проинтерпретировано как aa.operator+(2), но с 2+aa это не может быть сделано, потому что нет такого класса int, для которого можно было бы определить + так, чтобы это означало 2.operator+(aa). Даже если бы такой тип был, то для того, чтобы обработать и 2+aa и aa+2, понадобилось бы две различных функции члена. Так как компилятор не знает смысла +, определенного пользователем, то не может предполагать, что он коммутативен, и интерпретировать 2+aa как aa+2. С этим примером могут легко справиться функции друзья.

Все функции операции по определению перегружены. Функция операция задает новый смысл операции в дополнение к встроенному определению, и может существовать несколько функций операций с одним и тем же именем, если в типах их параметров имеются отличия, различимые для компилятора, чтобы он мог различать их при обращении (см. #4.6.7).

6.3 Определяемое Преобразование Типа

Приведенная во введении реализация комплексных чисел слишком ограничена, чтобы она могла устроить кого-либо, поэтому ее нужно расширить. Это будет в основном повторением описанных выше методов. Например:

---

* В некоторых системах компоновщик настолько "умен", что ругается, даже если неопределена неиспользуемая функция. В таких системах этим методом воспользоваться нельзя. (прим автора)

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }

    friend complex operator+(complex, complex);
    friend complex operator+(complex, double);
    friend complex operator+(double, complex);

    friend complex operator-(complex, complex);
    friend complex operator-(complex, double);
    friend complex operator-(double, complex);
    complex operator-()     // унарный -

    friend complex operator*(complex, complex);
    friend complex operator*(complex, double);
    friend complex operator*(double, complex);

    // ...
};
```

Теперь, имея описание complex, мы можем написать:

```
void f()
{
    complex a(1,1), b(2,2), c(3,3), d(4,4), e(5,5);
    a = -b-c;
    b = c*2.0*c;
    c = (d+e)*a;
}
```

Но писать функцию для каждого сочетания complex и double, как это делалось выше для operator+(), невыносимо нудно. Кроме того, близкие к реальности средства комплексной арифметики должны предоставлять по меньшей мере дюжину таких функций; посмотрите, например, на тип complex, описаннчй в .

6.3.1 Конструкторы

Альтенативу использованию нескольких функций (перегруженных) составлет описание конструктора, который по заданному double создает complex. Например:

```
class complex {
    // ...
    complex(double r) { re=r; im=0; }
};
```

Конструктор, требующий только один параметр, необязательно вызывать явно:

```
complex z1 = complex(23);
complex z2 = 23;
```

И z1, и z2 будут инициализированы вызовом complex(23).

Конструктор - это предписание, как создавать значение данного типа. Когда требуется значение типа, и когда такое значение может быть создано конструктором, тогда, если такое значение дается для присваивания, вызывается конструктор. Например, класс complex можно было бы описать так:

```
class complex {
```

```
    double re, im;
public:
    complex(double r, double i = 0) { re=r; im=i; }

    friend complex operator+(complex, complex);
    friend complex operator*(complex, complex);
};
```

и действия, в которые будут входить переменные complex и целые константы, стали бы допустимы. Целая константа будет интерпретироваться как complex с нулевой мнимой частью. Например, a=b*2 означает:

```
a=operator*( b, complex( double(2), double(0) ) )
```

Определенное пользователем преобразование типа применяется неявно только тогда, когда оно является едиственным.

Объект, сконструированный с помощью явного или неявного вызова конструктора, является автоматическим и будет уничтожен при первой возможности, обычно сразу же после оператора, в котором он был создан.

6.3.2 Операции Преобразования

Использование конструктора для задания преобразования типа является удобным, но имеет следствия, которые могут оказаться нежелательными:

[1] Не может быть неявного преобразования из определенного пользователем типа в основной тип (поскольку основные типы не являются классами);

[2] Невозможно задать преобразование из нового типа в старый, не изменяя описание старого; и

[3] Невозможно иметь конструктор с одним параметром, не имея при этом преобразования.

Последнее не является серьезной проблемой, а с первыми двумя можно справиться, определив для исходного типа операцию преобразования. Функция член X::operator T(), где T - имя типа, определяет преобразование из X в T. Например, можно определить тип tiny (крошечный), который может иметь значение только в диапазоне 0...63, но все равно может свободно сочетаться в целыми в арифметических операциях:

```
class tiny {
    char v;
    int assign(int i)
    { return v = (i&~63) ? (error("ошибка диапазона"),0) : i; }
public:
    tiny(int i)            { assign(i); }
    tiny(tiny& i)          { v = t.v; }
    int operator=(tiny& i) { return v = t.v; }
    int operator=(int i)   { return assign(i); }
    operator int()         { return v; }
}
```

Диапазон значения проверяется всегда, когда tiny инициализируется int, и всегда, когда ему присваивается int. Одно tiny может присваиваться другому без проверки диапазона. Чтобы разрешить выполнять над переменными tiny обычные целые операции, определяется tiny::operator int(), неявное преобразование из int в tiny. Всегда, когда в том месте, где требуется int, появляется tiny, используется соответствующее ему int. Например:

```
void main()
{
```

```
    tiny c1 = 2;
    tiny c2 = 62;
    tiny c3 = c2 - c1;  // c3 = 60
    tiny c4 = c3;       // нет проверки диапазона (необязательна)
    int i = c1 + c2;    // i = 64
    c1 = c2 + 2 * c1;   // ошибка диапазона: c1 = 0 (а не 66)
    c2 = c1 -i;         // ошибка диапазона: c2 = 0
    c3 = c2;            // нет проверки диапазона (необязательна)
}
```

Тип вектор из tiny может оказаться более полезным, поскольку он экономит пространство. Чтобы сделать этот тип более удобным в обращении, можно использовать операцию индексирования.

Другое применение определяемых операций преобразования - это типы, которые предоставляют нестандартные представления чисел (арифметика по основанию 100, арифметика с фиксированной точкой, двоично-десятичное представление и т.п.). При этом обычно переопределяются такие операции, как + и *.

Функции преобразования оказываются особенно полезными для работы со структурами данных, когда чтение (реалоизованное посредством операции преобразования) тривиально, в то время как присваивание и инициализация заметно более сложны.

Типы istream и ostream опираются на функцию преобразования, чтобы сделать возможными такие операторы, как

while (cin>>x) cout<<x выше возвращает istream&. Это значение неявно преобразуется к значению, которое указывает состояние cin, а уже это значение может проверяться оператором while (см. #8.4.2). Однако определять преобразование из оного типа в другой так, что при этом теряется информация, обычно не стоит.

6.3.3 Неоднозначности

Присваивание объекту (или инициализация объекта) класса X является допустимым, если или присваиваемое значение является X, или существует единственное преобразование присваиваемого значения в тип X.

В некоторых случаях значение нужного типа может сконструироваться с помощью нескольких применений конструкторов или операций преобразования. Это должно делаться явно; допустим только один уровень неявных преобразований, определенных пользователем. Иногда значение нужного типа может быть сконструировано более чем одним способом. Такие случаи являются недопустимыми. Например:

```
class x { /* ... */ x(int); x(char*); };
class y { /* ... */ y(int); };
class z { /* ... */ z(x); };

overload f;
x f(x);
y f(y);

z g(z);

f(1);         // недопустимо: неоднозначность f(x(1)) или f(y(1))
f(x(1));
f(y(1));
g("asdf");     // недопустимо: g(z(x("asdf"))) не пробуется
g(z("asdf"));
```

Определенные пользователем преобразования рассматриваются только в том случае, если без них вызов разрешить нельзя. Например:

```
class x { /* ... */ x(int); }
```

```
overload h(double), h(x);
h(1);
```

Вызов мог бы быть проинтерпретирован или как h(double(1)), или как h(x(1)), и был бы недупустим по правилу единственности. Но превая интерпретация использует только стандартное преобразование и она будет выбрана по правилам, приведеным в #4.6.7.
Правила преобразования не являются ни самыми простыми для реализации и документации, ни наиболее общими из тех, которые можно было бы разработать. Возьмем требование единственности преобразования. Более общий подход разрешил бы компилятору применять любое преобразование, которое он сможет найти; таким образом, не нужно было бы рассматривать все возможные преобразования перед тем, как объявить выражение допустимым. К сожалению, это означало бы, что смысл программы зависит от того, какое преобразование было найдено. В результате смысл программы неким образом зависел бы от порядка описания преобразования. Поскольку они часто находятся в разных исходных файлах (написанных разными людьми), смысл программы будет зависеть от порядка компоновки этих частей вместе. Есть другой вариант - запретить все неявные преобразования. Нет ничего проще, но такое правило приведет либо к неэлегантным пользовательским интефейсам, либо к бурному

росту перегруженных функций, как это было в предыдущем разделе с complex.

Самый общий подход учитывал бы всю имеющуюся информацию о типах и рассматривал бы все возможные преобразования. Например, если использовать предыдущее описание, то можно было бы обработать aa=f(1), так как тип aa определяет едиственность толкования. Если aa является x, то единственное, дающее в результате x, который требеутся присваиванием, - это f(x(1)), а если aa - это y, то вместо этого будет использоваться f(y(1)). Самый общий подход справился бы и с g("asdf"), поскольку единственной интерпретацией этого может быть g(z(x("asdf"))). Сложность этого подхода в том, что он требует расширенного анализа всего выражения для того, чтобы определить интерпретацию каждой операции и вызова функции. Это приведет к замеделению компиляции, а также к вызывающим удивление интерпретациям и сообщениям об ошибках, если компилятор рассмотрит преобразования, определенные в библиотеках и т.п. При таком подходе компилятор будет принимать во внимание больше, чем, как можно ожидать, знает пишущий программу программист!

6.4 Константы

Константы классового типа определить невозможно в том смысле, в каком 1.2 и 12e3 являются константой типа double. Вместо них, однако, часто можно использовать константы основных типов, если их реализация обеспечивается с помощью функций членов. Общий аппарат для этого дают конструкторы, получающие один параметр. Когда конструкторы просты и подставляются inline, имеет смысл рассмотреть в качестве константы вызов конструктора. Если, например, в есть описание класса comlpex, то выражение zz1*3+zz2*comlpex(1,2) даст два вызова функций, а не пять. К двум вызовам функций приведут две операции *, а операция + и конструктор, к которому обращаются для создания comlpex(3) и comlpex(1,2), будут расширены inline.

6.5 Большие Объеты

При каждом применении для comlpex бинарных операций, описанных выше, в функцию, которая реализует операцию, как параметр передается копия каждого операнда. Расходы на копирование каждого double заметны, но с ними вполне можно примириться. К сожалению, не все классы имеют небольшое и удобное представление. Чтобы избежать

ненужного копирования, можно описать функции таким образом, чтобы они получали ссылочные параметры. Например:

```
class matrix {
    double m[4][4];
public:
    matrix();
    friend matrix operator+(matrix&, matrix&);
    friend matrix operator*(matrix&, matrix&);
};
```

Ссылки позволяют использовать выражения, содержашие обычные арифметические операции над большими объектами, без ненужного

копирования. Указатели применять нельзя, потому что невозможно для применения к указателю смысл операции переопределить невозможно. Операцию плюс можно определить так:

```
matrix operator+(matrix&, matrix&);
{
    matrix sum;
    for (int i=0; i<4; i++)
        for (int j=0; j<4; j++)
            sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
    return sum;
}
```

Эта operator+() обращается к операндым + через ссылки, но возвращает значение объекта. Возврат сылки может оказаться более эффективным:

```
class matrix {
    // ...
    friend matrix& operator+(matrix&, matrix&);
    friend matrix& operator*(matrix&, matrix&);
};
```

Это является допустимым, но приводит к сложности с выделением памяти. Поскольку ссылка на результат будет передаваться из функции как ссылка на возвращаетмое значение, оно не может быть автоматической переменной. Поскольку часто операция используется в выражении больше одного раза, результат не может быть и статической переменной. Как правило, его размещают в свободной памяти. Часто копирование возвращаемого значения окаывается дешевле (по времени выполнения, объему кода и объему данных) и проще программируется.

6.6 Присваивание и Инициализация

Рассмотрим очень простой класс строк string:

```
struct string {
    char* p;
    int size;    // размер вектора, на который указывает p

    string(int sz) { p = new char[size=sz]; }
    ~string() { delete p; }
};
```

Строка - это структура данных, состоящая из вектора символов и длины этого вектора. Вектор создается конструктором и уничтожается деструктором. Однако, как показано в #5.10, это может привести к неприятностям. Например:

```
void f()
```

```
{
    string s1(10);
    string s2(20);
    s1 = s2;
}
```

будет размещать два вектора символов, а присваивание s1=s2 будет портить указатель на один из них и дублировать другой. На выходе из f() для s1 и s2 будет вызываться деструктор и уничтожать один и тот же вектор с непредсказуемо разрушительными последствиями. Решение этой проблемы состоит в том, чтобы соответствующим образом определить присваивание объектов типа string:

```
struct string {
    char* p;
    int size;    // размер вектора, на который указывает p

    string(int sz) { p = new char[size=sz]; }
    ~string() { delete p; }
    void operator=(string&)
};

void string::operator=(string& a)
{
    if (this == &a) return;      // остерегаться s=s;
    delete p;
    p=new char[size=a.size];
    strcpy(p,a.p);
}
```

Это определение string гарантирует,и что предыдущий пример будет работать как предполагалось. Однако небольшое изменение f() приведет к появлению той же проблемы в новом облике:

```
void f()
{
    string s1(10);
    s2 = s1;
}
```

Теперь создается только одна строка, а уничтожается две. К неинициализированному объекту определенная пользователем операция присваивания не применяется. Беглый взгляд на string::operator=() объясняет, почему было неразумно так делать: указатель p будет содержать неопределенное и совершенно случайное значение. Часто операция присваивания полагается на то, что ее аргументы инициализированы. Для такой инициализации, как здесь, это не так по определению. Следовательно, нужно определить похожую, но другую, функцию, чтобы обрабатывать инициализацию:

```
struct string {
    char* p;
    int size;    // размер вектора, на который указывает p

    string(int sz) { p = new char[size=sz]; }
    ~string() { delete p; }
    void operator=(string&)
    string(string&);
};

void string::string(string& a)
{
```

```
    p=new char[size=a.size];
    strcpy(p,a.p);
}
```

Для типа X инициализацию тем же типом X обрабатывает конструктор X(X&). Нельзя не подчеркнуть еще раз, что присваивание и инициализация - разные действия. Это особенно существенно при описании деструктора. Если класс X имеет конструктор, выполняющий нетривиальную работу вроде освобождения памяти, то скорее всего потребуется полный комплект функций, чтобы полностью избежать побитового копирования объектов:

```
class X {
    // ...
    X(something);  // конструктор: создаеть объект
    X(&X);         // конструктор: копирует в инициализации
    operator=(X&); // присваивание: чистит и копирует
    ~X();          // деструктор: чистит
};
```

Есть еще два случая, когда объект копируется: как параметр функции и как возвращаемое значение. Когда передается параметр, инициализируется неинициализированная до этого переменная - формальный параметр. Семантика идентична семантике инициализации. То же самое происходит при возврате из функции, хотя это менее очевидно. В обоих случаях будет применен X(X&), если он определен:

```
string g(string arg)
{
    return arg;
}

main()
{
    string s = "asdf";
    s = g(s);
}
```

Ясно, что после вызова g() значение s обязано быть "asdf". Копирование значения s в параметр arg сложности не представляет: для этого надо взывать string(string&). Для взятия копии этого значения из g() требуется еще один вызов string(string&); на этот раз инициализируемой является временная переменная, которая затем

присваивается s. Такие переменные, естественно, уничтожаются как положено с помощью string::~string() при первой возможности.

6.7 Индексирование

Чтобы задать смысл индексов для объектов класса используется функция operator[]. Второй параметр (индекс) функции operator[] может быть любого типа. Это позволяет определять ассоциативные массивы и т.п. В качестве примера давайте перепишем пример из #2.3.10, где при написании небольшой программы для подсчета числа вхождений слов в файле применялся ассоциативный массив. Там использовалась функция. Здесь определяется надлежащий тип ассоциативного массива:

```
struct pair {
    char* name;
    int val;
};

class assoc {
```

```
    pair* vec;
    int max;
    int free;
public:
    assoc(int);
    int& operator[](char*);
    void print_all();
};
```

В assoc хранится вектор пар pair длины max. Индекс первого неиспользованного элемента вектора находится в free. Конструктор выглядит так:

```
assoc::assoc(int s)
{
    max = (s<16) ? s : 16;
    free = 0;
    vec = new pair[max];
}
```

При реализации применяется все тот же простой и неэффективный метод поиска, что использовалася в #2.3.10. Однако при переполнении assoc увеличивается:

```
#include

int assoc::operator[](char* p)
/*
    работа с множеством пар "pair":
    поиск p,
    возврат ссылки на целую часть его "pair"
    делает новую "pair", если p не встречалось
*/
{
    register pair* pp;

    for (pp=&vec[free-1]; vec<=pp; pp--)
        if (strcmp(p,pp->name)==0) return pp->val;

    if (free==max) {    // переполнение: вектор увеличивается
        pair* nvec = new pair[max*2];
        for ( int i=0; iname = new char[strlen(p)+1];
    strcpy(pp->name,p);
    pp->val = 0;     // начальное значение: 0
    return pp->val;
}
```

Поскольку представление assoc скрыто, нам нужен способ его печати. В следующем разделе будет показано, как определить подходящий итератор, а здесь мы используем простую функцию печати:

```
vouid assoc::print_all()
{
    for (int i = 0; i>buf) vec[buf]++;
    vec.print_all();
}
```

6.8 Вызов Функции

Вызов функции, то есть запись выражение(список_выражений), можно проинтерпретировать как бинарную операцию, и операцию вызова можно перегружать так же, как и другие операции. Список параметров

функции operator() вычисляется и проверяется в соответствие с обычнчми правилами передачи параметров. Перегружающая функция может оказаться полезной главным образом для определения типов с единственной операцией и для типов, у которых одна операция настолько преобладает, что другие в большинстве ситуаций можно не принимать во внимание.

Для типа ассоциативного массива assoc мы не определили итератор. Это можно сделать, определив класс assoc_iterator, работа которого состоит в том, чтобы в определенном порядке поставлять элементы из assoc. Итератору нужен доступ к данным, которые хранятся в assoc, поэтому он сделан другом:

```
class assoc {
friend class assoc_iterator;
    pair* vec;
    int max;
    int free;
public:
    assoc(int);
    int& operator[](char*);
};
```

Итератор определяется как

```
class assoc_iterator{
    assoc* cs;  // текущий массив assoc
    int i;      // текущий индекс
public:
    assoc_iterator(assoc& s) { cs = &s; i = 0; }
    pair* operator()()
        { return (ifree)? &cs->vec[i++] : 0; }
};
```

Надо инициализировать assoc_iterator для массива assoc, после чего он будет возвращать указатель на новую pair из этого массива вский раз, когда его будут активизировать операцией (). По достижении конца массива он возвращает 0:

```
main()    // считает вхождения каждого слова во вводе
{
    const MAX = 256;  // больше самого большого слова
    char buf[MAX];
    assoc vec(512);
    while (cin>>buf) vec[buf]++;
    assoc_iterator next(vec);
    pair* p;
    while ( p = next() )
        cout << p->name << ": " << p->val << "\n";
}
```

Итераторный тип вроде этого имеет преимущество перед набором функций, которые выполняют ту же работу: у него есть собственные закрытые данные для хранения хода итерации. К тому же обычно существенно, чтобы одновременно могли работать много итераторов этого типа.

Конечно, такое применение объектов для представления итераторов никак особенно с перегрузкой операций не связано. Многие любят использовать итераторы с такими операциями, как first(), next() и last() (первый, следующий и последний).

6.9 Класс Строка

Вот довольно реалистичный пример класса string. В нем производится учет ссылок на строку с целью минимизировать

копирование и в качестве констант применяются стандартные символьные строки C++.

```
#include
#include

class string {
    struct srep {
        char* s;           // указатель на данные
        int   n;           // счетчик ссылок
};
    srep *p;

public:
    string(char *);        // string x = "abc"
    string();              // string x;
    string(string &);      // string x = string ...
    string& operator=(char *);
    string& operator=(string &);
    ~string();
    char& operator[](int i);

    friend ostream& operator<<(ostream&, string&);
    friend istream& operator>>(istream&, string&);

    friend int operator==(string& x, char* s)
        {return strcmp(x.p->s, s) == 0; }

    friend int operator==(string& x, string& y)
        {return strcmp(x.p->s, y.p->s) == 0; }

    friend int operator!=(string& x, char* s)
        {return strcmp(x.p->s, s) != 0; }

    friend int operator!=(string& x, string& y)
        {return strcmp(x.p->s, y.p->s) != 0; }

};
```

Конструкторы и деструкторы просты (как обычно):

```
string::string()
{
    p = new srep;
    p->s = 0;
    p->n = 1;
}

string::string(char* s)
{
    p = new srep;
    p->s = new char[ strlen(s)+1 ];
    strcpy(p->s, s);
    p->n = 1;
}

string::string(string& x)
{
    x.p->n++;
    p = x.p;
}

string::~string()
{
```

```
    if (--p->n == 0) {
        delete p->s;
        delete p;
    }
}
```

Как обычно, операции присваивания очень похожи на конструкторы. Они должны обрабатывать очистку своего первого (левого) операнда:

```
string& string::operator=(char* s)
{
    if (p->n > 1) {    // разъединить себя
        p-n--;
        p = new srep;
    }
    else if (p->n == 1)
        delete p->s;

    p->s = new char[ strlen(s)+1 ];
    strcpy(p->s, s);
    p->n = 1;
    return *this;
}
```

Благоразумно обеспечить, чтобы присваивание объекта самому себе работало правилньо:

```
string& string::operator=(string& x)
{
    x.p->n++;
    if (--p->n == 0) {
        delete p->s;
        delete p;
    }
    p = x.p;
    return *this;
}
```

Операция вывода задумана так, чтобы продемонстрировать применение учета ссылок. Она повторяет каждую вводимую строку (с помощью операции <<, которая определяется позднее):

```
ostream& operator<<(ostream& s, string& x)
{
    return s << x.p->s << " [" << x.p->n << "]\n";
}
```

Операция ввода использует стандартную функцию ввода символьной строки (#8.4.1).

```
istream& operator>>(istream& s, string& x)
{
    char buf[256];
    s >> buf;
    x = buf;
    cout << "echo: " << x << "\n";
    return s;
}
```

Для доступа к отдельным символам предоставлена операция индексирования. Осуществляется проверка индекса:

```
void error(char* p)
{
```

```
    cerr << p << "\n";
    exit(1);
}

char& string::operator[](int i)
{
    if (i<0 || strlen(p->s)s[i];
}
```

Головная программа просто немного опреобует действия над строками. Она читает слова со ввода в строки, а потом эти строки печатает. Она продолжает это делать до тех пор, пока не распознает строку done, которая завершает сохранение слов в строках, или не встретит конец файла. После этого она печатает строки в обратном порядке и завершается.

```
main()
{
    string x[100];
    int n;

    cout << "отсюда начнем\n";
    for (n = 0; cin>>x[n]; n++) {
        string y;
        if (n==100) error("слишком много строк");
        cout << (y = x[n]);
        if (y=="done") break;
    }
    cout << "отсюда мы пройдем обратно\n";
    for (int i=n-1; 0<=i; i--) cout << x[i];
}
```

6.10 Друзья и Члены

Теперь, наконец, можно обсудить, в каких случаях для доступа к закрытой части определяемого пользователем типа использовать члены, а в каких - друзей. Некоторые операции должны быть членами: конструкторы, деструкторы и виртуальные функции (см. следующую главу), но обчно это зависит от выбора.

Рассмотрим простой класс X:

```
class X {
    // ...
    X(int);
    int m();
    friend int f(X&);
};
```

Внешне не видно никаких причин делать f(X&) другом дополнительно к члену X::m() (или наоборот), чтобы реализовать действия над классом X. Однако член X::m() можно вызывать только для "настоящего объекта", в то время как друг f() может вызываться для объекта, созданного с помощью неявного преобразования типа. Например:

```
void g()
{
    1.m();      // ошибка
    f(1);       // f(x(1));
}
```

Поэтому операция, изменяющее состояние объекта, должно быть членом, а не другом. Для определяемых пользователем типов операции, требующие в случае фундаментальных типов операнд lvalue (=, *=, ++ и т.д.), наиболее естественно определяются как члены.

И наоборот, если нужно иметь неявное преобразование для всех операндов операции, то реализующая ее функция должна быть другом, а не членом. Это часто имеет место для функций, которые реализуют операции, не требующие при применении к фуныаментальным типам lvalue в качестве операндов (+, -, || и т.д.).

Если никакие преобразования типа не определены, то оказывается, что нет никаких существенных оснований в пользу члена, если есть друг, который получает ссылочный параметр, и наоборот. В некоторых случаях программист может предпочитать один синтаксис вызова другому. Например, оказывается, что большинство предпочитает для обращения матрицы m запись m.inv(). Конечно, если inv() действительно обращает матрицу m, а не просто возвращает новую матрицу, обратную m, ей следут быть другом.

При прочих равных условиях выбирайте, чтобы функция была членом: никто не знает, вдруг когда-нибудь кто-то определит операцию преобразования. Невозможно предсказать, потребуют ли будущие изменения изменить статус объекта. Синтаксис вызова функции члена ясно указывает пользователю, что объект можно изменить; ссылочный параметр является далеко не столь очевидным. Кроме того, выражения в члене могут быть заметно короче выражений в друге. В функции друге надо использовать явный параметр, тогда как в члене можно использовать неявный this. Если только не применяется перегрузка, имена членов обычно короче имен друзей.

6.11 Предостережение

Как и большая часть возможностей в языках программирования, перегрузка операций может применяться как правильно, так и неправильно. В частности, можно так воспользоваться возможность определять новые значения старых операций, что они станут почти совсем непостижимы. Представьте, например, с какими сложностями столкнется человек, читающий программу, в которой операция + была переопределена для обозначения вычитания.

Данный аппарат должен уберечь программиста/читателя от худших крайностей применения перегрузки, потому что программист предохранен от изменения значения операций для основных типов данных вроде int, а также потому, что синтаксис выражений и приоритеты операций сохраняются.

Может быть. разумно применять перегрузку операций главным образом так, чтобы подражать общепринятому применению операций. В тех случаях, когда нет общепринятой операции или имеющееся в C++ множество операций не подходит для имитации общепринятого применения, можно использовать запись вызова функции.

6.12 Упражнения

1. (*2) Определите итератор для класса string. Определите операцию конкатенации + и операцию "добавить в конец" +=. Какие еще операции над string вы хотели бы осуществлять?
2. (*1.5) Задайте с помощью перегрузки () операцию выделения подстроки для класса строк.
3. (*3) Постройте класс string так, чтобы операция выделения подстроки могла использоваться в левой части присваивания. Напишите сначала версию, в которой строка может присваиваться подстроке той же длины, а потом версию, где эти длины могут быть разными.
4. (*2) Постройте класс string так, чтобы для присваивания, передачи параметров и т.п. он имел семантику по значению, то

есть в тех случаях, когда копируется строковое представление, а не просто управляющая структура данных класса sring.

5. (*3) Модифицируйте класс string из предыдущего примера таким образом, чтобы строка копировалась только когда это необходимо. То есть, храните совместно используемое представление двух строк, пока одна из этих строк не будет изменена. Не пытайтесь одновременно с этим иметь операцию выделения подстроки, которая может использоваться в левой части.
6. (*4) Разработайте класс string с семантикой по значению, копированием с задержкой и операцией подстроки, которая может стоять в левой части.
7. (*2) Какие преобразования используются в каждом выражении следующей программы:

```
struct X {
   int i;
   X(int);
   operator+(int);
};

struct Y {
   int i;
   Y(X);
   operator+(X);
   operator int();
};

X operator* (X,Y);
int f(X);

X x = 1;
Y y = x;
int i = 2;

main()
{
   i + 10;
   y + 10;
   y + 10 * y;
   x + y + i;
   x * x + i;
   f(7);
   f(y);
   y + y;
   106 + y;
}
```

   Определите X и Y так, чтобы они оба были целыми типами. Измените программу так, чтобы она работала и печатала значения всех допустимых выражений.
8. (*2) Определите класс INT, который ведет себя в точности как int. Подсказка: определите INT::operator int().
9. (*1) Определите класс RINT, который ведет себя в точности как int за исключением того, что единственные возможные операции - это + (унарный и бинарный), - (унарный и бинарный), *, /, %. Подсказка: не определяйте $ (R?)INT::operator int().
10. (*3) Определите класс LINT, ведущий себя как RINT за исключением того, что имеет точность не менее 64 бит.
11. (*4) Определите класс, который реализует арифметику с произвольной точностью. Подсказка: вам надо управлять памятью аналогично тому, как это делалось для класса string.
12. (*2) Напишите программу, доведенную до нечитаемого состояния с помощью макросов и перегрузки операций. Вот идея: определите для INT + чтобы он означал - и наоборот, а потом с помощью

макроопределения определите int как INT. Переопределение часто употребляемых функций, использование параметров ссылочного типа и несколько вводящих в заблуждение комментариев помогут устроить полную неразбериху.

13. (*3) Поменяйтесь со своим другом программами, которые у вас получились в предыдущем упражнении. Не запуская ее попытайтесь понять, что делает программа вашего друга. После выполнения этого упражнения вы будете знать, чего следует избегать.
14. (*2) Перепишите примеры с comlpex (#6.3.1), tiny (#6.3.2) и string (#6.9) не используя friend функций. Используйте только функции члены. Протестируйте каждую из новых версий. Сравните их с версиями, в которых используются функции друзья. Еще раз посмотрите Упражнение 5.3.
15. (*2) Определите тип vec4 как вектор их четырех float. Определите operator[] для vec4. Определите операции +, -, *, /, =, +=, -=, *=, /= для сочетаний векторов и чисел с плавающей точкой.
16. (*3) Определите класс mat4 как вектор из четырех vec4. Определите для mat4 operator[], возвращающий vec4. Определите для этого типа обычные операции над матрицами. Определите функцию, которая производит исключение Гаусса для mat4.
17. (*2) Определите класс vector, аналогичный vec4, но с длиной, которая задается как параметр конструктора `vector::vector(int)`.
18. (*3) Определите класс matrix, аналогичный mat4, но с размерностью, задаемой параметрами конструктора `matrix::matrix(int,int)`.

Глава 7

Производные Классы

Не надо размножать объекты без необходимости
- У. Оккам

В этой главе описывается понятие производного класса в C++. Производные классы дают простой, гибкий и эффективный аппарат задания для класса альтернативного интерфейса и определения класса посредством добавления возможностей к уже имеющемуся классу без перепрограммирования или перекомпиляции. С помощью производных классов можно также обеспечить общий интерфейс для нескольких различных классов так, чтобы другие части программы могли работать с объектами этих классов одинаковым образом. При этом обычно в каждый объект помещается информация о типе, чтобы эти объекты могли обрабатываться соответствующим образом в ситуациях, когда их тип нельзя узнать во время компиляции. Для элегантиной и надежной обработки таких динамических зависимостей типов имеется понятие виртуальной функции. По своей сути производные классы существуют для того, чтобы облегчить программисту формулировку общности.

7.1 Введение

Представим себе процесс написания некоторого средства общего назначения (например, тип связанный список, таблица имен или планировщик для системы моделирования), кооторое предназначается для использоания многими разными людьми в различных обстоятельствах. Очевидно, что в кандидатах на роль таких средств недостатка нет, и выгоды от их стандартизации огромны. Кажется, любой опытный программист написал (и отладил) дюжину вариантов типов множества, таблицы имен, сортирующей функции и т.п., но оказывается, что таблиц имен каждый программист и каждая программа используют свою версию этих понятий, из-за чего программы слишком трудно читать, тяжело отлаживать и сложно модифицировать. Более того, в большой программе вполне может быть несколько копий идентичных (почти) частей кода для работы с такими фундаментальными

понятиями.

Причина этого хаоса частично состоит в том, что представить такие общие понятия в языке программирования сложно с концептульной точки зрения, а частично в том, что средства, обладающие достаточной общностью, налагают дополнительные расходы по памяти и/или по времени, что делает их неудобными для самых простых и наиболее напряженно используемых средств (связанные списки, вектора и т.п.), где они были бы наиболее полезны. Понятие производного класса в C++, описываемое в #7.2, не обеспечивают общего решения всех этих проблем, но оно дает способ справляться с довольно небольшим числом важных случаев. Будет, например, показано, как определить эффективный класс общего связанного списка таким образом, чтобы все его версии использовали код совместно.

Написание общецелевых средств - задача непростая, и часто основной акцент в их разработке другой, чем при разработке программ специального назначения. Конечно, нет четкой границы между средствами общего и специального назначения, и к методам и языковым

средствам, которые описываются в этой главе, можно относиться так, что они становятся все более полезны с ростом объема и сложности создаваемых программ.

7.2 Производные Классы

Чтобы разделить задачи понимания аппарата языка и методов его применения, знакомство с понятием производных классов делается в три этапа. Вначале с помощью небольших примеров, которые не надо воспринимать как реалистичные, будут описаны сами средства языка (заись и семантика). После этого демонстрируются некоторые неочевидные применения производных классов, и, наконец, приводится законченная программа.

7.2.1 Построение Производного Класса

Рассмотрим построение программы, которая имеет дело с людьми, служащими в некоторой фирме. Структура данных в этой программе может быть например такой:

```
struct employee {           // служащий
    char*    name;         // имя
    short    age;          // возраст
    short    department;   // подразделение
    int      salary;       //
    employee* next;
    // ...
};
```

Список аналогичных служащих будет свзяваться через поле next. Теперь давайте определим менеджера:

```
struct manager {            // менеджер
    employee emp;           // запись о менеджере как о служащем
    employee* group;        // подчиненные люди
    // ...
};
```

Менеджер также является служащим; относящиеся к служащему employee данные хранятся в члене emp объекта manager. Для читающего это человека это, может быть, очевидно, но нет ничего выделяющего член emp для компилятора. Указатель на менеджера (manager*) не является указателем на служащего (employee*), поэтому просто использовать один там, где требуется другой, нельзя. В частности, нельзя поместить менеджера в список служащих, не написав для этого специальную программу. Можно либо применить к manager* явное преобразование типа, либо поместить в список служащих адрес члена

emp, но и то и другое мало элегантно и довольно неясно. Корректный подход состоит в том, чтобы установить, что менеджер является служащим с некоторой добавочной информацией:

```
struct manager : employee {
    employee* group;
    // ...
};
```

manager является производным от employee и, обратно, employee есть базовый класс для manager. Класс manager дополнительно к члену group имеет члены класса employee (name, age и т.д.).

Имея определения employee и manager мы можем теперь создать список служащих, некоторые из которых являются менеджерами. Например:

```
void f()
{
    manager m1, m2;
    employee e1, e2;
    employee* elist;
    elist = &m1;     // поместить m1, e1, m2 и e2 в elist
    m1.next = &e1;
    e1.next = &m2;
    m2.next = &e2;
    e2.next = 0;
}
```

Поскольку менеджер является служащим, manager* может использоваться как employee*. Однако служащий необязательно является менеджером, поэтому использовать employee* как manager* нельзя.

7.2.2 Функции Члены

Просто структуры данных вроде employee и manager на самом деле не столь интересны и часто не особенно полезны, поэтому рассмотрим, как добавить к ним функции. Например:

```
class employee {
    char* name;
    // ...
public:
    employee* next;
    void print();
    // ...
};

class manager : public employee {
    // ...
public:
    void print();
    // ...
};
```

Надо ответить на некоторые вопросы. Как может функция член производного класса manager использовать члены его базового класса employee? Как члены базового класса employee могут использовать функции члены производного класса manager? Какие члены базового класса employee может использовать функция не член на объекте типа

manager? Каким образом программист может повлиять на ответы на эти вопросы, чтобы удовлетворить требованиям приложения?

Рассмотрим:

```
void manager::print()
{
    cout << " имя " << name << "\n";
    // ...
}
```

Член производного класса может использовать открытое имя из своего базового класса так же, как это могут делать другие члены последнего, то есть без указания объекта. Предполагается, что на объект указывает this, поэтому (корректной) ссылкой на имя name является this->name. Однако функция manager::print компилироваться не будет, член производного класса не имеет никакого особого права доступа к закрытым членам его базового класса, поэтому для нее name недоступно.

Это многим покажется удивительным, но представьте себе другой вариант: что функция член могла бы обращаться к закрытым членам своего базового класса. Возможность, позволяющая программисту получать доступ к закрытой части класса просто с помощью вывода из него другого класса, лишила бы понятие закрытого члена всякого смысла. Более того, нельзя было бы узнать все использования закрытого имени посмотрев на функции, описанные как члены и друзья этого класса. Пришлось бы проверять каждый исходный файл во всей программе на наличие в нем производных классов, потом исследовать каждую функцию этих классов, потом искать все классы, производные от этих классов, и т.д. Это по меньшей мере утомительно и скорее всего нереально.

С другой стороны, можно ведь использовать механизм friend, чтобы предоставить такой доступ или отдельным функциям, или всем функциям отдельного класса (как описывается в #5.3). Например:

```
class employee {
    friend void manager::print();
    // ...
};
```

решило бы проблему с manager::print(), и

```
class employee {
friend class manager;
    // ...
};
```

сделало бы доступным каждый член employee для всех функций класса manager. В частности, это сделает name доступным для manager::print().

Другое, иногда более прозрачное решение для производного класса,- использовать только открытые члены его базового класса. Например:

```
void manager::print()
{
    employee::print();    // печатает информацию о служащем
    // ...                // печатает информацию о  менеджере
}
```

Заметьте, что надо использовать ::, потому что print() была переопределена в manager. Такое повторное использование имен типично. Неосторожный мог бы написать так:

```
void manager::print()
{
    print();              // печатает информацию о служащем
    // ...                // печатает информацию о  менеджере
```

```
}
```

и обнаружить, что программа после вызова manager::print() неожиданно попадает в последовательность рекурсивных вызовов.

7.2.3 Видимость

Класс employee стал открытым (public) базовым классом класса manager в результате описания:

```
class manager : public employee {
    // ...
};
```

Это означает, что открытый член класса employee является также и открытым членом класса manager. Например:

```
void clear(manager* p)
{
    p->next = 0;
}
```

будет компилироваться, так как next - открытый член и employee и manager'а. Альтернатива - можно определить закрытый (private) класс, просто опустив в описании класса слово public:

```
class manager : employee {
    // ...
};
```

Это означает, что открытый член класса employee является закрытым членом класса manager. То есть, функции члены класса manager могут как и раньше использовать открытые члены класса employee, но для пользователей класса manager эти члены недоступны. В частности, при таком описании класса manager функция clear() компилироваться не будет. Друзья производного класса имеют к членам базового класса такой же доступ, как и функции члены.

Поскольку, как оказывается, описание открытых базовых классов встречается чаще описания закрытых, жалко, что описание открытого базового класса длиннее описания закрытого. Это, кроме того, служит источником запутывающих ошибок у начинающих.

Когда описывается производная struct, ее базовый класс по умолчанию является public базовым классом. То есть,

```
struct D : B { ...
```

означает

```
class D : public B { public: ...
```

Отсюда следует, что если вы не сочли полезным то скрытие данных, которое дают class, public и friend, вы можете просто не использовать эти ключевые слова и придерживаться struct. Такие средства языка, как функции члены, конструкторы и перегрузка операций, не зависят от механизма скрытия данных.

Можно также объявить некоторые, но не все, открытые $ члены базового класса открытыми членами производного класса. Например:

```
class manager : employee {
    // ...
public:
    // ...
    employee::name;
```

```
    employee::department;
};
```

Запись

```
имя_класса :: имя_члена ;
```

не вводит новый член, а просто делает открытый член базового класса открытым для производного класса. Теперь name и department могут использоваться для manager'a, а salary и age - нет. Естественно, сделать сделать закрытый член базового класса открытым членом производного класса невозможно. Невозможно с помощью этой записи также сделать открытыми перегруженные имена.

Подытоживая, можно сказать, что вместе с предоставлением средств дополнительно к имющимся в базовом классе, производный класс можно использовать для того, чтобы сделать средства (имена) недоступными для пользователя. Другими словами, с помощью производного класса можно обеспечивать прозрачный, полупрозрачный и непрозрачный доступ к его базовому классу.

7.2.4 Указатели

Если производный класс derived имеет открытый базовый класс base, то указатель на derived можно присваивать переменной типа указатель на base не используя явное преобразование типа. Обратное преобразование, указателя на base в указатель на derived, должно быть явным. Например:

```
class base { /* ... */ };
class derived : public base { /* ... */ };

derived m;
base* pb = &m;     // неявное преобразование
derived* pd = pb;  // ошибка: base* не является derived*
pd = (derived*)pb; // явное преобразование
```

Иначе говоря, объект производного класса при работе с ним через указател иможно рассматривать как объект его базового класса. Обратное неверно.

Будь base закрытым базовым классом класса derived, неявное преобразование derived* в base* не делалось бы. Неявное преобразование не может в этом случае быть выполнено, потому что к открытому члкну класса base можно обращаться через указатель на base, но нельзя через указатель на derived:

```
class base {
    int m1;
public:
    int m2;     // m2 - открытый член base
};

class derived : base {
    // m2 НЕ открытый член derived
};

derived d;
d.m2 = 2;       // ошибка: m2 из закрытой части класса
base* pb = &d;  // ошибка: (закрытый base)
pb->m2 = 2;     // ok
pb = (base*)&d; // ok: явное преобразование
pb->m2 = 2;     // ok
```

Помимо всего прочего, этот пример показывает, что используя явное приведение к типу можно сломать правила защиты. Ясно, делать это не

рекомендуется, и это приносит программисту заслуженую "награду". К несчасть , недисциплинированное использование явного преобразования может создать адские условия для невинных жертв, которые эксплуатируют программу, где это делается. Но, к счастью, нет способа воспользоваться приведением для получения доступа к закрытому имени m1. Закрытый член класса может использоваться только членами и друзьями этого класса.

7.2.5 Иерархия Типов

Производный класс сам может быть базовым классом. Например:

```
class employee { ... };
class secretary : employee { ... };
class manager : employee { ... };
class temporary : employee { ... };
class consultant : temporary { ... };
class director : manager { ... };
class vice_president : manager { ... };
class president : vice_president { ... };
```

Такое множество родственных классов принято называть иерархией классов. Поскольку можно выводить класс только из одного базового класса, такая иерархия является деревом и не может быть графом более общей структуры. Например:

```
class temporary { ... };
class employee { ... };
class secretary : employee { ... };

// не C++:
class temporary_secretary : temporary : secretary { ... };
class consultant : temporary : employee { ... };
```

И этот факт вызывает сожеление, потому что направленный ациклический граф производных классов был бы очень полезен. Такие структуры описать нельзя, но можно смоделировать с помощью членов соответствующий типов. Например:

```
class temporary { ... };
class employee { ... };
class secretary : employee { ... };

// Альтернатива:
class temporary_secretary : secretary
{ temporary temp; ... };
class consultant : employee
{ temporary temp; ... };
```

Это выглядит неэлегантно и страдает как раз от тех проблем, для преодоления которыз были изобретены производные классы. Например, поскольку consultant не является производным от temporary, consultant'а нельзя помещать с список временных служащих (temporary employee), не написав специальной программы. Однако во многих полезных программах этот метод успешно используется.

7.2.6 Конструкторы и Деструкторы

Для некоторых производных классов нужны конструкторы. Если у базового класса есть конструктор, он должен вызываться, и если для этого конструктора нужны параметры, их надо предоставить. Например:

```
class base {
    // ...
public:
    base(char* n, short t);
    ~base();
};

class derived : public base {
    base m;
public:
    derived(char* n);
    ~derived();
};
```

Параметры конструктора базового класса специфицируются в определении конструктора производного класса. В этом смысле базовый класс работакт точно также, как неименованный член производного класса (см. #5.5.4). Например:

```
derived::derived(char* n) : (n,10), m("member",123)
{
    // ...
}
```

Объекты класса конструируются снизу вверх: сначала базовый, потом члены, а потом сам производный класс. Уничтожаются они в обратном порядке: сначала сам производный класс, потом члены а потом базовый.

7.2.7 Поля Типа

Чтобы использовать производные классы не просто как удобную сокращенную запись в описаниях, надо разрешить следующую проблему: Если задан указатель типа base*, какому производному типу в действительности принадлежит указываемый объект? Есть три осповных способа решения этой проблемы:

[1] Обеспечить, чтобы всегда указывались только объекты одного типа (#7.3.3);

[2] Поместить в базовый класс поле типа, которое смогут просматривать функции; и

[3] Использовать виртуальные функции (#7.2.8).

Обыкновенно указатели на базовые классы используются при разработке контейнерных (или вмещающих) классов: множество, вектор, список и т.п. В этом случае решение 1 дает однородные списки, то есть списки объектов одного типа. Решения 2 и 3 можно использовать для построения неоднородных списков, то есть списков объектов (указателей на объекты) нескольких различных типов. Решение 3 - это специальный вариант решения 2, безопасный относительно типа.

Давайте сначала исследуем простое решение с помощью поля типа, то есть решение 2. Пример со служащими и менеджерами можно было бы переопределить так:

```
enum empl_type { M, E };

struct employee {
    empl_type type;
    employee* next;
    char*     name;
    short     department;
    // ...
};

struct manager : employee {
```

```
    employee* group;
    short     level;        // уровень
};
```

Имея это, мы можем теперь написать функцию, которая печатает информацию о каждом служащем:

```
void print_employee(employee* e)
{
    switch (e->type) {
    case E:
        cout << e->name << "\t" << e->department << "\n";
        // ...
        break;
    case M:
        cout << e->name << "\t" << e->department << "\n";
        // ...
        manager* p = (manager*)e;
        cout << " уровень " << p->level << "\n";
        // ...
        break;
    }
}
```

и воспользоваться ею для того, чтобы напечатать список служащих:

```
void f()
{
    for (; ll; ll=ll->next) print_employee(ll);
}
```

Это прекрасно работает,особенно в небольшой программе, написанной одним человеком, но имеет тот коренной недостаток, что неконтролируемым компилятором образом зависит от того, как программист работает с типами. В больших программах это обычно приводит к ошибкам двух видов. Первый - это невыполнение проверки поля типа, второй - когда не все случаи case помещаются в переключатель switch как в предыдущем примере. Оба избежать достаточно легко , когда программу сначала пишут на буммге $, но при модификации нетривиальной программы, особенно написанной другим человеком, очень трудно избежать и того, и другого. Часто от этих сложностей становится труднее уберечься из-за того, что функции вроде print() часто бывают организованы так, чтобы ползоваться общность классов, с которыми они работают. Например:

```
void print_employee(employee* e)
{
    cout << e->name << "\t" << e->department << "\n";
    // ...
    if (e->type == M)  {
        manager* p = (manager*)e;
        cout << " уровень " << p->level << "\n";
        // ...
    }
}
```

Отыскание всех таких операторов if, скрытых внутри большой функции, которая работает с бошим числом производных классов, может оказаться сложной задачей, и даже когда все они найдены, бывает нелегко понять, что же в них делается.

7.2.8 Виртуальные Функции

Виртуальные функции преодолевают сложности решения с помощью

полей типа, позволяя программисту описывать в базовом классе функции, которые можно переопределять в любом производном классе. Компилятор и загрузчик обеспечивают правильное соответствие между объектами и применяемыми к ним функциями. Например:

```
struct employee {
    employee* next;
    char*     name;
    short     department;
    // ...
    virtual void print();
};
```

Ключевое слово virtual указывает, что могут быть различные варианты функции print() для разных производных классов, и что поиск среди них подходящей для кажог вызова print() является задачей компилятора. Тип функции описывается в базовом классе и не может переописываться в производном классе. Виртуальная функция должна быть определена для класса, в котором она описана впервые. Например:

```
void employee::print()
{
    cout << e->name << "\t" << e->department << "\n";
    // ...
}
```

Виртуальная функция может, таким образом, использоваться даже в том случае, когда нет производных классов от ее класса, и в производном классе, в котором не нужен специальный вариант виртуальной функции, ее задавать не обязательно. Просто при выводе класса соответствующая функция задается в том случае, если она нужна. Например:

```
struct manager : employee {
    employee* group;
    short     level;
    // ...
    void print();
};

void manager::print()
{
    employee::print();
    cout << "\tуровень" << level << "\n";
    // ...
}
```

Функция print_employee() теперь не нужна, поскольку ее место заняли функции члены print(), и теперь со списком служащих можно работать так:

```
void f(employee* ll)
{
    for (; ll; ll=ll->next) ll->print();
}
```

Каждый служащий будет печататься в соответствии с его типом. Например:

```
main()
{
    employee e;
        e.name = "Дж.Браун";
```

```
        e.department = 1234;
        e.next = 0;
    manager m;
        m.name = "Дж.Смит";
        e.department = 1234;
        m.level = 2;
        m.next = &e;
    f(&m);
}
```

выдаст

```
Дж.Смит 1234
        уровень 2
Дж.Браун 1234
```

Заметьте, что это будет работать даже в том случае, если f() была написана и откомпилирована еще до того, как производный класс manager был задуман! Очевидно, при реализации этого в каждом объекте класса employee сохраняется некоторая информация о типе. Занимаемого для этого пространства (в текущей реализации) как раз хватает для хранения указателя. Это пространство занимается только в объектах классов с виртуальными функциями, а не во всех объектах классов и даже не во всех объектах производных классов. Вы платите эту пошлину только за те классы, для которых описали виртуальные функции.

Вызов функции с помощью операции разрешения области видимости ::, как это делается в manager::print(), гарантирует, что межанизм фиртуальных функций применяться не будет. Иначе manager::print() подвергалось бы бесконечной рекурсии. Применение уточненного имени имеет еще один эффект, который может оказаться полезным: если описанная как virtual функция описана еще и как inline (в чем ничего необычного нет), то там, где в вызове применяется :: может применяться inline-подстановка. Это дает программисту эффективный способ справляться с теми важными специальными случаями, когда одна виртуальная функция вызывает другую для того же объекта. Поскольку тип объекта был определен при вызове первой виртуальной функции, обычно его не надо снова динамически определять другом вызове для того же объекта.

7.3 Альтернативные Интерфейсы

После того, как описаны средства языка, которые относятся к производным классам, обсуждение снова может вернуться к стоящим задачам. В классах, которые описываются в этом рахделе, основополанающая идея состоит в том, что они однажды написаны, а потом их используют программисты, которые не могут изменить их определение. Физически классы состоят из одного или более фаголовочных файлов, определяющих интерфейс, и одного или более файлов, определяющих реализацию. Заголовочные файлы будут помещены куда-то туда, откуда пользователь может взять их копии с помощью директивы #include. Файлы, определяющие реализацию, обычно компилируют и помещают в библиотеку.

7.3.1 Интерфейс

Рассмотрим такое написание класса slist для однократно сязанного списка, с помощью которого можно создавать как однородные, так и неоднородные списки объектов тех типов, которые еще должны быть определены. Сначала мы определим тип ent:

```
typedef void* ent;
```

Точная сущность типа ent несущественна, но нужно, чтобы в нем мог храниться указатель. Тогда мы определим тип slink:

```
class slink {
friend class slist;
friend class slist_iterator;
    slink* next;
    ent e;
    slink(ent a, slink* p) { e=a; next=p;}
};
```

В одном звене может храниться один ent, и с помощью него реализуется класс slist:

```
class slist {
friend class slist_iterator;
    slink* last;        // last->next - голова списка
public:
    int insert(ent a);  // добавить в голову списка
    int append(ent a);  // добавить в хвост списка
    ent get();          // вернуться и убрать голову списка
    void clear();       // убрать все звенья

    slist()      { last=0; }
    slist(ent a) { last=new slink(a,0); last->next=last; }
    ~slist()     { clear(); }
};
```

Хотя список очевидным образом реализуется какк связанный список, реализацию можно изменить так, чтобы использовался вектор из ent'ов, не повлияв при этом на пользователей. То есть, применение slink'ов никак не видно в описаниях открытых функций slist'ов, а видно только в закрытой части и определениях функций.

7.3.2 Реализация

Реализующие slist функции в основном просты. Единственная настоящая сложность - что делать в случае ошибки, если, например, пользователь попытается get() что-нибудь из пустого списка. Мы обсудим это в #7.3.4. Здесь приводятся определения членов slist. Обратите внимание, как хранение указателя на последний элемент кругового списка дает возможность просто реализовать оба действия append() и insert():

```
int slist::insert(ent a)
{
    if (last)
        last->next = new slink(a,last->next);
    else {
        last = new slink(a,0);
        last->next = last;
    }
    return 0;
}

int slist::append(ent a)
{
    if (last)
        last = last->next = new slink(a,last->next);
    else {
        last = new slink(a,0);
```

```
        last->next = last;
    }
    return 0;
}

ent slist::get()
{
    if (last == 0) slist_handler("get fromempty list");
                                // взять из пустого списка
    slink* f = last->next;
    ent r  f->e;
    if (f == last)
        last = 0;
    else
        last->next = f->next;
    delete  f;
    return f;
}
```

Обратите внимание, как вызывается slist_handler (его описание можно найти в #7.3.4). Этот указатель на имя функции используется точно так же, как если бы он был именем функции. Это является краткой формой более явной записи вызова:

```
(*slist_handler)("get fromempty list");
```

И slist::clear(), наконец, удаляет из списка все элементы:

```
void slist::clear()
{
    slink* l = last;
    if (l == 0) return;
    do {
        slink* ll = l;
        l = l->next;

        delete ll;
    } while (l!=last);
}
```

Класс slist не обеспечивает способа заглянуть в список, но только средства для вставления и удаления элементов. Однако оба класса, и slist, и slink, описывают класс slist_iterator как друга, поэтому мы можем описать подходящий итератор. Вот один, написанный в духе #6.8:

```
class slist_iterator {
    slink* ce;
    slist* cs;
public:
    slist_iterator(slist& s) { cs = &s; ce = cs->last; }

    ent operator()() {
        // для индикации конца итерации возвращает 0
        // для всех типов не идеален, хорош для указателей
        ent ret = ce ? (ce=ce->next)->e : 0;
        if (ce == cs->last) ce= 0;
        return ret;
    }
};
```

7.3.3 Как Этим Пользоваться

Фактически класс slist в написанном виде бесполезен. В конечном

счете, зачем можно использовать список указателей void*? Штука в том, чтобы вывести класс из slist и получить список тех объектов, которые представляют интерес в конкретной программе. Представим компилятор языка вроде C++. В нем широко будут использоваться списки имен; имя - это нечто вроде

```
struct name {
    char* string;
    // ...
};
```

В список будут помещаться указатели на имена, а не сами объекты имена. Это позволяет использовать небольшое информационное поле е slist'a, и дает возможность имени находиться одновременно более чем в одном списке. Вот определение класса nlist, который очень просто выводится из класса slist:

```
#include "slist.h"
#include "name.h"

struct nlist : slist {
    void insert(name* a) { slist::insert(a); }
    void append(name* a) { slist::append(a); }
    name* get()          {}
    nlist(name* a) : (a) {}
};
```

Функции нового класса или наследуются от slist непосредственно, или ничего не делают кроме преобразования типа. Класс nlist - это ничто иное, как альтернативный интерфейс класса slist. Так как на

самом деле тип ent есть void*, нет необходимости явно преобразовывать указатели name*, которые используются в качестве фактических параметров (#2.3.4).

Списки имен можно использовать в классе, который представляет определение класса:

```
struct classdef {
    nlist friends;
    nlist constructors;
    nlist destructors;
    nlist members;
    nlist operators;
    nlist virtuals;
    // ...
    void add_name(name*);
    classdef();
    ~classdef();
};
```

и имена могут добавляться к этим спискам приблизительно так:

```
void classdef::add_name(name* n)
{
    if (n->is_friend()) {
        if (find(&friends,n))
            error("friend redeclared");
        else if (find(&members,n))
            error("friend redeclared as member");
        else
            friends.append(n);
    }
    if (n->is_operator()) operators.append(n);
    // ...
```

```
}
```

где is_iterator() и is_friend() являются функциями членами класса name. Фукнцию find() можно написать так:

```
int find(nlist* ll, name* n)
{
    slist_iterator ff(*(slist*)ll);
    ent p;
    while ( p=ff() ) if (p==n) return 1;
    return 0;
}
```

Здесь применяется явное преобразование типа, чтобы применить slist_iterator к nlist. Более хорошее решение,- сделать итератор для nlist'ов, приведено в #7.3.5. Печатать nlist может, например, такая функция:

```
void print_list(nlist* ll, char* list_name)
{
    slist_iterator count(*(slist*)ll);
    name* p;
    int n = 0;
    while ( count() ) n++;
    cout << list_name << "\n" << n << "members\n";
    slist_iterator print(*(slist*)ll);
    while ( p=(name*)print() ) cout << p->string << "\n";
}
```

7.3.4 Обработка Ошибок

Есть четыре подхода к проблеме, что же делать, когда во время выполнения общецелевое средство вроде slist сталкивается с ошибкой (в C++ нет никаких специальных средств языка для обработке ошибок):

[1] Возвращать недопустимое значение и позволить пользователю его проверять;
[2] Возвращать дополнительное значение состояния и разрешить пользователю проверять его;
[3] Вызывать функцию ошибок, заданную как часть класса slist; или
[4] Вызывать функцию ошибок, которую предположительно предоставляет пользователь.

Для небольшой программы, написанной ее единственным пользователем, нет фактически никаких особенных причин предпочесть одно из этих решений другим. Для средства общего назначения ситуация совершенно иная.

Первый подход, возвращать недопустимое значение, неосуществим. Нет совершенно никакого способа узнать, что некоторое конкретное значени будет недопустимым во всех применениях slist.

Второй подход, возвращать значение состояния, можно использовать в некоторых классах (один из вариантов этого плана применяется в стандартных потоках ввода/вывода istream и ostream; как - объясняется в #8.4.2). Здесь, однако, имеется серьезная проблема, вдруг пользователь не позаботится проверить значение состояния, если средство не слишком часто подводит. Кроме того, средство может использоваться в сотнях или даже тысячах мест программы. Проверка значения в каждом месте сильно затруднит чтение программы.

Третьему подходу, предоставлять функцию ошибок, недостает гибкости. Тот, кто реализует общецелевое средство, не может узнать, как пользователи захотят, чтобы обрабатывались ошибки. Например, пользователь может предпочитать сообщения на датском или венгерском.

Четвертый подход, позволить пользователю задавать функцию ошибок, имеет некоторую привлекательность при условии, что разработчик предоставляет класс в виде библиотеки (#4.5), в которой содержатся

стандартные функции обработки ошибок.

Решения 3 и 4 можно сделать более гибкими (и по сути эквивалентными), задав указатель на функцию, а не саму функцию. Это позволит разработчику такого средства, как slist, предоставить функцию ошибок, действующую по умолчанию, и при этом программистам, которые будут использовать списки, будет легко задать свои собственные функции ошибок, если нужно, и там, где нужно. Например:

```
typedef void (*PFC)(char*); // указатель на тип функция
extern PFC slist_handler;
extern PFC set_slist_handler(PFC);
```

Функция set_slist_hanlder() позволяет пользователю заменить стандартную функцию. Общепринятая реализация предоставляет действующую по умолчанию функцию обработки ошибок, которая сначала пишет сообщение об ошибке в cerr, после чего завершает программу с помощью exit():

```
#include "slist.h"
#include


void default_error(char* s)
{
    cerr << s << "\n";
    exit(1);
}
```

Она описывает также указател на функцию ошибок и, для удобства записи, функцию для ее установки:

```
PFC slist_handler = default_error;

PFC set_slist_handler(PFC handler);
{
    PFC rr = slist_handler;
    slist_handler = handler;
    return rr;
}
```

Обратите внимание, как set_slist_hanlder() возвращает предыдущий slist_hanlder(). Это делает удобным установку и переустановку обработчиков ошибок на манер стека. Это может быть в основном полезным в больших программах, в которых slist может использоваться в нескольких разных ситуациях, в каждой из которых могут, таким образом, задаваться свои собственные подпрограммы обработки ошибок. Например:

```
{
PFC old = set_slist_handler(my_handler);

    // код, в котором в случае ошибок в slist
    // будет использоваться мой обработчик my_handler

    set_slist_handler(old); // восстановление
}
```

Чтобы сделать управление более изящным, slist_hanlder мог бы быть сделан членом класса slist, что позволило бы различным спискам иметь одновременно разные обработчики.

7.3.5 Обобщенные Классы

Очевидно, можно было бы определить списки других типов (classdef*, int, char* и т.д.) точно так же, как был определен класс nlist: простым выводом из класса slist. Процесс определения таких новх типов утомителен (и потому чреват ошибками), но с помощью макросов его можно "механизировать". К сожалению, если пользоваться стандартным C препроцессором (#4.7 и #c.11.1), это тоже может оказаться тягостным. Однако полученными в результате макросами пользоваться довольно просто.

Вот пример того, как обобщенный (generic) класс slist, названный gslist, может быть задан как макрос. Сначала для написания такого рода макросов включаются некоторые инструменты из :

```
#include "slist.h"

#ifndef GENERICH
#include
#endif
```

Обратите внимание на использование #ifndef для того, чтобы гарантировать, что в одной компиляции не будет включен дважды. GENERICH определен в .

После этого с помощью name2(), макроса из для конкатенации имен, определяются имена новых обобщенных классов:

```
#define gslist(type) name2(type,gslist)
#define gslist_iterator(type) name2(type,gslist_iterator)
```

И, наконец, можно написать классы gslist(тип) и gslist_iterator(тип):

```
#define gslistdeclare(type)                                  \
struct gslist(type) : slist {                                \
    int insert(type a)                                       \
        { return slist::insert( ent(a) ); }                  \
    int append(type a)                                       \
        { return slist::append( ent(a) ); }                  \
    type get() { return type( slist::get() ); }              \
    gslist(type)() { }                                       \
    gslist(type)(type a) : (ent(a)) { }                      \
    ~gslist(type)() { clear(); }                             \
};                                                           \
                                                             \
struct gslist_iterator(type) : slist_iterator {              \
    gslist_iterator(type)(gslist(type)& a)                   \
        : ( (slist&)s ) {}                                   \
    type operator()()                                        \
        { return type( slist_iterator::operator()() ); } \
}
```

\ на конце строк указывает , что следуюшая строка является частью определяемого макроса.

С помощью этого макроса список указателей на имя, аналогичный использованному раньше классу nlist, можно определить так:

```
#include "name.h"

typedef name* Pname;
declare(gslist,Pname); // описать класс gslist(Pname)

gslist(Pname) nl;      // описать один gslist(Pname)
```

Макрос declare (описать) определен в . Он конкатенирует свои параметры и вызывает макрос с этим именем, в данном случае gslistdeclare, описанный выше. Параметр имя типа для declare должен быть простым именем. Используемый метод макроопределения не может обрабатывать имена типов вроде name*, поэтому применяется typedef.

Использования вывода класса гарантирует, что все частные случаи обобщенного класса разделяют код. Этот метод можно применять только для создания классов объектов того же размера или меньше, чем базовый класс, который используется в макросе. gslist применяется в #7.6.2.

7.3.6 Ограниченные Интерфейсы

Класс slist - довольно общего характера. Иногда подобная общность не требуется или даже нежелательна. Ограниченные виды списков, такие как стеки и очереди, даже более обычны, чем сам обобщенный список. Такие структуры данных можно задать, не описав базовый класс как открытый. Например, очередь целых можно определить так:

```
#include "slist.h"

class iqueue : slist {
               //предполагается sizeof(int)<=sizeof(void*)
public:
    void put(int a) { slist::append((void*)a); }
    int det()       { return int(slist::get()); }
    iqueue()        {}
};
```

При таком выводе осуществляются два логически разделенных действия: понятие списка ограничивается понятием очереди (сводится к нему), и задается тип int, чтобы свести понятие очереди к типу данных очередь целых, iqueue. Эти два деяствия можно выполнять и раздельно. Здесь первая часть - это список, ограниченный так, что он может использоваться только как стек:

```
#include "slist.h"

class stack : slist {
public:
    slist::insert;
    slist::get;
    stack() {}
    stack(ent a) : (a) {}
};
```

который потом используется для создания типа "стек указателей на символы":

```
#include "stack.h"

class cp : stack {
public:
    void push(char* a) { slist::insert(a); }
    char* pop() { return (char*)slist::get(); }
    nlist() {}
};
```

7.4 Добавление к Классу

В предыдущих примерах производный класс ничего не добавлял к базовому классу. Для производного класса функции определялись только чтобы обеспечить преобразование типа. Каждый производный класс просто задавал альтернативный интерфейс к общему множеству

программ. Этот специальный случай важен, но наиболее обычная причина определения новх классов как производных классов в том, что кто-то хочет иметь то, что предоставляет базовый класс, плюс еще чуть-чуть.

Для производного класса можно определить данные и функции дополнительно к тем, которые наследуются из его базового класса. Это дает альтернативную стратегию обеспечить средства связанного списка. Заметьте, когда в тот slist, который определялся выше, помещается элемент, то создается slink, содержащий два указателя. На их создание тратится время, а ведь без одного из указателей можно обойтись, при условии, что нужно только чтобы объект мог находиться в одном списке. Так что указатель next на следующий можно поместить в сам объект, вместо того, чтобы помещать его в отдельный объект slink. Идея состоит в том, чтобы создать класс olink с единственным полем next, и класс olist, который может обрабатывать указателями на такие звенья olink. Тогда olist сможет манипулировать объектами луюого класса, производного от olink. Буква "o" в названиях стоит для того, чтобы напоминать вам, что объект может находиться одновременно только в одном списке olist:

```
struct olink {
    olink* next;
};
```

Класс olist очень напоминает класс slist. Отличие состоит в том, что пользователь класса olist манипулирует объектами класса olink непосредсвенно:

```
class olist {
    olink* last;
public:
    void insert(olink* p);
    void append(olink* p);
    olink* get();
    // ...
};
```

Мы можем вывести из класса olink класс name:

```
class name : public olink {
    // ...
};
```

Теперь легко сделать список, который можно использовать без накладных расходов времени на размещение или памяти.

Объекты, помещаемы в olist, теряют свой тип. Это означает, что компилятор знает только то, что они olink'и. Правильный тип можно восстановить с помощью явного преобразования типа объектов, вынутых из olist. Например:

```
void f()
{
    olist ll;
    name nn;
    ll.insert(&nn);              // тип &nn потерян
    name* pn = (name*)ll.get();  // и восстановлен
}
```

Другой способ: тип можно восстановить, выведя еще один класс из olist для обработки преобразования типа:

```
class olist : public olist {
    // ...
    name* get() { return (name*)olist::get(); }
```

```
};
```

Имя name может одновременно находиться только в одном olist. Для имен это может быть и неподходит, но в классах, для которых это подойдет полностью, недостатка нет. Например, класс фигур shape использует для поддержки списка всех фигур именно этот метод. Обратите внимание, что можно было бы определить slist как производный от olist, объединяя таким образом оба понятия. Однако использование бызовых и производных классов на таком микроскопическом уровне может очень сильно исказить код.

7.5 Неоднородные Списки

Предыдущие списки были однородными. То есть, в список помещались только объекты одного типа. Это обеспечивалось аппаратом производных классов. Списки не обязательно должны быть одднородными. Список, заденный в виде указаетелей на класс, может содержать объекты любого класса, производного от этого класса. То есть, список может быть неоднородным. Вероятно, это единственный наиболее важный и полезный аспект производных классов, и он весьма существенно используется в стиле программирования, который демонстрируется приведенным выше примером. Этот стиль программирования часто называют объектно-основанным или объектно-ориентированным. Он опирается на то, что действия над объектами неоднородных списков выполняются одинаковым образом. Смысл этих действий зависит от фактического типа объектов, находящихся в списке (что становится известно только на стадии выполнения), а не просто от типа элементов списка (который компилятору известен).

7.6 Законченна Программа

Разберем процесс написания программы для рисования на экране геметрических фигур. Она естественным образом разделяется на три части:

[1] Администратор экрана: подпрограммы низкого уровня и структуры данных, определяющие экран; он ведает только точками и прямыми линиями;

[2] Библиотека фигур: набор определений оосновных фигур вроде прямоугольника и круга и стандартные программы для работы с ними; и

[3] Прикладная программа: множество определений, специализированных для данного приложения, и код, в котором они используются.

Эти три части скорее всего будут писать разные люди (в разных организациях и в разное время). При этом части будут скорее всего писать именно в указанном порядке с тем осложняющим обстоятельством, что у разработчиков нижнего уровня не будет точного представления, для чего их код в конечном счете будет использоваться. Это отражено в приводимом примере. Чтобы пример был короче, графическая библиотека предоставляет только весьма ограниченный сервис, а сама прикладная программа очень проста. Чтобы читатель смог испытать программу, даже если у него нет совсем никаких графических средств, используется чрезвычайно простая концепция экрана. Не должно составить труда заменить эту экранную часть программы чем-нибудь подходящим, не изменяя код библиотеки фигур и прикладной программы.

7.6.1 Администратор Экрана

Вначале было намерени написать администратор экрана на C (а не на C++), чтобы подчеркнуть разделение уровней реализации. Это оказалось слишком утомительным, поэтому пришлось пойти на компромисс: используется стиль C (нет функций членов, виртуальных

функций, определяемых пользователем операций и т.п.), однако применяются конструкторы, надлежащим образом описываются и проверяются параметры функций и т.д. Оглядываясь назад, можно сказать, что администратор экрана очень похож на С программу, которую потом модифицировали, чтобы воспользоваться средствами С++ не переписывая все полностью.

Экран представляется как двумерный массив символов, работу с которым осуществляют функции put_point() и put_line(), использующие при ссылке на экран структуру point:

```
// файл screen.h

const XMAX=40, YMAX=24;

struct point {
    int x,y;
    point() {}
    point(int a, int b) { x=a; y=b; }
};

overload put_point;
extern void put_point(int a, int b);
inline void put_point(point p) { put_point(p.x,p.y); }


overload put_line;
extern void put_line(int, int, int, int);
inline void put_line(point a, point b)
    { put_line(a.x,a.y,b.x,b.y); }

extern void screen_init();
extern void screen_refresh();
extern void screen_clear();

#include
```

Перед первым использованием функции put экран надо инициализировать с помощью screen_init(), а изменения в структуре данных экрана отображаются на экране только после вызова screen_refresh(). Как увидит пользователь, это "обновление" ("refresh") осуществляется просто посредством печати новой копии экрана под его предыдущим вариантом. Вот функции и определения данных для экрана:

```
#include "screen.h"
#include

enum color { black='*', white=' ' };

char screen[XMAX][YNAX];

void screen_init()
{
    for (int y=0; y=a || a<=b) y0 += dy, eps -= two_a;
    }
}
```

Предоставляются функции для очистки экрана и его обновления:

```
void screen_clear() { screen_init(); }  // очистка

void screen_refresh()                   // обновление
{
    for (int y=YMAX-1; 0<=y; y--) {     // сверху вниз
```

```
        for (int x=0; x
```

7.6.2 Библиотека Фигур

Нам нужно определить общее понятие фигуры (shape). Это надо сделать таким образом, чтобы оно использовалось (как базовый класс) всеми конкретными фигурами (например, кругами и квадратами), и так, чтобы любой фигурой можно было манипулировать исключительно через интерфейс, предоставляемый классом shape:

```
struct shape {
    shape() { shape_list.append(this); }

    virtual point north() { return point(0,0); } // север
    virtual point south() { return point(0,0); } // юг
    virtual point east() { return point(0,0); }  // восток
    virtual point neast() { return point(0,0); } // северо-восток
    virtual point seast() { return point(0,0); } // юго-восток

    virtual void draw() {};                      // нарисовать
    virtual void move(int, int) {};              // переместить
};
```

Идея состоит в том, что расположение фигуры задается с помощью move(), и фигура помещается на экран с помощью draw(). Фигуры можно располагать относительно друг друга, используя понятие точки соприкосновения, и эти точки перечисляются после точек на компасе (сторон света). Каждая конкретная фигура определяет свой смысл этих точек, и каждая определяет способ, которым она рисуется. Для экономии места здесь на самом деле определяются только необходимые в этом примере стороны света. Конструктор shape::shape() добавляет фигуру в список фигур shape_list. Этот список является gslist, то есть, одним из вариантов обобщенного односвязанного списка, определенного в #7.3.5. Он и соответствующий итератор были сделаны так:

```
typedef shape* sp;
declare(gslist,sp);

typedef gslist(sp) shape_lst;
typedef gslist_iterator(sp) sp_iterator;
```

поэтому shape_list можно описать так:

```
shape_lst shape_list;
```

Линию можно построить либо по двум точкам, либо по точке и целому.

В последнем случае создается горизонтальная линия, длину которой определяет целое. Знак целого указывает, каким концом является точка: левым или правым. Вот определение:

```
class line : public shape {
/*
    линия из 'w' в 'e'
    north() определяется как ``выше центра
    и на север как до самой северной точки''
*/
    point w,e;
public:
    point north()
        { return point((w.x+e.x)/2,e.ydraw();
    screen_refresh();
}
```

И вот, наконец, настоящая сервисная функция (утилита). Она кладет одну фигуру на верх другой, задавая, что south() одной должен быть сразу над north() другой:

```
void stack(shape* q, shape* p)   // ставит p на верх q
{
    point n = p->north();
    point s = q->south();
    q->move(n.x-s.x,n.y-s.y+1);
}
```

Теперь представим себе, что эта библиотека считается собственностью некоей компании, которая продает программное обеспечение, и что они продают вам только заголовочный файл, содержащий определения фигур, и откомпилированный вариант определений функций. И у вас все равно остается возможность определять новые фигуры и использовать для ваших собственных фигур сервисные функции.

7.6.3 Прикладная Программа

Прикладная программа чрезвычайно проста. Определяется новая фигура my_shape (на печати она немного похожа на рожицу), а потом пишется главная программа, которая надевает на нее шляпу. Вначале описание my_shape:

```
#include "shape.h"

class myshape : public rectangle {
    line* l_eye;                    // левый глаз
    line* r_eye;                    // правый глаз
    line* mouth;                    // рот
public:
    myshape(point, point);
    void draw();
    void move(int, int);
};
```

Глаза и рот - отдельные и независимые объекты, которые создает конструктор my_shape:

```
myshape::myshape(point a, point b) : (a,b)
{
    int ll = neast().x-swest().x+1;
    int hh = neast().y-swest().y+1;
    l_eye = new line(
        point(swest().x+2,swest().y+hh*3/4),2);
    r_eye = new line(
        point(swest().x+ll-4,swest().y+hh*3/4),2);
    mouth = new line(
        point(swest().x+2,swest().y+hh/4),ll-4);
}
```

Объекты глаза и рот порознь рисуются заново функцией shape_refresh(), и в принципе могут обрабатываться независимо из

объекта my_shape, которому они принадлежат. Это один способ определять средства для иерархически построенных объектов вроде my_shape. Другой способ демонстрируется на примере носа. Никакой нос не определяется, его просто добавляет к картинке функция draw():

```
void myshape::draw()
{
    rectangle::draw();
    put_point(point(
        (swest().x+neast().x)/2,(swest().y+neast().y)/2));
}
```

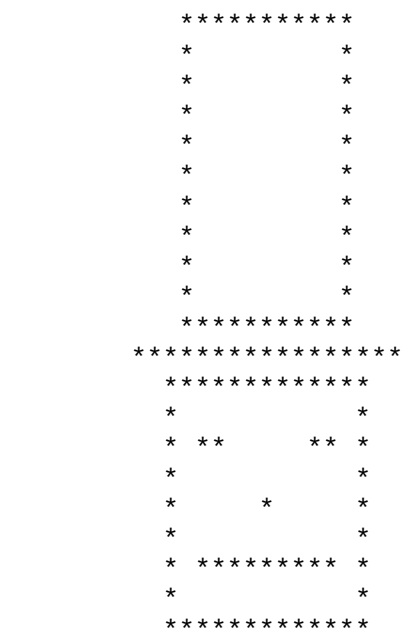
my_shape передвигается посредством перемещения базового прямоугольника rectangle и вторичных объектов l_eye, r_eye и mouth (левого глаза, правого глаза и рта):

```
void myshape::move()
{
    rectangle::move();
    l_eye->move(a,b);
    r_eye->move(a,b);
    mouth->move(a,b);
}
```

Мы можем, наконец, построить несколько фигур и немного их подвигать:

```
main()
{
    shape* p1 = new rectangle(point(0,0),point(10,10));
    shape* p2 = new line(point(0,15),17);
    shape* p3 = new myshape(point(15,10),point(27,18));
    shape_refresh();
    p3->move(-10,-10);
    stack(p2,p3);
    stack(p1,p2);
    shape_refresh();
    return 0;
}
```

Еще раз обратите внимание, как функции вроде shape_refresh() и stack() манипулируют объектами типов, определяемых гораздо позже, чем были написаны (и, может быть, откомпилированы) сами эти функции.



Если вы пользоались классом slist, вы могли обнаружить, что ваша программа тратит на заметное время на размещение и освобождение объектов класса slink. Класс slink - это превосходный пример класса, который может значительно выиграть от того, что программист возьмет под контроль управление свободной памятью. Для этого вида объектов идеально подходит оптимизирующий метод, который описан в #5.5.6. Поскольку каждый slink создается с помощью new и уничтожается с помощью delete членами класса slist, другой способ выделения памяти не представляет никаких проблемм.

Если производный класс осуществляет присваивание указателю this, то конструктор его базового класса будет вызываться только после этого присваивания, и значение указателя this в конструкторе базового класса будет тем, которое присвоено конструктором производного класса. Если базовый класс присваивает указателю this, то будет присвоено то значение, которое использует конструктор производного класса. Например:

```
#include

struct base { base(); };

struct derived : base { derived(); }

base::base()
{
    cout << "\tbase 1: this=" << int(this) << "\n";
    if (this == 0) this = (base*)27;
    cout << "\tbase 2: this=" << int(this) << "\n";
}

derived::derived()
{
    cout << "\tderived 1: this=" << int(this) << "\n";
    this = (this == 0) ? (derived*)43 : this;
    cout << "\tderived 2: this=" << int(this) << "\n";
}

main()
{
cout << "base b;\n";
base b;
cout << "new base b;\n";
new base;
cout << "derived d;\n";
derived d;
cout << "new derived d;\n";
new derived;
cout << "at the end\n";

}
```

порождает вывод

```
base b;
        base 1: this=2147478307
        base 2: this=2147478307
new base;
        base 1: this=0
        base 2: this=27
derived d;
        derived 1: this=2147478306
        base 1: this=2147478306
        base 2: this=2147478306
        derived 1: this=2147478306
```

```
new derived;
        derived 1: this=0
        base 1: this=43
        base 2: this=43
        derived 1: this=43
at the end
```

Если деструктор производного класса осуществляет присваивание указателю this, то будет присвоено то значение, которое всретил деструктор его базового класса. Когда кто-либо делает в конструкторе присваивание указателю this, важно, чтобы присваивание указателю this всречалось на всех путях в конструкторе*.

7.8 Упражнения

1. (*1) Определите

```
class base {
public:
   virtual void iam() { cout << "base\n"; }
};
```

   Выведите из base два класса и для каждого определите iam() ("я есть"), которая выводит имя класса на печать. Создайте объекты этих классов и вызовите для них iam(). Присвойте адреса объектов производных классов указателям base* и вызовите iam() через эти указатели.
2. (*2) Реализуйте примитивы экрана (#7.6.1) подходящим для вашей системы образом.
3. (*2) Определите класс triangle (треугольник) и класс circle (круг).
4. (*2) Определите функцию, которая рисует линию, соединяющую две фигуры, отыскивая две ближайшие "точки соприкосновения" и соединяя их.
5. (*2) Модифицируйте пример с фигурами так, чтобы line была rectangle и наоборот.
6. (*2) Придумайте и реализуйте дважды связанный список, который можно использовать без итератора.
7. (*2) Придумайте и реализуйте дважды связанный список, которым можно пользоваться только посредством итератора. Итератор должен иметь действия для движения вперед и назад, действия для вставления и удаления элементов списка, и способ доступа к текущему элементу.
8. (*2) Постройте обобщенный вариант дважды связанного списка.
9. (*4) Сделайте список, в котором вставляются и удаляются сами объекты (а не просто указатели на объекты). Проделайте это для класса X, для которого определены X::X(X&), X::~X() X::operator=(X&).
10. (*5) Придумайте и реализуйте библиотеку для написания моделей, управляемых прерываниями. Подсказка: . Только это - старая программа, а вы могли бы написать лучше. Должен быть класс task (- задача). Объект класса task должен мочь сохранять свое состояние и восстанавливаться в это состояние (вы можете определить task::save() и task::restore()), чтобы

---

* К сожалению, об этом присваивании легко забыть. Например, в первом издании этой книги (английском - перев.) вторая строка конструктор derived::derived() читалась так:

```
if (this == 0) this = (derived*)43;
```

И следовательно, для d конструктор базового класса base::base() не вызывался. Программа была допустимой и корректно выполнялась, но

очевидно делала не то, что подразумевал автор. (прим. автора)

он мог действовать как сопрограмма. Отдельные задачи можно определять как объекты классов, производных от класса task. Программа, которую должна исполнять задача, модет задаваться как виртуальная функция. Должна быть возможность передавать новой задаче ее параметры как параметры ее конструктора(ов). Там должен быть планировщик, реализующий концепцию виртуального времени. Обеспечьте функцию задержки task::delay(), которая "тратит" виртуальное время. Будет ли планировщик отдельным или частью класса task - это один из основных вопросов, которые надо решить при проектировании. Задача должна передавать данные. Для этого разработайте класс queue (очередь). Придумайте способ, чтобы задача ожидала ввода из нескольких очередей. Ошибки в ходе выполнения обрабатывайте единообразно. Как бы вы отлаживали программы, написанные с помощью такой библиотеки?

Глава 8

Потоки

Язык C++ не обеспечивает средств для ввода/вывода. Ему это и не нужно; такие средства легко и элегантно можно создать с помощью самого языка. Описанная зжесь стандартная библиотека потокового ввода/вывода обеспечивает гибкий и эффективный с гарантией типа метод обработки символьного ввода целых чисел, чисел с плавающей точкой и символьных строк, а также простую модель ее расширения для обработки типов, определяемых пользователем. Ее пользовательский интерфейс находится в . В этой главе описывается сама библиотека, некоторые способы ее применения и методы, которые использовались при ее реализации.

8.1 Введение

Разработка и реализация стандартных средств ввода/вывода для языка программирования зарекомдовала себя как заведомо трудная работа. Традиционно средства ввода/вывода разрабатывались исключительно для небольшого числа встроенных типов данных. Однако в C++ программах обычно используется много типов, определенных пользователем, и нужно обрабатывать ввод и вывод также и значений этих типов. Очевидно, средство ввода/вывода должно быть простым, удобным, надежным в употребелении, эффективным и гибким, и ко всему прочему полным. Ничье решение еще не смогло угодить всем, поэтому у пользователя должна быть возможность задавать альтернативные средства ввода/вывода и расширять стандартные средства ввода/вывода применительно к требованиям приложения.

C++ разработан так, чтобы у пользователя была возможность определять новые типы столь же эффективные и удобные, сколь и встроенные типы. Поэтому обоснованным является требование того, что средства ввода/вывода для C++ должны обеспечиваться в C++ с применением только тех средств, которые доступны каждому программисту. Описываемые здесь средства ввода/вывода предсавляют собой попытку ответить на этот вызов.

Средства ввода/вывода связаны исключительно с обработкой преобразования типизированнных объектов в последовательности символов и обратно. Есть и другие схемы ввода/вывода, но эта является основополагающей в системе UNIX, и большая часть видов бинарного ввода/вывода обрабатывается через рассмотрение символа просто как набор бит, при этом его общепринятая связь с алфавитом игнорируется. Тогда для программиста ключевая проблема заключается в задании соответствия между типизированным объектом и принципиально нетипизированной строкой.

Обработка и встроенных и определенных пользователем типов однородным образом и с гарантией типа достигается с помощью одного перегруженного имени функции для набора функций вывода. Например:

```
put(cerr,"x = "); // cerr - поток вывода обшибок
put(cerr,x);
put(cerr,"\n");
```

Тип параметра определяет то, какая из функций put будет вызываться для каждого параметра. Это решение применялось в нескольких языках. Однако ему недостает лаконичности. Перегрузка операции << значением "поместить в" дает более хорошую запись и позволяет программисту выводить ряд объектов одним оператором. Например:

```
cerr << "x = " << x << "\n";
```

где cerr - стандартный поток вывода ошибок. Поэтому, если x является int со значением 123, то этот оператор напечатает в стандартный поток вывода ошибок

```
x = 123
```

и символ новой строки. Аналогично, если X принадлежит определенному пользователем типу complex и имеет значение (1,2.4), то приведенный выше оператор напечатает в cerr

```
x = 1,2.4)
```

Этот метод можно применять всегда, когда для x определена операция <<, и пользователь может определять операцию << для нового типа.

8.2 Вывод

В этом разделе сначала обсуждаются средства форматного и бесформатного вывода встроенных типов, потом приводится стандартный способ спецификации действий вывода для определяемых пользователем типов.

8.2.1 Вывод Встроенных Типов

Класс ostream определяется вместе с операцией << ("поместить в") для обработки вывода встроенных типов:

```
class ostream {
    // ...
public:
    ostream& operator<<(char*);
    ostream& operator<<(int i) { return *this<
```

8.2.3 Некоторые Подробности Разработки

Операция вывода используется, чтобы избежать той многословности, которую дало бы использование функции вывода. Но почему <<?

Возможности изобрести новый лексический символ нет (#6.2). Операция присваивания была кандидатом одновременно и на ввод, и на вывод, но оказывается, большинство людей предпочитают, чтобы операция ввода отличалась от операции вывода. Кроме того, = не в ту сторону связывается (ассоциируется), то есть cout=a=b означает cout=(a=b).

Делались попытки оспользовать операции < и >, но значения "меньше" и "больше" настолько прочно вросли в сознание людей, что новые операции ввода/вывода во всех реальных случаях оказались нечитаемыми. Помимо этого, "<" находится на большинстве клавиатур

как раз на ",", и у людей получаются операторы вроде такого:

```
cout < x , y , z;
```

Для таких операторов непросто выдать хорошие сообщения об ошибках.

Операции << и >> к такого рода проблемам не приводят, они асимметричны в том смысле, что их можно проассоциировать с "в" и "из", а приоритет << достаточно низок, чтобы можно было не использовать скобки для арифметических выражений в роли операндов. Например:

```
cout << "a*b+c=" << a*b+c << "\n";
```

Естественно, при написании выражений, которые содержат операции с более низкими приоритетами, скобки использовать надо. Например:

```
cout << "a^b|c=" << (a^b|c) << "\n";
```

Операцию левого сдвига тоже можно применять в операторе вывода:

```
cout << "a<
```

8.2.4 Форматированный Вывод

Пока << применялась только для неформатированного вывода, и на самом деле в реальных программах она именно для этого главным образом и применяется. Помимо этого существует также несколько форматирующих функций, создающих представление своего параметра в виде строки, которая используется для вывода. Их второй (необязательный) параметр указывает, сколько символьных позиций должно использоваться.

```
char* oct(long, int =0);    // восьмеричное представление
char* dec(long, int =0);    // десятичное представление
char* hex(long, int =0);    // шестнадцатиричное представление
char* chr(int, int =0);     // символ
char* str(char*, int =0);   // строка
```

Если не задано поле нулевой длины, то будет производиться усечение или дополнение; иначе будет использоваться столько символов (ровно), сколько нужно. Например:

```
cout << "dec(" << x
     << ") = oct(" << oct(x,6)
     << ") = hex(" << hex(x,4)
     << ")";
```

Если x==15, то в результате получится:

```
dec(15) = oct(    17) = hex(   f);
```

Можно также использовать строку в общем формате:

```
char* form(char* format ...);
```

```
cout<
```

8.2.5 Виртуальная Функция Вывода

Иногда функция вывода должна быть virtual. Рассмотрим пример класса shape, который дает понятие фигуры (#1.18):

```
class shape {
    // ...
public:
    // ...
```

```
    virtual void draw(ostream& s); // рисует "this" на "s"
};

class circle : public shape {
    int radius;
public:
    // ...
    void draw(ostream&);
};
```

То есть, круг имеет все признаки фигуры и может обрабатываться как фигура, но имеет также и некоторые специальные свойства, которые должны учитываться при его обработке.

Чтобы поддерживать для таких классов стандартную парадигму вывода, операция << определяется так:

```
ostream& operator<<(ostream& s, shape* p)
{
    p->draw(s);
    return s;
}
```

Если next - итератор типа определенного в #7.3.3, то список фигур распечатывается например так:

```
while ( p = next() ) cout << p;
```

8.3 Файлы и Потоки

Потоки обычно связаны с файлами. Библиотека потоков создает стандартный поток ввода cin, стандартный поток вывода cout и стандартный поток ошибок cerr. Программист может открывать другие файлы и создавать для них потоки.

8.3.1 Инициализация Потоков Вывода

ostream имеет конструкторы:

```
class ostream {
    // ...
    ostream(streambuf* s);      // связывает с буфером потока
    ostream(int fd);            // связывание для файла
    ostream(int size, char* p); // связывет с вектором
};
```

Главная работа этих конструкторов - связывать с потоком буфер. streambuf - класс, управляющий буферами; он описывается в #8.6, как и класс filebuf, управляющий streambuf для файла. Класс filebuf является производным от класса streambuf.

Описание стандартных потоков вывода cout и cerr, которое находится в исходных кодах баблиотеки потоков ввода/вывода, выглядит так:

```
    // описать подходящее пространство буфера
char cout_buf[BUFSIZE]

    // сделать "filebuf" для управления этим пространством
    // связать его с UNIX'овским потоком вывода 1 (уже открытым)
filebuf cout_file(1,cout_buf,BUFSIZE);

    // сделать ostream, обеспечивая пользовательский интерфейс
ostream cout(&cout_file);

char cerr_buf[1];
```

```
    // длина 0, то есть, небуферизванный
    // UNIX'овский поток вывода 2 (уже открытый)
  filebuf cerr_file()2,cerr_buf,0;

  ostream cerr(&cerr_file);
```

Примеры двух других конструкторов ostream можно найти в #8.3.3 и #8.5.

8.3.2 Закрытие Потоков Вывода

Деструктор для ostream сбрасывает буфер с помощью открытого члена функции `ostream::flush()`:

```
  ostream::~ostream()
  {
      flush();       // сброс
  }
```

Сбросить буфер можно также и явно. Например:

```
  cout.flush();
```

8.3.3 Открытие Файлов

Точные детали того, как открываются и закрываются файлы, различаются в разных операционных системах и здесь подробно не описываются. Поскольку после включения становятся доступны cin, cout и cerr, во многих (если не во всех) программах не нужно держать код для открытия файлов. Вот, однако, программа, которая открывает два файла, заданные как параметры командной строки, и копирует первый во второй:

```
  #include

  void error(char* s, char* s2)
  {
      cerr << s << " " << s2 << "\n";
      exit(1);
  }

  main(int argc, char* argv[])
  {
      if (argc != 3) error("неверное число параметров","");

      filebuf f1;
      if (f1.open(argv[1],input) == 0)
          error("не могу открыть входной файл",argv[1]);
      istream from(&f1);

      filebuf f2;
      if (f2.open(argv[2],output) == 0)
          error("не могу создать выходной файл",argv[2]);
      ostream to(&f2);

      char ch;
      while (from.get(ch)) to.put(ch);

      if (!from.eof() !! to.bad())
          error("случилось нечто странное","");
  }
```

Последовательность действий при создании ostream для именованного файла та же, что используется для стандартных потоков: (1) сначала создается буфер (здесь это делается посредством описания filebuf); (2) затем к нему подсоединяется файл (здесь это делается посредством открытия файла с помощью функции filebuf::open()); и, накрнец, (3) создается сам ostream с filebuf в качестве параметра. Потоки ввода обрабатываются аналогично.

Файл может открываться в одной из двух мод:

```
enum open_mode { input, output };
```

Действие filebuf::open() возвращает 0, если не может открыть файл в соответствие с требованием. Если пользователь пытается открыть файл, которого не существует для output, он будет создан.

Перед завершением программа проверяет, находятся ли потоки в приемлемом состоянии (см. #8.4.2). При завершении программы открытые файлы неявно закрываются.

Файл можно также открыть одновременно для чтения и записи, но в тех случаях, когда это оказывается необходимо, парадигма потоков редко оказывается идеальной. Часто лучше рассматривать такой файл как вектор (гигантских размеров). Можно определить тип, котоырй позволяет программе обрабатывать файл как вектор; см. Упражнения 8-10.

8.3.4 Копирование Потоков

Есть возможность копировать потоки. Например:

```
cout = cerr;
```

В результате этого получаются две переменные, ссылающиеся на один и тот же поток. Гавным образом это бывает полезно для того, чтобы сделать стандартное имя вроде cin ввылающимся на что-то другое (пример этого см. в #3.1.6)

8.4 Ввод

Ввод аналогичен выводу. Имеется класс istream, который предоставляет операцию >> ("взять из") для небольшого множества стандартных типов. Функция operator>> может определяться для типа, определяемого пользователем.

8.4.1 Ввод Встроенных Типов

Класс istream определяется так:

```
class istream {
    // ...
public:
    istream& operator>>(char*);      // строка
    istream& operator>>(char&);      // символ
    istream& operator>>(short&);
    istream& operator>>(int&);
    istream& operator>>(long&);
    istream& operator>>(float&);
    istream& operator>>(double&);
    // ...
};
```

Функции ввода определяются в таком духе:

```
istream& istream::operator>>(char& c);
{
```

```
        // пропускает пропуски
    int a;
        // неким образом читает символ в "a"
    c = a;
}
```

Пропуск определяется как стандартнчй пропуск в C, через вызов isspase() в том виде, как она определена в (пробел, табуляция, символ новой строки, перевод формата и возврат каретки).

В качестве альтернативы можно использовать функции get():

```
class istream {
    // ...
    istream& get(char& c);                    // char
    istream& get(char* p, int n, int ='\n');  // строка
};
```

Они обрабатывают символы пропуска так же, как остальные символы. Функция istream::get(char) читает один и тот же символ в свой параметр; другая istream::get читает не более n символов в вектор символов, начинающийся в p. Необязательный третий параметр используется для задания символа остановки (иначе, терминатора или ограничителя), то есть этот символ читаться не будет. Если будет встречен символ ограничитель, он останется как первый символ потока. По умолчанию вторая функция get будет читать самое большее n символов, но не больше чем одну строку, '\n' является ограничителем по умолчанию. Необязательный третий параметр задает символ, который читаться не будет. Например:

```
cin.get(buf,256,'\t');
```

будет читать в buf не более 256 символов, а если встретится табуляция ('\t'), то это приведет к возврату из get. В этом случае следующим символом, который будет считан из cin, будет '\t'.

Стандартный заголовочный файл определяет несколько функций, которые могут оказаться полезными при осуществлении ввода:

```
int isalpha(char)    // 'a'..'z' 'A'..'Z'
int isupper(char)    // 'A'..'Z'
int islower(char)    // 'a'..'z'
int isdigit(char)    // '0'..'9'
int isxdigit(char)   // '0'..'9' 'a'..'f' 'A'..'F'
int isspase(char)    // ' ' '\t' возврат новая строка
                     // перевод формата
int iscntrl(char)    // управляющий символ
                     // (ASCII 0..31 и 127)
int ispunct(char)    // пунктуация: ниодин из вышеперечисленных
int isalnum(char)    // isalpha() | isdigit()
int isprint(char)    // печатаемый: ascii ' '..'-'
int isgraph(char)    // isalpha() | isdigit() | ispunct()
int isascii(char c)  { return 0<=c &&c<=127; }
```

Все кроме isascii() реализуются внешне одинаково, с применением символа в качестве индекса в таблице атрибутов символов. Поэтому такие выражения, как

```
(('a'<=c && c<='z') || ('A'<=c && c<='Z')) // алфавитный
```

не только утомительно пишутся и чреваты ошибками (на машине с набором символов EBCDIC оно будет принимать неалфавитные символы), они также и менее эффективны, чем применение стандартной функции:

```
isalpha(c)
```

8.4.2 Состояния Потока

Каждый поток (istream или ostream) имеет ассоциированное с ним состояние, и обработка ошибок и нестандартных условий осуществляется с помощью соответствующей установки и проверки этого состояния.

Поток может находиться в одном из следующих состояний:

```
enum stream_state { _good, _eof, _fail, _bad };
```

Если состояние _good или _eof, значит последняя операция ввода прошла успешно. Если состояние _good, то следующая операция ввода может пройти успешно, в противном случае она закончится неудачей. Другими словами, применение операции ввода к потоку, который не находится в состоянии _good, является пустой операцией. Если делается попытка читать в переменную v, и операция окацивается неудачей, значение v должно остаться неизменным (оно будет неизменным, если v имеет один из тех типов, которые обрабатываются функциями членами istream или ostream). Отличия между состояниями _fail и _bad очень незначительно и предсавляет интерес только для разработчиков операций ввода. В состоянии _fail предполагается, что поток не испорчен и никакие символы не потеряны. В состоянии _bad может быть все что угодно.

Состояние потока можно проверять например так:

```
switch (cin.rdstate()) {
case _good:
    // последняя операция над cin прошла успешно
    break;
case _eof:
    // конец файла
    break;
case _fail:
    // некоего рода ошибка форматирования
    // возможно, не слишком плохая
    break;
case _bad:
    // возможно, символы cin потеряны
    break;
}
```

Для любой переменной z типа, для которого определены операции << и >>, копирующий цикл можно написать так:

```
while (cin>>z) cout << z << "\n";
```

Например, если z - вектор символов, этот чикл будет брать стандартный ввод и помещать его в стандартный вывод по одному слову (то есть, последовательности символов без пробела) на строку.

Когда в качестве условия используется поток, происходит проверка состояния потока и эта проверка проходит успешно (то есть, значение условия не ноль) только если состояние _good. В частности, в предыдущем цикле проверялось состояние istream, которое возвращает cin>>z. Чтобы обнаружить, почему цикл или проверка закончились неудачно, можно исследовать состояние. Такая проверка потока реализуется операцией преобразования (#6.3.2).

Делать проверку на наличие ошибок каждого ввода или вывода действительно не очень удобно, и обычно источником ошибок служит программист, не сделавший этого в том месте, где это существенно. Например, операции вывода обычно не проверяются, но они могут случайно не сработать. Парадигма потока ввода/вывода построена так, чтобы когда в C++ появится (если это произойдет) механизм обработки исключительных ситуаций (как средство языка или как стандартная

библиотека) его будет легко применить для упрощения и стандартизации обработки ошибок в потоках ввода/вывода.

8.4.3 Ввод Типов, Определяемых Пользователем

Ввод для пользовательского типа может определяться точно так же, как вывод, за тем исключением, что для операции ввода важно, чтобы второй параметр был ссылочного типа. Например:

```
istream& operator>>(istream& s, complex& a)
/*
    форматы ввода для complex; "f" обозначает float:
    f
    ( f )
    ( f , f )
*/
{
    double re = 0, im = 0;
    char c = 0;

    s >> c;
    if (c == '(') {
        s >> re >> c;
        if (c == ',') s >> im >> c;
        if (c != ')') s.clear(_bad);    // установить state
    }
    else {
        s.putback(c);
        s >> re;
    }

    if (s) a = complex(re,im);
    return s;
}
```

Несмотря на то, что не хватает кода обработки ошибок, большую часть видов ошибок это на самом деле обрабатывать будет. Локальная переменная с инициализируется, чтобы ее значение не оказалось случайно '(' после того, как операция окнчится неудачно. Завершающая проверка состояния потока гарантирует, что значение параметра a будет изменяться только в том случае, если все идет хорошо.

Операция установки состояния названа clear() (очистить), потому что она чаще всего используется для установки сосояния потока заново как _good. _good является значением параметра по умолчанию и для `istream::clear()`, и для `ostream::clear()`.

Над операциями ввода надо поработать еще. Было бы, в частности, замечательно, если бы можно было задавать ввод в терминах шаблона (как в языках Снобол и Икон), а потом проверять, прошла ли успешна вся операция ввода. Такие операции должны были бы, конечно, обеспечивать некоторую дополнительную буферизацию, чтобы они могли воссанавливать поток ввода в его исходное состояние после неудачной попытки распознавания.

8.4.4 Инициализация Потоков Ввода

Естественно, тип istream, так же как и ostream, снабжен конструктором:

```
class istream {
    // ...
    istream(streambuf* s, int sk =1, ostream* t =0);
```

```
    istream(int size, char* p, int sk =1);
    istream(int fd, int sk =1, ostream* t =0);
};
```

Параметр sk задает, должны пропускаться пропуски или нет. Параметр t (необязательный) задает указатель на ostream, к которому прикреплен istream. Например, cin прикреплен к cout; это значит, что перед тем, как попытаться читать символы из своего файла, cin выполняет

```
cout.flush(); // пишет буфер вывода
```

С помощью функции istream::tie() можно прикрепить (или открепить, с помощью tie(0)) любой ostream к любому istream. Например:

```
int y_or_n(ostream& to, istream& from)
/*
     "to", получает отклик из "from"
*/
{
    ostream* old = from.tie(&to);
    for (;;) {
        cout << "наберите Y или N: ";
        char ch = 0;
        if (!cin.get(ch)) return 0;

        if (ch != '\n') { // пропускает остаток строки
            char ch2 = 0;
            while (cin.get(ch2) && ch2 != '\n') ;
        }
        switch (ch) {
        case 'Y':
        case 'y':
        case '\n':
            from.tie(old);        // восстанавливает старый tie
            return 1;
        case 'N':
        case 'n':
            from.tie(old);        // восстанавливает старый tie
            return 0;
        default:
            cout << "извините, попробуйте еще раз: ";
        }
    }
}
```

Когда используется буферизованный ввод (как это происходит по умолчанию), пользователь не может набрав только одну букву ожидать отклика. Система ждет появвения символа новой строки. y_or_n() смотрит на первыйй символ строки, а остальные игноиррует.

Символ можно вернуть в поток с помощью функции istream::putback(char). Это позволяет программе "заглядывать вперед" в поток ввода.

8.5 Работа со Строками

Можно осуществлять действия, подобные вводу/выводу, над символьным вектором, прикрепляя к нему istream или ostream. Например, если веатор содержит обычную строку, завершающуюся нулем, для печати слов из этого вектора можно использовать приведенный выше копирующий цикл:

```
void word_per_line(char v[], int sz)
/*
```

```
    печатет "v" размера "sz" по одному слову на строке
*/
{
    istream ist(sz,v); // сделать istream для v
    char b2[MAX];      // больше наибольшего слова
    while (ist>>b2) cout << b2 << "\n";
}
```

Завершающий нулевой символ в этом случае интерпретируется как символ конца файла.

В помощью ostream можно отформатировать сообщения, которые не нужно печатать тотчас же:

```
char* p = new char[message_size];
ostream ost(message_size,p);
do_something(arguments,ost);
display(p);
```

Такая операция, как do_something, может писать в поток ost, передавать ost своим подоперациям и т.д. спомощью стандартных операций вывода. Нет необходимости делать проверку не переполнение, поскольку ost знает свою длину и когда он будет переполняться, он будет переходить в состояние _fail. И, наконец, display может писать сообщения в "настоящий" поток вывода. Этот метод может оказаться наиболее полезным, чтобы справляться с ситуациями, в которых окончательное отображение данных включает в себя нечто более сложное, чем работу с традиционным построчным устройством вывода. Например, текст из ost мог бы помещаться в располагающуюся где-то на экране область фиксированного размера.

8.6 Буферизация

При задании операций ввода/вывода мы никак не касались типов файлов, но ведь не все устройства можно рассматривать одинаково с точки зрения стратегии буферизации. Например, для ostream, подключенного к символьной строке, требуется буферизация другого вида, нежели для ostream, подключенного к файлу. С этими пробемами можно справиться, задавая различные буферные типы для разных потоков в момент инициализации (обратите внимание на три конструктора класса ostream). Есть только один набор операций над этими буферными типами, поэтому в функциях ostream нет кода, их

различающего. Однако функции, которые обрабатывают переполнение сверху и снизу, виртуальные. Этого достаточно, чтобы справляться с необходимой в данное время стратегией буферизации. Это также служит хорошим примером применения виртуальных функций для того, чтобы

сделать возможной однородную обработку логически эквивалентных средств с различной реализацией. Описание буфера потока в выглядит так:

```
struct streambuf {      // управление буфером потока

    char* base;         // начало буфера
    char* pptr;         // следующий свободный char
    char* qptr;         // следующий заполненный char
    char* eptr;         // один из концов буфера
    char  alloc;        // буфер, выделенный с помощью new

        // Опустошает буфер:
        // Возвращает EOF при ошибке и 0 в случае успеха
    virtual int overflow(int c =EOF);

        // Заполняет буфер
```

```
        // Возвращет EOF при ошибке или конце ввода,
        // иначе следующий char
    virtual int underflow();

    int snextc()        // берет следующий char
    {
        return (++qptr==pptr) ? underflow() : *qptr&0377;
    }

    // ...

    int allocate()      // выделяет некоторое пространство буфера

    streambuf() { /* ... */}
    streambuf(char* p, int l) { /* ... */}
    ~streambuf() { /* ... */}
};
```

Обратите внимание, что здесь определяются указатели, необходимые для работы с буфером, поэтому обычные посимвольные действия можно определить (только один раз) в виде максимально эффективных inline-функций. Для каждой конкретной стратегии буферизации необходимо определять только функции переполнения overflow() и underflow(). Например:

```
struct filebuf : public streambuf {

    int fd;             // дескриптор файла
    char opened;        // файл открыт

    int overflow(int c =EOF);
    int underflow();

    // ...

        // Открывает файл:
        // если не срабатывает, то возвращет 0,
        // в случае успеха возвращает "this"
    filebuf* open(char *name, open_mode om);
    int close() { /* ... */ }

    filebuf() { opened = 0; }
    filebuf(int nfd) { /* ... */ }
    filebuf(int nfd, char* p, int l) : (p,l) { /* ... */ }
    ~filebuf() { close(); }
};

int filebuf::underflow()        // заполняет буфер из fd
{
    if (!opened || allocate()==EOF) return EOF;

    int count = read(fd, base, eptr-base);
    if (count < 1) return EOF;

    qptr = base;
    pptr = base + count;
    return *qptr & 0377;
}
```

8.7 Эффективность

Можно было бы ожидать, что раз ввод/вывод определен с помощью общедоступных средств языка, он будет менее эффективен, чем встроенное средство. На самом деле это не так. Для действий вроде

"поместить символ в поток" используются inline-функции, единственные необходимые на этом уровне вызовы функций возникают из-за переполнения сверху и снизу. Для простых объектов (целое, строка и т.п.) требуется по одному вызову на каждый. Как выясняется, это не отличается от прочих средств ввода/вывода, работающих с объектами на этом уровне.

8.8 Упражнения

1. (*1.5) Считайте файл чисел с плавающей точкой, составьте из пар считанных чисел комплексные числа и выведите комплексные числа.

2. (*1.5) Определите тип name_and_address (имя_и_адрес). Определите для него << и >>. Скопируйте поток объектов name_and_address.
3. (*2) Постройте несколько функций для запроса и чтения различного вида информации. Простейший пример - функция y_or_n() в #8.4.4. Идеи: целое, число с плавающей точкой, имя файла, почтовый адрес, дата, личные данные и т.д. Постарайтесь сделать их защищенными от дурака.
4. (*1.5) Напишите программу, которая печатает (1) все буквы в нижнем регистре, (2) все буквы, (3) все буквы и цифры, (4) все символы, которые могут встречаться в идентификаторах C++ на вашей системе, (5) все символы пунктуации, (6) целые значения всех управляющих символов, (7) все символы пропуска, (8) целые значения всех символов пропуска, и (9) все печатаемые символы.
5. (*4) Реализуйте стандартную библиотеку ввода/вывода C () с помощью стандартной библиотеки ввода/вывода C++ ().
6. (*4) Реализуйте стандартную библиотеку ввода/вывода C++ () с помощью стандартной библиотеки ввода/вывода C ().
7. (*4) Реализуйте стандартные библиотеки C и C++ так, чтобы они могли использоваться одновременно.
8. (*2) Реализуйте класс, для которого [] перегружено для реализации случайного чтения символов из файла.
9. (*3) Как Упражнение 8, только сделайте, чтобы [] работало и для чтения, и для записи. Подсказка: сделайте, чтобы [] возвращало объект "дескрипторного типа", для которого присваивание означало бы присвоить файлу через дескриптор, а неявное преобразование в char означало бы чтение из файла через дескриптор.
10. (*2) Как Упражнение 9, только разрешите [] индексировать записи некоторого вида, а не символы.
11. (*3) Сделайте обобщенный вариант класса, определенного в Упражнении 10.
12. (*3.5) Разработайте и реализуйте операцию ввода по сопосталению с образцом. Для спецификации образца используйте строки формата в духе printf. Должна быть возможность попробовать сопоставить со вводом несколько образцов для нахождения фактического формата. Можно было бы вывести класс ввода по образцу из istream.

13. (*4) Придумайте (и реализуйте) вид образцов, которые намного лучше.

Справочное руководство по C++

1. ВВЕДЕНИЕ

Язык программирования C++ - это C*, расширенный введением классов, inline-функций, перегруженных операций, перегруженных

имен функций, константных типов, ссылок, операций управления свободной памятью, проверки параметров функций. Коротко различия между C++ и "старым C" приведены в #15. В этом руководстве описывается язык по состоянию на Июнь 1985.

2. ДОГОВОРЕННОСТИ О ЛЕКСИКЕ

Есть шесть классов лексем: идентификаторы, ключевые слова, константы, строки, операторы и прочие разделители. Символы пробела, табуляции и новой строки, а также комментарии (собирательно - "белые места"), как описано ниже, игнорируются, за исключением тех случаев, когда они служат разделителями лексем. Некое пустое место необходимо для разделения идентификаторов, ключевых слов и констант, которые в противном случае окажутся соприкасающимися.

Если входной поток разобран на лексемы до данного символа, принимается, что следующая лексема содержит наиболее длинную строку символов из тех, что могут составить лексему.

2.1 Комментарии

Символы /* задают начало комментария, заканчивающегося символами */. Комментарии не могут быть вложенными. Символы // начинают комментарий, который заканчивается в конце строки, на которой они появились.

2.2 Идентификаторы (имена)

Идентификатор - последовательность букв и цифр произвольной длины; первый символ обязан быть буквой; подчерк '_' считается за букву; буквы в верхнем и нижнем регистрах являются различными.

2.3 Ключевые слова

Следующие идентификаторы зарезервированы для использования в качестве ключевых слов и не могут использоваться иным образом:

---

* "Язык программирования Си" Брайэна В. Кернигана и Денниса М. Ритчи. Это руководство было построено на основе "C Programming Language - Reference Manual" системы UNIX V с разрешения AT&T Bell Laboratories. (прим. автора)

```
asm       auto      break     case      char
class     const     continue  default   delete
do        double    else      enum      extern
float     for       friend    goto      if
inline    int       long      new       operator
overload  public    register  return    short
sizeof    static    struct    switch    this
typedef   union     unsigned  virtual   void
while
```

Идентификаторы signed и volatile зарезервированы для применения в будущем.

2.4 Константы

Как описано ниже, есть несколько видов констант. В #2.6 приводится краткая сводка аппаратных характеристик, которые влияют на их размеры.

2.4.1 Целые константы

Целая константа, состоящая из последовательности цифр, считается

восьмиричной, если она начинается с 0 (цифры ноль), и десятичной в противном случае. Цифры 8 и 9 не являются восьмиричными цифрами. Последовательность цифр, которой предшествует 0x или 0X, воспринимается как шестнадцатеричное целое. В шестнадцатеричные цифры входят буквы от a или A до f или F, имеющие значения от 10 до 15. Десятичная константа, значение которой превышает наибольшее машинное целое со знаком, считается длинной (long); восьмеричная и шестнадцатеричная константа, значение которой превышает наибольшее машинное целое со знаком, считается long; в остальных случаях целые константы считаются int.

2.4.2 Явно заданные длинные константы

Десятичная, восьмиричная или шестнадцатиричная константа, за которой непосредственно стоит l (латинская буква "эль") или L, считается длинной константой.

2.4.3 Символьные константы

Символьная константа состоит из символа, заключенного в одиночные кавычки (апострофы), как, например, 'x'. Значением символьной константы является численное значение символа в машинном наборе символов (алфавите). Символьные константы считаются данными типа int.

Некоторые неграфические символы, одиночная кавычка ' и обратная косая \, могут быть представлены в соответствие со следующей таблицей escape-последовательностей:

символ новой строки	NL(LF)	\n
горизонтальная табуляция	NT	\t
вертикальная табуляция	VT	\v
возврат на шаг	BS	\b
возврат каретки	CR	\r
перевод формата	FF	\f
обратная косая	\	\\
одиночная кавычка (апостроф)	'	\'
набор битов	0ddd	\ddd
набор битов	0xddd	\xddd

Escape-последовательность \ddd состоит из обратной косой, за которой следуют 1, 2 или 3 восьмеричных цифры, задающие значение требуемого символа. Специальным случаем такой консрукции является \0 (не следует ни одной цифры), задающая пустой символ NULL. Escape-последовательность \xddd состоит из обратной косой, за которой следуют 1, 2 или 3 шестнадцатиричных цифры, задающие значение требуемого символа. Если следующий за обратной косой символ не является одним из перечисленных, то обратная косая игнорируется.

2.4.4 Константы с плавающей точкой

Константа с плавающей точкой состоит из целой части, десятичной точки, мантиссы, e или E и целого показателя степени (возможно, но не обязательно, со знаком). Целая часть и мантисса обе состоят из последовательности цифр. Целая часть или мантисса (но не обе сразу) может быть опущена; или десятичная точка, или e(E) вместе с целым показателем степени (но не обе части одновременно) может быть опущена. Константа с плавающей точкой имеет тип double.

2.4.5 Перечислимые константы

Имена, описанные как перечислители, (см. #8.5) являются

константами типа int.

2.4.6 Описанные константы

Объект (#5) любого типа может быть определен как имеющий постоянное значение во всей области видимости (#4.1) его имени. В случае указателей для достижения этого используется декларатор *const; для объектов, не являющихся указателями, используется описатель const (#8.2).

2.5 Строки

Строка есть последовательность символов, заключенная в двойные кавычки: "...". Строка имеет тип "массив символов" и класс памяти static (см. #4 ниже), она инициализируется заданными символами. Все строки, даже если они записаны одинаково, различны. Компилятор располагает в конце каждой строки нулевой (пустой) байт \0 с тем, чтобы сканируюшая строку программа могла найти ее конец. В строке перед символом двойной кавычки " обяэательно должен стоять \; кроме того, могут использоваться те же escape-последовательности, что были описаны для символьных констант. И, наконец, символ новой строки может появляться только сразу после \; тогда оба,- \ и символ новой строки,- игнорируются.

2.6 Харктеристики аппаратного обеспечения

В нижеследующей таблице собраны некоторые харктеристики аппаратного обеспечения, различающиеся от машины к машине.

	DEC VAX-11 ASCII	Motorola 68000 ASCII	IBM 370 EBCDIC	AT&T 3B ASCII
char	8 бит	8 бит	8 бит	8 бит
int	32 бит	16 бит	32 бит	16 бит
short	16 бит	16 бит	16 бит	16 бит
long	32 бит	32 бит	32 бит	32 бит
float	32 бит	32 бит	32 бит	32 бит
double	64 бит	64 бит	64 бит	64 бит
указатель	32 бит	32 бит	24 бит	32 бит
диапазон float	+_10E+_38	+_10E+_38	+_10E+_76	+_10E+_38
диапазон double	+_10E+_38	+_10E+_38	+_10E+_76	+_10E+_308
тип char	знаковый	без знака	без знака	без знака
тип поля	знаковый	без знака	без знака	без знака
порядок полей	справа налево	слева направо	слева направо	слева направо

3. ЗАПИСЬ СИНТАКСИСА

По используемым в данном руководстве синтаксическим правилам записи синтаксические категории выделяются курсивом а литеральные слова и символы шрифтом постоянной ширины*. Альтернативные категории записываются на разных строках. Необязательный терминальный или нетерминальный символ обозначается нижним индексом "opt", так что

{ выражение opt }

указывает на необязательность выражения в фигурных скобках. Синтаксис кратко изложен в #14.

* !!! выделить "постоянной ширины" шрифтом, которым печатаются программы и английские слова!!!

4. ИМЕНА И ТИПЫ

Имя обозначает(денотирует) объект, функцию, тип, значение или метку. Имя вводится в программе описанием (#8). Имя может использоваться только внутри области текста программы, называемой его областью видимости. Имя имеет тип, определяющий его использование. Объект - это область памяти. Объект имеет класс памяти, определяющий его время жизни. Смысл значения, обнаруженного в объекте, определяется типом имени, использованного для доступа к нему.

4.1 Область видимости

Есть четыре вида областей видимости: локальная, файл, программа и класс.

Локальная: Имя, описанное в блоке (#9.2), локально в этом блоке и может использоваться только в нем после места описания и в охватываемых блоках. Исключение составляют метки (#9.12), которые могут использоваться в любом месте функции, в которой они описаны. Имена формальных параметров функции рассматриваются так, как если бы они были описаны в самом внешнем блоке этой функции.

Файл: Имя, описанное вне любого блока (#9.2) или класса (#8.5), может использоваться в файле, где оно описано, после места описания.

Класс: Имя члена класса локально для его класса и может использоваться только в функции члене этого класса (#8.5.2), после примененной к объекту его класса (#7.1) операции . или после примененной к указателю на объект его класса (#7.1) операции ->. На статические члены класса (#8.5.1) и функции члены можно также ссылаться с помощью операции :: там, где имя их класса находится в области видимости. Класс, описанный внутри класса (#8.5.15), не считается членом, и его имя принадлежит охватывающей области видимости.

Имя может быть скрыто посредством явного описания того же имени в блоке или классе. Имя в блоке или классе может быть скрыто только именем, описанным в охватываемом блоке или классе. Скрытое нелокальное имя также может использоваться, когда его область видимости указана операцией :: (#7.1). Имя класса, скрытое именем, которое не является именем типа, все равно может использоваться, если перед ним стоит class, struct или union (#8.2). Имя перечисления enum, скрытое именем, которое не является именем типа, все равно может использоваться, если перед ним стоит enum (#8.2).

4.2 Определения

Описание (#8) является определением, за исключением тех случаев, когда оно описывает функци, не задавая тела функции (#10), когда оно содержит спецификатор extern (1) и в нем нет инициализатора или тела функции, или когда оно является описанием класса (#8.8).

4.3 Компоновка

Имя в файловой области видимости, не описанное явно как static, является общим для каждого файла многофайловой программы. Таковым же является имя функции. О таких именах говорится, что они внешние. Каждое описание внешнего имени в программе относится к тому же

объекту (#5), функции (#8.7), классу (#8.5), перечислению (#8.10) или значению перечислителя (#8.10).

Типы, специфицированные во всех описаниях внешнего имени должны быть идентичны. Может быть больше одного определения типа, перечисления, inline-функции (#8.1) или несоставного const (#8.2), при условии, что определения идентичны, появляются в разных файлах и все инициализаторы являются константными выражениями (#12). Во всех остальных случаях должно быть ровно одно определение для внешнего имени в программе.

Реализация может потребовать, чтобы составное const, использованное там, где не всречено никакого определения const, должно быть явно описано extern и иметь в программе ровно одно определение. Это же ограничение может налагаться на inline-функции.

4.4 Классы памяти

Есть два описываемых класса памяти: автоматический и статический.

Автоматические объекты локальны для каждого вызова блока и сбрасываются по выходе из него.

Статические объекты существуют и сохраняют свое значение в течение выполнения всей програмы.

Некоторые объекты не связаны с именами и их времена жизни явно управляются операторами new и delete ; см. #7.2 и #9.14

4.5 Основные типы

Объекты, описанные как символы (char), достаточны для хранения любого элемента машинного набора символов, и если принадлежащий этому набору символ хранится в символьной переменной, то ее значение равно целому коду этого символа.

В настоящий момент имеются целые трех размеров, описываемые как **short int**, **int** и **long int**. Более длинные целые (long int) предоставляют не меньше памяти, чем более короткие целые (short int), но при реализации или длинные, или короткие, или и те и другие могут стать эквивалентными обычным целым. "Обычные" целые имеют естественный размер, задаваемый архитектурой центральной мащины; остальные размеры делаются такими, чтобы они отвечали специальным потребностям.

Каждое перечисление (#8.9) является набором именованных констант. Свойства enum идентичны свойствам int.

Целые без знака, описываемые как unsigned, подчиняются правилам арифметики по модулю 2n, где n - число бит в их представлении.

Числа с плавающей точкой одинарной (float) и двойной (double) точности в некоторых машинных реализациях могут быть синонимами.

Поскольку объекты перечисленных выше типов вполне можно интерпретировать как числа, мы будем говорить о них как об арифметических типах. Типы char, int всех размеров и enum будут собирательно называться целыми типами. Типы float и double будут собирательно называться плавающими типами.

Тип данных void (пустой) определяет пустое множество значений. Значение (несуществующее) объекта void нельзя использовать никаким образом, не могут применяться ни явное, ни неявное преобразования. Поскольку пустое выражение обозначает несуществующее значение, такое выражение такое выражение может использоваться только как оператор выражение (#9.1) или как левый операнд в выражении с запятой (#7.15). Выражение может явно преобразовываться к типу void (#7.2).

4.4 Производные типы

Кроме основных арифметических типов концептуально существует

бесконечно много производных типов, сконструированных из основных типов следующим образом:

массивы объектов данного типа;

функции, получающие аргументы данного типа и возвращающие объекты данного типа;

указатели на объекты данного типа;

ссылки на объекты данного типа;

константы, являющиеся значениями данного типа;

классы, содержащие последовательность объектов различных типов, множество функций для работы с этими объектами и набор ограничений на доступ к этим объектам и функциям;
структуры, являющиеся классами без ограничений доступа;

объединения, являющиеся структурами, которые могут в разное время содержать объекты разных типов.

В целом эти способы конструирования объектов могут применяться рекурсивно.
Объект типа void* (указатель на void) можно использовать для указания на объекты неизвестного типа.

5. ОБЪЕКТЫ И LVALUE(АДРЕСА)

Объект есть область памяти;lvalue (адрес) есть выражение, ссылающееся на объект. Очевидный пример адресного выражения - имя объекта. Есть операции, дающие адресные выражения: например, если E - выражение типа указатель, то *E - адресное выражение, ссылающееся на объект, на который указывает E. Термин "lvalue" происходит из выражения присваивания E1=E2, в котором левый операнд E1 должен быть адресным (value) выражением. Ниже при обсуждении каждого оператора указывается, требует ли он адресные операнды и возвращает ли он адресное значение.

6. ПРЕОБРАЗОВАНИЯ

Определенные операции могут в зависимости от их операндов вызывать преобразование значения операнда от одного типа к другому. В этой части объясняется, каков ожидаемый результат таких преобразований. В #6.6 содержится краткое описание преобразований, требуемых наиболее стандартными операциями; оно будет дополняться по мере надобности в процессе обсуждения каждой операции. В #8.5.6 описываются преобразования, определяемые пользователем.

6.1 Символы и целые

Символ или короткое целое могут использоваться, если может использоваться целое. Во всех случаях значение преобразуется к целому. Преобразование короткого целого к длинному всегда включает в себя знаковое расширение; целые являются величинами со знаком. Содержат символы знаковый разряд или нет, является машинно

зависимым; см. #2.6. Более явный тип unsigned char ограничивает изменение значения от 0 до машинно зависимого максимума.
В машинах, где символы рассматриваются как имеющие знак (знаковые), символы множества кода ASCII являются положительными.

Однако, символьная константа, заданная восьмеричной esc-последовательностью подвергается знаковому расширению и может стать отрицательным числом; так например, '\377' имеет значение -1.

Когда длинное целое преобразуется в короткое или в char, оно урезается влево; избыточные биты просто теряются.

6.2 Float и double

Для выражений float могут выполняться действия арифметики с плавающей точкой одинарной точности. Преобразования между числами одинарной и двойной точности выполняются настолько математически корректно, насколько позволяет аппаратура.

6.3 Плавающие и целые

Преобразования плавающих значений в интегральный тип имеет склонность быть машинно зависимым. В частности, направление усечения отрицательных чисел различается от машины к машине. Если предоставляемого пространства для значения не хватает, то результат неопределен.

Преобразование интегрального значения в плавающий тип выполняются хорошо. При нехватке в аппаратной реализации требуемых бит возникает некоторая потеря точности.

6.4 Указатели и целые

Выражение целого типа можно прибавить к указателю или вычесть из него; в таком случае первый преобразуется, как указывается при обсуждении операции сложения.

Можно производить вычитание над двумя указателями на объекты одного типа; в этом случае результат преобразуется к типу int или long в зависимости от машины; см. #7.4.

6.5 Unsigned

Всегда при сочетании целого без знака и обычного целого обычное целое преобразуется к типу unsigned и результат имеет тип unsigned. Значением является наименьшее целое без знака, равное целому со знаком (mod 2**(размер слова)) (т.е. по модулю 2**(размер слова)). В дополнительном двоичном представлении это преобразование является пустым, и никаких реальных изменений в двоичном представлении не происходит.

При преобразовании целого без знака в длинное значение результата численно совпадает со значением целого без знака. Таким образом, преобразование сводится к дополнению нулями слева.

6.6 Арифметические преобразования

Большое количество операций вызывают преобразования и дают тип результата одинаковым образом. Этот стереотип будет называться "обычным арифметическим преобразованием".

Во-первых, любые операнды типа char, unsigned char или short преобразуются к типу int.

Далее, если один из операндов имеет тип double, то другой преобразуется к типу double и тот же тип имеет результат.

Иначе, если один из операндов имеет тип unsigned long, то другой преобразуется к типу unsigned long и таков же тип результата.

Иначе, если один из операндов имеет тип long, то другой преобразуется к типу long и таков же тип результата.

Иначе, если один из операндов имеет тип unsigned, то другой преобразуется к типу unsigned и таков же тип результата.

Иначе оба операнда должны иметь тип int и таков же тип результата.

6.7 Преобразования указателей

Везде, где указатели присваиваются, инициализируются, сравниваются и т.д. могут выполняться следующие преобразования.

Константа 0 может преобразовываться в указатель, и гарантируется, что это значение породит указатель, отличный от указателя на любой объект.

Указатель любого типа может преобразовываться в void*.

Указатель на класс может преобразовываться в указатель на открытый базовый класс этого класса; см. #8.5.3.

Имя вектора может преобразовываться в указатель на его первый элемент.

Идентификатор, описанный как "функция, возвращающая ...", всегда, когда он не используется в позиции имени функции в вызове, преобразуется в "указатель на функцию, возвращающую ...".

6.8 Преобразования ссылок

Везде, где инициализируются ссылки, может выполняться ссследующее преобразование.

Ссылка на класс может преобразовываться в ссылку на открытый базовый класс этого класса; см. #8.6.3.

7. ВЫРАЖЕНИЯ

Приоритет операций в выраженях такой же, как и порядок главных подразделов в этом разделе, наибольший приоритет у первого. Так например, выражения, о которых говорится как об операндах операции + (#7.4) - это те выражения, которые определены в ##7.1-7.4. Внутри каждого подраздела операции имеют одинаковый приоритет. В каждом подразделе для рассматриваемых в нем операций определяется их левая или правая ассоциативность (порядок обработки операндов). Приоритет и ассоциативность всех операций собран вместе в описании грамматики в #14.

В остальных случаях порядок вычисления выражения неопределен. Точнее, компилятор волен вычислять подвыражения в том порядке, который он считает более эффективным, даже если подвыражения вызывают побочные эффекты. Порядок возникновения побочных эффектов неопределен. Выражения, включающие в себя коммутативные и асссоциативные операции (*, +, &, |, ^), могут быть реорганизованы произвольным образом, даже при наличии скобок; для задания определенного порядка вычисления выражения необходимо использовать явную временную переменную.

Обработка переполнения и контроль деления при вычислении выражения машинно зависимы. В большинстве существующих реализаций C++ переполнение целого игнорируется; обработка деления на 0 и всех исключительных ситуаций с числами с плавающей точкой различаются от машины к машине и обычно могут регулироваться библиотечными функциями.

Кроме стандартного значения, описанного в #7.2-7.15, операции могут быть перегружены*, то есть, могут быть заданы их значения для случая их применения к типам, определяемым пользователем; см.

#7.16.

7.1 Основные выражения

Основные выражения, включающие в себя . , -> , индексирование и вызовы функций, группируются слева направо.

список_выражений:
выражение
список_выражений , выражение

id:
идентификатор

---

* Этот термин применяется для описания использования в языке одной и той же лексемы для обозначения различных процедур; вид процедуры выбирается компилятором на основании дополнительной информации в виде числа и типа аргументов и т.п.

имя_функции_операции
typedef-имя
:: идентификатор
typedef-имя :: имя_функции_операции

первичное_выражение:
id
:: идентификатор
константа
строка
this
(выражение)
первичное_выражение [выражение]
первичное_выражение (список_выражений opt)
первичное_выражение . id
первичное_выражение -> id

Идентификатор есть первичное выражение, причем соответственно описанное (#8). Имя_функции_операции есть идентификатор со специальным значением; см. #7.16 и #8.5.1.

Операция ::, за которой следует идентификатор из файловой области видимости, есть то же, что и идентификатор. Это позволяет ссылаться на объект даже в том случае, когда его идентификатор скрыт (#4.1).

Typedef-имя (#8.8) , за которым следует ::, после чего следует идентификатор, является первичным выражением. Typedef-имя должно обозначать класс (#8.5), и идентификатор должен обозначать член этого класса. Его тип специфицируется описанием идентификатора. Typedef-имя может быть скрыто именем, которое не является именем типа. В этом случае typedef-имя все равно может быть найдено и его можно использовать.

Константа является первичным выражением. Ее тип должен быть int, long или double в зависимости от ее формы.

Строка является первичным выражением. Ее тип - "массив символов". Обычно он сразу же преобразуется в указатель на ее первый символ (#6.7).

Ключевое слово this является локальной переменной в теле функции члена (см. #8.5) . Оно является указателем на объект, для которого функция была вызвана.

Выражение, заключенное в круглые скобки, является первичным выражением, чей тип и значение те же, что и у незаключенного в скобки выражения. Наличие скобок не влияет на то, является выражение lvalue или нет.

Первичное выражение, за которым следует выражение в квадратных скобках, является первичным выражением. Интуитивный смысл - индекс. Обычно первичное выражение имеет тип "указатель на ...", индексирующее выражение имеет тип int и тип результата есть "...". Выражение E1[E2] идентично (по определению) выражению *((E1)+(E2)).

Все тонкие места, необходимые для понимания этой записи, содержатся в этом разделе вместе с обсуждением в ## 7.1, 7.2 и 7.4, соответственно, идентификаторов, * и + ; ниже, в #8.4.2 приводятся следствия из этого.

Вызов функции является первичным выражением, за которым следуют скобки, содержащие список (возможно, пустой) разделенных запятыми выражений, составляющих фактические параметры для функции. Первичное выражение должно иметь тип "функция, возвращающая ..." или "указатель на функцию, возвращающую ...", и результат вызова функции имеет тип "...".

Каждый формальный параметр инициализируется фактическим параметром (#8.6). Выполняются стандартные (#6.6-8) и определяемые пользователем преобразования (#8.5.6). Функция может изменять значения своих формальных параметров, но эти изменения не могут повлиять на значения фактических параметров за исключением случая, когда формальный параметр имеет ссылочный тип.

Функция может быть описана как получающая меньше или больше параметров, чем специфицировано в описании функции (#8.4). Каждый фактический параметр типа float, для которого нет формального параметра, преобразуются к типу double; и, как обычно, имена массивов преобразуются к указателям. Порядок вычисления параметров не определен языком; имейте в виду различия между компиляторами.

Допустимы рекурсивные вызовы любых функций.

Первичное выражение, после которого стоит точка, за которой следует идентификатор (или идентификатор, уточненный typedef-именем с помощью операции ::) является выражением. Первое выражение должно быть объектом класса, а идентификатор должен именовать член этого класса. Значением является именованный член объекта, и оно является адресным, если первое выражение является адресным. Следует отметить, что "классовые объекты" могут быть структурами (#8.5.12) или объединениями (#8.5.13).

Первичное выражение, после которого стоит стрелка (->), за которой следует идентификатор (или идентификатор, уточненный typedef-именем с помощью операции ::) является выражением. Первое выражение должно быть указателем на объект класса, а идентификатор должен именовать член этого класса. Значение является адресом, ссылающимся на именованный член класса, на который указывает указательное выражение. Так, выражение E1->MOS есть то же, что и (*E1).MOS. Классы обсуждаются в #8.5.

Если первичное выражение дает значение типа "указатель на ..." (см. #8.4 and #8.6.3), значением выражения был объект, обозначаемый ссылкой. Ссылку можно считать именем объекта; см. #8.6.3.

7.2 Унарные операции

Выражения с унарными операциями группируют справа налево:

```
унарное_выражение:
     унарная_операция                                  выражение
     выражение                                                ++
     выражение                                                --
     sizeof                                            выражение
     sizeof             (              имя_типа              )
     (            имя_типа             )              выражение
     простое_имя_типа       (        список_выражений        )
     new               имя_типа       инициализатор        opt
     new              (              имя_типа               )
     delete                                            выражение
     delete [ выражение ] выражение
унарная_операция:                  одна                    из
     *  &  -  !  ~  ++  --
```

Унарная операция * означает косвенное обращение: выражение должно

быть указателем и результатом будет lvalue, ссылающееся на объект, на который указывает выражение. Если выражение имеет тип "указатель на ...", то тип результата есть "...".

Результатом унарной операции & является указатель на объект, на который ссылается операнд. Операнд должен быть lvalue. Если выражение имеет тип "...", то тип результата есть "указатель на ...".

Результатом унарной операции + является значение ее операнда после выполнения обычных арифметических преобразований. Операнд должен быть арифметического типа.

Результатом унарной операции - является отрицательное значение ее операнда. Операнд должен иметь целый тип. Выполняются обычные арифметические преобразования. Отрицательое значение беззнаковой величины вычислятся посредством вычитания ее значения из 2n, где n -число битов в целом типа int.

Результатом операции логического отрицания ! является 1, если значение операнда 0, и 0, если значение операнда не 0. Результат имеет тип int. Применима к любому арифметическому типу или к указателям.

Операция ~ дает дополнение значения операнда до единицы. Выполняются обычные арифметические преобразования. Операнд должен иметь интегральный тип.

7.2.1 Увеличение и Уменьшение

Операнд префиксного ++ получает приращение. Операнд должен быть адресным . Значением является новое значение операнда, но оно не адресное. Выражение ++x эквивалентно x+=1. По поводу данных о преобразованиях см. обсуждение операций сложения (#7.4) и присваивания (#7.14).

Операнд префиксного -- уменьшается аналогично действию префиксной операции ++.

Значение, получаемое при использовании постфиксного ++, есть значение операнда. Операнд должен быть адресным. После того, как результат отмечен, объект увеличивается так же, как и в префиксной операции ++. Тип результата тот же, что и тип операнда.

Значение, получаемое при использовании постфиксной --, есть значение операнда. Операнд должен быть адресным. После того, как результат отмечен, объект увеличивается так же, как и в префиксной операции ++. Тип результата тот же, что и тип операнда.

7.2.2 Sizeof

Операция sizeof дает размер операнда в байтах. (Байт не определяется языком иначе, чем через значение sizeof. Однако, во всех существующих реализациях байт есть пространтсво, необходимое для хранения char.) При применении к массиву результатом является полное количество байтов в массиве. Размер определяется из описаний объектов, входящих в выражение. Семантически это выражение является беззнаковой константой и может быть использовано в любом месте, где требуется константа.

Операцию sizeof можно также применять к заключенному в скобки имени типа. В этом случае она дает размер, в байтах, объекта указанного типа.

7.2.3 Явное Преобразование Типа

Простое_имя_типа (#8.2), возможно, заключенное в скобки, за которым идет заключенное в скобки выражение (или список_выражений, если тип является классом с соответствующим образом описанным конструктором #8.5.5) влечет преобразование значения выражения в названный тип. Чтобы записать преобразование в тип, не имеющий простого имени, имя_типа (#8.7) должно быть заключено в скобки.

Если имя типа заключено в скобки, выражение заключать в скобки необязательно. Такая запись называется приведением к типу.

Указатель может быть явно преобразован к любому из интегральных типов, достаточно по величине для его хранения. То, какой из int и long требуется, является машинно зависимым. Отображающая функция также является машинно зависимой, но предполагается, что она не содержит сюрпризов для того, кто знает структуру адресации в машине. Подробности для некоторых конкретных машин были приведены в #2.6.

Объект интегрального типа может быть явно преобразован в указатель. Отображающая функция всегда превращает целое, полученное из указателя, обратно в тот же указатель, но в остальных случаях является машинно зависимой.

Указатель на один тип может быть явно преобразован в указатель на другой тип. Использование полученного в результате указателя может привести к исключительной ситуации адресации, если исходный указатель не указывает на объект, соответствующим образом выравненный в памяти. Гарантируется, что указатель на объект данного размера может быть преобразован в указатель на объект меньшего размера и обратно без изменений. Различные машины могут различаться по числу бит в указателях и требованиям к выравниванию объектов. Составные объекты выравниваются по самой строгой границе, требуемой каким-либо из его составляющих.

Объект может преобразовываться в объект класса только если был описан соответствующий конструктор или операция преобразования (#8.5.6).

Объект может явно преобразовываться в ссылочный тип &X, если указатель на этот объект может явно преобразовываться в X*.

7.2.4 Свободная Память

Операция new создает объект типа имя_типа (см. #8.7), к которому он применен. Время жизни объекта, созданного с помощью new, не ограничено областью видимости, в которой он создан. Операция new возвращает указатель на созданный ей объект. Когда объект является массивом, возвращается указаетль на его первый элемент. Например, и new int и new int[10] возвращают int*. Для объектов некоторых классов надо предоставлять инициализатор (#8.6.2). Операция new (#7.2) для получения памяти вызывает функцию

```
void* operator new (long);
```

Параметр задает требуемое число байтов. Память будет инициализирована. Если operator new() не может найти требуемое количество памяти, то она возвращает ноль.

Операция delete уничтожает объект, созданный операцией new. Ее результат является void. Операнд delete должен быть указателем, возвращенным new. Результат применения delete к указателю, который не был получен с помощью операции new. Однако уничтожение с помощью delete указателя со значением ноль безвредно.

Чтобы освободить указанную память, операция delete вызывает функцию

```
void operator delete (void*);
```

В форме

```
delete [ выражение ] выражение
```

второй параметр указывает на вектор, а первое выражение задает число элементов этого вектора. Задание числа элементов является избыточным за исключением случаев уничтожения векторов некоторых классов; см. #8.5.8.

7.3 Мультипликативные операции

Мультипликативные операции *, / и % группируют слева направо. Выполняются обычные арифметические преобразования.

```
мультипликативное_выражение:
    выражение * выражение
    выражение  /  выражение
    выражение  % выражение
```

Бинарная операция * определяет умножение. Операция * ассоциативна и выражения с несколькими умножениями на одном уровне могут быть реорганизованы компилятором.

Бинарная операция / определяет деление. При делении положительных целых округление осуществляется в сторону 0, но если какой-либо из операндов отрицателен, то форма округления является машинно зависимой. На всех машинах, охватываемых данным руководством, остаток имеет тот же знак, что и делимое. Всегда истиинно, что (a/b)*b + a%b равно a (если b не 0).

Бинарная операция % дает остаток от деления первого выражения на второе. Выполняются обычные арифметические преобразования. Операнды не должны быть числами с плавающей точкой.

7.4 Аддитивные операции

Аддитивные операции + и - группируют слева направо. Выполняюься обычные арифметические преобразования. Каждая операция имеет некоторые дополнительные возможности, связанные с типами.

```
аддитивное_выражение:
    выражение + выражение
    выражение  -  выражение
```

Результатом операции + является сумма операндов. Можно суммировать указатель на объект массива и значение целого типа. Последнее во всех случаях преобразуется к смещению адреса с помощью умножения его на длину объекта, на который указывает указатель. Результатом является указатель того же типа, что и исходный указатель, уазывающий на другой объект того же массива и соответствующим образом смещенный от первоначального объекта. Так, если P есть указатель на объект массива, то выражение P+1 есть указатель на следующий объект массива.

Никакие другие комбинации типов для указателей не допустимы.

Операция + ассоциативна и выражение с несколькими умножениями на одном уровне может быть реорганизовано компилятором.

Результатом операции - является разность операндов. Выполняюься обычные арифметические преобразования. Кроме того, значение любого целого типа может вычитаться из указателя, в этом случае применяются те же преобразования, что и к сложению.

Если вычитаются указатели на объекты одного типа, то результат преобразуется (посредством деления на длину объекта) к целому, представляющему собой число объектов, разделяющих объекты, указанные указателями. В засисимости от машины результирующее целое может быть или типа int, или типа long; см. #2.6. Вообще говоря, это преобразование будет давать неопределенный результат кроме тех случаев, когда указатели указывают на объекты одного массива, поскольку указатели, даже на объекты одинакового типа, не обязательно различаются на величину, кратную длине объекта.

7.5 Операции сдвига

Операции сдвига << и >> группируют слева направо. Обе выполняют

одно обычное арифметическое преобразование над своими операндами, каждый из которых должен быть целым. В этом случае правый операнд преобразуется к типу int; тип результата совпадает с типом левого операнда. Результат неопределен, если правый операнд отрицателен или больше или равен длине объекта в битах.

сдвиговое_выражение:
выражение << выражение
выражение >> выражение

Значением E1 << E2 является E1 (рассматриваемое как битовое представление), сдвинутое влево на E2 битов; освободившиеся биты заполняются нулями. Значением E1 >> E2 является E1 , сдвинутое вправо на E2 битовых позиций. Гарантируется, что сдвиг вправо является логическим (заполнение нулями), если E1 является unsigned; в противном случае он может быть арифметичевким (заполнение копией знакового бита).

7.6 Операции отношения

Операции отношения (сравнения) группируют слева направо, но этот факт не очень-то полезен: a < b < c не означает то, чем кажется.

выражение_отношения:
выражение < выражение
выражение > выражение
выражение <= выражение
выражение >= выражение

Операции < (меньше чем), > (больше чем), <= и >= все дают 0, если заданное соотношение ложно, и 1, если оно истинно. Тип результата int. Выполняются обычные арифметические преобразования. Могут сравниваться два указателя; результат зависит от относительного положения объектов, на которые указывают указатели, в адресном

пространстве. Сравнение указателей переносимо только если указатели указывают на объекты одного массива.

7.7 Операции равенства

выражение_равенства:
выражение == выражение
выражение != выражение

Операции == и != в точности аналогичны операциям сравнения за исключением их низкого приоритета. (Так, a
7.8 Операция побитовое И

И-выражение: выражение & выражение

Операция & ассоциативна, и выражения, содержащие &, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция И операндов. Операция применяется только к целым операндам.

7.9 Операция побитовое исключающее ИЛИ

исключающее_ИЛИ_выражение:
выражение ^ выражение

Операция ^ ассоциативна, и выражения, содержащие ^, могут реорганизовываться. Выполняются обычные арифметические

преобразования; результатом является побитовая функция исключающее ИЛИ операндов. Операция применяется только к целым операндам.

7.10 Операция побитовое включающее ИЛИ

включающее_ИЛИ_выражение:
выражение | выражение

Операция | ассоциативна, и выражения, содержащие |, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция включающее ИЛИ операндов. Операция применяется только к целым операндам.

7.11 Операция логическое И

логическое_И_выражение:
выражение && выражение

Операция && группирует слева направо. Она возвращает 1, если оба операнда ненулевые, и 0 в противном случае. В противоположность операции & операция && гарантирует вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд есть 0.

Операнды не обязаны иметь один и тот же тип, но каждый из них должен иметь один из основных типов или быть указателем. Результат всегда имеет тип int.

7.12 Операция логическое ИЛИ

логическое_ИЛИ_выражение:
выражение || выражение

Операция || группирует слева направо. Она возвращает 1, если хотя бы один из ее операндов ненуелвой, и 0 в противном случае. В противоположность операции | операция || гарантирует вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд не есть 0.

Операнды не обязаны иметь один и тот же тип, но каждый из них должен иметь один из основных типов или быть указателем. Результат всегда имеет тип int.

7.13 Условная операция

условное_выражение:
выражение ? выражение : выражение

Условная операция группирует слева направо. Вычисляется первое выражение, и если оно не 0, то результатом является значение второго выражения, в противном случае значение третьего выражения. Если это возможно, то выполняются обычные арифметические преобразования для приведения второго и третьего выражения к общему типу. Если это возможно, то выполняются преобразования указателей для приведения второго и третьего выражения к общему типу. Вычисляется только одно из второго и третьего выражений.

7.14 Операции присваивания

Есть много операций присваивания, все группируют слева направо. Все в качестве левого операнда требуют lvalue, и тип выражения присваивания тот же, что и у его левого операнда. Это lvalue не может ссылаться на константу (имя массива, имя функции или const). Значением является значение, хранящееся в левом операнде просле выполнения присваивания.

выражение_присваивания:
выражение операция_присваивания выражение

операция_присваивания: одна из
= += -= *= /= %= >>= <<= &= ~= |=

В простом присваивании с = значение выражения замещает собой значение объекта, на который ссылается операнд в левой части. Если оба операнда имеют арифметический тип, то при подготовке к присваиванию правый операнд преобразуется к типу левого. Если

аргумент в левой части имеет указательный тип, аргумент в правой части должен быть того же типа или типа, который может быть преобразован к нему, см. #6.7. Оба операнда могут быть объектами одного класса. Могут присваиваться объекты некоторых производных классов; см. #8.5.3.

Присваивание объекту типа "указатель на ..." выполнит присваивание объекту, денотируемому ссылкой.

Выполнение выражения вида E1 op= E2 можно представить себе как эквивалентное E1 = E1 op (E2); но E1 вычисляется только один раз. В += и -= левый операнд может быть указателем, и в этом случае (интегральный) правый операнд преобразуется так, как объяснялось в #7.4; все правые операнды и не являющиеся указателями левые должны иметь арифметический тип.

7.15 Операция запятая

```
запятая_выражение:
    выражение , выражение
```

Пара выражений, разделенных запятой, вычисляется слева направо, значение левого выражения теряется. Тип и значение результата являются типом и значением правого операнда. Эта операция группирует слева направо. В контексте, где запятая имеет специальное значение, как например в списке фактических параметров функции (#7.1) и в списке инициализаторов (#8.6), операция запятая, как она описана в этом разделе, может появляться только в скобках; например,

```
f (a,(t=3,t+2),c)
```

имеет три параметра, вторым из которых является значение 5.

7.16 Перегруженные операции

Большинство операций может быть перегружено, то есть, описано так, чтобы они получали в качестве операндов объекты классов (см. #8.5.11). Изменить приоритет операций невозможно. Невозможно изменить смысл операций при применении их к неклассовым объектам. Предопределенный смысл операций = и & (унарной) при применении их к объектам классов может быть изменен.

Эквивалентность операций, применяемых к основным типам (например, ++a эквивалентно a+=1), не обязательно выполняется для операций, применяемых к классовым типам. Некоторые операции, например, присваивание, в случае применения к основным типам требуют, чтобы операнд был lvalue; это не требуется для операций, описанных для классовых типов.

7.16.1 Унарные операции

Унарная операция, префиксная или постфиксная, может быть определена или с помощью функции члена (см. #8.5.4), не получающей параметров, или с помощью функции друга (см. #8.5.10), получающей один параметр, но не двумя способами одновременно. Так, для любой унарной операции @, x@ и @x могут интерпретироваться как x.операция@() или операция@(x). При перегрузке операций ++ и -- невозможно различить префиксное и постфиксное использование.

7.16.2 Бинарные операции

Бинарная операция может быть определена или с помощью функции члена (см. #8.5.4), получающей один параметр, или с помощью функции друга (см. #8.5.9), получающей два параметра, но не двумя способами одновременно. Так, для любой бинарной операции @, x@y может быть

проинтерпретировано как x.операция@(y) или операция@(x,y).

7.16.3 Особые операции

Вызов функции

```
первичное_выражение ( список_выражений opt )
```

и индексирование

```
первичное_выражение [ выражение ]
```

считаются бинарными операциями. Именами определяющей функции являются соответсвенно operator() и operator[]. Обращение x(arg) интерпретируется как x.operator()(arg) для классового объекта x. Индексирование x[y] интерпретируется как x.operator[](y).

8. ОПИСАНИЯ

Описания используются для определения интерпретации, даваемой каждому идентификатору; они не обязательно резервируют память, связанную с идентификатором. Описания имеют вид:

```
описание:
    спецификаторы_описания opt список_описателей opt ;
    описание_имени
    asm_описание
```

Описатели в списке_описателей содержат идентификаторы, подлежащие описанию. Спецификаторы_описания могут быть опущены только в определениях внешних функций (#10) или в описаниях внешних функций. Список описателей может быть пустым только при описании класса (#8.5) или перечисления (#8.10), то есть, когда спецификаторы_описания - это class_спецификатор или enum_спецификатор. Описания имен описываются в #8.8; описания asm описаны в #8.11.

```
спецификатор_описания:
    sc_спецификатор
    спецификатор_типа
    фнк_спецификатор
    friend
    typedef

спецификаторы_описания:
    спецификатор_описания спецификатор_описания opt
```

Список должен быть внутренне непротиворечив в описываемом ниже смысле.

8.1 Спецификаторы класса памяти

Спецификаторы "класса памяти" (sc-спецификатор) это:

```
sc-спецификатор:
    auto
    static
    extern
    register
```

Описания, использующие спецификаторы auto, static и register также служат определениями тем, что они вызывают резервирование соответствующего объема памяти. Если описание extern не является

определением (#4.2), то где-то еще должно быть определение для данных идентификаторов.

Описание register лучше всего представить как описание auto (автоматический) с подсказкой компилятору, что описанные переменные усиленно используются. Подсказка может быть проигнорирована. К ним не может применяться операция получения адреса &.

Спецификаторы auto или register могут применяться только к именам, описанным в блоке, или к формальным параметрам. Внутри

блока не может быть описаний ни статических функций, ни статических формальных параметров.

В описании может быть задан максимум один sc_спецификатор. Если в описании отсутсвует sc_спецификатор, то класс памяти принимается автоматическим внутри функции и статическим вне. Исключение: функции не могут быть автоматическими.

Спецификаторы static и extern могут использоваться только для имен объектов и функций.

Некоторые спецификаторы могут использоваться только в описаниях функций:

```
фнк-спецификатор:
    overload
    inline
    virtual
```

Спецификатор перегрузки overload делает возможным использование одного имени для обозначения нескольких функций; см. #8.9.

Спецификатор inline является только подсказкой компилятору, не влияет на смысл программы и может быть проигнорирован. Он используется, чтобы указать на то, что при вызове функции inline-подстановка тела функции предпочтительнее обычной реализацци вызова функции. Функция (#8.5.2 и #8.5.10), определенная внутри описания класса, является inline по умолчанию.

Спецификатор virtual может использоваться только в описаниях членов класса; см. #8.5.4.

Спецификатор friend используется для отмены правил скрытия имени для членов класса и может использоваться только внутри описаний классов; см. #8.5.9.

С помощью спецификатора typedef вводится имя для типа; см. #8.8.

8.2 Спецификаторы Типа

Спецификаторами типов (спецификатор_типа) являются:

```
спецификатор_типа:
    простое_имя_типа
    class_спецификатор
    enum-спецификатор
    сложный_спецификатор_типа
    const
```

Слово const можно добавлять к любому допустимому спецификатору_типа. В остальных случаях в описании может быть дано не более одного спецификатора_типа. Объект типа const не является lvalue. Если в описании опущен спецификатор типа, он принимается int.

```
простое_имя_типа:
    char
    short
    int
    long
```

```
    unsigned
    float
    double
    const
    void
```

Слова long, short и unsigned можно рассматривать как прилагательные. Они могут применяться к типу int; unsigned может также применяться к типам char, short и long.

Спецификаторы класса и перечисления обсуждаются в #8.5 и #8.10 соответственно.

```
сложный_спецификатор_типа:
    ключ typedef-имя
    ключ идентификатор

ключ:
    class
    struct
    union
    enum
```

Сложный спецификатор типа можно использовать для ссылки на имя класса или перечисления там, где имя может быть скрыто локальным именем. Например:

```
class x { ... };

void f(int x)
{
    class x a;
    // ...
}
```

Если имя класса или перечисления ранее описано не было, сложный_спецификатор_типа работает как описание_имени; см. #8.8.

8.3 Описатели

Список_описателей, появляющийся в описании, есть разделенная запятыми последовательность описателей, каждый из которых может иметь инициализатор.

```
список_описателей:
    иниц_описатель
    иниц_описатель  ,  список_описателей



иниц_описатель:
    описатель  инициализатор opt
```

Инициализаторы обсуждаются в #8.6. Спецификатор в описании указывает тип и класс памяти объектов, к которым относятся описатели. Описатели имеют синтаксис:

```
описатель:
    оп_имя
    ( описатель )
    * const opt  описатель
    & const opt  описатель
    описатель   ( список_описаний_параметров )
    описатель     [ константное_выражение opt ]

оп-имя:
    простое_оп_имя
```

typedef-имя :: простое_оп_имя

простое_оп_имя:
идентификатор
typedef-имя
~ typedef-имя
имя_функции_операции
имя_функции_преобразования

Группировка та же, что и в выражениях.

8.4 Смысл описателей

Каждый описатель считается утверждением того, что если в выражении возникает конструкция, имеющаяя ту же форму, что и описатель, то она дает объект указанного типа и класса памяти. Каждый описатель содержит ровно одно оп_имя; оно определяет описываемый идентификатор. За исключением описаний некоторых специальных функций (см. #8.5.2) , оп_имя будет простым идентификатором.

Если в качестве описателя возникает ничем не снабженный идентификатор, то он имеет тип, указанный спецификатором, возглавляющим описание.

Описатель в скобках эквивалентен описателю без скобок, но связку сложных описателей скобки могут изменять.

Теперь представим себе описание

```
T D1
```

где T - спецификатор типа (как int и т.д.), а D1 - описатель. Допустим, что это описание заставляет идентификатор иметь тип "... T", где "..." пусто, если идентификатор D1 есть просто обычый идентификатор (так что тип x в "int x" есть просто int). Тогда, если D1 имеет вид

```
*D
```

то тип содержащегося идентификатора есть "... указатель на T."

Если D1 имеет вид

```
* const D
```

то тип содержащегося идентификатора есть "... константный указатель на T", то есть, того же типа, что и *D, но не lvalue.

Если D1 имеет вид

```
&D
```

или

```
& const D
```

то тип содержащегося идентификатора есть "... ссылка на T." Поскольку ссылка по определению не может быть lvalue, использование const излишне. Невозможно иметь ссылку на void (void&).

Если D1 имеет вид

```
D (список_описаний_параметров)
```

то содержащийся идентификатор имеет тип "... функция, принимающая параметр типа список_описаний_параметров и возвращающая T."

список_описаний_параметров:

```
    список_описаний_парам opt ... opt

  список_описаний_парам:
    список_описаний_парам  ,  описание_параметра
    описание_параметра

  описание_параметра:
    спецификаторы_описания  описатель
    спецификаторы_описания  описатель  = выражение
    спецификаторы_описания  абстракт_описатель
    спецификаторы_описания  абстракт_описатель  =  выражение
```

Если список_описаний_параметров заканчивается многоточием, то о числе параметров известно лишь, что оно равно или больше числа специфицированных типов параметров; если он пуст, то функция не получает ниодного параметра. Все описания для функции должны согласовываться и в типе возвращаемого значения, а также в числе и типе параметров.

Список_описаний_параметров используется для проверки и преобразования фактических параметров и для контроля присваивания указателю на функцию. Если в описании параметра специфицировано выражение, то это выражение используется как параметр по умолчанию. Параметры по умолчанию будут использоваться в вызовах, где опущены стоящие в хвосте параметры. Параметр по умолчанию не может переопределяться более поздними описаниями. Однако, описание может добавлять параметры по умолчанию, не заданные в предыдущих описаниях.

Идентификатор может по желанию быть задан как имя параметра. Если он присутствует в описании функции, его использовать нельзя, поскольку он сразу выходит из области видимости. Если он присутствует в определении функции (#10), то он именует фармальный параметр.

Если D1 имеет вид

```
  D[ константное_выражение]
```

или

```
  D[]
```

то тип содержащегося идентификатора есть "... массив объектов типа T". В первом случае константное_выражение есть выражение, значение которого может быть определено во время компиляции, и тип которого int. (Константные выражения определены в #12.) Если подряд идут несколько спецификаций "массив из", то создается многомерный массив; константное выражение, определяющее границы массива, может быть опущено только для первого члена последовательности. Этот пропуск полезен, когда массив является внешним, и настоящее определение, которое резервирует память, находится в другом месте. Первое константное выражение может также быть опущено, когда за описателем следует инициализация. В этом случае используется размер, вычисленный исходя из числа начальных элементов.

Массив может быть построен из одного из основных типов, из указателей, из структуры или объединения или из другого массива (для получения многомерного массива).

Не все возможности, которые позволяет приведенный выше синтаксис, допустимы. Ограничения следующие: функция не может возвращать массив или функцию, хотя она может возвращать указатели на эти объекты; не существует массивов функций, хотя могут быть массивы указателей на функции.

8.4.1 Примеры

В качестве примера, описание

```
int i;
int *ip;
int f ();
int *fip ();
int (*pfi) ();
```

описывает целое i, указатель ip на целое, функцию f, возвращающую целое, функцию fip , возвращающую указатель на целое, и указатель pfi на функцию, возвращающую целое. Особенно полезно сравнить последние две. Цепочка *fip() есть *(fip()), как предполагается в описании, и та же конструкция требуется в выражении, вызов функции fip, и затем косвенное использование результата через (указатель) для получения целого. В описателе (*pfi)() внешние скобки необходимы, поскольку они также входят в выражение, для указания того, что функция получается косвенно через указатель на фунцию, которая затем вызывается; это возвращает целое. Функции f и fip описаны как не получающие параметров, и fip как указывающая на функцию, не получающую параметров.

Описание

```
const a = 10, *pc = &a, *const cpc = pc;
int b, *const cp = &b;
```

описывает a: целую константу, pc: указатель на целую константу, cpc: константный указатель на целую константу, b: целое и cp: константный указатель на целое. Значения a, cpc и cp не могут быть изменены после инициализации. Значение pc может быть изменено, как и объект, указываемый cp. Примеры недопустимых выражений :

```
a = 1;
a++;
*pc = 2;
cp = &a;
cpc++;
```

Примеры допустимых выражений :

```
b = a;
*cp = a;
pc++;
pc = cpc;
```

Описание

```
fseek (FILE*,long,int);
```

описывает функцию, получающую три параметра специальных типов. Поскольку тип возвращаемого значения не определен, принимается, что он int (#8.2). Описание

```
point (int = 0,int = 0);
```

описывает функцию, которая может быть вызвана без параметров, с одним или двумя параметрами типа int. Например

```
point (1,2);
point (1)             /* имеет смысл point (1,0); */
point ()              /* имеет смысл point (0,0); */
```

Описание

```
printf (char* ... );
```

описывает функцию, которая может быть вызываться с различными числом и типами параметров. Например

```
printf ("hello, world");
printf ("a=%d b=%d",a,b);
printf ("string=%s",st);
```

Однако, она всегда должна иметь своим первым параметром char*.
В качестве другого примера,

```
float fa[17], *afp[17];
```

описывает массив чисел с плавающей точкой и массив указателей на числа с плавающей точкой. И, наконец,

```
static int x3d[3][5][7];
```

описывает массив целых, размером 3x6x7. Совсем подробно: x3d является массивом из трех элементов; каждый из элементов является массивом из пяти элементов; каждый из последних элементов является массивом из семи целых. Появление каждое из выражений x3d, x3d[i], x3d[i][j], x3d[i][j][k] может быть приемлемо. Первые три имеют тип "массив", последний имеет тип int.

8.5 Описания классов

Класс специфицирует тип. Его имя становится typedef-имя (см. #8.8), которое может быть использовано даже внутри самого спецификатора класса. Объекты класса состоят из последовательности членов.

```
спецификатор_класса:
        заголовок_класса     {      список_членов      opt      }
        заголовок_класса         {  список_членов  opt  public  :
список_членов                        opt                       }

заголовок_класса:
        агрег                  идентификатор                  opt
        агрег идентификатор opt : public opt typedef-имя
агрег:
        class
        struct
        union
```

Структура является классом, все члены которого общие; см. #8.5.8. Объединение является классом, содержащим в каждый момент только один член; см. #8.5.12. Список членов может описывать члены вида: данные, функция, класс, определение типа, перечисление и поле. Поля обсуждаются в #8.5.13. Список членов может также содержать описания, регулирующие видимость имен членов; см. #8.5.8.

```
список_членов:
        описание_члена  список_членов opt
описание_члена:
        спецификаторы_описания        opt        описатель_члена;
описатель_члена:
        описатель                          идентификатор   opt   :
константное_выражение
```

Члены, являющиеся классовыми объектами, должны быть объектами предварительно полностью описанных классов. В частности, класс cl не может содержать объект класса cl, но он может содержать указатель на объект класса cl.

Имена обектов в различных классах не конфликтуют между собой и с обычными переменными.

Вот простой пример описания структуры:

```
struct tnode
    {
      char tword[20];
      int count;
      tnode *left;
      tnode *right;
    };
```

содержащей массив из 20 символов, целое и два указателя на такие же структуры. Если было дано такое описание, то описание

```
tnode s, *sp
```

описывает s как структуру данного сорта и sp как указатель на структуру данного сорта. При наличии этих описаний выражение

```
sp->count
```

ссылается на поле count структуры, на которую указывает sp;

```
s.left
```

ссылается на указатель левого поддерва структуры s; а

```
 s.right->tword[0]
```

ссылается на первый символ члена tword правого поддерва структуры s.

8.5.1 Статические члены

Член-данные класса может быть static; члены-функции не могут. Члены не могут быть auto, register или extern. Есть единственная копия статического члена, совместно используемая всеми членами класса в программе. На статический член mem класса cl можно ссылаться cl:mem, то есть без ссылки на объект. Он существует, даже если не было создано ниодного объекта класса cl.

8.5.2 Функции члены

Функция, описанная как член, (без спецификатора friend (#8.5.9)) называется функцией членом и вызывается с помощью синтаксиса члена класса (#7.1). Например:

```
struct tnode
  {
    char tword[20];
    int count;
    tnode *left;
    tnode *right;
    void set (char* w,tnode* l,tnode* r);
  };

tnode n1, n2;

n1.set ("asdf",&n2,0);
n2.set ("ghjk",0,0);
```

Определение функции члена рассматривается как находящееся в области видимости ее класса. Это значит, что она может непосредственно использовать имена ее класса. Если определение функции члена находится вне описания класса, то имя функции члена должно быть уточнено именем класса с помощью записи

typedef-имя . простое_оп_имя
см. 3.3. Определения функций обсуждаются в #10.1. Например:

```
void tnode.set (char* w,tnode* l,tnode* r)
  {
    count = strlen (w);
    if (sizeof (tword) <= count) error ("tnode string too long");
    strcpy (tword,w);
    left = l;
    right = r;
  }
```

Имя функции tnode.set определяет то, что множество функций является членом класса tnode. Это позволяет использовать имена членов **word**, **count**, **left** и **right**. В функции члене имя члена ссылается на объект, для которого была вызвана функция. Так, в вызове n1.set(...) tword ссылается на n1.tword, а в вызове n2.set(...) он ссылается на n2.tword. В этом примере предполагается, что функции strlen, error и strcpy описаны где-то в другом месте как внешние функции (см. #10.1).

В члене функции ключевое слово this указывает на объект, для которого вызвана функция. Типом this в функции, которая является членом класса cl, является cl*. Если mem - член класса cl,то mem и **this->mem** - синонимы в функции члене класса cl (если mem не был использован в качестве имени локальной переменной в промежуточной области видимости).

Функция член может быть определена (#10.1) в описании класса. Помещение определения функции члена в описание класса является кратким видом записи описания ее в описаниии класса и затем определения ее как inline (#8.1) сразу после описания класса. Например:

```
int b;
struct x
  {
    int f () { return b; }
    int f () { return b; }
    int b;
  };
```

означает

```
int b;
struct x
  {
    int f ();
    int b;
  };
inline x.f () { return b; }
```

Для функций членов членов не нужно использование спецификатора overload (#8.2): если имя описывается как означающее несколько имен в классе, то оно перегружено (см. #8.9).

Применение операции получения адреса к функциям членам допустимо. Тип параметра результирующей функции указатель на есть (...), то есть, неизвестен (#8.4). Любое использование его является зависимым от реализации, поскольку способ инициализации указателя для вызова функции члена неопределен.

8.5.3 Производные классы

В конструкции
агрег идентификатор:public opt typedef-имя
typedef-имя должно означать ранее описанный класс, называемый

базовым классом для класса, подлежащего описанию. Говорится, что последний выводится из предшествующего. На члены базового класса можно ссылаться, как если бы они были членами производного класса, за исключением тех случаев, когда имя базового члена было переопределено в производном классе; в этом случае для ссылки на скрытое имя может использоваться такая запись (#7.1):

typedef-имя :: идентификатор

Например:

```
struct base
  {
    int a;
    int b;
  };

struct derived : public base
  {
    int b;
    int c;
  };

derived d;

d.a = 1;
d.base::b = 2;
d.b = 3;

d.c = 4;
```

осуществляет присваивание четырем членам d.

Производный тип сам может использоваться как базовый.

8.5.4 Виртуальные функции

Если базовый класс base содержит (виртуальную) virtual (#8.1) функцию vf, а производный класс derived также содержит функцию vf, то вызов vf для объекта класса derived вызывает derived::vf. Например:

```
struct base
  {
    virtual void vf ();
    void f ();
  };

 struct derived : public base
  {
    void vf ();
    void f ();
  };

derived d;
base* bp = &d;

bp->vf ();
bp->f ();
```

Вызовы вызывают, соответственно, derived::vf и base::f для объекта класса derived, именованного d. Так что интерпретация вызова виртуальной функции зависит от типа объекта, для которого она вызвана, в то время как интерпретация вызова невиртуальной функции зависит только от типа указателя, обозначающего объект.

Из этого следует, что тип объектов классов с виртуальными функциями и объектов классов, выведенных из таких классов, могут

быть определены во время выполнения.

Если производный класс имеет член с тем же именем, что и у виртуальной функции в базовом классе, то оба члена должны иметь одинаковый тип. Виртуальная функция не может быть другом (friend) (#8.5.9). Функция f в классе, выведенном из класса, который имеет виртуальную функцию f, сама рассматривается как виртуальная. Виртуальная функция в базовом классе должна быть определена. Виртуальная функция, которая была определена в базовом классе, не нуждается в определении в производном классе. В этом случае функция, определенная для базового класса, используется во всех вызовах.

8.5.5 Конструкторы

Член функция с именем, совпадающим с именем ее класса, называется конструктором. Конструктор не имеет типа возвращаемого значения; он используется для конструирования значений с типом его класса. С помощью конструктора можно создавать новые объекты его типа, используя синтаксис

typedef-имя (список_параметров opt)

Например,

```
complex zz = complex (1,2.3);

cprint (complex (7.8,1.2));
```

Объекты, созданные таким образом, не имеют имени (если конструктор не использован как инициализатор, как это было с zz выше), и их время жизни ограничено областью видимости, в которой они созданы. Они не могут рассматриваться как константы их типа. Если класс имеет конструктор, то он вызывается для каждого объекта этого класса перед тем, как этот объект будет как-либо использован; см. #8.6.

Консруктор может быть overload, но не virtual или friend.

Если класс имеет базовый класс с конструктором, то конструктор для базового класса вызывается до вызова конструктора для производного класса. Конструкторы для объектов членов, если таковые есть, выполняются после конструктора базового класса и до конструктора объекта, содержащего их. Объяснение того, как могут быть специфицированы параметры для базового класса, см. в #8.6.2, а того, как конструкторы могут использоваться для управления свободной памятью, см. в #17.

8.5.6 Преобразования

Конструктр, получающий один параметр, определяет преобразование из типа своего параметра в тип своего класса. Такие преобразования неявно применяются дополнительно к обычным арифметическим пробразованиям. Поэтому присваивание объекту из класса X допустимо, если или присваиваемое значение является X, или если X имеет конструктор, который получает присваиваемое значение как свой единственный параметр. Аналогично конструкторы используются для преобразования параметров функции (#7.1) и инициализаторов (#8.6). Например:

```
class X { ... X (int); };
f (X arg)
  {
    X a = 1;          /* a = X (1) */
    a = 2;            /* a = X (2) */
```

```
    f (3);            /* f (X (3)) */
  }
```

Если для класса X не найден ниодин конструктор, принимающий присваиваемый тип, то не делается никаких попыток отыскать конструктор для преобразования присваиваемого типа в тип, который мог бы быть приемлем для конструкторов класса X. Например:

```
class X { ... X (int); };
class X { ... Y (X); };

Y a = 1;              /* недопустимо: Y (X (1)) не пробуется */
```

8.5.7 Деструкторы

Функция член класса cl с именем ~cl называется деструктором. Деструктор не возвращает никакого значения и не получает никаких параметров; он используется для уничтожения значений типа cl непосредственно перед уничтожением содержащего их объекта. Деструктор не может быть overload, virtual или friend.

Деструктор для базового класса выполняется после деструктора производного от него класса. Как деструкторы используютя для управления свободной памятью, см. объяснение в #17.

8.5.8 Видимость имен членов

Члены класса, описанные с ключевым словом class, являются закрытыми, это значит, что их имена могут использоваться только функциями членами (#8.5.2) и друзьями (см. #8.5.10), пока они не появятся после метки public: . В этом случае они являются общими. Общий член может использоваться любой функцией. Структура является классом, все члены которого общие; см. #8.5.11.

Если перед именем базового класса в описании производного класса стоит ключевое слово public, то общие члены базового класса являются общими для производного клаcа; если нет, то они являются закрытыми. Общий член mem закрытого базового класса base может быть описан как общий для производного класса с помощью опиисания вида

typedef-имя . идентификатор;

в котором typedef-имя означает базовый класс, а идентификатор есть имя члена базового класса. Такое описание может появляться в общей части производного класса.

Рассмотрим

```
class base
  {
    int a;
public:
    int b,c;
    int bf ();
  };

class derived : base
  {
    int d;
public:
    base.c;
    int e;
    int df ();
  };

int ef (derived&);
```

Внешняя функция ef может использовать только имена c, e и df.

Являясь членом derived, функция df может использовать имена b, c, bf, d, e и df, но не a. Являясь членом base, функция bf может использовать члены a, b, c и bf.

8.5.9 Друзья (friends)

Другом класса является функция не-член, которая может использовать имена закрытых членов. Следующий пример иллюстрирует различия между членами и друзьями:

```
class private
  {
    int a;
    friend void friend_set (private*,int);
public:
    void member_set (int);
  };

void friend_set (private* p,int i) { p->a=i; }

void private.member_set (int i) { a = i; }

private obj;

friend_set (&obj,10);

obj.member_set (10);
```

Если описание friend относится к перегруженному имени или операции, то другом становится только функция с описанными типами параметров. Все функции класса cl1 могут быть сделаны друзьями класса cl2 с помощью одного описания

```
class cl2
  {
    friend cl1;
    . . .
  };
```

8.5.10 Функция операция

Большинство операций могут быть перегружены с тем, чтобы они могли получать в качестве операндов объекты класса.

имя_функции_операции: operator op

```
op:             +   -   *   /   %   ^   &   |   ~
                !   =   <   >   +=  -=  *=  /=  %=
                ^=  &=  |=  <<  >>  <<= >>= ==  !=
                <=  >=  &&  ||  ++  --  ()  []
```

Последние две операции - это вызов функции и индексирование. Функция операция может или быть функцией членом, или получать по меньшей мере один параметр класса. См. также #7.16.

8.5.11 Структуры

Структура есть класс, все члены которого общие. Это значит, что

```
struct s { ... };
```

эквивалентно

```
class s { public: ... };
```

Структура может иметь функции члены (включая конструкторы и деструкторы).

8.5.12 Объединения

Объединение можно считать структурой, все объекты члены которой начинаются со смещения 0, и размер которой достаточен для содержания любого из ее объектов членов. В каждый момент времени в объединеии может храниться не больше одного из объектов членов. Объединение может иметь функции члены (включая конструкторы и деструкторы).

8.5.13 Поля бит

Описатель члена вида

идентификатор opt: константное_выражение

определяет поле; его длина отделяется от имени поля двоеточием. Поля упаковываются в машинные целые; они не являются альтернативой слов. Поле , не влезающее в оставшееся в целом место, помещается в следующее слово. Поле не может быть шире слова. На некоторых машинах они размещаются справа налево, а на некоторых слева направо; см. #2.6.

Неименованные поля полезны при заполнении для согласования внешне предписанных размещений (форматов). В особых случаях неименованные поля длины 0 задают выравнивание следующего поля по границе слова. Не требуется аппаратной поддержки любых полей, кроме целых. Более того, даже целые поля могут рассматриваться как unsigned. По этим причинам рекомендуется описывать поля как unsigned. К полям не может применяться операция получения адреса &, поэтому нет указателей на поля.

Поля не могут быть членами объединения.

8.5.14 Вложенные классы

Класс может быть описан внутри другого класса. В этом случае область видимости имен внутреннего класса его и общих имен ограничивается охватывающим классом. За исключением этого ограничения допустимо, чтобы внутренний класс уже был описан вне охватывающего класса. Описание одного класса внутри другого не влияет на правила доступа к закрытым членам и не помещает функции члены внутреннего класса в область видимости охватывающего класса. Например:

```
int x;

class enclose   /* охватывающий */
  {
    int x;
    class inner
      {
        int y;
        f () { x=1 }
        ...
      };
    g (inner*);
    ...
  };

int inner;  /* вложенный */

enclose.g (inner* p) { ... }
```

В этом примере x в f ссылается на x, описанный перед классом enclose. Поскольку y является закрытым членом inner, g не может его использовать. Поскольку g является членом enclose, имена,

использованные в g, считаются находящимися в области видимости класса enclose. Поэтому inner в описании параметров g относится к охваченному типу inner, а не к int.

8.6 Инициализация

Описание может задавать начальное значение описываемого идентификатора. Инициализатору предшествует =, и он состоит из выражения или списка значений, заключенного в фигурные скобки.

```
инициализатор:
        =                                              expression
        =           {           список_инициализаторов           }
        =       {       список_инициализаторов       ,       }
        (                   список_выражений                   )

список_инициализаторов                                         :
        выражение
        список_инициализаторов      ,      список_инициализаторов
        { список_инициализаторов }
```

Все выражения в инициализаторе статической или внешней переменной должны быть константными выражениями, которые описаны в #15, или выражениями, которые сводятся к адресам ранее описанных переменных, возможно со смещением на константное выражение. Автоматические и регистровые переменные могут инициализироваться любыми выражениями, включающими константы, ранее описанные переменные и функции.

Гарантируется, что неинициализированные статические и внешние переменные получают в качестве начального значения "пустое значение"*. Когда инициализатор применяется к скаляру (указатель или объект арифметического типа), он состоит из одного выражения, возможно, заключенного в фигурные скобки. Начальное значение объекта находится из выражения; выполняются те же преобразования, что и при присваивании.

Заметьте, что поскольку () не является инициализатором, то "X a();" является не описанием объекта класса X, а описанием функции, не получающей значений и возвращающей X.

8.6.1 Список инициализаторов

Когда описанная переменная является составной (класс или массив), то инициализатор может состоять из заключенного в фигурные скобки, разделенного запятыми списка инициализаторов для членов составного объекта, в порядке возрастания индекса или по порядку членов. Если массив содерхит составные подобъекты, то это правило рекурсивно применяется к членам составного подобъекта. Если инициализаторов в списке меньше, чем членов в составном подобъекте, то составной подобъект дополняется нулями.

Фигурные скобки могут опускаться следующим образом. Если инициализатор начинается с левой фигурной скобки, то следующий за ней список инициализаторов инициализирует члены составного объекта; наличие числа инициализаторов, большего, чем число членов, считается ошибочным. Если, однако, инициализатор не начинается с левой фигурной скобки, то из списка берутся только элементы, достаточные для сопоставления членам составного объекта, частью которого является текущий составной объект.

Например,

```
int x[] = { 1, 3, 5 };
```

описывает и инициализирует x как одномерный массив, имеющий три члена, поскольку размер не был указан и дано три инициализатора.

---

* В английском "garbage", означающее затертое место [памяти], т.е. если переменная целая, то 0, если char, то '\0', если

указатель на T, то (T*) NULL.

```
float y[4][3] =
    {
      { 1, 3, 5 },
      { 2, 4, 6 },
      { 3, 5, 7 }
    };
```

является полностью снабженной квадратными скобками инициализацией: 1,3 и 5 инициализируют первый ряд массива y[0], а именно, y[0][2]. Аналогично, следующие две строки инициализируют y[1] и y[2]. Инициализатор заканчивается раньше, поэтому y[3] инициализируется значением 0. В точности тот же эффект может быть достигнут с помощью

```
float y[4][3] =
  {
    1, 3, 5, 2, 4, 6, 3, 5, 7
  };
```

Инициализатор для y начинается с левой фигурной скобки, но не начинается с нее инициализатор для y[0], поэтому используется три значения из списка. Аналогично, следующие три успешно используются для y[1] и следующие три для y[2].

```
float y[4][3] = { { 1 }, { 2 }, { 3 }, { 4 } };
```

инициализирует первый столбец y (рассматриваемого как двумерный массив) и оставляет остальные элементы нулями.

8.6.2 Классовые объекты

Объект с закрытыми членами не может быть инициализован с помщью простого присваивания, как это описывалось выше; это же относится к объекту объединение. Если класс имеет конструктор, не получающий значений, то этот конструктор используется для объектов, которые явно не инициализированы.

Параметры для конструктора могут также быть представлены в виде заключенного в круглые скобки списка. Например:

```
struct complex
  {
    float re;
    float im;
    complex (float r,float i) { re=r; im=i; }
    complex (float r) { re=r; im=0; }
  };

complex zz (1,2.3);
complex* zp = new complex (1,2.3);
```

Инициализация может быть также выполнена с помощью явного присваивания; преобразования производятся. Например,

```
complex zz1 = complex (1,2.3);
complex zz2 = complex (123);
complex zz3 = 123;
complex zz4 = zz3;
```

Если конструктор ссылается на объект своего собственного класса, то он будет вызываться при инициализации объекта другим объектом

этого класса, но не при инициализации объекта конструктором.
Объект класса, имеющего конструкторы, может быть членом составного объекта только если он сам не имеет конструктора или если его конструкторы не имеют параметров. В последнем случае конструктор вызывается при создании составного объекта. Если член составного объекта является членом класса с деструкторами, то этот деструктор вызывается при уничтожении составного объекта.

8.6.3 Ссылки

Когда переменная описана как T&, что есть "ссылка на тип T", она может быть инициализирована или указателем на тип T, или объектом типа T. В последнем случае будет неявно применена операция взятия адреса &. Например:

```
int i;
int& r1 = i;
int& r2 = &i;
```

И r1 и r2 будут указывать на i.
Обработка инициализации ссылки очень сильно зависит от того, что ей присваивается. Как описывалось в #7.1, ссылка неявно переадресуется при ее использовании. Например

```
r1 = r2;
```

означает копирование целого, на которое указывает r2, в целое, на которое указывает r1.
Ссылка должна быть инициализована. Таким образом, ссылку можно считать именем объекта.
Чтобы получить указатель pp, обозначающий тот обект, что и ссылка rr, можно написать pp=&rr. Это будет проинтерпретировано как pp=&*rr.
Если инициализатор для ссылки на тип T не является адресным выражением, то будет создан и инициализован с помощью правил инициализации объект типа T. Тогда значением ссылки станет адрес объекта. Время жизни объекта, созданного таким способом, будет в той области видимости, в которой он создан. Например:

```
double& rr = 1;
```

допустимо, и rr будет указывать на объект типа double, в котором хранится значение 1.0.
Ссылки особенно полезны в качестве типов параметров.

8.6.4 Массивы символов

Последняя сокращенная запись позволяет инициализировать строкой массив данных типа char. В этом случае последовательные символы строки инициализируют члены массива. Например:

```
char msg[] = "Syntax error on line %d\n";
```

демонстрирует массив символов, члены которого инициализированы строкой.

8.7 Имена типов

Иногда (для неявного задания преобразования типов и в качестве параметра sizeof или new) нужно использовать имя типа данных. Это выполняется при помощи "имени типа" которое по сути является описанием для объекта этого типа, в котором опущено имя объекта.

имя_типа:
спецификатор_типа абстрактный_описатель

абстрактный_описатель :
пустой
* абстрактный_описатель
абстрактный_описатель (списоко_писателей_параметров)
абстрактный_описатель [константное_выражение opt]
(абстрактный_описатель)

Является возможным идентифицировать положение в абстрактном_описателе, где должен был бы появляться идентификатор в случае, если бы конструкция была описателем в описании. Тогда именованный тип является тем же, что и тип предполагаемого идентификатора. Например:

```
int
int *
int *[3]
int *()
int (*)()
```

именует, соответсвенно, типы "целое", "указатель на целое", "указатель на массив из трех целых", "функция, возвращающая указатель на функцию, возвращающую целое" и "указатель на целое".

Простое имя типа есть имя типа, состоящее из одного идентификатора или ключевого слова.

простое_имя_типа:
typedef-имя
char
short
int
long
unsigned
float
double

Они используются в альтернативном синтаксисе для преобразования типов. Например:

```
(double) a
```

может быть также записано как

```
double (a)
```

8.8 Определение типа typedef

Описания, содержащие спецификатор_описания typedef, определяют идентификаторы, которы позднее могут использоваться так, как если бы они были ключевыми словами типа, именующее основные или производные типы.

typedef-имя:
идентификатор

Внутри области видимости описания, содержащего typedef, каждый идентификатор, возникающий как часть какого-либо описателя, становится в этом месте синтаксически эквивалентным ключевому слову типа, которое именует тип, ассоциированный с идентификатором таким обрахом, как описывается в #8.4. Имя класса или перечисления также является typedef-именем. Например, после

```
typedef int MILES, *KLICKSP;
struct complex { double re, im; };
```

каждая из конструкций

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

является допустимым описанием; distance имеет тип int, metricp имеет тип "указатель на int".

typedef не вводит новых типов, но только синонимы для типов, которые могли бы быть определены другим путем. Так в приведенном выше примере distance рассматривается как имеющаяя в точности тот же тип, что и любой другой int объект.

Но описание класса вводит новый тип. Например:

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

описывает три переменных трех различных типов.

Описание вида

описание_имени:
агрег идентификатор ;
enum идентификатор ;

определяет то, что идентификатор является именем некоторого (возможно, еще не определенного) класса или перечисления. Такие описания позволяют описывать классы, ссылающихся друг на друга. Например:

```
class vector;
class matrix
  {
    ...
    friend matrix operator* (matrix&,vector&);
  };

class vector
  {
    ...
    friend matrix operator* (matrix&,vector&);
  };
```

8.9 Перегруженные имена функций

В тех случаях, когда для одного имени определено несколько (различных) описаний функций, это имя называется перегруженным. При использовании этого имени правильная функция выбирается с помощью сравнения типов фактических параметров с типами параметров в описаниях функций. К перегруженным именам неприменима операция получения адреса &.

Из обычных арифметических преобразований, определенных в #6.6, для вызова перегруженной функции выполняются только char->short->int, int->double, int->long и float->double. Для того, чтобы перегрузить имя функции не-члена описание overload должно предшествовать любому описанию функции; см. #8.2.

Например:

```
overload abs;
int abs (int);
double abs (double);
```

Когда вызывается перегруженное имя, по порядку производится сканирование списка функций для нахождения той, которая может быть вызвана. Например, abs(12) вызывает abs(int), а abs(12.0) будет вызывать abs(double). Если бы был зарезервирован порядок вызова, то оба обращения вызвали бы abs(double).

Если в случае вызова перегруженного имени с помощью

вышеуказанного метода не найдено ниодной функции, и если функция получает параметр типа класса, то констукторы классов параметров (в этом случае существует единственный набор преобразований, делающий вызов допустимым) применяются неявным образом. Например:

```
class X { ... X (int); };
class Y { ... Y (int); };
class Z { ... Z (char*); };

overload int f (X), f (Y);
overload int g (X), g (Y);

f (1);        /* неверно: неоднозначность f(X(1)) или f(Y(1)) */
g (1);        /* g(X(1))                                      */
g ("asdf");   /* g(Z("asdf"))                                 */
```

Все имена функций операций являются автомаатически перегруженными.

8.10 Описание перечисления

Перечисления являются int с именованными константами.

enum_спецификатор:
enum идентификатор opt { enum_список }

enum_список:
перечислитель
enum_список, перечислитель

перечислитель:
идентификатор
идентификатор = константное_выражение

Идентификаторы в enum-списке описаны как константы и могут появляться во всех местах, где требуются константы. Если не появляется ни одного перечислителя с =, то значения всех соответствующих констант начинаются с 0 и возрастают на 1 по мере чтения описания слева нарпаво. Перечислитель с = дает ассоциированному с ним идентификатору указанное значение; последующие идентификаторы продолжают прогрессию от присвоеннного зеначения.

Имена перечислителей должны быть отличными от имен обычных переменных. Значения перечислителей не обязательно должны быть различными.

Роль идентификатора в спецификаторе перечисления enum_спецификатор полностью аналогична роли имени класса; он именует определенный нутератор. Например:

```
enum color { chartreuse, burgundy, claret=20, winedark };
  ...
  color *cp, col;
  ...
  col = claret;
  cp = &col;
  ...
  if (*cp == burgundy) ...
```

делает color именем типа,описывающего различные цвета, и затем описывает cp как указатель на объект этого типа. Возможные значения извлекаются из множества { 0, 1, 20, 21 }.

8.11 Описание Asm

Описание Asm имеет вид

```
asm (строка);
```

Смысл описания asm неопределен. Обычно оно используется для передачи информации ассемблеру через компилятор.

9. ОПЕРАТОРЫ

Операторы выполняются последовательно во всех случаях кроме особо оговоренных.

9.1 Оператор выражение

Большинство операторов является операторами выражение, которые имеют вид

```
выражение ;
```

Обычно операторы выражение являются присваиваниями и вызовами функций.

9.2 Составной оператор, или блок

Составной оператор (называемый также "блок", что эквивалентно)

дает возможность использовать несколько операторов в том месте, где предполагается использование одного:

```
составной_оператор:
     { список_описаний opt список_операторов opt }
список_описаний:
     описание
     описание список_описаний
список_операторов:
     оператор
     оператор список_операторов
```

Если какой-либо из идентификаторов в списке_описаний был ранее описан, то внешенее описание выталкивается на время выполнения блока, и снова входит в силу по его окончании.

Каждая инициализиция auto или register переменных производится всякий раз при входе в голову блока. В блок делать передачу; в этом случае инициализации не выполняются. Инициализации переменных, имеющих класс памяти static (#4.2) осуществляются только один раз в начале выполнения программы.

9.3 Условный оператор

Есть два вида условных операторов

```
if ( выражение ) оператор
if ( выражение ) оператор else оператор
```

В обоих случаях вычисляется выражение, и если оно не ноль, то выполняется первый подоператор. Во втором случае второй подоператор выполняется, если выражение есть 0. Как обычно, неоднозначность "else" разрешается посредством того, что else связывается с последним встречнным if, не имеющим else.

9.4 Оператор while

Оператор while имеет вид

```
while ( выражение ) оператор
```

Выполнение подоператора повторяется, пока значение выражения остается ненулевым. Проверка выполняется передкаждым выполнением оператора.

9.5 Оператор do

Оператор do имеет вид

```
do оператор while (выражение);
```

Выполнение подоператора повторяется до тех пор, пока значение выражения не станет нулем. Проверка выполняется после каждого выполнения оператора.

9.6 Оператор for

Оператор for имеет вид

```
for ( выражение_1 opt ; выражение_2 opt ; выражение_3 opt )
      оператор
```

Этот оператор эквивалентен следующему:

```
выражение_1;
while                                                    (выражение_2)
                                                                  {
                                                            оператор

                                                          выражение_3;
  }
```

Первое выражение задает инициализацию цикла; второе выражение задает осуществляемую перед каждой итерацией проверку, по которой производится выход из цикла, если выражение становится нулем; третье выражение часто задает приращение, выполняемое после каждой итерации.

Каждое или все выражения могут быть опущены. Отсутствие выражения_2 делает подразумеваемое while-предложение эквивалентым while(1); остальные опущенные выражения просто пропускаются в описанном выше расширении.

9.7 Оператор switch

Оператор switch вызывает передачу управления на один из нескольких операторов в зависимости от значения выражения. Он имеет вид

```
switch ( выражение ) оператор
```

Выражение должно быть целого типа или типа указателя. Любой оператор внутри оператора может быть помечен одним или более префиксом case следующим образом:

```
case      константное_выражение :
```

где константное выражение должно иметь тот же тип что и выоажение-переключатель; производятся обычные арифметические преобразования. В одном операторе switch никакие две константы, помеченные case, не могут иметь одинаковое значение. Константные выражения точно определяются в #15.

Может также быть не более чем один префикс оператора вида

```
default :
```

Когда выполнен оператор switch, проведено вычисление его выражения и сравнение его с каждой case константой. Если одна из констант равна значению выражения, то управление передается на выражение, следующее за подошедшим префиксом case. Если никакая case константа не соответствует выражению, и есть префикс default, то управление передается на выражение, которому он предшествует. Если нет соответсвующих вариантов case и default отсутствует, то никакой из операторов в операторе switch не выполняется.

Префиксы case и default сами по себе не изменяют поток управления, который после задерки идет дальше, перескакивая через эти префиксы. Для выхода из switch см. break, #9.8.

Обычно зависящий от switch оператор является составным. В голове этого оператора могут стоять описания, но инициализации автоматических и регистровых переменных являются безрезультатными.

9.8 Оператор break

Оператор

```
break ;
```

прекращает выполнение ближайшего охватывающего while, do, for или switch оператора; управление передается на оператор, следующий за законченным.

9.9 Оператор continue

Оператор

```
continue ;
```

вызывает передачу управления на управляющую продолжением цикла часть наименьшего охватывающего оператора while, do или for; то есть на конец петли цикла. Точнее, в каждом из операторов

```
while (...)        do                 for (...)
  {                  {                  {
    ...                ...                ...
    contin:;           contin:;           contin:;
  }                  }                  }
                   while (...);
```

`continue` эквивалентно `goto contin`. (За contin: следует пустой оператор, #9.13.)

9.10 Оператор return

Возврат из функции в вызывающеую программу осуществляется с помощью оператора return, имеющего один из двух видов:

```
return ;
return выражение ;
```

Первый может использоваться только в функциях, не возвращающих значения, т.е. в функциях с типом возвращаемого значения void. Вторая форма может использоваться только в функциях, не возвращающих значение; вызывающей функцию программе возвращается значение выражения. Если необходимо, то выражение преобразуется, как это делается при присваивании, к типу функции, в которой оно возникло. Обход конца функции эквивалентен возврату return без возвращаемого значения.

9.11 Оператор goto

Можно осуществлять безусловную передачу упраления с помощью оператора

```
goto идентификатор ;
```

Идентификатор должен быть меткой (#9.12), расположенной в текущей функции.

9.12 Помеченные операторы

Перед любым оперотором может стоять префикс метка, имеющий вид

```
идентификатор :
```

который служит для описания идентификатора как метки. Метка используется только как объект для goto. Областью видимости метки является текущая функция, исключая любой подблок, в котором был переописан такой же идентияикатор. См. #4.1.

9.13 Пустой оператор

Пустой оператор имеет вид

```
;
```

Пустой оператор используется для помещения метки непосредственно перед } составного оператора или того, чтобы снабдить такие операторы, как while, пустым телом.

9.14 Оператор delete

Оператор delete имеет вид

```
delete выражение ;
```

Результатом выражения должен быть указатель. Объект, на который он указывает, уничтожается. Это значит, что после оператора уничтожения delete нельзя гарантировать, что объект имеет определенное значение; см. #17. Эффект от применеия delete к указателю, не полученному из операции new (#7.1), неопределен. Однако, уничтожение указателя с нулевым значением безопасно.

9.15 Оператор asm

Оператор asm имеет вид

```
asm ( строка) ;
```

Смысл оператора asm неопределен. Обычно он используется для передачи информации черз компилятор ассемблеру.

10. ВНЕШНИЕ ОПРЕДЕЛЕНИЯ

Программа на C++ состоит из последовательности внешних определений. Внешенее определение описывает идентификатор как имеющий класс памяти static и определяет его тип. Спецификатор типа (#8.2) может также быть пустым, и в этом случае принимается тип int. Область видимости внешних определений простирается до конца файла, в котором они описаны, так же, как действие описаний сохраняется до конца блока. Синтаксис внешних определений тот же, что и у описаний, за исключением того, что только на этом уровне и внутри описаний классов может быть задан код (текст программы) функции.

10.1 Определения функций

Определения функций имеют вид

```
определение_функции:
     спецификаторы_описания описатель_функции  opt инициализа-
тор_базового_класса                                          opt
     тело_функции
```

Единственными спецификаторами класса памяти (sc-спецификаторами), допустимыми среди спецификаторов описания, являются extern, static, overload, inline и virtual. Описатель функции похож на описатель "функции, возвращающей ...", за исключением того, что он включает в себя имена формальных параметров определяемой функции. Описатель функции имеет вид

```
описатель_функции:
     описатель ( список_описаний_параметров )
```

Форма списка описаний параметров определена в #8.4. Единственный класс памяти, который может быть задан, это тот, при котором соответствующий фактический параметр будет скопирован, если это возможно, в регистр при входе в функцию. Если в качестве инициализатора для параметра задано константное выражение, то это значение используется как значение параметра по умолчанию.

Тело функции имеет вид

```
тело_функции:
     составной_оператор
```

Вот простой пример полного определения функции:

```
int max (int a,int b,int c)
  {
```

```
    int m = (a > b) ? a : b;
    return  (m > c) ? m : c;
  }
```

Здесь int является спецификатором типа ; max (int a, int b, int c) является описателем функции ; { ... } - блок, задающий текст программы (код) оператора.

Поскольку в контексте выражения имя (точнее, имя как формальный параметр) считается означающим указатель на первый элемент массива, то описания формальных параметров, описанных как "массив из ...", корректируются так, чтобы читалось "указатель на ...".

Инициализатор базового класса имеет вид

инициализатор_базового_класса:
: (список_параметров opt)

Он используется для задания параметров конструктора базового класса в конструкторе производного класса. Например:

```
struct base { base (int); ... };
struct derived : base { derived (int); ... };

derived.derived (int a) : (a+1) { ... }

derived d (10);
```

Конструктор базового класса вызывается для объекта d с параметром 11.

10.2 Определения внешних данных

Определения внешних данных имеют вид

определение_данных:
описание

Класс памяти таких данных статический.

Если есть более одного определения внешних данных одного именеи, то определения должны точно согласовываться по типу и классу памяти, и инициализаторы (если они есть), должны иметь одинаковое значение.

11. ПРАВИЛА ОБЛАСТИ ВИДИМОСТИ

См. #4.1.

12. КОМАНДНЫЕ СТРОКИ КОМПИЛЯТОРА

Компилятор языка C++ содержит препроцессор, способный выполнять макроподстановки, условную компиляцию и включение именованных файлов. Строки, начинающиеся с #, относятся к препроцессору. Эти строки имеют независимый от остального языка синтаксис; они могут появляться в любом месте оказывать влияние, которое распространяется (независимо от области видимости) до конца исходного файла программы.

Заметьте, что определения const и inline дают альтернативы для большинства использований #define.

12.1 Замена идентификаторов

Командная строка компилятора имеет вид

#define идент строка_символов

вызывает замену препроцессором последующих вхождений идентификатора, заданного строкой символов. Точка с запятой внутри (или в конце) строки символов является частью этой строки.

Строка вида

#define идент(идент , ..., идент) строка_символов

где отсутсвует пробел между первым идентификатором и (, является макроопределением с параметрами. Последующие вхождения первого идентификатора с идущими за ним (, последовательностью символов, разграниченной запятыми, и), заменяются строкой символов, заданной в определении. Каждое местоположение идентификатора, замеченного в списке параметров определения, заменяется соответствующей строкой из вызова. Фактическими параметрами вызова являются строки символов, разделенные запятыми; однако запятые в строке, заключенной в кавычки, или в круглых скобках не являются разделителями параметров. Число формальных и фактических параметров должно совпадать. Строки и символьные константы в символьной строке сканируются в поисках формальных параметров, но строки и символьные константы в остальной программе не сканируются в поисках определенных (с помощью define) идентификаторов.

В обоих случаях строка замещения еще раз сканируется в поисках других определнных идентификаторов. В обоих случаях длинное определение может быть продолжено на другой строке с помощью записи \ в конце продолжаемой строки.

Командная строка вида

```
#undef идент
```

влечет отмену препроцессорного определения идентификатора.

12.2 Включение файлов

Командная строка компилятора вида

```
#include "имя_файла"
```

вызывает замену этой строки полным содержимым файла имя_файла. Сначала именованный файл ищется в директории первоначального исходного файла, а затем в стандартных или заданных местах. Альтернативный вариант, командная строка вида

```
#include <имя_файла>
```

производит поиск только в стандартном или заданном месте, и не ищет в директории первоначального исходного файла. (То, как эти места задаются, не является частью языка.)

Включения с помощью #include могут быть вложенными.

12.3 Условная компиляция

Командная строка компилятора вида

```
#if выражение
```

проверяет, является ли результатом вычисления выражения не-ноль. Выражение должно быть константным выражением, которые обсуждаются в #15; применительно к использованию данной директивы есть дополнительные ограничения: константное выражение не может содержать sizeof или перечислимые константы. Кроме обычных операций C может использоваться унарная операция defined. В случае применения к идентификатору она дает значение не-ноль, если этот идентификатор был ранее определен с помощью #define и после этого не было отмены определения с помощью #undef; иначе ее значение 0.

Командная строка вида

```
#ifdef идент
```

проверяет, определен ли идентификатор в препроцессоре в данный момент; то есть, был ли он объектом командной строки #define.

Командная строка вида

```
#ifndef идент
```

проверяет, является ли идентификатор неопределенным в препроцессоре в данный момент.

После строки каждого из трех видов может стоять произвольное количество строк, возможно, содержащих командную строку

```
#else
```

и далее до командной строки

```
#endif
```

Если проверенное условие истинно, то все строки между #else и

#endif игнорируются. Если проверенное условие ложно, то все строки между проверкой и #else или, в случае отсутствия #else, #endif, игнорируются.

Эти конструкции могут быть вложенными.

12.4 Управление строкой

Для помощи другим препроцессорам, генерирующим программы на C, строка вида

```
#line константа "имя_файла"
```

заставляет компилятор считать, например, в целях диагностики ошибок, что константа задает номер следущей строки исходного файла,

и текущий входной файл именуется идентификатором. Есло идентификатор отсутствует, то запомненное имя файла не изменяется.

13. НЕЯВНЫЕ ОПИСАНИЯ

См. #8.1.

14. ОБЗОР ТИПОВ

В этом разделе кратко собрано описание действий, которые могут совершаться над объектами различных типов.

14.1 Классы

Классовые объекты могут присваиваться, передаваться функциям как параметры и возвращаться функциями. Другие возможные операции, как, например, проверка равенства, могут быть определены пользователем; см. #8.5.10.

14.2 Функции

Есть только две вещи, которые можно проделывать с функцией: вызывать ее и брать ее адрес. Если в выражении имя функции возникает не в положении имени функции в вызове, то генерируется указатель на функцию. Так, для передачи одной функции другой можно написать

```
typedef int (*PF) ();
extern g (PF);
extern f ();
...
g (f);
```

Тогда определение g может иметь следующий вид:

```
g (PF funcp)
  {
    ...
    (*funcp) ();
    ...
  }
```

Заметьте, что f должна быть описана явно в вызывающей программе, поскольку ее появление в g(f) не сопровождалось (.

14.3 Массивы, указатели и индексирование

Всякий раз, когда в выражении появляется идентификатор типа массива, он преобразуется в указатель на первый член массива. Из-за преобразований массивы не являются адресами. По определению операция индексирования [] интерпретируется таким образом, что

E1[E2] идентично *((E1)+(E2)). В силу правил преобразования, применяемых к +, если E1 массив и E2 целое, то E1[E2] относится к E2-ому члену E1. Поэтому, несмотря на такое проявление асимметрии, индексирование является коммутативной операцией.

Это правило сообразным образом применяется в случае многомерного массива. Если E является n-мерным массивом ранга i*j*...*k, то возникающее в выражении E преобразуется в указатель на (n-1)-мерный массив ранга j*...*k. Если к этому указателю, явно или неявно, как результат индексирования, применяется операция *, ее результатом является (n-1)-мерный массив, на который указывалось, который сам тут же преобразуется в указатель.

Рассмотрим, например,

```
int x[3][5];
```

Здесь x - массив целых размером 3*5. Когда x возникает в выражении, он преобразуется в указатель на (первый из трех) массив из 5 целых. В выражении x[i], которое эквивалентно *(x+1), x сначала преобразуется, как описано, в указатель, затем 1 преобразуется к типу x, что включает в себя умножение 1 на длину объекта, на который указывает указатель, а именно объект из 5 целых. Результаты складываются, и используется косвенная адресация для получения массива (из 5 целых), который в свою очередь преобразуется в указатель на первое из целых. Если есть еще один индекс, снова используется тот же параметр; на этот раз результат является целым.

Именно из всего этого проистекает то, что массивы в C хранятся по строкам (быстрее всего изменяется последний индекс), и что в описании первый индекс помогает определить объем памяти, поглощаемый массивом, но не играет никакой другой роли в вычислениях индекса.

14.4 Явные преобразования указателей

Определенные преобразования, включающие массивы, выполняются, но имеют зависящие от реализации аспекты. Все они задаются с помощью явной операции преобразования типов, см. ##7.2 и 8.7.

Указатель иожет быть преобразован к любому из целых типов, достаточно больших для его хранения. То, какой из int и long требуется, является машинно зависимым. Преобразующая функция также является машинно зависимой, но предполагается, что она не содержит сюрпризов для того, кто знает структуру адресации в машине. Подробности для некоторых конкретных машин были даны в #2.6.

Объект целого типа может быть явно преобразован в указатель. Преобразующая функция всегда превращает целое, полученное из указателя, обратно в тот же указатель, но в остальных случаях является машинно зависимой.

Указатель на один тип может быть преобразован в указатель на другой тип. Использование результирующего указателя может вызывать особые ситуации, если исходный указатель не указывает на объект, соответствующим образом выравненный в памяти. Гарантируется, что указатель на объект данного размера может быть преобразован в указатель на объект меньшего размера и обратно без изменений.

Например, программа, выделяющая память, может получать размер (в байтах) размещаемого объекта и возвращать указатель на char; это можно использовать следующим образом.

```
extern void* alloc ();
double* dp;
```

```
dp = (double*) alloc (sizeof (double));
*dp= 22.0 / 7.0;
```

alloc должна обеспечивать (машинно зависимым образом) то, что возвращаемое ею значение подходит для преобразования в указатель на double; в этом случае использование функции мобильно. Различные машины различаются по числу бит в указателях и требованиям к выравниванию объектов. Составные объекты выравниваются по самой строгой границе, требуемой каким-либо из его составляющих.

15. КОНСТАНТНЫЕ ВЫРАЖЕНИЯ

В нескольких местах C++ требует выражения, вычисление которых дает константу: в качестве границы массива (#8.3), в case выражениях (#9.7), в качестве значений параметров функции, присваиваемых по умолчанию, (#8.3), и в инициализаторах (#8.6). В первом случае выражение может включать только целые константы, символьные константы, константы, описанные как имена, и sizeof выражения, возможно, связанные бинарными операциями

```
+ - * / % & | ^ << >> == != < > <= >= && ||
```

или унарными операциями

```
- ~ !
```

или тернарными операциями

```
? :
```

Скобки могут использоваться для группирования, но не для вызова функций.

Большая широта допустима для остальных трех случаев использования; помимо константных выражений, обсуждавшихся выше, допускаются константы с плавающей точкой, и можно также применять унарную операцию & к внешним или статическим объектам, или к внешним или статическим массивам, индексированным константным выражением. Унарная операция & может также быть применена неявно с помощью употребления неиндексированных массивов и функций. Основное правило состоит в том, что инициализаторы должны при вычислении давать константу или адрес ранее описанного внешнего или статического обйекта плюс или минус константа.

Меньшая широта допустима для константных выражений после #if: константы, описанные как имена, sizeof выражения и перечислимые константы недопустимы.

16. СООБРАЖЕНИЯ МОБИЛЬНОСТИ

Определенные части C++ являются машинно зависимыми по своей сути. Следующий ниже список мест возможных затруднений не претендует на полноту, но может указать на основные из них.

Как показала практика, характеристики аппаратуры в чистом виде, такие, как размер слова, свойства плавающей арифметики и целого деления, не создают особых проблем. Другие аппаратные аспекты отражаются на различных программных разработках. Некоторые из них, особенно знаковое расширение (преобразование отрицательного символа в отрицательное целое) и порядок расположения байтов в слове, являются досадными помехами, за которыми надо тщательно следить. Большинство других являются всего лишь мелкими сложностями.

Число регистровых переменных, которые фактически могут быть помещены в регистры, различается от машины к машине, как и множество фактических типов. Тем не менее, все компиляторы на "своей" машине все делают правильно; избыточные или недействующие описания register игнорируются.

Некоторые сложности возникают при использовании двусмысленной манеры программирования. Писать программы, зависящие от какой-либо из этих особенностей, райне неблагоразумно.

В языке неопределен порядок вычисления параметров функции. На некоторых машинах он слева направо, а на некоторых справо налево. Порядок появления некотрых побочных эффектов также недетерминирован.

Поскольку символьные константы в действительности являются объектами типа int, то могут быть допустимы многосимвольные константы. Однако конкретная реализация очень сильно зависит от машины, поскольку порядок, в котором символы присваиваются слову, различается от машины к машине. На некоторых машинах поля в слове присваиваются слева направо, на других справо налево.

Эти различия невидны для отдельных программ, не позволяющих себе каламбуров с типами (например, преобразования int указателя в char указатель и просмотр памяти, на которую указывает указатель), но должны приниматься во внимание при согласовании внешне предписанных форматов памяти.

17. СВОБОДНАЯ ПАМЯТЬ

Операция new (#7.2) вызывает функцию

```
extern void* _new (long);
```

для получения памяти. Параметр задает число требуемых байтов. Память будет инициализирована. Если _new не может найти требуемое количество памяти, то она возвращает ноль.

Операция delete вызывает функцию

```
extern void _delete (void*);
```

чтобы освободить память, указанную указателем, для повторного использования. Результат вызова _delete() для указателя, который не был получен из _new(), неопределен, это же относится и к повторному вызову _delete() для одного и того же указателя. Однако уничтожение с помощью delete указателя со значением ноль безвредно.

Предоставляются стандартные версии _new() и _delete(), но пользователь может применять другие, более подходящие для конкретных приложений.

Когда с помощью операции new создается классовый объект, то для получения необходимой памяти конструктор будет (неявно) использовать new. Конструктор может осуществить свое собственное резервирование памяти посредством присваивания указателю this до каких-либо использований. С помощью присваивания this значения ноль деструктор может избежать стандартной операции дерезервирования памяти для объекта его класса. Например:

```
class cl
  {
    int v[10];
    cl () { this = my_own_allocator (sizeof (cl)); }
    ~cl () { my_own_deallocator (this); this = 0; }
  }
```

На входе в конструктор this являеется не-нулем, если резервирование памяти уже имело место (как это имеет место для автоматических объектов), и нулем в остальных случаях.

Если производный класс осуществляет присваивание this, то вызов конструктора (если он есть) базового класса будет иметь место после присваивания, так что конструктор базового класса ссылаться на объект посредством конструктора производного класса. Если конструктор базового класса осуществляет присваивание this, то

значение также будет использоваться конструктором (если таковой есть) производного класса.

18. КРАТКОЕ ИЗЛОЖЕНИЕ СИНТАКСИСА

Мы надеемся, что эта краткая сводка синтаксиса C++ поможет пониманию. Она не является точным изложением языка.

18.1 Выражения

```
выражение:
        терм
        выражение            бинарная_операция            выражение
        выражение       ?        выражение       :       выражение
        список_выражений
терм:
        первичный
        *                                                       терм
        &                                                       терм
        -                                                       терм
        !                                                       терм
        ~                                                       терм
        ++терм
        --терм
        терм++
        терм--
        (                    имя_типа)                      выражение
        имя_простого_типа            (            список_выражений)



        sizeof                                              выражение
        sizeof              (              имя_типа              )
        new                                                  имя_типа
        new ( имя_типа )
первичный:
        id
        ::                                              идентификатор
        константа
        строка
        this
        (                         выражение                         )
        первичный[                    выражение                    ]
        первичный       (       список_выражений       opt       )
        первичный.id
        первичный->id
id:
        идентификатор
        typedef-имя :: идентификатор
список_выражений:
        выражение
        список_выражений, выражение
операция:
        унарная_операция
        бинарная_операция
        специальная_операция
```

Бинарные операции имеют приоритет, убывающий в указанном порядке:

```
бинарная_операция:
        *                           /                           %
        +                                                       -
        <<                                                     >>
        <                                                       >
        ==                                                     !=
        &
        ^
        |
```

```
     &&
     ||
     =   +=   -=   *=   /=   %=   ^=   &=   |=   >>=   <<=
                                                  унарная_операция:
      *   &   -   ~   !   ++   --
специальная_операция:
     ()       []
имя_типа:
     спецификаторы_описания абстрактный_описатель
абстрактный_описатель:
     пустой
     *                                    абстрактный_описатель
     абстрактный_описатель  (   список_описаний_параметров   )
     абстрактный_описатель  [   константное_выражение  opt   ]
     ( абстрактный_описатель )
простое_имя_типа:
     typedef-имя
     char
     short
     int
     long



     unsigned
     float
     double
typedef-имя:
     идентификатор
```

18.2 Описания

```
описание:
     спецификаторы_описания  opt   список_описателей   opt   ;
     описание_имени
     asm-описание
описание_имени:
     агрег                  идентификатор                   ;
     enum  идентификатор ;
агрег:
     class
     struct
     union
asm-описание:
     asm ( строка );
спецификаторы_описания:
     спецификатор_описания спецификатор_описания opt
спецификатор_описания:
     имя_простого_типа
     спецификатор_класса
     enum_спецификатор
     sc_спецификатор
     фнк_спецификатор
     typedef
     friend
     const
     void
sc_спецификатор:
     auto
     extern
     register
     static
фнк-спецификатор:
     inline
     overload
     virtual
```

```
список_описателей:
     иниц-описатель
     иниц-описатель , список_описателей
иниц-описатель:
     описатель инициализатор opt
описатель:
     оп_имя
     (                        описатель                        )
     *             const             opt             описатель
     &             const             opt             описатель
     описатель      (       список_описаний_параметров       )
     описатель [ константное_выражение opt ]



оп_имя:
     простое_оп_имя
     typedef-имя . простое_оп_имя
простое_оп_имя:
     идентификатор
     typedef-имя
     -                                                typedef-имя
     имя_функции_операции
имя_функции_операции:
     операция операция



список_описаний_параметров:
     список_описаний_прм opt ... opt
список_описаний_прм                                          :
     список_описаний_прм         ,          описание_параметра
     описание_параметра
описание_параметра:
     спецификаторы_описания                              описатель
     спецификаторы_описания описатель = константное_выражение
спецификатор_класса:
     заголовок_класса       {список_членов        opt        }
     заголовок_класса    {список_членов     opt    public    :
список_членов opt }
заголовок_класса                                             :
     агрег                  идентификатор                  opt
     агрег идентификатор opt : public opt typedef-имя
список_членов                                                :
     описание_члена список_членов opt
описание_члена:
     спецификаторы_описания opt описатель_члена ;
описатель_члена:
     описатель
     идентификатор opt : константное_выражение
инициализатор:
     =                                                  выражение
     =                {                 список_инициализаторов}
     =       {        список_инициализаторов,                 }

     (список_выражений )
список_инициализаторов                                       :
     выражение
     список_инициализаторов        ,    список_инициализаторов
     { список_инициализаторов }
enum-спецификатор:
     enum идентификатор opt { enum-список }
enum-список:
     перечислитель
     enum-список , перечислитель
перечислитель:
```

```
     идентификатор
     идентификатор = константное_выражение

18.3 Операторы

составной_оператор:
     { список_описаний opt список_операторов opt }
список_описаний:
     описание
     описание список_описаний
список_операторов:
     оператор
     оператор список_операторов
оператор:
     выражение                                                    ;
     if         (         выражение         )         оператор
     if    (    выражение    )    оператор    else    оператор
     while        (         выражение        )        оператор



     do     оператор     while     (     выражение     )     ;
     for (  выражение opt  ; выражение  opt ;  выражение opt )
          оператор
     switch        (        выражение        )        оператор
     case      константное      выражение      :      оператор
     default                    :                    оператор
     break;
     continue;
     return            выражение             opt             ;
     goto                 идентификатор                      ;
     идентификатор              :                    оператор
     delete                   выражение                      ;
     asm          (            строка           )           ;
     ;

18.4 Внешние определения

программа:
     внешнее_определение
     внешнее_определение программа
внешнее_определение:
     определение_функции
     описание
определение_функции:
     спецификаторы_описания       opt        описатель_функции
инициализатор_базового_класса opt тело_функции
описатель_функции:
     описатель ( список_описаний_параметров)
тело_функции:
     составной_оператор
инициализатор_базового_класса:
     : ( список_параметров opt )

18.5 Препроцессор

#define идент строка_символов
#define идент( идент,...,идент ) строка символов
#else
#endif
#if выражение
#ifdef идент
#ifndef идент
#include "имя_файла"
#include <имя_файла>
#line константа "имя_файла"
```

```
#undef идент
```

19. ОТЛИЧИЯ ОТ "СТАРОГО С"

19.1 Расширения

Типы параметров функции могут быть заданы (#8.4) и будут проверяться (#7.1). Могут выполняться преобразования типов.

Для выражений с числами с плавающей точкой может использоваться плавающая арифметика одинарной точности; #6.2.

Имена функций могут быть перегружены; #8.6

Операции могут быть перегружены; 7.16, #8.5.10.

Может осуществляться inline-подстановка функций; #8.1.

Объекты данных могут быть константными (const); #8.3.

Могут быть описаны объекты ссылочного типа; #8.3, #8.6.3

Операции new и delete обеспечивают свободное хранение в памяти; #17.

Класс может обеспечивать скрытые данные (#8.5.8), гарантированную инициализацию (#8.6.2), оперделяемые пользователем преобразвания (#8.5.6), и динамическое задание типов через использование виртуальных функций (#8.5.4).

Имя класса является именем типа; #8.5.

Любой указатель может присваиваться [указателю] void* без приведения типов; #7.14.

Сергей Деревяго.
C++ 3rd: комментарии

Origin: http://cpp3.virtualave.net/cpp3comm/toc

toc
Сергей Деревяго
C++ 3rd: комментарии

intro

Введение

Вашему вниманию предлагается "еще одна" книга по C++. Что в ней есть? В ней есть все, что нужно для глубокого понимания C++. Дело в том, что практически весь материал основан на блестящей книге Б.Страуструпа "Язык программирования C++", 3е издание. Я абсолютно уверен, что интересующийся C++ программист обязан прочитать "Язык программирования C++", а после прочтения он вряд ли захочет перечитывать описание C++ у других авторов -- маловероятно, что кто-то напишет собственно о C++ лучше д-ра Страуструпа. Моя книга содержит исправления, комментарии и дополнения, но нигде нет повторения уже изложенного материала.

В процессе чтения (и многократного) перечитывания C++ 3rd у меня возникало множество вопросов, большая часть которых отпадала после изучения собственно стандарта и продолжительных раздумий, а за некоторыми приходилось обращаться непосредственно к автору. Хочется выразить безусловную благодарность д-ру Страуструпу за его ответы на все мои, заслуживающие внимания, вопросы и разрешение привести данные ответы здесь.

Как читать эту книгу. Прежде всего, нужно прочитать "Язык программирования C++" и только на этапе второго или третьего перечитывания обращаться к моему материалу, т.к. здесь кроме исправления ошибок русского перевода излагаются и весьма нетривиальные вещи, которые вряд ли будут интересны среднему программисту на C++. Моей целью было улучшить перевод C++ 3rd, насколько это возможно и пролить свет на множество интересных особенностей C++. Кроме того, оригинальное (английское) издание пережило довольно много тиражей, и каждый тираж содержал некоторые исправления, я постарался привести все существенные исправления здесь.

В процессе чтения было бы неплохо иметь под рукой стандарт C++ (ISO/IEC 14882 Programming languages -- C++, First edition, 1998-09-01), или его Final Draft. Также не помешает ознакомиться с классической STL, ведущей начало непосредственно от Алекса Степанова. И самое главное -- не забудьте заглянуть к самому Бьерну Страуструпу.

С уважением, Сергей Деревяго.

За исключением операторов new, delete, type_id, dynamic_cast, throw и блока try, отдельные выражения и инструкции C++ не требуют поддержки во время выполнения. Хотелось бы отметить, что есть еще несколько очень важных мест, где мы имеем неожиданную и порой весьма существенную поддержку времени выполнения. Это конструкторы/деструкторы (особенно "сложных" объектов), пролог/эпилог создающих объекты функций, отчасти, вызовы виртуальных функций.
Для демонстрации данной печальной особенности рассмотрим следующую программу:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

long Var, Count;

struct A {
       A();  // не inline
       ~A();
};

void ACon();
void ADes();

void f1()
{
 A a;
}

void f2()
{
 ACon();
 ADes();
}

int main(int argc,char** argv)
{
 if (argc>1) Count=atol(argv[1]);

 clock_t c1,c2;
 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000000; j++)
          f1();

  c2=clock();
  printf("f1(): %ld mlns calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000000; j++)
          f2();

  c2=clock();
  printf("f2(): %ld mlns calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
}

A::A()  { Var++; }
A::~A() { Var++; }

void ACon() { Var++; }
```

```
void ADes() { Var++; }
```

В ней функции f1() и f2() делают одно и то же, только первая неявно, с помощью конструктора и деструктора класса A, а вторая с помощью явного вызова ACon() и ADes().
Для работы программа требует одного параметра -- сколько миллионов раз вызывать тестовые функции. Выберите значение, позволяющее f1() работать несколько секунд и посмотрите на результат для f2(). В зависимости от компилятора и платформы разница может достигать 10 раз!
А теперь немного о структуре данного примера. Во-первых, определения вызываемых A-функций я поместил в конце файла, т.к. некоторые в меру интеллектуальные оптимизаторы встраивают тело функции непосредственно в место ее вызова (даже без явной спецификации inline), если определение функции было доступно в точке вызова и тело функции было тривиальным. Даже более того: после работы качественного оптимизатора приведенный выше цикл

```
for (long i=0; i<Count; i++)
    for (long j=0; j<1000000; j++)
        f2();
```

может превратиться в

```
Var+=Count*2*1000000;
```

И это просто замечательно, но, к сожалению, не в нашем конкретном случае.
Кстати, проверьте качество вашего оптимизатора, перенеся определения A-функций до их вызова в f1() и f2().
В принципе, встраивание тела функции можно осуществить и в случае присутствия определения функции после ее вызова, но, как правило, этого не происходит.
Итак, накладные расходы присутствуют. А что же inline? Давайте внесем очевидные изменения:

```
struct A {
       A() { Var++; }
       ~A() { Var++; }
};

void f1()
{
 A a;
}

void f2()
{
 Var++;
 Var++;
}
```

После данных изменений разницы во времени работы f1() и f2() не должно быть. Но, к несчастью, она будет присутствовать для большинства компиляторов.
Что же происходит? Наблюдаемый нами эффект называется abstraction penalty, т.е. обратная сторона абстракции или налагаемое на нас некачественными компиляторами наказание за использование объектно-ориентированных абстракций.
Посмотрим, как abstraction penalty проявляется в нашем случае. Что же из себя представляет

```
void f1()
{
 A a;
}
```

эквивалентное

```
void f1()  // псевдокод
{
 A::A();
 A::~A();
}
```

И чем оно отличается от простого вызова двух функций:

```
void f2()
{
 ACon();
 ADes();
}
```

В данном случае -- ничем! Но, давайте рассмотрим похожий пример:

```
void f1()
{
 A a;
 fun();
}

void f2()
{
 ACon();
 fun();
 ADes();
}
```

Как вы думаете, эквивалентны ли данные функции? Правильный ответ -- нет, т.к. f1() представляет собой

```
void f1()  // псевдокод
{
 A::A();

 try {
     fun();
 }
 catch (...) {
       A::~A();
       throw;
 }

 A::~A();
}
```

Т.е. если конструктор успешно завершил свою работу, то языком гарантируется, что обязательно будет вызван деструктор. Т.е. там, где создаются некоторые объекты, компилятор специально вставляет блоки обработки исключений для гарантии вызова соответствующих деструкторов. А накладные расходы в оригинальной f1() чаще всего будут вызваны присутствием ненужных в данном случае блоков обработки исключений:

```
void f1()  // псевдокод
{
 A::A();

 try {
     // пусто
 }
 catch (...) {
       A::~A();
       throw;
 }

 A::~A();
}
```

Дело в том, что компилятор обязан корректно обрабатывать все возможные случаи, поэтому для упрощения компилятора его разработчики часто не принимают во внимание "частные случаи", в которых можно не генерировать ненужный код. Увы, подобного рода упрощения компилятора очень плохо сказываются на производительности интенсивно использующего средства абстракции и inline функции кода. Хорошим примером подобного рода кода является STL, чье использование, при наличие плохого оптимизатора, вызывает чрезмерные накладные расходы.
Итак, занимаясь оптимизацией "узких мест" обратите внимание на abstraction penalty.

2.5.5. Виртуальные функции, стр. 73
Требования по памяти составляют один указатель на каждый объект класса с виртуальными функциями, плюс одна vtbl для каждого такого класса.
На самом деле первое утверждение неверно, т.е. объект полученный в результате множественного наследования от полиморфных классов будет содержать несколько "унаследованных" указателей на vtbl.
Рассмотрим следующий пример. Пусть у нас есть полиморфный (т.е. содержащий виртуальные функции) класс B1:

```
struct B1 {  // я написал struct чтобы не возиться с правами доступа
       int a1;
       int b1;

       virtual ~B1() { }
};
```

Пусть имеющаяся у нас реализация размещает vptr (указатель на таблицу виртуальных функций класса) перед объявленными нами членами, тогда данные объекта класса B1 будут расположены в памяти следующим образом:

```
vptr_1  // указатель на vtbl класса B1
a1      // объявленные нами члены
b1
```

Если теперь объявить аналогичный класс B2 и производный класс D

```
struct D: B1, B2 {
       virtual ~D() { }
};
```

то его данные будут расположены следующим образом:

```
vptr_d1  // указатель на vtbl класса D, для B1 здесь был vptr_1
a1       // унаследованные от B1 члены
b1
vptr_d2  // указатель на vtbl класса D, для B2 здесь был vptr_2
a2       // унаследованные от B2 члены
b2
```

Почему здесь два vptr? Потому, что была проведена оптимизация, иначе их было бы три. Я, конечно, понял, что вы имели ввиду: "Почему не один"? Не один, потому что мы имеем возможность преобразовывать указатель на производный класс в указатель на любой из базовых. И полученный указатель должен указывать на корректный объект базового класса. Т.е. если я напишу:

```
D d;
B2* ptr=&d;
```

то в нашем примере ptr укажет в точности на vptr_d2. А собственным vptr класса D будет являться vptr_d1. Значения этих указателей, вообще говоря, различны. Почему? Потому что у B1 и B2 в vtbl по одному и тому же индексу могут быть расположены разные функции, а D должен иметь возможность их правильно заместить. Т.о. vtbl класса D состоит из нескольких частей: часть для B1, часть для B2 часть для собственных нужд.
Подводя итог, можно сказать, что если мы используем множественное наследование от большого числа полиморфных классов, то накладные расходы по памяти могут быть достаточно существенными. Конечно, от этих расходов можно отказаться, реализовав вызов виртуальной функции специальным образом, а именно: каждый раз вычисляя положение vptr относительно this. Однако это спровоцирует существенные относительные расходы времени выполнения, компрометируя тем самым вызов виртуальных функций.
И раз уж так много слов было сказано про эффективность, давайте реально измерим относительную стоимость вызова виртуальной функции.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

long Var, Count;
```

```
struct B {
       void f();
       virtual void vf();
};

struct D : B {
       void vf();  // замещаем B::vf
};

void f1(B* ptr)
{
 ptr->f();
}

void f2(B* ptr)
{
 ptr->vf();
}

int main(int argc,char** argv)
{
 if (argc>1) Count=atol(argv[1]);

 clock_t c1,c2;

 D d;
 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000000; j++)
          f1(&d);

  c2=clock();
  printf("f1(): %ld mlns calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000000; j++)
          f2(&d);

  c2=clock();
  printf("f2(): %ld mlns calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
}

void B::f()  { Var++; }
void B::vf() { }

void D::vf() { Var++; }
```

В зависимости от компилятора, накладные расходы на вызов виртуальной функции составили от 5 до 20 процентов. Т.о. вызов виртуальной функции не несет существенных накладных расходов -- очень хорошая новость.

2.7.2. Обобщенные алгоритмы, стр. 79

Встроенные в C++ типы низкого уровня, такие как указатели и массивы, имеют соответствующие операции, поэтому мы можем записать:

```
char vc1[200];
char vc2[500];

void f()
{
```

```
 copy(&vc1[0],&vc1[200],&vc2[0]);
}
```

На самом деле, если подойти к делу формально, то писать так мы не должны, хотя практически всегда ничего неожиданного не произойдет. Вот что говорит об этом д-р Страуструп:

The issue is whether taking the address of one-past-the-last element of an array is conforming C and C++. I could make the example clearly conforming by a simple rewrite:

```
 copy(vc1,vc1+200,vc2);
```

However, I don't want to introduce addition to pointers at this point of the book. It is a surprise to most experienced C and C++ programmers that &vc1[200] isn't completely equivalent to vc1+200. In fact, it was a surprise to the C committee also and I expect it to be fixed in the upcoming revision of the standard. (also resolved for C9x - bs 10/13/98).

Суть вопроса в том, означает ли взятие адреса элемента, следующего за последним одни и те же действия в C и C++. Я мог сделать пример более корректным простой заменой

```
 copy(vc1,vc1+200,vc2);
```

Однако, я не хотел вводить сложение с указателем в этой части книги. Даже для самых опытных программистов на C и C++ большим сюрпризом является тот факт, что &vc1[200] не полностью эквивалентно vc1+200. Фактически, это оказалось неожиданностью и для C комитета, и я надеюсь, что это недоразумение будет устранено в следующих редакциях стандарта.

Так в чем же нарушается эквивалентность? По стандарту C++ мы имеем следующие эквивалентные преобразования:

```
&vc1[200] == &(*((vc1)+(200))) == &*(vc1+200)
```

Действительно ли равенство &*(vc1+200) == vc1+200 неверно?

It is false in C89 and C++, but not in K&R C or C9x. The C89 standard simply said that &*(vc1+200) means dereference vc1+200 (which is an error) and then take the address of the result, and the C++ standard copiled the C89 wording. K&R C and C9x say that &* cancels out so that &*(vc1+200) == vc2+200.

Это неверно в C89 и C++, но не в K&R C или C9x. Стандарт C89 говорит, что &*(vc1+200) означает разыменование vc1+200 (что является ошибкой) и затем взятие адреса результата. И стандарт C++ просто взял эту формулировку из C. Однако K&R C и C9x устанавливают, что &* взаимно уничтожаются, т.е. &*(vc1+200) == vc1+200. Спешу вас успокоить, что на практике в выражении &*(vc1+200) некорректное разыменование *(vc1+200) практически никогда не произойдет, т.к. результатом всего выражения является адрес и ни один серьезный компилятор не станет выбирать значение по некоторому адресу (операция разыменования) чтобы потом получить тот же самый адрес с помощью операции &.

5.1.1. Ноль, стр. 128

Если вы чувствуете, что просто обязаны определить NULL, воспользуйтесь

```
const int NULL=0;
```

Лично я бы советовал быть более осмотрительным. Существуют платформы, где размер указателя не равен размеру int. Рассмотрим, например, следующий код:

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

void error(int stat ...)
{
 va_list ap;
 va_start(ap,stat);

 while (const char* sarg=va_arg(ap,const char *))
       printf("%s",sarg);

 va_end(ap);
 exit(stat);
```

```
}

int main()
{
 error(1,"Случилось ","страшное",NULL);
}
```

Здесь описана функция error() с переменным числом параметров. Она печатает переданные ей строки, пока не встретит нулевой указатель и прекращает выполнение программы с заданным кодом завершения. Если в нашей реализации sizeof(char*)>sizeof(int), и NULL определен через int, то при запуске приведенного кода нас ждут большие неприятности.
Как же быть? Прежде всего, не следует использовать подобного рода функции, провоцирующие ошибки. Ну а если они действительно нужны, то важно помнить, что функциям с переменным числом параметров нужно передавать в точности ожидаемые типы. Т.е. приведенный вызов следует переписать так:

```
error(1,"Случилось ","страшное",static_cast<const char *>(0));
```

Каюсь, ненужная писанина выводит меня из себя, и в реальной программе я бы использовал эквивалентное:

```
error(1,"Случилось ","страшное",(char *)0);
```

хоть это и не педагогично.

7.4. Перегруженные имена функций, стр. 192
Процесс поиска подходящей функции из множества перегруженных заключается в ...
Приведенный в книге пункт 2 нужно заменить на:
[2] Соответствие, достигаемое "продвижением" ("повышением в чине") интегральных типов (например, bool в int, char в int, short в int; ¶ B.6.1), float в double.
Также следует отметить, что доступность функций-членов не влияет на процесс поиска подходящей функции, например:

```
struct A {
 private:
       void f(int);
 public:
       void f(...);

};

void g()
{
 A a;
 a.f(1);  // ошибка: выбирается A::f(int), использование
          // которой в g() запрещено
}
```

Отсутствие данного правила породило бы тонкие ошибки, когда выбор подходящей функции зависел бы от места вызова: в функции-члене или в обычной функции.

7.6. Неуказанное количество аргументов, стр. 199
До выхода из функции, где была использована va_start(), необходимо осуществить вызов va_end(). Причина состоит в том, что va_start() может модифицировать стек таким образом, что станет невозможен нормальный выход из функции.
Ввиду чего возникают совершенно незаметные подводные камни. Общеизвестно, что обработка исключения предполагает раскрутку стека. Следовательно, если в момент возбуждения исключения функция изменила стек, то у вас гарантированно будут неприятности. Таким образом, до вызова va_end() следует воздерживаться от потенциально вызывающих исключения операций. Специально добавлю, что ввод/вывод C++ может генерировать исключения, т.е. "наивная" техника вывода в std::cout до вызова va_end() чревата неприятностями.
Д-р Страуструп пишет по этому поводу следующее:
It is your job to ensure that va_end() is called. The language and standard library offers no guarantees. I don't recommend the use of varargs.
Это ваше дело следить за тем, чтобы va_end() была вызвана. Язык и стандартная

библиотека не дают никаких гарантий. Я бы не рекомендовал использовать технику stdarg.

7.7. Указатель на функцию, стр. 202
Причина в том, что разрешение использования cmp3 в качестве аргумента ssort() нарушило бы гарантию того, что ssort() вызовется с аргументами mytype*.
Здесь имеет место досадная опечатка, совершенно искажающая смысл предложения. Следует читать так: Причина в том, что разрешение использования cmp3 в качестве аргумента ssort() нарушило бы гарантию того, что cmp3() вызовется с аргументами mytype*.

10.4.6.2. Члены-константы, стр. 296
Можно проинициализировать член, являющийся статической константой интегрального типа, добавив к объявлению члена константное выражение в качестве инициализирующего значения.
Вроде бы все хорошо, но почему только интегрального типа? В чем причина подобной дискриминации? Д-р Страуструп пишет по этому поводу следующее:
The reason for "discriminating against" floating points in constant expressions is that the precision of floating point traditionally varied radically between processors. In principle, constant expressions should be evaluated on the target processor if you are cross compiling.
Причина подобной "дискриминации" плавающей арифметики в константных выражениях в том, что обычно точность подобных операций на разных процессорах существенно отличается. В принципе, если вы осуществляете кросс-компиляцию, то такие константные выражения должны вычисляться на целевом процессоре.
От себя добавлю, что звучит это немного нелогично, т.к. я могу написать так:
```
class Curious {
      static const float c5;
};

const float Curious::c5=7.5f;
```

а вот такой код:
```
class Curious { static const float c5=7.5f; };
```
уже вызовет ошибку. К счастью, далеко не все компиляторы страдают такими надуманными ограничениями, но, к сожалению, использовав эти послабления, вы получите непереносимый код, что может превратить в кошмар "простой перенос тривиального кода" в другую среду.

10.4.7. Массивы, стр. 297
Не существует способа явного указания аргументов конструктора (за исключением использования списка инициализации) при объявлении массива.
Но вы легко можете его получить, не изменяя при этом исходный класс, что, вообще говоря, важно. Для этого достаточно воспользоваться локальным классом:
```
#include <stdio.h>

struct A {  // исходный класс
       int a;
       A(int a_) : a(a_) { printf("%d\n",a); }
};

void fun()
{
 static int s;

 struct A_fun : public A {  // вспомогательный локальный
        A_fun() : A(s++) { }
 };

 A_fun arr[5];
 // и далее используем как A arr[5];
}
```

```
int main()
{
 fun();
}
```

К сожалению, локальные классы и их использование остались за рамками книги. Я попытаюсь исправить данное упущение. Вот часть стандарта о локальных классах:
9.8. Объявления локальных классов (Local class declarations [class.local])
- 1 - Класс может быть определен внутри функции; такой класс называется локальным (local) классом. Имя локального класса является локальным в окружающем контексте (enclosing scope). Локальный класс находится в окружающем контексте и имеет тот же доступ к именам вне функции, что и у самой функции. Объявления в локальном классе могут использовать только имена типов, статические переменные, extern переменные и функции, перечисления из окружающего контекста. Например:

```
int x;
void f()
{
 static int s;
 int x;
 extern int g();

 struct local {
        int g() { return x; }   // ошибка, auto x
        int h() { return s; }   // OK
        int k() { return ::x; } // OK
        int l() { return g(); } // OK
 };
 //  ...
}

local* p = 0;  // ошибка: нет local в текущем контексте
```

- 2 - Окружающая функция не имеет специального доступа к членам локального класса, она имеет обычные права доступа (см. class.access). Функции-члены локального класса, если они есть, должны быть определены внутри определения класса.
- 3 - Если класс X -- локальный класс, то в нем может быть объявлен вложенный класс Y, который может быть определен позднее, в определении класса X; или он может быть определен позднее в том же контексте, что и X. Вложенный класс локального класса сам является локальным.
- 4 - Локальный класс не может иметь статических данных-членов.

11.3.1. Операторы-члены и не-члены, стр. 316

```
complex r1=x+y+z;  // r1=operator+(x,operator+(y,z))
```

На самом деле данное выражение будет проинтерпретировано так:

```
complex r1=x+y+z;  // r1=operator+(operator+(x,y),z)
```

Потому что операция сложения левоассоциативна: (x+y)+z.

11.5.1. Поиск друзей, стр. 328
Приведенный в конце данной страницы пример нужно заменить на:

```
// нет f() в данной области видимости

class X {
      friend void f();          // бесполезно
      friend void h(const X&);  // может быть найдена по аргументам
};

void g(const X& x)
{
 f();   // нет f() в данной области видимости
 h(x);  // h() -- друг X
```

```
}
```

Он взят из списка авторских исправлений к 8-му тиражу.

11.7.1. Явные конструкторы, стр. 333
Разница между
```
String s1='a';  // ошибка: нет явного преобразования char в String
String s2(10);  // правильно: строка для хранения 10 символов
```

может показаться очень тонкой...
На самом деле разница сразу же заметна невооруженным глазом. Если мы пишем
```
X a=b;
```

то это подразумевает, что мы хотим создать объект a класса X , проинициализировав его значением объекта b некоторого класса. А инициализация всегда означает копирование, т.е. если b того же типа, что и a, то сразу вызывается конструктор копирования, что эквивалентно записи
```
X a(b);
```

Если же объект b имеет другой тип, то с помощью соответствующего конструктора (неявно) создается временный объект класса X и далее все идет по накатанной колее. То есть
```
X a(X(b));
```

Надеюсь, что после этих объяснений разница между объявлением s1 и s2 стала очевидной.
Ввиду всего вышесказанного хотелось бы отметить, что для типов, определяемых пользователем, не стоит использовать нотацию
```
X a=b;
```

т.к. в лучшем случае она эквивалентна более явному
```
X a(b);
```

а в худшем приводит к совершенно излишнему созданию временных объектов, как, например, в объявлении
```
complex z=5;
```

Вместе с тем, реализация вправе не создавать ненужные временные объекты. Вот соответствующая часть стандарта.
12.8. Копирование объектов класса (Copying class objects)
- 15 - Каждый раз, когда временный объект класса копируется при помощи конструктора копирования, и данный объект и копия имеют идентичные cv-квалификаторы, реализации разрешено считать оригинал и копию двумя ссылками на один и тот же объект и не осуществлять копирование, даже если копирующий конструктор или деструктор имеет побочный эффект. Для возвращающей класс функции, если выражение в операторе return является именем локального объекта и cv-квалификаторы локального объекта совпадают с типом возвращаемого значения, реализации разрешено не создавать временный объект для хранения возвращаемого значения, даже если копирующий конструктор или деструктор имеет побочный эффект. В этих случаях объект будет уничтожен позднее (время его жизни определяется копией).
Давайте не поленимся и напишем маленький класс, позволяющий отследить возникающие при этом спецэффекты.
```
#include <stdio.h>
#include <string.h>

struct A {
       static const int nsize=10;

       char n[nsize];

       A(char cn) {
         n[0]=cn;
         n[1]=0;
         printf("%5s.A::A()\n",n);
```

```
      }

      A(const A& a) {
        if (strlen(a.n)<=nsize-2) {
           n[0]='?';
           strcpy(n+1,a.n);
        }
        else strcpy(n,"беда");
        printf("%5s.A::A(const A& %s)\n",n,a.n);
      }

      ~A() {
         printf("%5s.A::~A()\n",n);
      }

      A& operator=(const A& a) {
           if (strlen(a.n)<=nsize-2) {
              n[0]='=';
              strcpy(n+1,a.n);
           }
           else strcpy(n,"беда");
           printf("%5s.A::operator=(const A& %s)\n",n,a.n);
           return *this;
      }

};

A f1(A a)
{
 printf("A f1(A %s)\n",a.n);
 return a;
}

A f2()
{
 printf("A f2()\n");
 A b('b');
 return b;
}

A f3()
{
 printf("A f3()\n");
 return A('c');
}

int main()
{
 {
  A a('a');
  A b='b';
  A c(A('c'));
  A d=A('d');
 }
 printf("----------\n");
 {
  A a('a');
  A b=f1(a);
  printf("b это %s\n",b.n);
 }
 printf("----------\n");
 {
  A a=f2();
  printf("a это %s\n",a.n);
 }
 printf("----------\n");
```

```
 {
  A a=f3();
  printf("a это %s\n",a.n);
 }
}
```

Прежде всего, в main() разными способами создаются объекты a, b, c и d. В нормальной реализации вы получите следующий вывод:

```
    a.A::A()
    b.A::A()
    c.A::A()
    d.A::A()
    d.A::~A()
    c.A::~A()
    b.A::~A()
    a.A::~A()
```

Там же, где разработчики компилятора схалтурили, появятся ненужные временные объекты, например:

```
    . . .
    c.A::A()
   ?c.A::A(const A& c)
    c.A::~A()
    d.A::A()
    d.A::~A()
   ?c.A::~A()
    . . .
```

Т.е. A c(A('c')) превратилось в A tmp('c'), c(tmp). Далее, вызов f1() демонстрирует неявные вызовы конструкторов копирования во всей красе:

```
    a.A::A()
   ?a.A::A(const A& a)
A f1(A ?a)
  ??a.A::A(const A& ?a)
   ?a.A::~A()
b это ??a
  ??a.A::~A()
    a.A::~A()
```

На основании a создается временный объект ?a, и передается f1() качестве аргумента. Далее, внутри f1() на основании ?a создается другой временный объект -- ??a, он нужен для возврата значения. А вот тут-то и происходит исключение нового временного объекта -- b это ??a, т.е. локальная переменная main() b -- это та самая, созданная в f1() переменная ??a, а не ее копия (специально для сомневающихся: будь это не так, мы бы увидели b это ???a).
Полностью согласен -- все это действительно очень запутанно, но разобраться все же стоит. Для более явной демонстрации исключения временной переменной я написал f2() и f3():

```
A f2()
    b.A::A()
   ?b.A::A(const A& b)
    b.A::~A()
a это ?b
   ?b.A::~A()

A f3()
    c.A::A()
a это c
    c.A::~A()
```

В f2() оно происходит, а в f3() -- нет, как говорится, все дело в волшебных пузырьках. Другого объяснения нет, т.к. временная переменная должна была исключиться в обоих случаях (ох уж мне эти писатели компиляторов!).
А сейчас рассмотрим более интересный случай -- перегрузку операторов. Внесем в наш класс соответствующие изменения:

```
#include <stdio.h>
```

```
#include <string.h>

struct A {
       static const int nsize=10;
       static int tmpcount;

       int val;
       char n[nsize];

       A(int val_) : val(val_) {  // для создания временных объектов
         sprintf(n,"_%d",++tmpcount);
         printf("%5s.A::A(int %d)\n",n,val);
       }

       A(char cn,int val_) : val(val_) {
         n[0]=cn;
         n[1]=0;
         printf("%5s.A::A(char,int %d)\n",n,val);
       }

       A(const A& a) : val(a.val) {
         if (strlen(a.n)<=nsize-2) {
            n[0]='?';
            strcpy(n+1,a.n);
         }
         else strcpy(n,"беда");
         printf("%5s.A::A(const A& %s)\n",n,a.n);
       }

       ~A() {
          printf("%5s.A::~A()\n",n);
       }

       A& operator=(const A& a) {
            val=a.val;
            if (strlen(a.n)<=nsize-2) {
               n[0]='=';
               strcpy(n+1,a.n);
            }
            else strcpy(n,"беда");
            printf("%5s.A::operator=(const A& %s)\n",n,a.n);
            return *this;
       }

       friend A operator+(const A& a1,const A& a2) {
         printf("operator+(const A& %s,const A& %s)\n",a1.n,a2.n);
         return A(a1.val+a2.val);
       }
};

int A::tmpcount;

int main()
{
 A a('a',1), b('b',2), c('c',3);
 A d=a+b+c;
 printf("d это %s\n",d.n);
 printf("d.val=%d\n",d.val);
}
```

После запуска вы должны получить следующие результаты:

```
    a.A::A(char,int 1)
    b.A::A(char,int 2)
    c.A::A(char,int 3)
operator+(const A& a,const A& b)
   _1.A::A(int 3)
```

```
operator+(const A& _1,const A& c)
   _2.A::A(int 6)
   _1.A::~A()
d это _2
d.val=6
   _2.A::~A()
    c.A::~A()
    b.A::~A()
    a.A::~A()
```

Все довольно наглядно, так что объяснения излишни. А для демонстрации работы оператора присваивания попробуйте

```
 A d('d',0);
 d=a+b+c;
```

В данном случае будет задействовано на одну временную переменную больше:

```
    a.A::A(char,int 1)
    b.A::A(char,int 2)
    c.A::A(char,int 3)
    d.A::A(char,int 0)
operator+(const A& a,const A& b)
   _1.A::A(int 3)
operator+(const A& _1,const A& c)
   _2.A::A(int 6)
  =_2.A::operator=(const A& _2)
   _2.A::~A()
   _1.A::~A()
d это =_2
d.val=6
  =_2.A::~A()
    c.A::~A()
    b.A::~A()
    a.A::~A()
```

11.9. Вызов функции, стр. 337

Функция, которая вызывается повторно, -- это operator()() объекта Add(z). Использование шаблонов и смысл их параметров может стать для вас совершенно непонятным, если раз и навсегда не уяснить одну простую вещь: при вызове функции-шаблона вы передаете объекты, но критически важной для инстанцирования шаблонов информацией являются типы переданных объектов. Сейчас я проиллюстрирую данную идею на приведенном в книге примере.

Рассмотрим, например, определение функции-шаблона for_each()

```
template <class InputIter, class Function>
Function for_each(InputIter first, InputIter last, Function f) {
 for ( ; first != last; ++first)
     f(*first);
 return f;
}
```

Данное определение я взял непосредственно из SGI STL (предварительно убрав символы подчеркивания для улучшения читаемости). Если его сравнить с приведенным в книге, то сразу бросается в глаза исправление типа возвращаемого значения (по стандарту должен быть аргумент-функция) и отказ от использования потенциально менее эффективного постинкремента итератора.

Когда мы вызываем for_each() с аргументом Add(z),

```
for_each(ll.begin(),ll.end(),Add(z));
```

то Function -- это Add, т.е. тип, а не объект Add(z). И по определению for_each() компилятором будет сгенерирован следующий код:

```
Add for_each(InputIter first, InputIter last, Add f) {
 for ( ; first != last; ++first)
     f.operator()(*first);
 return f;
}
```

Т.о. в момент вызова for_each() будет создан временный объект Add(z), который затем и будет передан в качестве аргумента. После чего, внутри for_each() для копии этого объекта будет вызываться Add::operator()(complex&). Конечно, тип InputIter также будет заменен типом соответствующего итератора, но в данный момент это нас не интересует.
На что же я хочу обратить ваше внимание? Я хочу отметить, что шаблон -- это не макрос в который передается что-то, к чему можно приписать скобки с соответствующими аргументами. Если бы шаблон был макросом, непосредственно принимающим переданный объект, то мы бы получили

```
Add for_each(. . .) {
 for (. . .)
     Add(z).operator()(*first);
 return f;
}
```

что, в принципе, тоже корректно, только крайне неэффективно: при каждом проходе цикла создается временный объект, к которому затем применяется операция вызова функции.

11.12. Класс String, стр. 344
Обратите внимание, что для неконстантного объекта, s.operator[](1) означает Cref(s,1).
А вот здесь хотелось бы по-подробнее. Почему в одном классе мы можем объявить const и не const функции-члены? Как осуществляется выбор перегруженной функции? Рассмотрим следующее объявление:

```
struct X {

  void f(int);
  void f(int) const;
};

void h()
{
 const X cx;
 cx.f(1);

 X x;
 x.f(2);
}
```

Ввиду того, что функция-член всегда имеет скрытый параметр this, компилятор воспринимает данное объявление как

```
// псевдокод
struct X {
  void f(      X *const this);
  void f(const X *const this);
};

void h()
{
 const X cx;
 X::f(&cx,1);

 X x;
 X::f(&x,2);
}
```

и выбор перегруженной функции осуществляется по обычным правилам. В общем -- никакой мистики.

12.2. Производные классы, стр. 351
Базовый класс иногда называют суперклассом, а производный -- подклассом. Однако

подобная терминология вводит в заблуждение людей, которые замечают, что данные в объекте производного класса являются надмножеством данных базового класса. Вместе с тем, данная терминология совершенно естественна в теоретико-множественном смысле. А именно: каждый объект производного класса является объектом базового класса, а обратное, вообще говоря, неверно. Т.о. базовый класс шире, поэтому он и суперкласс. Путаница возникает из-за того, что больше сам класс, а не его объекты, которые ввиду большей общности класса должны иметь меньше особенностей -- членов.

12.2. Производные классы, стр. 361
Стоит помнить, что традиционной и очевидной реализацией вызова виртуальной функции является просто косвенный вызов функции...
Это, вообще говоря, неверно. При применении множественного наследования "просто косвенного вызова" оказывается недостаточно. Рассмотрим следующую программу:

```
#include <iostream.h>

struct B1 {
       int b1;  // непустая
};

struct B2 {
       int b2;  // непустая
       virtual void vfun() { }
};

struct D : B1, B2 {  // множественное наследование от непустых классов
       virtual void vfun() {
              cout<<"D::vfun(): this="<<this<<"\n";
       }
};

int main()
{
 D d;

 D* dptr=&d;
 cout<<"dptr\t"<<dptr<<"\n";
 dptr->vfun();

 B2* b2ptr=&d;
 cout<<"b2ptr\t"<<b2ptr<<"\n";
 b2ptr->vfun();
}
```

На своей машине я получил следующие результаты:

```
dptr    0x283fee8
D::vfun(): this=0x283fee8
b2ptr   0x283feec
D::vfun(): this=0x283fee8
```

Т.е. при вызове через указатель на производный класс dptr, внутри D::vfun() мы получим this=0x283fee8. Но несмотря на то, что после преобразования исходного указателя в указатель на (второй) базовый класс b2ptr, его значение (очевидно) изменилось, внутри D::vfun() мы все равно видим исходное значение, что полностью соответствует ожиданиям D::vfun() относительно типа и значения своего this.
Что же все это означает? А означает это то, что если бы вызов виртуальной функции

```
struct D : B1, B2 {
 virtual void vfun(D *const this) {  //  псевдокод
         // . . .
 }
};
```

через указатель ptr->vfun() всегда сводился бы к вызову vfun(ptr), то в нашей программе мы бы получили this==b2ptr==0x283fee8.

Вопрос номер два: как они это делают? Элементарно, Ватсон! (Не переборщил?)
Известно, что виртуальная функция производного класса замещает (override) соответствующую функцию базового класса, а именно: в соответствующую позицию vtbl записывается адрес функции производного класса. Только вот в часть vtbl, относящуюся лично к классу D будет записан адрес собственно D::vfun(), а в часть, относящуюся к подъобекту B2 будет записан адрес некоторой, сгенерированной компилятором функции vfun_hack():

```
// псевдокод
void vfun_hack(B2 *const this)  // обратите внимание: указатель на B2
{
 return D::vfun(static_cast<D*>(this));  // да, return void;
}
```

Хотя наверняка оптимизатор исключит вложенный вызов функции с помощью простого goto D::vfun, однако для простоты восприятия об этом лучше не говорить.

13.2.3. Параметры шаблонов, стр. 382
В частности, строковый литерал не допустим в качестве аргумента шаблона. Потому что строковый литерал -- это объект с внутренней компоновкой (internal linkage).

13.6.2. Параметризация и наследование, стр. 399
Любопытно, что конструктор шаблона никогда не используется для генерации копирующего конструктора (так, чтобы при отсутствии явно объявленного копирующего конструктора, генерировался бы копирующий конструктор по умолчанию).
Прекрасный образчик дословного перевода. Сразу же вспоминается другой, не менее удачный перевод:
-- How do you do? // Как вы это делаете?
-- It's all right. // Все время правой.
Так что же имелось ввиду под этой туманной фразой? А имелось ввиду, что если копирующий конструктор не был явно объявлен, то он будет сгенерирован компилятором самостоятельно, причем предоставленные пользователем конструкторы-шаблоны не принимаются во внимание, т.е. по ним конструктор копирования сгенерирован быть не может.
Далее хочу отметить, что постоянно встречающуюся в переводе фразу "конструктор шаблона" следует понимать как "конструктор-шаблон".

14.4.1. Использование конструкторов и деструкторов, стр. 419
Итак, там, где годится подобная простая модель выделения ресурсов, автору конструктора нет необходимости писать явный код обработки исключений.
Если вы решили, что тем самым должна повыситься производительность, ввиду того, что в теле функции отсутствуют блоки try/catch, то должен вас огорчить -- они будут автоматически сгенерированы компилятором, для корректной обработки раскрутки стека. Но все-таки, какая версия выделения ресурсов обеспечивает большую производительность? Давайте протестируем следующий код:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

long Var, Count;

void ResourceGet();
void ResourceReturn();
void Work();

struct SafeResource {
       SafeResource()  { ResourceGet(); }
       ~SafeResource() { ResourceReturn(); }
};


void f1()
```

```
{
 ResourceGet();
 try {
     Work();
 }
 catch (...) {
       ResourceReturn();
       throw;
 }
 ResourceReturn();
}

void f2()
{
 SafeResource sr;
 Work();
}

int main(int argc,char** argv)
{
 if (argc>1) Count=atol(argv[1]);

 clock_t c1,c2;

 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000000; j++)
          f1();

  c2=clock();
  printf("f1(): %ld mln calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000000; j++)
          f2();

  c2=clock();
  printf("f2(): %ld mln calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
}

void ResourceGet()     { Var++; }

void ResourceReturn()  { Var--; }

void Work() { Var+=2; }
```

Как выдумаете, какая функция работает быстрее? А вот и нет! В зависимости от компилятора быстрее работает то f1(), то f2(), а иногда они работают совершенно одинаково из-за полной идентичности сгенерированного компилятором кода. Все зависит от используемых принципов обработки исключений и качества оптимизатора. Как же работают исключения? Мне известны две принципиально разных стратегии: метод раскрутки стека (РС) и метод естественных возвратов (ЕВ). Рассмотрим их действие на следующем примере:

```
void g1(), g2();

void f()
{
 g1();
 g2();
}
```

```
void g1()
{
 // 1
 throw E();
 // 2
}
```

Для метода EB сгенерированный код будет выглядеть следующим образом:

```
extern bool InException;

void f()
{
 g1();
 if (InException) return;

 g2();
}

void g1()
{
 // 1
 CopyAndSave(E());
 InException=true;
 return;
 // 2
}
```

Т.е. происходит простое отображение исключений на традиционно применяемую в C/C++ технику проверки кодов возврата. Конечно, для возвращающих значения и создающих объекты функций картина значительно усложняется.
Метод PC реализуется с помощью разбора содержимого стека, ввиду чего каждая использующая исключения функция обязана устанавливать специальный стековый фрейм, по которому можно найти адреса созданных в стеке объектов и корректно их уничтожить.
Каждый из методов обладает своими достоинствами и недостатками, так, например, для метода PC пролог/эпилог функций становится довольно "тяжелым", каждая функция обязана устанавливать стековый фрейм, метод EB здесь существенно выигрывает в производительности. С другой стороны, если мы должны осуществить возврат из глубоко рекурсивной функции, не создававшей объектов, то для метода PC работа заключается в простом сбросе стека, а EB будет вынужден проделать долгий обратный путь.
В целом, любая стратегия реализации исключений всегда приводит к накладным расходам и разбуханию сгенерированного кода. Все известные мне реализации при генерации кода с поддержкой исключений приводили к значительным накладным расходам. Вместе с тем, д-р Страуструп пишет по этому поводу:

It is possible to drive the exception handling overhead down below 1% run-time overhead in real implementations - I know because it has been done.

В принципе, вызываемые поддержкой исключений накладные расходы времени выполнения (в реальных реализациях) могут быть сделаны менее 1%, я могу это утверждать, т.к. это уже было реализовано.
Вместе с тем, там, где от исключений можно отказаться, не стоит пренебрегать открывшейся возможностью. Как правило, критические участки кода изолируют в отдельных файлах и компилируют с отключенной поддержкой исключений, конечно, при этом вы обязаны гарантировать, что любой код в данном файле не приведет к генерации исключения, даже посредством невинного new или вызова внешней по отношению к оптимизированному модулю функции (любые возникающие в ней исключения не должны ее покинуть).
Резюмируя данный раздел, хочется отметить, что как расширенное использование шаблонов совершенно изменило внешний вид стандартного C++, исключения кардинально изменили вид внутренний. C++ начал свою жизнь в UNIX с транслятора cfront, который просто переводил код C++ в эквивалентный код C, однако после года работы над cfront 4.0, когда он уже был написан, команда разработчиков cfront решила от него отказаться, т.к. удовлетворяющий всем требованиям стандарта механизм обработки исключений оказался совершенно неэффективным при его реализации непосредственно на C.

14.4.2. auto_ptr, стр. 421

В стандартном заголовочном файле <memory> auto_ptr объявлен следующим образом... Ввиду того, что после выхода первых (английских) тиражей стандарт претерпел некоторые изменения в части auto_ptr, концовку данного раздела следует заменить следующим текстом (он взят из списка авторских исправлений к 4 тиражу).

Для достижения данной семантики владения (также называемой семантикой разрушения после копирования (destructive copy semantics)), семантика копирования шаблона auto_ptr радикально отличается от семантики копирования обычных указателей: когда один auto_ptr копируется или присваивается другому, исходный auto_ptr очищается (эквивалентно присваиванию 0 указателю). Т.к. копирование auto_ptr приводит к его изменению, то const auto_ptr не может быть скопирован.

Шаблон auto_ptr определен в <memory> следующим образом:

```
template<class X> class std::auto_ptr {
 // вспомогательный класс
 template <class Y> struct auto_ptr_ref { /* ... */ };

 X* ptr;
public:
 typedef X element_type;

 explicit auto_ptr(X* p =0) throw() { ptr=p; }
 ~auto_ptr() throw() { delete ptr; }

 // обратите внимание: конструкторы копирования и операторы
 // присваивания имеют неконстантные аргументы

 // скопировать, потом a.ptr=0
 auto_ptr(auto_ptr& a) throw();

 // скопировать, потом a.ptr=0
 template<class Y> auto_ptr(auto_ptr<Y>& a) throw();

 // скопировать, потом a.ptr=0
 auto_ptr& operator=(auto_ptr& a) throw();

 // скопировать, потом a.ptr=0
 template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw();

 X& operator*() const throw() { return *ptr; }
 X* operator->() const throw() { return ptr; }

 // вернуть указатель
 X* get() const throw() { return ptr; }

 // передать владение
 X* release() throw() { X* t = ptr; ptr=0; return t; }

 void reset(X* p =0) throw() { if (p!=ptr) { delete ptr; ptr=p; } }

 // скопировать из auto_ptr_ref
 auto_ptr(auto_ptr_ref<X>) throw();

 // скопировать в auto_ptr_ref
 template<class Y> operator auto_ptr_ref<Y>() throw();

 // разрушающее копирование из auto_ptr
 template<class Y> operator auto_ptr<Y>() throw();
};
```

Назначение auto_ptr_ref -- обеспечить семантику уничтожения после копирования, ввиду чего копирование константного auto_ptr становится невозможным. Конструктор-шаблон и оператор присваивания-шаблон обеспечивают возможность неявного пребразования auto_ptr<D> в auto_ptr<B> если D* может быть преобразован в B*, например:

```
void g(Circle* pc)
{
 auto_ptr<Circle> p2 = pc;  // сейчас p2 отвечает за удаление

 auto_ptr<Circle> p3 = p2;  // сейчас p3 отвечает за удаление,
                            // а p2 уже нет

 p2->m = 7;                 // ошибка программиста: p2.get()==0

 Shape* ps = p3.get();      // извлечение указателя

 auto_ptr<Shape> aps = p3;  // передача прав собственности и
                            // преобразование типа

 auto_ptr<Circle> p4 = pc;  // ошибка: теперь p4 также отвечает за удаление
}
```

Эффект от использования нескольких auto_ptr для одного и того же объекта неопределен; в большинстве случаев объект будет уничтожен дважды, что приведет к разрушительным результатам.
Следует отметить, что семантика уничтожения после копирования не удовлетворяет требованиям для элементов стандартных контейнеров или стандартных алгоритмов, таких как sort(). Например:

```
// опасно: использование auto_ptr в контейнере
void h(vector<auto_ptr<Shape> >& v)
{
 sort(v.begin(),v.end());  // не делайте так: элементы не будут отсортированы
}
```

Понятно, что auto_ptr не является обычным "умным" указателем, однако он прекрасно справляется с предоставленной ему ролью -- обеспечивать безопасную относительно исключений работу с автоматическими указателями, и делать это без существенных накладных расходов.

14.4.4. Исключения и оператор new, стр. 422
При некотором использовании этого синтаксиса выделенная память затем освобождается, при некотором -- нет.
Т.к. приведенные в книге объяснения немного туманны, вот соответствующая часть стандарта:
5.3.4. New [expr.new]
- 17 - Если инициализация объекта завершается из-за возбуждения исключения и может быть найдена подходящая функция освобождения памяти, она вызывается для освобождения выделенной для размещения объекта памяти, а само исключение передается окружающему контексту. Если подходящая функция освобождения не может быть однозначно определена, освобождение выделенной памяти не производится (это удобно, когда функция выделения памяти на самом деле память не выделяет, если же память была выделена, то это, вероятно, приведет к утечке памяти).

14.6.1. Проверка спецификаций исключений, стр. 431
Спецификация исключений не является частью типа функции, и typedef не может ее содержать.
Сразу же возникает вопрос: в чем причина этого неудобного ограничения? Д-р Страуструп пишет по этому поводу следующее:
The reason is the exception spacification is not part of the type; it is a constraint that is checked on assignment and exforced at run time (rather than at compile time). Some people would like it to be part of the type, but it isn't. The reason is to avoid difficulties when updating large systems with parts from different sources. See "The Design and Evolution of C++" for details.
Спецификация исключений не является частью типа, данное ограничение проверяется при присваивании указателей и является механизмом времени выполнения (а не времени компиляции). Некоторым людям хотелось бы, чтобы спецификация исключений была частью типа, но это не так. Причина в том, что мы хотим избежать трудностей, возникающих при внесении изменений в большие системы, состоящие из отдельных частей, полученных из разных источников. Для детального описания

смотрите "Проект и эволюция C++".
С великому сожалению русскоязычного читателя, смотреть оказывается нечего, и не ясно получим ли мы перевод этой книги вообще.

14.6.3. Отображение исключений, стр. 431
В настоящее время стандарт не поддерживает отображение исключений в std::bad_exception описанным в данном разделе образом. Вот что об этом пишет д-р Страуструп:
The standard doesn't support the mapping of exceptions as I describe it in

14.6.3. It specifies mapping to std::bad_exception for exceptions thrown explicitly within an unexpected() function. This makes std::bad_exception an ordinary and rather pointless exception. The current wording does not agree with the intent of the proposer of the mechanism (Dmitry Lenkov of HP) and what he thought was voted in. I have raised the issue in the standards committee.
Стандарт не поддерживает отображение исключений в том виде, как это было описано в разделе 14.6.3. Он специфицирует отображение в std::bad_exception только для исключений, сгенерированных в unexpected(). Это лишает std::bad_exception первоначального смысла, делая его обычным и отчасти бессмысленным исключением. Текущая формулировка (стандарта) не совпадает с первоначально предложенной Дмитрием Ленковым из HP. Комитет стандартов поставлен в известность.
Ну и раз уж столько слов было сказано про формулировку из стандарта, думаю, что стоит здесь ее привести:
15.5.2 Функция unexpected() [except.unexpected]
- 1 - Если функция со спецификацией исключений возбуждает исключение не принадлежащее ее спецификации, будет вызвана функция
```
 void unexpected();
```

сразу же после завершения раскрутки стека (stack unwinding).
- 2 - Функция unexpected() не может вернуть управление, но может (пере)возбудить исключение. Если она возбуждает новое исключение, которое разрешено нарушенной до этого спецификацией исключений, то поиск подходящего обработчика будет продолжен с точки вызова сгенерировавшей неожиданное исключение функции. Если же она возбудит недозволенное исключение, то: Если спецификация исключений не содержит класс std::bad_exception (18.6.2.1), то будет вызвана terminate(), иначе (пере)возбужденное исключение будет заменено на определяемый реализацией объект типа std::bad_exception и поиск соответствующего обработчика будет продолжен описанным выше способом.
- 3 - Таким образом, спецификация исключений гарантирует, что могут быть возбуждены только перечисленные исключения. Если спецификация исключений содержит класс std::bad_exception, то любое неописанное исключение может быть заменено на std::bad_exception внутри unexpected().

15.3.2.1. Множественное наследование и управление доступом, стр. 461
... доступ разрешен только в том случае, если он разрешен по каждому из возможных путей.
Тут, конечно, имеет место досадная опечатка, что, кстати сказать, сразу видно из приведенного примера. Т.е. читать следует так: ... если он разрешен по некоторому из возможных путей.

15.6. Свободная память, стр. 478
В принципе, освобождение памяти осуществляется тогда внутри деструктора (который знает размер).
Именно так. Т.е. если вы объявили деструктор некоторого класса
```
A::~A()
{
 // тело деструктора
}
```

то компилятором (чаще всего) будет сгенерирован следующий код
```
// псевдокод
A::~A(A *const this,bool flag)
{
```

```
 if (this) {
    // тело деструктора
    if (flag) delete(this,sizeof(A));
 }
}
```

Ввиду чего функция

```
void f(Employee* ptr)
{
 delete ptr;
}
```

превратится в

```
// псевдокод
void f(Employee* ptr)
{
 Employee::~Employee(ptr,true);
}
```

и т.к. класс Employee имеет виртуальный деструктор, то это в конечном итоге приведет к вызову соответствующего метода.

15.3.2.1. Множественное наследование и управление доступом, стр. 480
... допускаются некоторые ослабления по отношению к типу возвращаемого значения. Следует отметить, что эти "некоторые ослабления" не являются простой формальностью. Рассмотрим следующий пример:

```
#include <iostream.h>

struct B1 {
       int b1;  // непустая
};

struct B2 {
       int b2;  // непустая
       virtual B2* vfun() {
              cout<<"B2::vfun()\n";  // этого мы не должны увидеть
              return this;
       }
};

struct D : B1, B2 {  // множественное наследование от непустых классов
       virtual D* vfun() {
              cout<<"D::vfun(): this="<<this<<"\n";
              return this;
       }
};

int main()
{
 D d;

 D* dptr=&d;
 cout<<"dptr\t"<<dptr<<"\n";

 void* ptr1=dptr->vfun();
 cout<<"ptr1\t"<<ptr1<<"\n";

 B2* b2ptr=&d;
 cout<<"b2ptr\t"<<b2ptr<<"\n";

 void* ptr2=b2ptr->vfun();
 cout<<"ptr2\t"<<ptr2<<"\n";
}
```

Обратите внимание: в данном примере я воспользовался "некоторыми ослаблениями"

для типа возвращаемого значения D::vfun(), и вот к чему это привело:
```
dptr    0012FF6C
D::vfun(): this=0012FF6C
ptr1    0012FF6C
b2ptr   0012FF70
D::vfun(): this=0012FF6C
ptr2    0012FF70
```

Т.о. оба раза была вызвана D::vfun(), но возвращаемое ей значение зависит от способа вызова (ptr1!=ptr2), как это, собственно говоря, и должно быть. Делается это точно так же, как уже было описано, -- через автоматическую генерацию vfun_hack() и запись ее адреса в соответствующую часть vtbl. Только сама vfun_hack() "существенно" усложняется:
```
// псевдокод
B2* vfun_hack(B2 *const this)
{
 D* ptr= D::vfun(static_cast<D*>(this));

 return static_cast<B2*>(ptr);
}
```

Коль скоро мы должны обеспечить как пре- так и постобработку, вложенного вызова функции не избежать.
Подводя итог хочется отметить, что в общем случае вызов виртуальной функции становится все меньше похож на "просто косвенный вызов функции". К счастью, виртуальные функции с ковариантным (covariant) типом возврата встречаются довольно редко.
И раз уж речь зашла о виртуальных функциях с ковариантным типом возврата, стоит привести соответствующую часть стандарта:
10.3. Виртуальные функции (Virtual functions [class.virtual])
- 5 - Тип возвращаемого значения замещающей функции может быть или идентичен типу замещаемой функции или быть ковариантным (covariant). Если функция D::f замещает функцию B::f, типы возвращаемых ими значений будут ковариантными, если они удовлетворяют следующим условиям:
  они оба являются указателями или ссылками на класс (многоуровневые указатели или ссылки на многоуровневые указатели запрещены)
  класс в возвращаемом значении B::f идентичен классу в возвращаемом значении D::f или он является однозначно определенным открытым прямым или косвенным базовым классом возвращаемого значения D::f
  как указатели так и ссылки имеют идентичные cv-квалификаторы и, при этом, класс возвращаемого значения D::f имеет те же или меньшие cv-квалификаторы, что и класс в возвращаемом значении B::f.
Если тип возвращаемого значения D::f отличается от типа возвращаемого значения B::f, то тип класса в возвращаемом значении D::f должен быть завершен в точке определения D::f или он должен быть типом D. Когда замещающая функция будет вызывана (как последняя заместившая функция), тип ее возвращаемого значения будет (статически) преобразован в тип возвращаемого значения замещаемой функции (см. expr.call). Например:
```
class B {};
class D : private B { friend class Derived; };
struct Base {
 virtual void vf1();
 virtual void vf2();
 virtual void vf3();
 virtual B*   vf4();
 virtual B*   vf5();

 void f();
};

struct No_good : public Base {
 D* vf4();  // ошибка: базовый класс B недоступен
};

class A;
struct Derived : public Base {
```

```
 void vf1();     // виртуальная и замещает Base::vf1()
 void vf2(int);  // не виртуальная, скрывает Base::vf2()
 char vf3();     // ошибка: неправильный тип возвращаемого значения
 D*   vf4();     // OK: возвращает указатель на производный класс
 A*   vf5();     // ошибка: возвращает указатель на незавершенный класс
 void f();
};

void g()
{
 Derived d;
 Base* bp=&d;      // стандартное преобразование: Derived* в Base*
 bp->vf1();        // вызов  Derived::vf1()
 bp->vf2();        // вызов  Base::vf2()
 bp->f();          // вызов  Base::f()  (не виртуальная)
 B* p=bp->vf4();   // вызов  Derived::pf() и преобразование
                   // возврата в B*
 Derived* dp=&d;
 D* q=dp->vf4();   // вызов  Derived::pf(), преобразование
                   // результата в B* не осуществляется
 dp->vf2();        // ошибка: отсутствует аргумент
}
```

А что означает загадочная фраза "меньшие cv-квалификаторы"?

3.9.3. CV-квалификаторы [basic.type.qualifier]

- 4 - Множество cv-квалификаторов является частично упорядоченным:

Нет cv-квалификатора<const
Нет cv-квалификатора<volatile
Нет cv-квалификатора<const volatile
Const<const volatile
Volatile<const volatile

16.2.3. STL-контейнеры, стр. 498

Она явилась результатом целенаправленного поиска бескомпромиссно эффективных общих алгоритмов.
Вместе с тем, не стоит думать, что STL не содержит снижающих эффективность компромиссов. Очевидно, что специально написанный для решения конкретной проблемы код будет работать эффективнее, вопрос в том, насколько эффективнее? Например, если нам нужно просто сохранить в памяти заранее неизвестное количество элементов, а затем их последовательно использовать, то (односвязный) список будет наиболее адекватной структурой данных. Однако STL не содержит односвязных списков, как много мы на этом теряем?
Рассмотрим следующий пример:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <list>

struct List {  // односвязный список
       struct Data {
              int val;
              Data* next;

              Data(int v, Data* n=0) : val(v), next(n) {}
       };

       Data *head, *tail;

       List() { head=tail=0; }
       ~List() {
            for (Data *ptr=head, *n; ptr; ptr=n) {  // удаляем все элементы
                n=ptr->next;
                delete ptr;
            }
       }
```

```
       void push_back(int v) {  // добавляем элемент
         if (!head) head=tail=new Data(v);
         else tail=tail->next=new Data(v);
       }
};

long Count, Var;

void f1()
{
 List lst;
 for (int i=0; i<1000; i++)
     lst.push_back(i);

 for (List::Data* ptr=lst.head; ptr; ptr=ptr->next)
     Var+=ptr->val;
}

void f2()
{
 std::list<int> lst;
 for (int i=0; i<1000; i++)
     lst.push_back(i);

 for (std::list<int>::const_iterator ci=lst.begin(); ci!=lst.end(); ++ci)
     Var+=*ci;
}


int main(int argc,char** argv)
{
 if (argc>1) Count=atol(argv[1]);

 clock_t c1,c2;
 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000; j++)
          f1();

  c2=clock();
  printf("f1(): %ld ths calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000; j++)
          f2();

  c2=clock();
  printf("f2(): %ld ths calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
}
```

В нем f1() использует определенный нами List: вставляет 1000 элементов, а затем проходит по списку.
Т.к. STL использует собственный распределитель памяти (вскоре вы увидите, что делает она это совсем не напрасно), то то же самое следует попробовать и нам:

```
struct List {  // односвязный список
       struct Data {
              int val;
              Data* next;
```

```
            Data(int v, Data* n=0) : val(v), next(n) {}

            // для собственного распределения памяти
            static Data* free;
            static void allocate();
            void* operator new(size_t);
            void operator delete(void*,size_t);
      };

      // . . .
};

List::Data* List::Data::free;

void List::Data::allocate()
{
 const int sz=100;  // выделяем блоки по sz элементов
 free=reinterpret_cast<Data*>(new char[sz*sizeof(Data)]);

 // сцепляем свободные элементы
 for (int i=0; i<sz-1; i++)
     free[i].next=free+i+1;
 free[sz-1].next=0;
}

inline void* List::Data::operator new(size_t)
{
 if (!free) allocate();

 Data* ptr=free;
 free=free->next;

 return ptr;
}

inline void List::Data::operator delete(void* dl,size_t)
{  // добавляем в начало списка свободных элементов
 Data* ptr=static_cast<Data*>(dl);
 ptr->next=free;
 free=ptr;
}
```

Обратите внимание, что в данном примере наш распределитель памяти не возвращает полученную память системе.
И, для чистоты эксперимента , в заключение попробуем двусвязный список -- его по праву можно назвать вручную написанной альтернативой std::list<int>:

```
struct DList {  // двусвязный список
      struct Data {
            int val;
            Data *prev, *next;

            Data(int v, Data* p=0, Data* n=0) : val(v), prev(p), next(n) {}

            // для собственного распределения памяти
            static Data* free;
            static void allocate();
            void* operator new(size_t);
            void operator delete(void*,size_t);
      };

      Data *head, *tail;

      DList() { head=tail=0; }
      ~DList() {
           for (Data *ptr=head, *n; ptr; ptr=n) {  // удаляем все элементы
               n=ptr->next;
```

```
                delete ptr;
            }
        }

        void push_back(int v) {  // добавляем элемент
          if (!head) head=tail=new Data(v);
          else tail=tail->next=new Data(v,tail);
        }
};
```

Итак, все готово, и можно приступать к тестированию. Данные три теста я попробовал на двух разных компиляторах, вот результат:

односвязныйодносвязный с собственным распределителем памятидвусвязный с собственным распределителем памяти
f1()f2()f1()f2()f1()f2()
реализация 19.612.11.112.11.312.1
реализация 220.22.51.82.51.92.5

И что же мы здесь видим?

- добавление собственного распределителя памяти существенно ускоряет программу (в нашем примере, от 9 до 11 раз). Также замечу , что это приводит и к существенной экономии памяти (как правило, для борьбы с внутренней фрагментацией стандартные распределители памяти выделяют отрезки памяти, кратные некоторому числу, например 16 байт, плюс память теряется на поддерживающих список выделенных блоков структурах. Т.е. желательно выделять память большими блоками и как можно реже.
- в зависимости от качества компилятора/реализации STL, использование STL может быть как почти приемлемым (замедление 30% для двусвязного и 40% для односвязного списков), так и неприемлемым совершенно (замедление от 9 до 11 раз!)
- двусвязный список является вполне приемлемой альтернативой односвязному (замедление 5 -- 20%)

Итак, наши измерения показывают, что бескомпромиссная эффективность STL является мифом. Даже более того, если вы используете недостаточно хороший оптимизатор, то использование STL вызовет существенные накладные расходы.

16.3.4. Конструкторы, стр. 505

То есть каждый из 10 000 элементов vr инициализируется конструктором Record(), а каждый из s1 элементов контейнера vi инициализируется int().
Инициализация 10 000 элементов конструктором по умолчанию не может не впечатлять -- только в очень редком случае нужно именно это. Если вы выделяете эти 10 000 элементов про запас, для последующей перезаписи, то стоит подумать о следующей альтернативе:

```
vector<X> vx;          // объявляем пустой вектор
vx.resize(10000);      // резервируем место воизбежание "дорогих"
                       // перераспределений в push_back()
// . . .
vx.push_back(x_work);  // добавляем элементы по мере надобности
```

О ней тем более стоит подумать, т.к. даже в отличной реализации STL 3.2 от SGI конструктор

```
vector<int> vi(s1);
```

подразумевает собой явный цикл заполнения нулями:

```
for (int i=0; i<s1; i++)
    vi.elements[i]=0;
```

и требуется достаточно интеллектуальный оптимизатор для превращения этого цикла в вызов memset()

```
memset(vi.elements, 0, sizeof(int)*s1);
```

что значительно улучшит производительность (конечно не программы вообще, а только данного отрезка кода). Matt Austern поставлен в известность, и в будущих версиях SGI STL можно ожидать повышения производительности данного конструктора.

16.3.5. Операции со стеком, стр. 508
Сноска: То есть память выделяется с некоторым запасом (обычно на десять элементов). -- Примеч. ред.
Очень жаль, что дорогая редакция сочла возможным поместить в книгу такую глупость. Для приведения количества "дорогих" перераспределений к приемлемому уровню O(log(N)), в STL используется увеличение объема зарезервированной памяти в два раза, а при простом добавлении некоторого количества (10, например) мы, очевидно, получим O(N), что есть плохо. Также отмечу, что для уменьшения количества перераспределений стоит воспользоваться resize(), особенно, если вы заранее можете оценить предполагаемую глубину стека.

17.4.1.2. Итераторы и пары, стр. 541
Также обеспечена функция, позволяющая удобным образом создавать pair.
Честно говоря, при первом знакомстве с шаблонами от всех этих многословных объявлений начинает рябить в глазах, и не всегда понятно, что именно удобно в такой вот функции:
```
template <class T1,class T2>
pair<T1,T2> std::make_pair(const T1& t1, const T2& t2)
{
 return pair<T1,T2>(t1,t2);
}
```

А удобно следующее: Если нам нужен экземпляр класса-шаблона, то мы обязаны предоставить все необходимые для инстанцирования класса параметры, т.к. на основании аргументов конструктора они не выводятся. С функциями-шаблонами дела обстоят получше:
```
char c=1;
int  i=2;

// пробуем создать "пару"
pair(c,i);            // неправильно -- pair<char,int> не выводится
pair<char,int>(c,i);  // правильно
make_pair(c,i);       // правильно
```

17.4.1.5. Сравнения, стр. 543
Поэтому для константных ассоциативных массивов не существует версии operator[](), стр. 543
Вообще говоря, существует, т.к. она объявлена в классе, но, ввиду ее неконстантности, применена быть не может -- при попытке инстанцирования вы получите ошибку компиляции.

17.5.3.3. Другие операции, стр. 555
К сожалению, вызов явно квалифицированного шаблона члена требует довольно сложного и редкого синтаксиса.
А вот тут мы имеем дело с откровенным "ляпом" в стандарте, увы! Во-первых, приведенную фразу следует читать так: К сожалению, вызов явно квалифицированного члена-шаблона всегда требует довольно сложного и редкого синтаксиса. Вот тут-то и скрывается "ляп"; если я могу написать так:
```
f<int>();  // f -- функция-шаблон
```

то почему это запрещено для функции-члена
```
obj.f<int>();           // ошибка: (obj.f)<(int) . . .
obj.template f<int>();  // правильно
```

Это нелогично, и действительно есть компиляторы (GNU C 2.95, например), способные правильно разбираться с явной квалификацией параметров при вызове члена-шаблона без использования "template". Д-р Страуструп пишет по этому поводу:
"template" is not always needed. A compiler can hold lots of information that it can use to disambiguate constructs. However, to ensure portability it is important not to rely on information that is not used by all compilers and tools.
Квалификатор template необходим не всегда. Компилятор может хранить достаточно

информации для разрешения неоднозначных конструкций, однако для гарантии переносимости не стоит использовать информацию, которая не поддерживается всеми компиляторами.
Остается надеяться, что в стандарт внесут соответствующие изменения.

17.6. Определение нового контейнера, стр. 556
... а потом применяйте поддерживаемый hash_map.
А вот еще один "ляп", и нет ему оправдания! Дело в том, что в стандарте понятия "поддерживаемый hash_map" не существует. Еще больше пикантности данной ситуации придает тот факт, что в самой STL, которая является основной частью стандартной библиотеки C++, hash_map есть (и есть уже давно). Д-р Страуструп пишет по этому поводу, что hash_map просто проглядели, а когда хватились, то было уже поздно -- никакие существенные изменения внести в стандарт уже было нельзя. Ну что ж, бывает...

18.4.4.1. Связыватели, стр. 583
Читаемо? Эффективно?
Что же нам советуют признать читаемым и эффективным (впрочем, к эффективности претензий действительно нет; теоретически).
```
list<int>::const_iterator p=find_if(c.begin(),c.end(),bind2nd(less<int>(),7));
```

Осмелюсь предложить другой вариант:
```
for (list<int>::const_iterator p=c.begin(); p!=c.end(); ++p)
    if (*p<7) break;
```

Трудно ли это написать? По-видимому, нет. Является ли этот явный цикл менее читаемым? По моему мнению, он даже превосходит читаемость примера с использованием bind2nd(). А если нужно написать условие вида *p>=5 && *p<100, что, в принципе, встречается не так уж и редко, то вариант с использованием связывателей и find_if() проигрывает однозначно. Стоит добавить и чисто психологический эффект -- вызов красивой функции часто подсознательно воспринимается атомарной операцией и не лишне подчеркнуть, что, порой, за красивым фасадом скрывается крайне неэффективный последовательный поиск.
В целом, я агитирую против потери здравого смысла при использовании предоставленного нам пестрого набора свистулек и колокольчиков. Увы, следует признать, что для сколько-нибудь сложного применения они не предназначены, да и на простом примере польза практически не видна.

18.4.4.2. Адаптеры функций-членов, стр. 584
Сначала рассмотрим типичный случай, когда мы хотим вызвать функцию-член без аргументов...
Теперь немного про вызовы функций-членов для элементов контейнера с помощью механизма mem_fun(). Действительно, вариант
```
for_each(lsp.begin(),lsp.end(),mem_fun(&Shape::draw));  // рисуем все фигуры
```

подкупает своим изяществом. И даже более того, предоставляемые mem_fun() возможности действительно могут быть востребованы, например, при реализации некоторого абстрактного шаблона разработки (design pattern). Но за красивым фасадом скрывается вызов функции через указатель на член -- операция отнюдь не дешевая и далеко не все компиляторы умеют встраивать вызов функции через такой указатель, будем рисковать?
А что, если нам нужно повернуть все фигуры на заданный угол? bind2nd(), говорите? А если на разные углы да причем не все элементы контейнера, и эти углы рассчитываются по сложному алгоритму? По-моему, такой вариант в реальных программах встречается гораздо чаще.
Выходит, что и механизм mem_fun() не очень-то предназначен для серьезного использования. Изучить его, конечно, стоит, а вот использовать или нет -- решать вам.

18.6. Алгоритмы, модифицирующие последовательность, стр. 592
Вместо вставки и удаления элементов такие алгоритмы изменяют значения элементов...

Вот это да! Т.е. если я попытаюсь удалить элемент из списка с помощью такого remove(), то вместо удаления элемента я получу перезаписывание (в среднем) половины его элементов? Поймите меня правильно, среди приведенных в этом разделе алгоритмов будут и практически полезные, но держать в стандартной библиотеке не только неэффективные, но даже не соответствующие своему названию алгоритмы -- это уже слишком!

18.6.1. Копирование, стр. 592
Определения базовых операций копирования тривиальны...
Но в таком виде они будут совершенно неэффективны в приложении ко встроенным типам, ведь общеизвестно, что для копирования больших объемов информации (если без него действительно никак нельзя обойтись) следует использовать функции стандартной библиотеки C memcpy() и memmove(). Вы нечасто используете векторы встроенных типов? Осмелюсь заметить, что вектор указателей встречается не так уж и редко и как раз подходит под это определение. К счастью, у меня есть хорошая новость: в качественной реализации STL (например от SGI) вызов операции копирования для vector<int> как раз и приведет к эффективному memmove().
Выбор подходящего алгоритма производится на этапе компиляции с помощью специально определенного шаблона __type_traits<> -- свойства типа. Который (по умолчанию) имеет безопасные настройки для сложных типов с нетривиальными конструкторами/деструкторами и оптимизированные специализации для POD типов, которые можно копировать простым перемещением блоков памяти.
В C++ вы часто будете встречать аббревиатуру POD (Plain Old Data). Что же она обозначает? POD тип -- это тип, объекты которого можно безопасно перемещать в памяти (с помощью memmove(), например). Данному условию очевидно удовлетворяют встроенные типы (в том числе и указатели) и классы без определяемой пользователем операции присваивания и деструктора.
Почему я об этом говорю? Потому что, например, очевидное определение класса Date является POD типом:

```
class Date {
      int day, mon, year;
      // или даже
      long val;  // yyyymmdd
 public:
      // ...
};
```

Поэтому стоит разрешить оптимизацию предоставив соответствующую специализацию __type_traits<>:

```
template<> struct __type_traits<Date> {
 // ...
};
```

Только обратите внимание: __type_traits<> -- не часть стандартной библиотеки, разные реализации могут использовать различные имена или даже не производить оптимизацию вообще. Изучите то, что есть у вас.

19.2.5. Обратные итераторы, стр. 622
Это приводит к тому, что * возвращает значение *(current-1) ...
Да, по смыслу именно так:

```
24.4.1.3.3 - operator* [lib.reverse.iter.op.star]
 reference operator*() const;

-1- Effects:
 Iterator tmp = current;
 return *--tmp;
```

Т.е. каждый раз, когда вы применяете разыменование обратного итератора, происходит создание временного итератора, его декремент и разыменование. Не многовато ли, для такой простой и часто используемой (как правило, в цикле для каждого элемента) операции? Д-р Страуструп пишет по этому поводу следующее:
I don't think anyone would use a reverse iterator if an iterator was an alternative, but then you never know what people might know. When you actually need to go through a sequence in reverse order a reverse iterator is often quite

efficient compared to alternatives. Finally, there may not be any overhead because where the iterator is a vector the temporary isn't hard to optimize into a register use. One should measure before worrying too much about overhead.
Я не думаю, что кто-то будет использовать обратный итератор, когда можно использовать обычный, но мы никогда не можем знать, что думают другие люди. Когда вам действительно нужно пройти последовательноть в обратном порядке, обратный итератор является вполне приемлемой альтернативой. В принципе, иногда можно избежать накладных расходов вообще, например в случае обратного прохода по вектору (когда итератором является просто T*). В любом случае, прежде чем беспокоиться о производительности, следует произвести реальные измерения. Вместе с тем, обратный итератор все-таки несет в себе ненужные накладные расходы, и для обратного прохода по последовательности лучше использовать обычный итератор с явным (пре)декрементом.
И раз уж речь зашла о реальных измерениях, давайте их произведем.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <list>

long Count, Var;

std::list<int> lst;

void f1()
{
 typedef std::list<int>::reverse_iterator RI;
 for (RI p=lst.rbegin(); p!=lst.rend(); ++p)
     Var+=*p;
}

void f2()
{
 std::list<int>::iterator p=lst.end();
 if (p!=lst.begin())
    do {
       --p;
       Var+=*p;
    } while (p!=lst.begin());
}

int main(int argc,char** argv)
{
 if (argc>1) Count=atol(argv[1]);

 for (int i=0; i<10000; i++)
     lst.push_back(i);

 clock_t c1,c2;
 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000; j++)
          f1();

  c2=clock();
  printf("f1(): %ld ths calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
 {
  c1=clock();

  for (long i=0; i<Count; i++)
      for (long j=0; j<1000; j++)
          f2();

  c2=clock();
```

```
  printf("f2(): %ld ths calls per %.1f sec\n",Count,double(c2-c1)/CLK_TCK);
 }
}
```

В данном примере список из 10 000 элементов проходится несколько тысяч раз (задается параметром) с использованием обратного (в f1()) и обычного (в f2()) итераторов. При использовании качественного оптимизатора разницы времени выполнения замечено не было, а для "обычных" реализаций она составила от 45% до 2.4 раза.
И еще одна проблема: приводит ли постинкремент итератора к существенным накладным расходам по сравнению с преинкрементом? Давайте внесем соответствующие изменения:

```
void f1()
{
 typedef std::list<int>::iterator I;
 for (I p=lst.begin(); p!=lst.end(); ++p)
     Var+=*p;
}

void f2()
{
 typedef std::list<int>::iterator I;
 for (I p=lst.begin(); p!=lst.end(); p++)
     Var+=*p;
}
```

И опять все тот же результат: разницы может не быть, а там, где она проявлялась, ее величина находилась в пределах 5 -- 30 процентов.
В целом, не стоит использовать потенциально более дорогие обратные итераторы и постинкременты, если вы не убедились в интеллектуальности используемого оптимизатора.

19.4.2. Распределители памяти, определяемые пользователем, стр. 637

```
template<class T>

T* Pool_alloc<T>::allocate(size_type n, void* =0)
{
 if (n==1) return static_cast<T*>(mem_alloc());
 // ...
}
```

Как всегда, самое интересное скрывается за многоточием. Как же нам реализовать часть allocate<>() для n!=1? Простым вызовом в цикле mem_alloc()? Увы, в данном случае очевидное решение не подходит совершенно. Почему? Давайте рассмотрим поведение Pool_alloc<char>. Глядя на конструктор оригинального Pool:

```
Pool::Pool(unsigned int sz)
      : esize(sz<sizeof(Link*) ? sizeof(Link*) : sz)
{
 // ...
}
```

можно заметить, что для sz==sizeof(char) для каждого char мы будем выделять sizeof(Link*) байт памяти. Для "обычной" реализации это означает четырехкратный перерасход памяти! Т.о. выделение памяти для массивов объектов типа X, где sizeof(X)<sizeof(Link*) становится нетривиальной задачей, равно как и последующее их освобождение в deallocate<>(), фактически, придется принципиально изменить алгоритм работы аллокатора.

19.4.4. Неинициализированная память, стр. 641

```
template<class T, class A> T* temporary_dup(vector<T,A>& v)
{
 T* p=get_temporary_buffer<T>(v.size()).first;
 if (p==0) return 0;
 copy(v.begin(),v.end(),raw_storage_iterator<T*,T>(p));
```

```
 return p;
}
```

Вообще говоря, приведенная функция написана некорректно, т.к. не проверяется второй элемент возвращаемой get_temporary_buffer<>() пары. Т.к. get_temporary_buffer<>() может вернуть меньше памяти, чем мы запросили, то необходима другая проверка:

```
template<class T, class A> T* temporary_dup(vector<T,A>& v)
{
 pair<T*,ptrdiff_t> p(get_temporary_buffer<T>(v.size()));

 if (p.second<v.size()) {
    if (p.first) return_temporary_buffer(p.first);
    return 0;
 }

 copy(v.begin(),v.end(),raw_storage_iterator<T*,T>(p));
 return p;
}
```

20.2.1. Особенности символов, стр. 647

Вызов assign(s,n,x) при помощи assign(s[i],x) присваивает n копий x строке s. Функция compare() использует для сравнения символов lt() и eq().

К счастью, для обычных символов char_traits<char> это не так, в том смысле, что не происходит вызов в цикле lt(), eq(), assign(s[i],x), а используются специально для этого предназначенные memcmp() и memset(), что, впрочем, не влияет на конечный результат. Т.е. используя strcmp() мы ничего не выигрываем, даже более того, в специально проведенных мной измерениях производительности, сравнения string оказались на 30% быстрее, чем принятое в C сравнение char* с помощью strcmp(). Что и не удивительно: для string размеры сравниваемых массивов char известны заранее.

20.3.4. Конструкторы, стр. 652

Реализация basic_string хранит длину строки, не полагаясь на завершающий символ (ноль).

Вместе с тем, хорошо оптимизированные реализации хранят строку вместе с завершающим нулем, дабы максимально ускорить функцию basic_string::c_str(). Не секрет, что большинство используемых функций (традиционно) принимают строку в виде [const] char* вместо эквивалентного по смыслу [const] string&, исходя из того простого факта, что мы не можем ускорить "безопасную" реализацию, но можем скрыть эффективную за безопасным интерфейсом.

К слову сказать, исходя из моего опыта, слухи об опасности манипулирования простыми char* в стиле C оказываются сильно преувеличенными. Да, вы должны следить за всеми мелочами, но, например, ни у кого не возникает протеста по поводу того, что если в формуле корней квадратного уравнения мы вместо ▒-▒ напишем ▒+', то результат будет неверен.

Резюмируя данный абзац, хочу сказать, что string использовать можно и нужно, но если логика работы вашей программы интенсивно использует манипуляции со строками, стоит подумать о разработке собственных средств, основанных на функциях типа memcpy(), а в "узких" местах без этого просто не обойтись.

20.3.6. Присваивание, стр. 655

Это делает использование строк, которые только считываются и задаются в качестве аргумента, гораздо более дешевым, чем кто-то мог по наивности предположить. Однако было бы так же наивно со стороны программистов не проверять имеющиеся у них реализации перед написанием кода, который полагается на оптимизацию копирования строк.

Я бы попросил вас серьезно отнестись к данному совету (т.е. к проверке имеющейся реализации). Например, SGI STL 3.2 всегда копирует символы строки, не полагаясь на основанную на подсчете ссылок версию. Авторы библиотеки объясняют это тем, что использующие модель подсчета ссылок строки не подходят для многопоточных приложений, ими утверждается, что использующие данную реализацию строк многопоточные приложения аварийно завершают свою работу один раз в несколько

месяцев и именно из-за строк. Мне, честно говоря, данное утверждение кажется несерьезным, но факт остается фактом -- существуют отлично оптимизированные реализации стандартной библиотеки, которые по тем или иным причинам отказались от использования основанных на подсчете ссылок строк.
Резюмируя данный материал хочу отметить, что я всегда, где это возможно, стараюсь избегать копирования строк, например путем передачи const string&.

21.2.2. Вывод встроенных типов, стр. 676
... будет интерпретировано так:
```
(cerr.operator<<("x=")).operator<<(x);
```

Конечно же на самом деле все не так: в новых потоках ввода-вывода оператор вывода строки больше не является функцией-членом, следовательно оно будет интерпретировано так:
```
 operator<<(cerr,"x=").operator<<(x);
```

Товарищи программисты! Еще раз повторю: никогда не копируйте блоками старый текст, а если это все-таки необходимо, обязательно проверяйте каждую загогулину. Вот гражданин Страуструп забыл проверить, и, в результате, новый релиз его монографии содержит очевидную ошибку.

21.3.4. Ввод символов, стр. 687
Как уже было сказано, главная сила языка C -- в его способности считывать символы и решать, что с ними ничего не надо делать -- причем выполнять это быстро. Это действительно важное достоинство, которое нельзя недооценивать, и цель C++ -- не утратить его.
Вынужден вас огорчить -- потоки C++ не обладают данным важным свойством, они всегда работают медленнее C, а в некоторых реализациях -- медленно до смешного (правда, объективности ради стоит отметить, что мне попадались и совершенно отвратительно реализованные FILE* потоки C, в результате чего код C++ работал быстрее, но это просто недоразумение, если не сказать крепче!). Рассмотрим следующую программу:
```
#include <stdio.h>
#include <time.h>
#include <io.h>  // для open()
#include <fcntl.h>
#include <iostream.h>
#include <fstream.h>

void workc(char*);
void workcpp(char*);
void work3(char*);

int main(int argc, char **argv)
{
 if (argc==3)
    switch (*argv[2]-'0') {
           case 1: {
                workc(argv[1]);
                break;
           }
           case 2: {
                workcpp(argv[1]);
                break;
           }
           case 3: {
                work3(argv[1]);
                break;
           }
    }
}

void workc(char* fn)
{
```

```
 FILE* fil=fopen(fn,"rb");
 if (!fil) return;

 time_t t1; time(&t1);

 long count=0;
 while (getc(fil)!=EOF)
       count++;

 time_t t2; time(&t2);

 fclose(fil);
 cout<<count<<" bytes per "<<t2-t1<<" sec.\n" ;
}

void workcpp(char* fn)
{
 // для старых реализаций: ifstream fil(fn,ios::in|ios::binary);
 ifstream fil(fn,ios_base::in|ios_base::binary);
 if (!fil) return;

 time_t t1; time(&t1);

 long count=0;
 while (fil.get()!=EOF)
       count++;

 time_t t2; time(&t2);
 cout<<count<<" bytes per "<<t2-t1<<" sec.\n" ;
}

class File {
      int            fd;           // дескриптор файла
      unsigned char  buf[BUFSIZ];  // буфер стандартного размера
      unsigned char* gptr;         // следующий читаемый символ
      unsigned char* gend;         // конец данных

      int uflow();
 public:
      File(char* fn) : gptr(0), gend(0) { fd=open(fn,O_RDONLY|O_BINARY); }
      ~File() { if (Ok()) close(fd); }

      int Ok() { return fd!=-1; }

      int gchar() { return (gptr<gend) ? *gptr++ : uflow(); }
};

int File::uflow()
{
 if (!Ok()) return EOF;

 int rd=read(fd,buf,BUFSIZ);
 if (rd<=0) {  // ошибка или EOF
    close(fd);
    fd=-1;

    return EOF;
 }

 gptr=buf;
 gend=buf+rd;


 return *gptr++;
}
```

```
void work3(char* fn)
{
 File fil(fn);
 if (!fil.Ok()) return;

 time_t t1; time(&t1);

 long count=0;
 while (fil.gchar()!=EOF)
       count++;

 time_t t2; time(&t2);

 cout<<count<<" bytes per "<<t2-t1<<" sec.\n" ;
}
```

Ее нужно запускать с двумя параметрами. Первый параметр -- это имя (большого) файла для чтения, а второй -- цифра 1, 2 или 3, выбирающая функцию workc(), workcpp() или work3() соответственно. Только не забудьте про дисковый кэш, т.е. для получения объективных результатов программу нужно запустить несколько раз для каждого из вариантов.
Необычным местом здесь является функция work3(). Она написана специально для проверки "честности" реализации стандартных средств ввода-вывода C -- FILE*. Если вдруг окажется, что workc() работает существенно медленнее work3(), то вы имеете полное право назвать создателей такой библиотеки, как минимум, полными неучами.
А сейчас попробуем получить информацию к размышлению -- проведем серию контрольных запусков и посмотрим на результат. И что же нам говорят безжалостные цифры? На отлично оптимизированных реализациях мы получим разницу "всего" в 40% (например, 8 сек. против 11), а вот на "обычных"... Для одного широко распространенного коммерческого пакета (не будем показывать пальцем) я даже получил отличие в 11 раз! (7 сек. против 79).
Почему же все так плохо? Все становится очевидным, стоит только взглянуть на определения вызываемых функций.
Для C с его getc() в типичной реализации мы имеем:

```
#define getc(f) \
  ((--((f)->level) >= 0) ? (unsigned char)(*(f)->curp++) : \
    _fgetc (f))
```

Т.е. коротенький макрос вместо функции. Как говорится -- всего-ничего. А вот для C++ все гораздо сложнее. Вот код из оптимизированной реализации-чемпиона:

```
inline
int get() { if (!ipfx1()) return EOF;
  else { int ch = _strbuf->sbumpc();
         if (ch == EOF) set(ios::eofbit);
         return ch;
       } }
```

Проследим вызовы функций при обычном поведении:

```
inline
int ipfx1() { // Optimized version of ipfx(1).
 if (!good()) { set(ios::failbit); return 0; }
 else {
   _IO_flockfile(_strbuf);
   if (_tie && rdbuf()->in_avail() == 0) _tie->flush();
   return 1;
 }
    }

// _IO_getc() -- это похожая на getc() работа с буфером
inline int sbumpc() { return _IO_getc(this); }
```

Даже при беглом взгляде становится ясно, что количество выполняемого кода весьма сильно превосходит то, что мы имеем в C. Но самое интересное заключается в том, что количество "лишнего" кода существенно превышает показанную разницу в 40%. В чем же дело? А дело в том, что узким местом данного цикла является обращение к

диску, и именно из-за стоимости дисковых операций получаются эти неожиданные 40%. Более подробно данная тема будет рассмотрена в посвященном оптимизации кода разделе.
А что будет, если мы выделим большой буфер и зададим его нашим функциям?

```
void workc(char* fn)
{
 FILE* fil=fopen(fn,"rb");
 if (!fil) return;

 if (Bsize>0 && setvbuf(fil,Buf,_IOFBF,Bsize)) return;

 // . . .
}

void workcpp(char* fn)
{
 // для старых реализаций: ifstream fil(fn,ios::in|ios::binary);
 ifstream fil(fn,ios_base::in|ios_base::binary);
 if (!fil) return;

 // для старых реализаций: fil.rdbuf()->setbuf(Buf,Bsize);
 if (Bsize>0) fil.rdbuf()->pubsetbuf(Buf,Bsize);

 // . . .
}
```

Как ни странно, по сути ничего не изменится! Дело в том, что современные ОС при работе с диском используют очень качественные алгоритмы кэширования, так что еще один уровень буферизации внутри приложения оказывается излишним (в том смысле, что используемые по умолчанию буферы потоков оказываются вполне адекватными). Кстати, одним из хороших примеров необходимости использования многопоточных программ является возможность ускорения работы программ копирования файлов, когда исходный файл и копия расположены на разных устройствах. В этом случае программа запускает несколько потоков, осуществляющих асинхронные чтение и запись. Но в современных ОС в этом нет никакого смысла, т.к. предоставляемое системой кэширование кроме всего прочего обеспечивает и прозрачное для прикладных программ асинхронное чтение и запись.
Подводя итог, хочется отметить, что если ввод-вывод является узким местом вашего приложения, то следует воздержаться от использования стандартных потоков C++ и использовать проверенные десятилетиями методы.

21.4.6.3. Манипуляторы, определяемые пользователем, стр. 701

Коль скоро с эффективностью потоков ввода-вывода мы уже разобрались, следует поговорить об удобстве. К сожалению, для сколько-нибудь сложного форматирования предоставляемые потоками средства не предназначены. Не в том смысле, что средств нет, а в том, что данные средства совершенно неудобны и легко выводят из себя привыкшего к элегантному формату ...printf() программиста. Не верите? Давайте попробуем вывести обыкновенную дату в формате dd.mm.yyyy:

```
int day= 31,
    mon= 1,
    year=1974;

printf("%02d.%02d.%d\n",day,mon,year);  // 31.01.1974

cout<<setfill('0')<<setw(2)<<day<<'.'<<setw(2)<<mon<<setfill(' ')<<'.'
    <<year<<"\n";  // тоже 31.01.1974
```

Думаю, что комментарии излишни.
За что же не любят потоки C и чем потоки C++ могут быть удобнее? У потоков C++ есть только одно существенное достоинство -- типобезопасность. Т.к. потоки C++ все же нужно использовать, я написал специальный манипулятор, который, оставаясь типобезопасным, позволяет использовать формат ...printf(). Он не вызывает существенных накладных расходов и с его помощью приведенный выше пример будет выглядеть следующим образом:

```
cout<<c_form(day,"02")<<'.'<<c_form(mon,"02")<<'.'<<year<<'\n';
```

Вот исходный код:

```
#include <iostream>
#include <ctype.h>

class c_formBase {  // класс базовых операций
 protected:
      // для компактной записи
      typedef std::ios_base::fmtflags                              fmt_t;
      typedef std::basic_ostream<char,std::char_traits<char> > ostr_t;
      typedef std::ios_base                                        ios;
 private:
      const char* form;   // формат
      int   arg1, arg2;   // аргументы ширины/точности
      fmt_t       flags;  // сохраняемые флаги
      int         width;  // ширина
      int         prec;   // точность
      char        fill;   // символ-заполнитель
      fmt_t       myfl;   // устанавливаемые флаги
 protected:
      // конструктор с разбором формата
      // формат: [-|0] [число|*] [.[число|*]] [e|f|g|o|x]
      c_formBase( const char* form_,
                  const int arg1_,
                  const int arg2_
                )
      : form(form_), arg1(arg1_), arg2(arg2_) {
        const char* iptr=form;  // текущий символ строки формата

        myfl=0;
        fill=0;
        if (*iptr=='-') {  // выравнивание влево
           myfl|=ios::left;
           iptr++;
        }
        else if (*iptr=='0') {  // добавляем '0'ли только если !left
                fill='0';
                iptr++;
             }

        width=0;
        if (*iptr=='*') {  // читаем ширину, если есть
           width=arg1;
           iptr++;

           arg1=arg2;  // сдвигаем агрументы влево
        }
        else if (isdigit(*iptr)) width=getval(iptr);

        prec=0;
        if (*iptr=='.') {  // есть точность
           if (*++iptr=='*') {
              prec=arg1;
              iptr++;
           }
           else if (isdigit(*iptr)) prec=getval(iptr);
                else throw std::invalid_argument("c_form");
        }

        switch (*iptr++) {
               case   0: return;  // конец строки формата
               case 'e': myfl|=ios::scientific; break;
               case 'f': myfl|=ios::fixed;      break;
               case 'g':                        break;
               case 'o': myfl|=ios::oct;        break;
               case 'x': myfl|=ios::hex;        break;
```

```
                default: throw std::invalid_argument("c_form");
          }

          if (*iptr) throw std::invalid_argument("c_form");
        }

        static int getval(const char*& iptr) {  // чтение числа
               int ret=0;
               do ret=ret*10+ *iptr -'0';
                  while (isdigit(*++iptr));

               return ret;
        }

        void prologue(ostr_t& os) {  // настройка
             flags=os.flags();
             // очищаем floatfield и устанавливаем свои флаги
             os.flags((flags & ~ios::floatfield) | myfl);

             if (width) os.width(width);
             if (fill)  fill=os.fill(fill);
             if (prec)  prec=os.precision(prec);
        }

        void epilogue(ostr_t& os) {  // восстановление
             os.flags(flags);

             if (fill) os.fill(fill);
             if (prec) os.precision(prec);
        }
};

template <class T>
class c_formClass : private c_formBase {  // класс-шаблон для вывода типа T
        T val;  // выводимое значение
 public:
        c_formClass( T val_,
                     const char* form,
                     const int arg1,
                     const int arg2
                   )
        : c_formBase(form,arg1,arg2), val(val_) { }

        // оператор вывода
        friend ostr_t& operator<<(ostr_t& os, const c_formClass& cf_) {
               c_formClass& cf=const_cast<c_formClass&>(cf_);

               cf.prologue(os);
               os<<cf.val;
               cf.epilogue(os);

               return os;
        }
};

template <class T>  // функция-помощник
inline c_formClass<T> c_form( T val,             // выводимое значение
                              const char* form,  // формат вывода
                              const int arg1=0,  // необязательные
                              const int arg2=0   // параметры
                            )
{ return c_formClass<T>(val,form,arg1,arg2); }
```

Принцип его работы основан на следующей идее: функция c_form<>() возвращает объект класса c_formClass<>, для которого определена операция вывода в ostream. Для удобства использования, c_form<>() является функцией, т.к. если бы мы сразу

использовали конструктор некоторого класса-шаблона c_form<>, то нам пришлось бы явно задавать его параметры:

```
cout<<c_form<int>(day,"02");
```

что, мягко говоря, неудобно. Далее, мы, в принципе, могли бы не использовать базовый класс c_formBase, но это привело бы к совершенно ненужной повторной генерации общего для всех вариантов шаблона кода.
И еще. Несмотря на то, что конструктор c_formBase возбуждает исключения (std::invalid_argument), я не стал специфицировать это явным образом. Дело в том, что большинство компиляторов не умеет встраивать функции с явно заданной спецификацией возбуждаемых исключений, а одной из целей создания нашего манипулятора являлась производительнотсь.
Для старых реализаций приходится использовать другой код, что, кстати сказать, позволяет нам реально ощутить некоторые удобные изменения, привнесенные в C++ стандартом:

```
#include <iostream.h>
#include <ctype.h>

class c_formBase {  // класс базовых операций
 private:
      const char* form;   // формат
      int   arg1, arg2;   // аргументы ширины/точности
      long        flags;  // сохраняемые флаги
      int         width;  // ширина
      int         prec;   // точность
      char        fill;   // символ-заполнитель
      long        myfl;   // устанавливаемые флаги
 protected:
      // конструктор с разбором формата
      // формат: [-|0] [число|*] [.[число|*]] [e|f|g|o|x]
      c_formBase( const char* form_,
                  const int arg1_,
                  const int arg2_
                )
      : form(form_), arg1(arg1_), arg2(arg2_) {
        const char* iptr=form;  // текущий символ строки формата

        myfl=0;
        fill=0;
        if (*iptr=='-') {  // выравнивание влево
           myfl|=ios::left;
           iptr++;
        }
        else if (*iptr=='0') {  // добавляем '0'ли только если !left
                fill='0';
                iptr++;
             }

        width=0;
        if (*iptr=='*') {  // читаем ширину, если есть
           width=arg1;
           iptr++;

           arg1=arg2;  // сдвигаем агрументы влево
        }
        else if (isdigit(*iptr)) width=getval(iptr);

        prec=0;
        if (*iptr=='.') {  // есть точность
           if (*++iptr=='*') {
              prec=arg1;
              iptr++;
           }
           else if (isdigit(*iptr)) prec=getval(iptr);
        }
```

```
        switch (*iptr++) {
               case   0: return;  // конец строки формата
               case 'e': myfl|=ios::scientific; break;
               case 'f': myfl|=ios::fixed;      break;
               case 'g':                        break;
               case 'o': myfl|=ios::oct;        break;
               case 'x': myfl|=ios::hex;        break;
        }
      }

      static int getval(const char*& iptr) {  // чтение числа
             int ret=0;
             do ret=ret*10+ *iptr -'0';
                while (isdigit(*++iptr));

             return ret;
      }

      void prologue(ostream& os) {  // настройка
           flags=os.flags();
           // очищаем floatfield и устанавливаем свои флаги
           os.flags((flags & ~ios::floatfield) | myfl);

           if (width) os.width(width);
           if (fill)  fill=os.fill(fill);
           if (prec)  prec=os.precision(prec);
      }

      void epilogue(ostream& os) {  // восстановление
           os.flags(flags);

           if (fill) os.fill(fill);
           if (prec) os.precision(prec);
      }
};

template <class T>
class c_formClass : private c_formBase {  // класс-шаблон для вывода типа T
      T val;  // выводимое значение
 public:
      c_formClass( T val_,
                   const char* form,
                   const int arg1,
                   const int arg2
                 )
      : c_formBase(form,arg1,arg2), val(val_) { }

      // оператор вывода
      friend ostream& operator<<(ostream& os, const c_formClass<T>& cf_) {
             c_formClass<T>& cf=(c_formClass<T>&) cf_;

             cf.prologue(os);
             os<<cf.val;
             cf.epilogue(os);

             return os;
      }
};

template <class T>  // функция-помощник (два аргумента)
inline c_formClass<T> c_form(T val, char* form, int arg1, int arg2)
{ return c_formClass<T>(val,form,arg1,arg2); }

template <class T>  // функция-помощник (один аргумент)
inline c_formClass<T> c_form(T val, char* form, int arg1)
{ return c_formClass<T>(val,form,arg1,0); }
```

```
template <class T>  // функция-помощник (без аргументов)
inline c_formClass<T> c_form(T val, char* form)
{ return c_formClass<T>(val,form,0,0); }
```

21.6.2. Потоки ввода и буфера, стр. 711
Функция readsome() является операцией нижнего уровня, которая позволяет...
Т.к. приведенное в книге описание readsome() туманно, вот часть стандарта, относящаяся к ней:

```
streamsize readsome(char_type* s, streamsize n);
```

Эффект: Если !good() вызвать setstate(failbit), которая может возбудить исключение. Иначе извлечь символы и поместить их в массив, на который указывает s. Если rdbuf()->in_avail() == -1, вызвать setstate(eofbit) (которая может возбудить исключение ios_base::failure (lib.iostate.flags)) и не извлекать символы;
  Если rdbuf()->in_avail() == 0, не извлекать символы
  Если rdbuf()->in_avail() > 0, извлечь min(rdbuf()->in_avail(),n)).
Возвращает: Количество прочитанных символов.

23.4.3.1. Этап 1: выявление классов, стр. 773
Например, в математике окружность -- это частный случай эллипса, но в большинстве программ окружность не нужно выводить из эллипса, или делать эллипс потомком окружности.
Думаю, что стоит по-подробнее рассмотреть данный конкретный случай, т.к. он иллюстрирует довольно распространенную ошибку проектирования. На первый взгляд может показаться, что идея сделать класс Circle производным от класса Ellipse является вполне приемлемой, ведь они связаны отношением is-a: каждая окружность является эллипсом. Некорректность данной идеи станет очевидной, как только мы приступим к написанию кода.
У эллипса, кроме прочих атрибутов, есть два параметра: полуоси a и b. И производная окружность их унаследует. Более того, нам нужен один единственный радиус для окружности и мы не можем для этих целей использовать один из унаследованных атрибутов, т.к. это изменит его смысл и полученный от эллипса код перестанет работать. Следовательно мы вынуждены добавить новый атрибут -- радиус и поддерживать в корректном состоянии унаследованные атрибуты. Очевидно, что подобного рода наследование лишено смысла, т.к. не упрощает, а усложняет разработку.
В чем же дело? А дело в том, что понятие окружность в математическом смысле является ограничением понятия эллипс, т.е. его частным случаем. А наследование будет полезно, если конструируемый нами объект содержит подобъект базового класса и все унаследованные операции имеют смысл (рассмотрите, например, операцию изменения значения полуоси b -- она ничего не знает об инварианте окружности и легко его разрушит). Другими словами, объект производного класса должен быть расширением объекта базового класса, но не его частным случаем (изменением), т.к. мы не можем повлиять на поведение базового класса, если он нам не предоставил соответутвующих возможностей, например в виде подходящего набора виртуальных функций.

A.5. Выражения, стр. 879
То есть "если нечто можно понять как объявление, это и есть объявление".
Т.к. сложные объявления C++ могут быть непонятны даже неновичку, стоит прокомментировать приведенные в книге объявления. Неочевидность всех приведенных объявлений основана на добавлении лишних скобок:

T(*e)(int(3)); эквивалентно T* e(int(3)); То, что инициализация указателя с помощью int запрещена, синтаксичестим анализатором не принимается во внимание: будет распознано объявление указателя и выдана ошибка.

T(f)[4]; эквивалентно T f[4];

T(a); эквивалентно T a;

T(a)=m; эквивалентно T a=m;

T(*b)(); объявление указателя на функцию.

T(x),y,z=7; эквивалентно T x,y,z=7;

B.13.2. Друзья, стр. 931
Приведенный в конце страницы пример нужно заменить на:

```
template<class C> class Basic_ops {  // базовые операции с контейнерами
 friend bool operator==<>(const C&, const C&);  // сравнение элементов
 friend bool operator!=<>(const C&, const C&);
 // ...
};
```

Уголки (<>) после имен функций означают, что друзьями являются функции-шаблоны (поздние изменения стандарта).
Этот текст взят из списка авторских исправлений к 10 тиражу.
Почему в данном случае необходимы <>? Потому что иначе мы объявляем другом operator==() не шаблон, т.к. до объявления класса в окружающем контексте не было объявления operator==()-шаблона. Вот формулировка стандарта:
14.5.3. Друзья [temp.friend]
-1- Другом класса или класса-шаблона может быть функция-шаблон, класс-шаблон, их специализации или обычная (не шаблон) функция или класс. Для объявления функций-друзей которые не являются объявлениями шаблонов:
  если имя друга является квалифицированным или неквалифицированным идентификатором шаблона, то объявление друга ссылается на специализацию функции-шаблона, иначе,
  если имя друга является квалифицированным и в специфицированном классе или пространстве имен найдена соответствующая функция не шаблон, то объявление друга ссылается на эту функцию, иначе,
  если имя друга является квалифицированным и в специфицированном классе или пространстве имен найдена соответствующая функция-шаблон, то объявление друга ссылается на эту функцию, иначе,
  имя должно быть неквалифицированным и объявлять (или переобъявлять) обычную (не шаблон) функцию.
Например:

```
template<class T> class task;
template<class T> task<T>* preempt(task<T>*);

template<class T> class task {
 //  ...
 friend void next_time();
 friend void process(task<T>*);
 friend task<T>* preempt<T>(task<T>*);
 template<class C> friend int func(C);

 friend class task<int>;
 template<class P> friend class frd;
 //  ...
};
```

здесь функция next_time является другом каждой специализации класса-шаблона task; т.к. process не имеет явных template-arguments, каждая специализация task имеет функцию-друга process соответствующего типа и этот друг не является специализацией функции-шаблона; т.к. друг preempt имеет явный template-argument <T>, каждая специализация класса-шаблона task имеет другом соответствующую специализацию функции-шаблона preempt; и, наконец, каждая специализация класса-шаблона task имеет другом все специализации функции-шаблона func. Аналогично, каждая специализация класса-шаблона task имеет другом класс-специализацию task<int>, и все специализации класса-шаблона frd.

Оптимизация
Поговорим об оптимизации. Что нужно оптимизировать? Когда? И нужно ли вообще? В

этих вопросах очень легко заблудиться, если сразу же не выбрать правильную точку зрения. Как только мы посмотрим на наш код с точки зрения пользователя, сразу же все становится на свои места:

Программа должна делать то, что от нее требуется.

Она должна делать это хорошо.

Именно так -- глупо оптимизировать неправильно работающий код. Если же пользователя устраивает текущее быстродействие, не стоит искать неприятности.

Итак, анализ проведен, решение принято -- ускоряемся. Что может ускорить нашу программу? Да все, что угодно, вопрос поставлен некорректно. Что может существенно ускорить нашу программу? А вот над этим уже стоит подумать.

Прежде всего, стоит подумать о "внешнем" ускорении, т.е. о не приводящих к изменению исходного кода действиях. Самый широкораспространенный метод -- использование более мощного аппаратного обеспечения. Увы, иногда это не самый эффективный способ. Зачастую гораздо большего можно добиться путем правильного конфигурирования того, что есть. Например, работа с БД -- практически всегда самое узкое место. Должно быть очевидно, что правильная настройка сервера БД -- одно из самых важных действий и за него всегда должен отвечать компетентный специалист. Вы будете смеяться, но грубые оплошности сисадминов происходят слишком часто, чтобы на них не обращать внимание (из моей практики: неоднократно время работы приложения уменьшалось с нескольких часов до нескольких минут (!) из-за очевидной команды UPDATE STATISTICS, фактически, перед анализом плана испонения тяжелых SQL-запросов всегда полезно невзначай поинтересоваться актуальностью статистики. Не менее редким происшествием является "случайная потеря" индекса важной таблицы в результате реорганизации или резервного копирования БД).

Коль скоро среда исполнения правильно сконфигурирована, стоит обратить внимание непосредственно на код. Очевидно, что максимальная скорость эскадры определяется скоростью самого медленного корабля. Он-то нам и нужен. Если "эскадрой" является набор SQL-запросов работающего с БД приложения, то, как правило, никаких трудностей с определением узких мест не возникает. Трудности возникают с определением узких мест "обычных" приложений.

Узкие места нужно искать только с помощью объективных измерений, интуиция в данной области часто не работает (не стоит утверждать, что не работает вообще). Причем измерять относительную производительность имеет смысл только при "релиз"-настройках компилятора (при отключенной оптимизации узкие места могут быть найдены там, где их нет. Увы, данного рода ошибки допускают даже опытные программисты). Действительно серьезным подспорьем здесь являются профайлеры -- неотъемлемая часть любой профессиональной среды разработки.

Когда критический участок кода локализован, можно приступать к непосредственному анализу. С чего начать? Начинать нужно с самых ресурсоемких операций. Как правило, по требуемому для исполнения времени, операции легко разделяются на слои, отличающиеся друг от друга на несколько порядков:

работа с внешними устройствами

системные вызовы

вызовы собственных функций

локальные управляющие структуры

Например, не стоит заниматься вопросами размещения управляющей переменной цикла в соответствующем регистре процессора, если в данном цикле происходит обращение к диску. Вызовы собственных функций существенно отличаются от системных вызовов тем, что когда мы обращаемся к системе, происходит переключение контекста потока (системный код имеет больше привелегий, обращаться к нему можно только через специальные шлюзы) и обязательная проверка достоверности переданных аргументов (например, система проверяет действительно ли ей передана корректная строка путем ее посимвольного сканирования, если при этом произойдет нарушение прав доступа или ошибка адресации, то приложение будет об этом проинформировано; тем самым исключается возможность сбоя внутри ядра системы, когда неясно что делать и кто виноват, наиболее вероятный результат этого -- blue death screen, system trap и т.д., т.е. невосстановимый сбой самой системы).

Как правило, только в исключительных случаях заметного ускорения работы можно достичь путем локальных улучшений (которыми пестрят древние наставления: a+a вместо 2*a, for (register int i; и т.д.), современные компиляторы прекрасно справляются с ними без нас. Серьезные улучшения обычно приносит только изменение алгоритма работы. Первым делом стоит обратить внимание на сам алгоритм (классическим примером является сортировка с алгоритмами O(N*N) и O(N*log(N)) стоимости или выбор подходящего контейнера).

Если же принципиальный алгоритм изначально оптимален, можно попробовать

использовать замену уровней ресурсоемкости. Классическим примером является кэширование. Например вместо дорогостоящего считывания данных с диска, проиходит обращение к заранее подготовленной копии в памяти, тем самым мы переходим с уровня 1 на 2-3тий. Стоит отметить, что техника кэширования находит свое применение не только в работе с внешними устройствами. Например, в если в игровой программе узким местом становится вычисление sin(x), то стоит подумать об использовании заранее рассчитанной таблицы синусов (обычно достаточно 360 значений типа int вместо дорогостоящей плаваючей арифметики). Более "прикладной" пример -- это длинный switch по типам сообщений в их обработчике. Если он стал узким местом, то стоит подумать об использовании таблицы переходов или хэширования (стоимость O(1)) или же специальной древовидной структуры (стоимость O(log(N))) -- существенно лучше обеспечиваемого switch O(N).
Все эти замечания применимы в равной степени к любому языку. Давайте посмотрим на что стоит обратить внимание программистам на C++.
Прежде всего, стоит отметить, что все существенные маленькие хитрости уже были рассмотрены в предыдущих примерах так же как и скрытые накладные расходы. Быть может, за кадром осталась только возможность "облегченного вызова функции", т.к. она является не частью (стандартного) C++, а особенностью конкретных реализаций. C++ как и C при вызове функции размещает параметры в стеке. Т.е. имея параметр в регистре, компилятор заносит его в стек, вызывает функцию, а в теле функции опять переносит параметр в регистр. Всего этого можно избежать использовав соответствующее соглашение вызова (в некоторых реализациях используется зарезервированное слово _fastcall), когда параметры перед вызовом размещаются непосредственно в регистрах, исключая тем самым ненужные стековые операции.
Например в простом тесте:

```
void f1(int arg)
{
 Var+=arg;
}

void _fastcall f2(int arg)
{
 Var+=arg;
}
```

функция f1() работала на 50% медленнее. Конечно, реальную выгоду из этого факта можно получить только при массовом использовании функций облегченного вызова во всем проекте. Эта совершенно бесплатная разница может быть достаточно существенной.
Еще один немаловажный фактор -- размер программ. Откуда взялись все эти современные мегабайты? Увы, большая их часть -- мертвый код, реально, более 90% загруженного кода никогда не будет вызвано! Не беда, если эти мегабайты просто лежат на диске, реальные трудности появляются, когда вы загружаете на выполнение несколько таких монстров. Падение производительности системы во время выделения дополнительной виртуальной памяти может стать просто катастрофическим.
Если при разработке большого проекта изначально не придерживаться политики строгого определения зависимостей между исходными файлами (и не принимать серьезных мер для их минимизации), то в итоге, для успешной линковки будет необходимо подключить слишком много мусора из стандартного инструментария данного проекта. В несколько раз больше, чем полезного кода. Из-за чего это происходит? Если функция f() из file1.cpp вызывает g() из file2.cpp, то очевидно, что мы обязаны подключить file2.cpp к нашему проекту. При этом, если не было принято специальных мер, то в file2.cpp почти всегда найдется какая-нибудь g2(), как правило не нужная для работы g() и вызывающая функции еще какого-либо файла; и пошло-поехало. А когда каждое приложение содержит свыше полусотни исходных файлов, а в проекте несколько сотен приложений, то навести порядок постфактум уже не представляется возможным.

notes2

Макросы
В C++ макросы не нужны! До боли знакомое высказывание, не так ли? Я бы его немного уточнил: не нужны, если вы не хотите существенно облегчить себе жизнь. Я полностью согласен с тем, что чрезмерное и необдуманное использование макросов может вызвать большие неприятности, особенно при повторном использовании кода. Вместе с тем, я не знаю ни одного средства C++, которое могло бы принести пользу

при чрезмерном и необдуманном его использовании.
Итак, когда макросы могут принести пользу?
Макрос как надъязыковое средство. Хороший примером является простой, но удивительно полезный отладочный макрос _VAL_, выводящий имя и значение переменной:

```
#define _VAL_(var) #var "=" << var << " "
```

Надъязыковой является работа с переменной, как с текстом, путем перевода имени переменной (оно существует только в исходном коде программы) в строковый литерал, реально существующий в коде бинарном. Данную возможность могут предоставить только макросы.
Информация о текущем исходном файле и строке -- ее пользу при отладке трудно переоценить. Для этого я использую специальный макрос _ADD_. Например:

```
cout<<_ADD_("Ошибка чтения");
```

выведет что-то вроде
Ошибка чтения <file.cpp:34>
А если нужен перевод строки, то стоит попробовать

```
cout<<"Ошибка чтения" _ADD_("") "\n";
```

Такой метод работает, потому что макрос _ADD_ возвращает строковый литерал.
Вроде бы эквивалентная функция

```
char* _ADD_(char*);
```

вполне подошла бы для первого примера, но не для второго. Конечно, для вывода в cout это не имеет никакого значения, но в следующем пункте я покажу принципиальную важность подобного поведения.
Рассмотрим устройство _ADD_:

```
#define _ADD_tmp_tmp_(str,arg) str " <" __FILE__ ":" #arg ">"
#define _ADD_tmp_(str,arg) _ADD_tmp_tmp_(str,arg)
#define _ADD_(str) _ADD_tmp_(str,__LINE__)
```

Почему все так сложно? Дело в том, что __LINE__ в отличие от __FILE__ является числовым, а не строковым литералом и чтобы привести его к нужному типу приходится проявить некоторую смекалку. Мы, очевидно, не можем сразу написать #__LINE__, т.к. результатом будет "__LINE__". Из-за этого мы передаем __LINE__ другому макросу -- _ADD_tmp_, который уже увидит значение __LINE__ (34, например), и в следующей итерации мы можем использовать #arg без всяких неожиданностей.
Получение значения числового макроса в виде строки. Как показывает практика, данное действие необходимо не только в виде части сложных макросов. Допустим, что для взаимодействия с SQL-сервером у нас определен класс DB::Query с соответствующей функцией

```
void DB::Query::Statement(const char *);
```

и мы хотим выбрать все строки некоторой таблицы, имеющие равное некому "магическому числу" поле somefield:

```
#define FieldOK 7
// . . .
DB::Int tmp(FieldOK);
q.Statement(" SELECT * "
            " FROM sometable "
            " WHERE somefield=? "
);
q.SetParam(),tmp;
```

Чересчур многословно. Как бы это нам использовать FieldOK напрямую?
Недостаточно знакомые с возможностями макросов программисты делают это так:

```
#define FieldOK 7
// . . .
#define FieldOK_CHAR "7"
// . . .
q.Statement(" SELECT * "
            " FROM sometable "
            " WHERE somefield=" FieldOK_CHAR
);
```

В результате чего вы получаете все прелести синхронизации изменений взаимосвязанных наборов макросов со всеми вытекающими из этого ошибками. Правильным решением будет

```
#define FieldOK 7
// . . .
q.Statement(" SELECT * "
            " FROM sometable "
            " WHERE somefield=" _GETSTR_(FieldOK)
);
```

где _GETSTR_ определен следующим образом:

```
#define _GETSTR_(arg) #arg
```

Кстати, приведенный пример наглядно демонстрирует невозможность полностью эквивалентной замены всех числовых макросов на принятые в C++

```
const int FieldOK=7;
enum { FieldOK=7 };
```

макрос _GETSTR_ с ними не сможет работать.
Многократно встречающиеся части кода. Рассмотрим еще один пример из области работы с SQL-сервером. Предположим, что нам нужно выбрать данные из некоторой таблицы. Это можно сделать влоб:

```
struct Table1 {  // представление данных таблицы
       DB::Date  Field1;
       DB::Int   Field2;
       DB::Short Field3;
};

void f()
{
 Table1 tbl;
 DB::Query q;
 q.Statement(" SELECT Field1, Field2, Field3 "
             " FROM Table1 "
 );
 q.BindCol(),tbl.Field1, tbl.Field2, tbl.Field3;
 // . . .
}
```

И этот метод действительно работает. Но что, если представление таблицы изменилось? Теперь нам придется искать и исправлять все подобные места -- чрезвычайно утомительный процесс. Об этом стоило позаботиться заранее:

```
#define TABLE1_FLD      Field1, Field2, Field3
#define TABLE1_FLD_CHAR "Field1, Field2, Field3"

struct Table1 {  // представление данных таблицы
       DB::Date  Field1;
       DB::Int   Field2;
       DB::Short Field3;

       // вспомогательная функция
       void BindCol(DB::Query& q) { q.BindCol(),TABLE1_FLD; }
};

void f()
{
 Table1 tbl;
 DB::Query q;
 q.Statement(" SELECT " TABLE1_FLD_CHAR
             " FROM Table1 "
 );
 tbl.BindCol(q);
 // . . .
}
```

Теперь изменение структуры таблицы обойдется без зубовного скрежета. Стоит отметить, что в определении TABLE1_FLD_CHAR я не мог использовать очевидное _GETSTR_(TABLE1_FLD), т.к. TABLE1_FLD содержит запятые. К сожалению, данное печальное ограничение в примитивном препроцессоре C++ никак нельзя обойти. Многократно встречающиеся подобные части кода. Представим себе, что мы пишем приложение для банковской сферы и должны выбрать информацию по некоторым счетам. В России, например, счет состоит из многих полей, которые для удобства работы собирают в специальную структуру, а в таблице он может быть представлен смежными полями с одинаковым префиксом:

```
q.Statement(" SELECT Field1, AccA_bal, AccA_cur, AccA_key, AccA_brn, "
                    " AccA_per, Field2 "
            " FROM Table1 "
);
q.BindCol(),tbl.Field1, tbl.AccA.bal, tbl.AccA.cur, tbl.AccA.key,
            tbl.AccA.brn, tbl.AccA.per, tbl.Field2;
// . . .
```

Можете себе представить, сколько писанины требуется для выбора четырех счетов (tbl.AccA, tbl.AccB, tbl.KorA, tbl.KorB). И снова на помощь приходят макросы:

```
#define _SACC_(arg) #arg"_bal, "#arg"_cur, "#arg"_key, "#arg"_brn, " \
                    #arg"_per "
#define _BACC_(arg) arg.bal, arg.cur, arg.key, arg.brn, arg.per

// . . .

q.Statement(" SELECT Field1, " _SACC_(AccA) " , Field2 "
            " FROM Table1 "
);
q.BindCol(),tbl.Field1, _BACC_(tbl.AccA), tbl.Field2;
// . . .
```

Думаю, что комментарии излишни.

Рассмотрим более тонкий пример подобия. Пусть нам потребовалось создать таблицу для хранения часто используемой нами структуры данных:

```
struct A {
       MyDate Date;
       int    Field2;
       short  Field3;
};
```

Мы не можем использовать идентификатор Date для имени столбца таблицы, т.к. DATE является зарезервированным словом SQL. Эта проблема легко обходится с помощью приписывания некоторого префикса:

```
struct TableA {
       DB::Date  xDate;
       DB::Int   xField2;
       DB::Short xField3;

       void Clear();
       TableA& operator=(A&);
};
```

А теперь определим функции-члены:

```
void TableA::Clear()
{
 xDate="";
 xField2="";
 xField3="";
}

TableA& TableA::operator=(A& a)
{
 xDate=ToDB(a.Date);
 xField2=ToDB(a.Field2);
 xField3=ToDB(a.Field3);
```

```
 return *this;
}
```

Гарантирую, что если TableA содержит хотя бы пару-тройку десятков полей, то написание подобного кода вам очень быстро наскучит, мягко говоря! Нельзя ли это сделать один раз, а потом использовать результаты? Оказывается можно:

```
void TableA::Clear()
{
#define CLR(arg) arg=""
 CLR(xDate);
 CLR(xField2);
 CLR(xField3);
#undef CLR
}

TableA& TableA::operator=(A& a)
{
// используем склейку лексем: ##
#define ASS(arg) x##arg=ToDB(a.arg);
 ASS(xDate);
 ASS(xField2);
 ASS(xField3);
#undef ASS

 return *this;
}
```

Теперь определение `TableA::operator=()` по `TableA::Clear()` не несет никакой нудной работы, если, конечно, ваш текстовый редактор поддерживает команды поиска и замены. Так же просто можно определить и обратное присваивание: `A& A::operator=(TableA&)`.

Надеюсь, что после приведенных выше примеров вы по-новому посмотрите на роль макросов в C++.