

ПРОГРАММИРОВАНИЕ

ЯЗЫКИ
ПРОГРАММИРОВАНИЯ

на C++



Программирование на C++



Programming in C++
Stephen C. Dewhurst and Kathy T. Stark

AT&T

Bell Laboratories



Prentice Hall, Englewood Cliffs, New Jersey 07632

Программирование на C++

Стефан Дьюхарст, Кэти Старк

Перевод с английского
В. А. Кравчук



Киев НИПФ «ДиаСофт» 1993

Стефан Дьюхарст, Кэти Старк

П 47

Программирование на C++. Пер. с англ. -
Киев: «ДиаСофт», 1993. - 272 с., ил.

ISBN 5-87458-441-2

Эта книга написана для студентов и профессиональных программистов, которые хотят больше узнать о языке объектно-ориентированного программирования C++. Она будет полезна тем, кто хорошо относится к С. Авторы описывают особенности C++, а также современные парадигмы программирования - абстракцию данных и объектно-ориентированное программирование.

Ряд уникальных особенностей, имеющихсся в книге, как, например, новые взгляды на процесс программирования, детальное описание современных парадигм абстракции данных и объектно-ориентированного программирования, практические обсуждения проблем наследования, повторного использования кода и эффективного построения библиотек, позволяет ее считать в своем роде уникальным учебником по объектно-ориентированному программированию в широком понимании этого термина.

Для более надежного усвоения материала в конце глав помещены упражнения; решения некоторых упражнений приведены в приложении.

Для специалистов в области программирования, студентов и аспирантов ВУЗов.

ББК 32.971

ISBN 013723156-3 (англ.)

© 1989 by AT&T

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

ISBN 5-87458-441-2 (русск.)

© НИПФ «ДиаСофт», 1993,

обработка и оформление

© В.А. Кравчук, 1993,

перевод на русский язык

© С.В. Мальцев, 1993,

иллюстрации

От издателей

Глубокоуважаемые читатели!

Перед Вами одна из первых книг, выпущенных издательским отделом молодой научно-исследовательской проектной фирмы ДиаСофт.

У читателя, возможно, возникает вопрос: почему опять C++, литературы по этому языку уже предостаточно. Чтобы самостоятельно ответить на этот вопрос, мы рекомендуем хотя бы бегло ознакомиться с книгой, которую трудно не назвать замечательной в плане того, как легко и доступно, от простого к сложному, иллюстрируя примерами, авторы описывают процесс освоения и программирования на одном из ведущих языков. Несмотря на относительную сложность C++ как языка программирования, в интерпретации Стефана Дьюхарста и Кэти Старк C++ выглядит гораздо проще.

На вопрос читателей о соблюдении фирмой авторских прав издательства Prentice-Hall, Inc. ответить гораздо сложнее. После длительных и весьма бурных обсуждений коллектив нашей фирмы пришел к единому мнению: наши читатели не должны испытывать информационного голода. Со страниц изданной нами книги фирма ДиаСофт приносит свои искренние извинения за публикацию издания Prentice-Hall без их разрешения. Авторские права Prentice-Hall мы сохранили соответствующей надписью, хотя может быть это ничтожная компенсация ущерба, нанесенного интересам Prentice-Hall.

Мы готовы рассмотреть вопрос о сотрудничестве с ведущими отечественными и иностранными издательствами на территории Украины с целью популяризации их изданий на зарождающемся здесь рынке интеллектуального труда.

Пользуясь возможностью, которую Вы нам предоставили тем, что любезно согласились прочитать это предисловие, мы хотим проинформировать Вас о том, что основными видами деятельности нашей фирмы являются издательство научно-технической литературы по программированию и производство программных продуктов, в том числе и инструментальных средств для программистов.

Мы надеемся на Вашу помощь при формировании наших тематических планов.

Мы хотим с Вами сотрудничать по следующим вопросам:

- информация о рынке программных средств и печатной продукции;
- информационные материалы для издательской деятельности;
- перспективные идеи и разработки;
- оптовые закупки нашей продукции;
- оказание полиграфических услуг;
- организация информационного обмена на регулярной основе.

Всех заинтересованных лиц, которые оказывали бы неоценимую помощь по обеспечению нас информацией или оказывали другие услуги, мы могли бы поощрять различными удобными для них способами.

Обращайтесь к нам по адресу: 252055, Украина, г. Киев-55, а/я 100, фирма ДиаСофт. Тел./факс: (044)-441-9766.

Коллектив фирмы ДиаСофт

ПРЕДИСЛОВИЕ ПЕРЕВОДЧИКА.

Язык программирования C++ стал одним из наиболее популярных инструментов системных и прикладных программистов не только "на Западе", но и в странах СНГ. Благодаря непрерывным усилиям фирм Borland, Zortech и других по созданию все новых версий компиляторов C++ для операционной системы MS DOS, язык стал доступен широкому кругу пользователей. Но, если те или иные версии языка C++ легко доступны (в том числе и легальные с фирменной документацией), то литературы на русском языке, объясняющей как и для чего можно использовать C++, крайне мало. Предлагаемая книга, которая не является просто справочником или описанием языка, я надеюсь, поможет читателю понять, какие преимущества он может получить при использовании C++ и как правильно использовать новые выразительные возможности языка.

Изложение материала ведется на основе многочисленных примеров, как ставших уже классическими при рассмотрении языка (комплексные числа, графическая библиотека), так и более интересных для профессиональных программистов (драйверы устройств, элементы трансляторов, имитационное моделирование). Авторы, обладающие большим опытом использования языка и создания трансляторов для C++, дают примеры, достаточно близкие к реальным задачам. Примеры применения новых (по сравнению с языком C) механизмов управления памятью и некоторых программистских трюков, помогут читателю создавать эффективные по скорости выполнения и памяти приложения.

В отличие от других книг, посвященных языку C++, авторы обращают особое внимание на объяснение новых парадигм программирования, которые поддерживаются языком. Правильное и уместное использование абстракции данных и объектно ориентированного подхода позволит получить при переходе на C++ качественный скачок эффективности работы программиста.

В конце каждой главы приводятся упражнения, решение которых позволит читателю не только глубже понять осо-

бенности программирования на C++, но и получить большой практический опыт и множество программ, которые могут послужить основой коммерческих продуктов. Приведенные в приложении решения некоторых упражнений также могут помочь внимательному читателю при практическом использовании C++.

Некоторые примеры (в частности, моделирование работы аэропорта и системы управления полетами) нельзя проверить на компьютерах с ОС MS DOS. Тут могут помочь только компиляторы C++ под ОС UNIX, в среде которой и развивался язык в момент написания книги. Для версий UNIX на персональных компьютерах, в частности, SCO UNIX System V ver. 3.2v2, такие компиляторы доступны.

Книга вышла в 1989 году. Тем не менее, она содержит (в отличие от различных изданий на русском языке книги Б.Страуструпа "Язык программирования C++", являющейся основным источником информации о C++ на просторах СНГ) обсуждение такой довольно новой черты языка, как множественное наследование. Что же касается шаблонов и обработки исключительных ситуаций, то основным источником информации о них пока является документация, предоставляемая вместе с компилятором Borland C++.

В книге, к сожалению, не содержалось списка литературы. Хотя для наших читателей он имел бы скорее теоретическое значение. Поэтому, я решил взять на себя смелость порекомендовать несколько изданий на русском языке, которые будут полезны изучающим C++. В первую очередь, это уже упоминавшаяся книга Б.Страуструпа "Программирование на C++", М., "Р и С", 1991. Хотя многие говорят, что ее трудно читать, и описываемая в ней версия C++ несколько устарела, в ней имеется множество интересных примеров, дополняющих предлагаемый здесь материал. Описание особенностей последних версий C++ фирмы Borland можно найти в справочнике И.М.Двоеглазова "Язык программирования C++. Справочное пособие", К., ЕВРОИндекс, 1993. Полное изложение объектно ориентированной методологии (анализа, проектирования, программирования, в том числе на C++) можно найти в фундаментальной монографии Г.Бу-

ча "Объектно-ориентированное проектирование с примерами применения", М., Конкорд, 1992. В ней содержатся многочисленные фрагменты программ на C++, а также полностью описана, начиная с этапа анализа до сопровождения и модификации, реализация на C++ системы регистрации ошибок в программных средствах.

В заключение хотелось бы выразить надежду, что предлагаемый перевод поможет читателям, программирующим на C++, повысить свой профессиональный уровень. Я также был бы благодарен читателям за пожелания и замечания по поводу предложенной терминологии языка C++, которые можно направлять электронной почтой по адресу: valera@galagan.ts.kiev.ua

Желаю Вам удачи

Валерий Кравчук

15 мая 1993 г.

ПРЕДИСЛОВИЕ.

Язык программирования С++ становится все более популярным. Это происходит не только из-за популярности его предшественника С, но и благодаря ориентации на абстракцию данных и наличию черт объектной ориентированности. Развитие С++ было отмечено активным диалогом сообщества пользователей, которые плодотворным обменом идеями помогли его формированию. В результате, язык С++ приобрел множество черт, которые поддерживают различные методы и приемы программирования.

Язык С++ лучше всего изучать, рассматривая каким образом взаимодействуют черты языка при написании программ. Как и большинство пользователей и реализаторов С++, во время становления языка, мы имеем необычную возможность увидеть, почему конкретные элементы языка были развиты и поучаствовать в их усовершенствовании через их применение и использование. В этой книге мы надеемся отразить наше понимание и оценку С++ как инструмента программирования.

Мы написали эту книгу в то время, когда С++ еще развивался. Наша презентация С++ - эта попытка избежать деталей, которые могут вызвать непонимание (противоречивое толкование) у пользователей различных версий языка, поэтому новые черты, такие, как множественное наследование и уточнения определения и реализации языка, могут сделать представляемый нами язык в чем-то отличающимся от доступной читателю реализации. Мы не пытались написать справочное руководство по языку или описание некоторой реализации. Мы попытались провести наших читателей между деталями к общим концепциям разработки программ, которые помогут программистам эффективно использовать С++.

Эта книга стала лучше благодаря вкладу, сделанному многими из тех, кто участвовал в развитии С++. Мы отмечаем значительный вклад, сделанный ими в развитие методологии и техники программирования на С++. Их мысли оказали на нас большое влияние, и многие их идеи отраже-

ны в этой книге. Том Каргилл (Tom Cargill), Джим Коплин (Jim Coplien), Кейт Торлен (Keith Gorlen), Билл Хопкинс (Bill Hopkins), Эндо Кениг (Andy Koenig), Джерри Шварц (Jerry Schwarz) и Джон Шопиро (Jon Shopiro) были главными из этих пионеров. Мы также хотим поблагодарить наших коллег Лору Ивз (Laura Eaves), Фила Брауна (Phil Brown) и Стан Липпман (Stan Lippman), которые работали вместе с нами над созданием компилятора C++; Сару Хевинз (Sarah Hewins), Боба Вилсона (Bob Wilson) и Вэйн Вулф (Wayne Wolf), которые просмотрели различные черновики книги и Джеффа Мони́на (Jeff Monin), который посвятил нам свое время и выступил в роли эксперта по многим ключевым вопросам. Джон Вэйт (John Wait) из Prentice-Hall направлял и воодушевлял этот проект от начала до конца.

Особо мы благодарны Брайану Кернигану (Brian Kernighan), который упорно пробирался среди различных вариантов книги, ведя нас от непонятности к ясности. То, что мы, возможно, не закончили это путешествие, произошло скорее из-за нашего упрямства, чем из-за его упорства и убежденности.

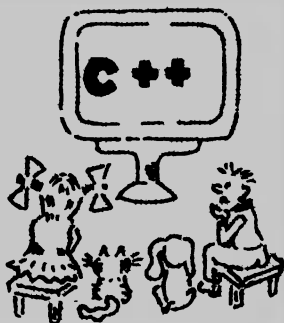
Бьерн Страуструп (Bjarne Stroustrup) не только создал C++ и превратил его в богатый язык программирования, но и хотел попутно его обсуждать. Мы благодарим Бьерна за многие содержательные дискуссии и обсуждения и за поддержку этого проекта.

S.C.D.

K.T.S.

Глава 0

ВВЕДЕНИЕ



C++ - это язык программирования общего назначения, происходящий от C. К своему родительскому языку он добавил многие черты, важнейшие из которых - это поддержка абстракции данных и объектно-ориентированного программирования. C++ обеспечивает основные типы данных, операции, синтаксис операторов и структуру программы языка C. Дополнительные черты расширяют C-подобные части языка, одновременно открывая возможности по использованию новых стилей программирования.

Языки программирования создаются не спонтанно, а в результате изменения представления людей о процессе программирования. Обобщенные и закодированные, эти мыслительные процессы достигают уровня парадигм, и новые языки создаются для их поддержки. Парадигмы программирования, таким образом, это модели, которые предлагают множество методов для использования при разработке и реализации программ. Эти методы содержат разделы о связи разработки программы с задачей, об использовании абстракции и организации программы. Язык поддерживает конкретную парадигму, если свойства языка облегчают использование соответствующих приемов. Свойства C++ обеспечивают несколько парадигм программирования.

Большинство книг по новым языкам программирования учит особенностям языка, но не дает читателю возможно-

стей самому освоить процесс программирования на этом языке. Для языка C++, который значительно богаче большинства языков, с которыми читатель уже, может быть, знаком, этот подход неудовлетворителен. Эта книга о том, как программировать, используя C++. Мы обсуждаем детали C++ параллельно с обсуждением использования парадигм для разработки и реализации программ.

Наша презентация развивает концепции путем обсуждения парадигм программирования и особенностей языка в контексте. Хотя мы и представляет основные особенности, мы не даем детального описания языка. Во многих примерах используются функции ОС UNIX без детального объяснения, но их использование достаточно очевидно, чтобы не затруднить понимание для читателя, незнакомого с системой UNIX. Мы предполагаем, что наши читатели - опытные программисты, знакомые с C, и наиболее детально представляем то, что им покажется новым при программировании на C++.

0.1. Язык C++.

Основная добавка, сделанная в C++ по сравнению с C - это введение классов. Классы позволяют пользователю определять сложные типы, которые включают не только элементы данных, но и функции, применимые к этому типу. Скрытие данных в классах дает механизм для абстракции данных. Наследование классов расширяет абстракцию данных до ООП. Определяемые пользователем операторные функции и преобразования облегчают интегрирование классов в предопределенную систему типов включением операндов классового типа в выражения и преобразованиями между классовыми и неклассовыми типами.

Классы включают особенности, позволяющие встроить управление памятью в динамические структуры данных. Язык также содержит обычные операторы выделения и освобождения памяти. Эта комбинация позволяет программисту специально приспособлять схему управления памятью к конкретной задаче.

Кроме возможностей, которые поддерживают приемы

построения структур данных, есть и возможности, расширяющие использование функций. Все объявления функций в C++ должны содержать информацию о типе аргумента. Это позволяет проверять тип аргумента и переопределять функции. Описания аргументов функции могут также содержать предопределенные значения для аргументов, отсутствующих в вызове функции. Ссылочные типы в C++ позволяют передавать аргументы не только по значению, но и по ссылке.

0.2. Парадигмы программирования.

Парадигмы программирования - это модели разработки и реализации программ. Разные модели приводят к разным приемам программирования. То, что приемы различны, не означает, что они противоречат друг другу; они могут быть взаимодополняющими. Общим для всех моделей является то, что разработка программы должна базироваться на абстракциях, отвечающих элементам задачи, и что реализация должна представлять собой совокупность модулей, предпочтительно таких, которые можно использовать и в дальнейшем. Они различаются в том, как формировать абстракции и что содержит модуль.

Хорошо развитые методы процедурного программирования базируются на модели построения программы, как совокупности функций. Приемы программирования объясняют как разрабатывать, организовывать и реализовывать функции, составляющие программу. Метод функциональной декомпозиции определяет функции как абстрактные операции, которые решают задачу программирования. Файловая организация позволяет группировать функции в отдельные модули, а приемы структурного программирования делают реализацию функций читабельной и понятной.

Абстракция данных концентрируется на структурах данных, которыми пренебрегают объектно-ориентированные приемы. Модель абстракции данных состоит в том, что структура данных определяется операциями над ней в боль-

шей степени, чем структурой ее реализации. Техника, используемая при абстракции данных, состоит в инкапсуляции структуры данных в абстрактный тип. Доступ к структуре возможен через набор операций, которые являются частью типа. Абстракция данных дополняет взгляд процедурного программирования на функции как на абстрактные операции, поскольку ни одна из этих абстракций не является полной без другой.

ООП базируется на модели построения программ как набора объектов абстрактного типа данных. При объектно ориентированной разработке определяются типы, которые представляют объекты представленной задачи. Операции в объектных типах, как и функции в процедурной модели, представляют собой абстрактные операции, решающие задачу. Объектный тип служит модулем, который может быть использован для решения другой задачи в той же области.

Ни одна из парадигм не позволяет хорошо решать все задачи. Программирование включает инженерную экспертизу, но это все же не наука. Приемы программирования необходимо применять гибко, учитывая насколько они подходят для решаемой задачи. Слепое применение наиболее популярной в настоящее время парадигмы никогда не заменит внимательного исследования и вдумчивого абстрагирования задачи. Основная цель этой книги - подвести читателя к критическому и гибкому подходу к задачам разработки и реализации программ.

0.3. Организация книги.

Мы намеревались научить читателя разрабатывать и писать программы на C++. Организация этой книги сочетает описание языка C++ с обсуждением приемов и парадигм, для которых были созданы те или иные возможности. Мы начинаем с описания типов данных и операций языка, прослеживаем способы организации структур данных и операций для программ и заканчиваем развернутым обсуждением вопросов управления памятью и построения библиотек.

Один из взглядов на программы состоит в том, что они описывают последовательность операций над данными, по-

этому в Главе 1 описаны типы данных и операций в C++. Классы, определяемые пользователем, рассматриваются как добавление соответствующих типов данных и операций к языку.

В Главе 2 обсуждаются приемы процедурного программирования для организации последовательностей операций в функциях. Функциональная декомпозиция и структурное программирование раскрываются как парадигмы разработки вместе с поддержкой таких черт языка, как видимость и соединение имен, передача параметров по значению и по ссылке, перекрытие имен функций, предопределенные аргументы и подставляемые функции.

Глава 3 охватывает основные особенности классов в C++. Это простое описание черт языка представляет собой основу, на которой базируется дальнейшее изложение. Мы обсуждаем компоненты данных и функциональные компоненты, конструкторы и деструкторы, переопределение операторов и защиту.

В Главе 4 обсуждается использование классов для абстракции данных. Мы показываем как развивать новый тип данных, используя механизмы, предоставляемые классами, чтобы сделать реализацию надежной и сопровождаемой, и как определять преобразования и операции, интегрирующие новый тип в существующую систему типов.

Глава 5 посвящена наследованию классов - свойству, позволяющему определять новый класс через уже существующие. Наследование может быть использовано для модификации существующего абстрактного типа данных, создания иерархии связанных абстрактных типов, комбинирования свойств несвязанных типов и обеспечивает гибкое связывание вызовов функций во время исполнения.

В Главе 6 обсуждается использование абстракции данных и наследования для ООП. Мы представляем способы разработки и организации программы из модулей объектно-го типа и демонстрируем динамический объектно-ориентированный стиль программирования.

Глава 7 представляет средства управления памятью в C++. Эффективные и понятные схемы управления памятью

могут быть разработаны и приспособлены как для общего использования, так и для данной библиотеки классов или специальных приложений.

В Главе 8 мы обсуждаем создание и использование библиотек как совокупностей необходимых программных модулей. Мы показываем способы создания библиотек, позволяющих пользователям расширять и модифицировать их без изменения исходного кода библиотек.

Последняя часть каждой главы содержит упражнения. Для тех, которые отмечены значком +, в Приложении даны решения.

Мы связываем детали языка C++ с более крупными проблемами разработки программного обеспечения. Мы надеемся, что наши читатели не только научатся эффективно использовать C++ как инструмент программирования, но и получат толчок к размышлению о новых путях проектирования и создания программ.

Глава 1

ТИПЫ ДАННЫХ И ОПЕРАЦИИ



Система типов C++ состоит из основных predefined языком типов, классов, определяемых пользователем, и типов, которые могут быть получены из основных типов и классов. Язык поддерживает операции и стандартные преобразования для встроенных типов.

Для классов также могут быть определены операции и преобразования, позволяющие использовать классы как содержательные расширения системы predefined типов.

Встроенные типы данных в C++ можно интерпретировать как абстрактные концепции, такие как числа или логические значения, или, в соответствии с их представлением в компьютере, как последовательность битов. Интерпретация зависит от операций, использующихся для работы со значениями различных типов.

Арифметические и логические операторы дают результаты, сохраняющие математическую и логическую интерпретацию типов. Другие операторы дают результаты, зависящие от побитового представления.

1.1. Числовые типы.

В C++ есть как целый, так и с плавающей точкой числовые типы. Язык предоставляет переопределяемые арифме-

тические операторы, которые работают с целыми числами и числами с плавающей точкой и определяют преобразования числовых типов. Для перегружаемых операторов один и тот же знак операции представляет более чем одну реализацию операции. Числовые типы в арифметических операторах интерпретируются как представления чисел.

Следующая программа выполняет расчеты с использованием переменных и значений числовых типов и арифметических операторов. Для представления целых использован тип `int`, для вещественных - `double`.

```
#include <stdio.h>
main() {
    /*
       Расстояние падающего объекта от
       точки начала падения
       в первые 10 секунд, в метрах
    */
    const double g = 9.80; //ускорение свободного
                           // падения

    for (int t = 1; t <= 10; t++) {
        double distance = g * t * t / 2;
        printf( "\t%2d %7.2f\n", t, distance );
    }
}
```

Эта программа печатает расстояние до падающего объекта в каждую из первых десяти секунд после начала падения:

1	4.90
2	19.60
3	44.10
4	78.40
5	122.50
6	176.40
7	240.10
8	313.60
9	396.90
10	490.00

В выражении, вычисляющем расстояние, смешаны опе-

панды типа `double` и `int`. Результат - значение типа `double` - запоминается в переменной `distance` типа `double`:

```
double distance = g * t * t / 2;
```

Хотя и первый, и второй операторы имеют тип `int`, вычисления будут выполняться с плавающей точкой. Так как для типа `double` операции умножения и деления выполняются слева направо, то каждый оператор в вычислении содержит один операнд типа `double`, что вызывает преобразование `int` операнда в `double`-операнд.

Арифметические операторы для числовых типов перегружаются. Типы операндов определяют, какая операция (с плавающей точкой или целого типа) будет выполнена. Если изменить выражение так, чтобы операции выполнялись в другом порядке, типы некоторых операций тоже могут измениться.

```
distance = g * ( t * t / 2 );
```

Если скобки группируют целые операнды, будет целое умножение типа на тип, результат которого имеет тип `int`. Деление также применяется к целым операндам. Целое деление дает результат целого типа; если `t` нечетное, дробная часть результата теряется, и это вычисление приведет к результату, отличающемуся от предыдущего, который был получен при вычислении с плавающей точкой. Использование литерала с плавающей точкой для делителя вызывает операцию деления с плавающей точкой, и мы получим результат с прежней точностью.

```
distance = g * ( t * t / 2.0 );
```

Для перегружаемого оператора, тип операнда определяет, какая из реализаций оператора будет использована для выполнения операции.

Арифметические операторы определены для всех числовых типов. Перечислим унарные операторы:

- инкремент ++
- декремент --
- изменение знака -
- неоперационный +.

Перечислим бинарные арифметические операции:

- умножение *
- деление /
- сложение +
- вычитание -.

Кроме того, оператор получения остатка %, применяется только к целым операндам.

Операторы инкремента и декремента изменяют значение объекта, являющегося их операндом по-разному в зависимости от того, с какой стороны от операнда они находятся. Они могут быть использованы как в префиксной, так и в постфиксной форме с различными результатами. В префиксной форме результатом выражения будет аргумент, увеличенный или уменьшенный на 1. В постфиксной форме значением выражения будет значение аргумента до его изменения. Другие арифметические операторы не изменяют значения своих операндов.

Унарные операторы имеют высший приоритет по сравнению с бинарными. Бинарные операторы типа умножения (*, /, %) имеют более высокий приоритет по сравнению со сложением и вычитанием.

Целые типы - это `char`, `short`, `int` и `long`. Каждый тип может представлять то же множество значений, что и предыдущий; каждый тип можно представлять как более широкий или эквивалентный предшествующему типу. Целые типы различают со знаком (`signed`) или без знака (`unsigned`). В объявлениях описатели `short`, `int` и `long` по умолчанию `signed`, в то время как `unsigned` необходимо указывать явно. Когда различные типы целых операндов в выражении комбинируются, они преобразуются в один тип. Операнды типа `char` и `short` обычно преобразуются в `int`. Другие преобразования обычно происходят к типу с более широким диапазоном области допустимых значений. Преобразования применяются к `unsigned`-версии типа только в

случаях, когда тип одного из операндов - `unsigned` и больший или равный по размеру типу другого операнда.

Типы с плавающей точкой - это `float`, `double` и `long double`. Каждый тип может представлять множество значений предыдущего типа в списке, и каждый можно рассматривать как тип, содержащий предшествующий тип. Если операнды арифметического выражения различных плавающих типов, операнд меньшего размера преобразуется в тип другого операнда. Если сочетаются операнды целого типа и с плавающей точкой, целый преобразуется в тип операнда с плавающей точкой.

Любое числовое значение может быть присвоено объекту любого другого числового типа; в этом случае значение преобразуется в тип переменной, находящейся в левой стороне оператора.

```
double d;  
d = 42;
```

Преобразование может привести к потере части значения.

```
int i;  
i = 3.1415;  
char c;  
c = 777;
```

В примере, представленном выше, `i` получит значение 3. Значение 777 слишком велико, чтобы быть представленным как `char`, поэтому значение `c` после выполнения оператора зависит от способа, которым конкретная реализация производит преобразование из большего в меньший целый тип.

Операторы преобразования или `cast`-ы (явные указания типов операндов) можно использовать для явного преобразования. Например, в следующем примере значение типа `int` переменной `count` явно преобразуется в `double`.

```
int count = 1066, total = 1337;  
double ratio;
```

```
ratio = double( count ) / total;
```

Этот пример использует преобразование типов в функциональном стиле:

```
double ( count )
```

Альтернативная форма такого же оператора:

```
( double ) count
```

Форма типа вызова функции обычно более понятна для преобразования простых выражений.

Числовые значения могут быть прямо представлены в программе как числовые литералы. Литералы с плавающей точкой могут иметь десятичную точку и дробную часть, а также могут быть записаны в виде, предусматривающем указание мантиссы и порядка:

```
9.80  
0.98e1  
98e-1
```

Все эти литералы представляют одно и то же значение и имеют тип `double`. Для значений различных типов с плавающей точкой, суффикс `L` или `l` обозначает `long double`, а `F` или `f` обозначает `float`.

```
0.98e1L  
9.80f
```

Целые литералы имеют тип `int`, если их значение не слишком большое или их тип не определен суффиксом.

Литерал, слишком большой для представления как `int`, может иметь тип `long`. Суффикс `L` или `l` указывает на `long double`, а `U` или `u` определяет `unsigned`. Эти суффиксы могут комбинироваться, например:

```
1642UL
```

Восьмиричный литерал определяется цифрой 0 в первой позиции:

```
0777
```

Шестнадцатиричный литерал определяется добавлением 0x или 0X:

```
0x1ff
```

Числовые значения также могут быть представлены символически при помощи задания константы с начальным значением.

```
const double g = 9.80
```

Так как `const` определяет, что значение `g` не может быть изменено, идентификатор всегда представляет значение 9.80.

Целые числа могут также быть представлены в символьном виде как элементы списка `enum` (перечислимый тип). Если нет других определений, элементы `enum` имеют последовательные целые значения (`int`), первое значение равно 0.

```
enum { mon, tues, wed, thur, fri, sat, sun };
```

Здесь `fri` имеет значение 4. Когда элементы `enum` явно инициализированы, неинициализированные элементы списка имеют значения на единицу больше, чем у предыдущего значения списка.

```
enum { mon = 1, tues, wed, thur, fri, sat = -1, sun };
```

В этом случае значение `fri` это 5, значение `sun` - 0.

Множество символов представляется целыми значениями, которые могут быть включены в данные типа `char`. Символьные литералы, которые представляют собой символы или `esc` последовательности в одинарных кавычках, дают

удобное представление символьных значений:

```
char digit = '9';  
char w = 'w';  
char newline = '\n';  
char tab = '\t';  
char null = '\0';
```

Значения символов всегда положительное, даже, если они типа `signed char`. Они могут быть использованы как целые операнды в выражениях:

```
int value = digit - '0';
```

Полезные вычисления над числовыми значениями символов обычно предполагают известные стандартные наборы символов, таких, как ASCII и EBCDIC.

1.2. Скалярные типы. Операторы сравнения и логические операторы.

В C++ нет логического типа. Скалярные типы, такие, как целые, работают по аналогии с булевыми, в которых ноль представляет значение FALSE, а любое ненулевое значение - TRUE. Указатели тоже имеют скалярный тип и описываются в этой главе позже. Операторы сравнения используются для сравнения значений и возвращают TRUE и FALSE в форме численного значения 1 или 0. Логические операторы работают со скалярными операторами, интерпретируемые как TRUE и FALSE и аналогично возвращают 1 или 0.

Операторы сравнения вырабатывают целый результат 1 или 0. Пример в предыдущем разделе, вычисляющий расстояние пойдённое при падении, содержит сравнение для управления циклом.

```
for (int t = 1; t <= 10; t++) {  
    // и т.д.  
}
```

Чтобы подчеркнуть управляющее условие, мы заменим цикл `for` эквивалентным циклом `while`.

```
int t = 1;
```

```
while (t <= 10) {  
    // и т.д.  
    t++;  
}
```

Тело цикла выполняется, пока условие $t \leq 10$ не обратится в 0. Выражение имеет результат 1 до тех пор, пока t меньше или равно 10; таким образом, цикл выполняется для значений t от 1 до 10.

К операциям сравнения относят:

- меньше <
- больше >
- меньше или равно <=
- больше или равно >=

Есть также операции равенства равно $=$ или не равно $!=$. Эти операторы - перегружаемые для целых чисел и чисел с плавающей точкой точно также, как и операнды указателя, однако всегда возвращают результат 1 или 0 типа `int`.

Так как любое ненулевое значение представляет `TRUE`, то любое выражение (а не только то, которое принимает значение 1 или 0 типа `int`) может использоваться как условие.

Например:

```
int t = 11;  
while (--t) {  
    // и т.д.  
}
```

Тело цикла выполняется для значений t от 10 до 1.

Перечислим логические операторы:

- AND &&
- OR ||
- NOT !

Выражение с унарным оператором `!` возвращает 0, если его операнд ненулевой и 1 в противном случае. Например, следующий фрагмент присваивает значение предварительно равной нулю переменной:

```
if (!p) //т.е. if (p == 0)
```

```
p = get_a_val();
```

Выражение, использующее `||`, равно 0, если оба операнда равны 0 и 1 иначе. В следующем примере функция `error` вызывается только в случае, если значение `x` не лежит в пределах от 0 до 10.

```
if (x < 0 || x > 10)
    error("value outside range");
```

Выражение `&&` равно 1, если оба операнда ненулевые и 0 в противном случае. В следующем примере то же условие, что и в предыдущем, использует другое логическое выражение:

```
if (!(x >= 0 && x <= 10))
    error("value outside range");
```

Первое выражение предпочтительнее, так как его легче понять.

Второй операнд операции `&&` или `||` вычисляется только, если это необходимо для определения результата. Если первый операнд выражения `||` ненулевой, результат равняется 1, независимо от второго операнда. Аналогично, если первый операнд `&&` равен 0, второй операнд не вычисляется и результат - 0. Например, ниже, если `b` имеет значение 0, равенство не проверяется и подвыражение деления пропускается.

```
if (b && a/b == c) {
    // и т.д.
}
```

При использовании `&&` проверяется, что `b` не 0 и, тем самым, исключается возможность деления на 0.

1.3. Неабстрактные операции.

В C++ есть несколько операций, позволяющих программисту игнорировать абстрактную интерпретацию типов, основываясь на их машинном представлении.

Целые типы реализуются в компьютере как последовательность битов различной длины. Когда целые используются в арифметических, логических операциях и операциях сравнения, их значения интерпретируются абстрактно как числа или как TRUE и FALSE, и детали битового представления программист может игнорировать. Тем не менее, иногда возникают задачи, когда программист желает работать с битами.

Ниже представлена хэш-функция для строк, предложенная Питером Вейнбергером. Хэш-функция вычисляет числовое значение из строки символов и используется для определения расположения в памяти ключевой информации строки. Параметр - это указатель на последовательность символов, которые используются как числовые значения при вычислении. С переменной hash обращаются как с 32-битовой последовательностью до тех пор, пока она не используется как числовое значение для взятия остатка от деления ее на prime.

```
int
hashpjw(char *s) {
    const prime = 211;
    unsigned hash = 0, g;
    for (char *p = s; *p; p++) {
        hash = (hash << 4) + *p;
        // предполагаем, что int имеет размер 32 бита
        if (g = hash & 0xf0000000) {
            hash ^= g >> 24;
            hash ^= g;
        }
    }
    return hash % prime;
}
```

Здесь использованы поразрядные операции: << - сдвиг влево и >> - сдвиг вправо, поразрядное И & и поразрядное исключающее ИЛИ. Имеются также поразрядное включа-

ющее ИЛИ `|` и унарное поразрядное дополнение `~`.

Оператор `^` в примере использован в форме оператора присваивания `^=`. Выражение

```
hash ^= g;
```

эквивалентно

```
hash = hash ^ g;
```

Бинарные операторы `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^` и `|` могут сочетаться с присваиванием таким же образом.

Оператор `sizeof` выдает число байт, необходимое для представления типа. Например, результат

```
sizeof( int )
```

определяет число байт, необходимое для представления `int`. Значение

```
sizeof( char )
```

всегда 1. Если операнд не тип, а выражение, результатом будет размер типа выражения в байтах. Этот оператор в основном используется для определения пространства, необходимого для динамического создания объекта. Он используется совместно с другими операторами управления памятью в C++, т. е. `new` и `delete`. При использовании этих операторов, берущих на себя заботу о выделении пространства, программисту редко нужно знать размер представления типа.

1.4. Типы, определяемые пользователем.

Для расширения базовой системы типов C++ пользователь может определять классы. Операции и преобразования можно определять для классов так, что их можно будет комбинировать с другими типами.

Программа в следующем примере вычисляет напряжение в цепи переменного тока, состоящей из проводников, резистора и конденсатора с использованием формулы $Z=R+j\omega L+1/(j\omega C)$ для импеданса и $V=ZI$ для напряжения. Напряжение, ток и импеданс цепи переменного тока имеют две составляющих, представляемые действительной и мнимой частью комплексного числа. В языке C++ нет предопределенного типа для комплексных чисел. Программист

использует для представления комплексных чисел определенный пользователем классовый тип.

```
#include "complex.h"

main() {
/*
    вычисление напряжения в цепи переменного тока
*/
    const complex j(0,1); // мнимая 1
    const double pi = 3.1415926535897931;

    double
        L = .03, // индуктивность в генри
        R = 5000, // сопротивление в омах
        C = .02, // емкость в фарадах
        freq = 60, // частота в герцах
        omega = 2 * pi * freq;
        // частота в радианах/сек
    complex
        I = 12,
        Z, // импеданс
        V; // напряжение

    Z = R + j * omega * L + 1/( j * omega * C );
    V = Z * I;
    V.print();
}
```

Программа выдает

(60000.00, 134.13)

Заголовочный файл `complex.h` содержит определение класса *complex*, реализующего математические действия с комплексными числами. Можно, например, использовать следующую сокращенную версию класса *complex*.

```
class complex {
    double re, im;
public:
    complex (double r = 0, double i = 0)
        { re = r; im = i; }
```

```
void print();  
friend complex operator +(complex, complex);  
friend complex operator *(complex, complex);  
friend complex operator /(complex, complex);  
};
```

Определение класса содержит объявления элементов и объявления *friend*-функций, которые имеют специальный доступ к элементам класса. Элементы, объявленные после метки *public* доступны без ограничений, тогда как приватные элементы *private* и *protected* доступны только через элементы и *friend*-функции. Элементы данных, представляющие *complex*, скрыты в приватной части класса, так что тип можно использовать только через открыто доступные функции.

Функциональный элемент, имеющий то же имя, что и класс носит название конструктора. Конструктор используется для создания и инициализации объектов типа *complex*, или для преобразования значений других типов в тип класса. Этот конструктор объявляется с предопределенными значениями аргументов, поэтому его можно вызывать без одного или обоих аргументов; в этом случае будут подставлены предопределенные значения.

Объявление

```
const complex j(0,1);
```

содержит инициализатор, предоставляющий значения обоим аргументам конструктора, действительная и мнимая части *j* получают соответственно значения 0 и 1. Объявление

```
complex l = 12;
```

соответственно эквивалентно

```
complex l(12,0);
```

Предопределенное значение 0 будет присвоено аргументу, представляющему мнимую часть *i*. *z* и *v* инициализированы предопределенными аргументами конструктора, так как никакие начальные значения в их объявлении не указаны.

Другие функции, объявленные в классе *complex*, определяются вне описания класса:

```

void
complex::print() {
    printf("( %5.2f, %5.2f)\n", re, im);
}

complex
operator +( complex a1, complex a2 ) {
    return complex( a1.re + a2.re, a1.im + a2.im );
}

complex
operator *( complex a1, complex a2 )
{
    return complex(a1.re * a2.re - a1.im * a2.im,
        a1.re * a2.im + a1.im * a2.re);
}

complex
operator /( complex a1, complex a2 )
{
    double r = a2.re; /* (r,i) */
    double i = a2.im;
    double ti;        /* (tr,ti) */
    double tr;
    tr = r < 0? -r : r;
    ti = i < 0? -i : i;
    if (tr <= ti) {
        ti = r/i;
        tr = i * (1 + ti*ti);
        r = a1.re;
        i = a1.im;
    }
    else {
        ti = -i/r;
        tr = r * (1 + ti*ti);
        r = -a1.im;
        i = a1.re;
    }
    return complex( (r*ti+i)/tr, (i*ti-r)/tr);
}

```

Элемент-функция `print` используется в примере для печати полученного в результате значения `V`.

```
V.print();
```

Как элемент-функция, `print` получает доступ к другим элементам класса без ограничений. Она форматирует и печатает элементы `re` и `im` объекта типа *complex*, к которому она применяется.

Операторные функции объявляются внутри класса *complex* как `friend` (дружественные). Дружественные функции не являются элементами класса, но как и элементы-функции они позволяют получить доступ к приватным элементам объекта типа *complex*. Операторные функции реализуют арифметические операции для значений типа *complex* и позволяют использовать операнды типа *complex* в выражениях в инфиксной нотации:

$$Z = R + j * \text{omega} * L + 1 / (j * \text{omega} * C);$$

Выражение, вычисляющее импеданс, содержит операнды типа `int`, `double` и *complex*. Когда операнды типа `int` или `double` используются вместе с *complex*, автоматически вызывается конструктор для преобразования операндов в *complex* до того, как они станут аргументами операторных функций. Для получения нужного типа аргументов конструктора используются predefined преобразования встроенных типов. Таким образом, один конструктор служит для преобразования как `int`, так и `double`-операндов. Когда для преобразования производится вызов конструктора *complex*, второй аргумент получает predefined значение.

Благодаря скрытию представления, операторам, определенным пользователем, и преобразованиям типов, класс *complex* представляет собой абстрактный числовой тип, который естественным образом сочетается с predefined числовыми типами.

1.5. Указатели и массивы.

Указатели и массивы являются производными типами. Указатели представляют адреса объектов других типов. Они используются для поддержки динамически размещаемых объектов, гибких структур данных и вместе с арифметиче-

скими операциями над указателями служат для доступа к элементам массивов. Массивы представляют собой последовательности элементов определенного типа и широко используются как сложные структуры данных. Одно из стандартных использований массивов - это строки, являющиеся последовательностями символов.

Указатели определяются по модификатору типа `*` вместе с другой информацией о типе в объявлении. Тот же символ используется для оператора разыменования, который возвращает объект, на который ссылается указатель. Результат разыменования можно использовать как значение объекта или в левой части оператора присваивания.

```
int *p;      // p - указатель на int
int i = 33;
p = &i;      // p присваивается указатель на i
*p = *p + 1; // i присваивается 34
```

Оператор `&` возвращает адрес операнда. Адресное значение имеет тип *указатель-на-объект-типа*.

Оператор динамического размещения `new` создает объект, тип которого определяется операндом и возвращает указатель на новый объект. Объект, созданный `new`, может быть уничтожен оператором `delete`, который интерпретирует свой операнд как указатель на сброшенный объект.

```
int *p = 0;

if (!p)
    p = new int;

delete p;
p = 0;
```

Этот фрагмент объявляет указатель и присваивает ему 0, получая специальный указатель `null` на несуществующий адрес. Для указателя производится проверка на предмет того, не был ли он установлен, и если нет, ему присваивается адрес вновь созданного объекта. Затем объект удаляется

и указатель устанавливается в `null`. Комбинация указателей, использование указателей `null` как флагов, а также размещение и уничтожение объектов операторами `new` и `delete` - это рудиментарные средства для построения динамических структур в C++.

Между указательными типами нет автоматических преобразований, за исключением некоторых случаев при присваивании и инициализации. Существует специальный тип указателя `void*`, который охватывает значения указателей любого типа. Можно представлять `void*` как тип *указатель на нечто*. Любой другой указательный тип автоматически преобразуется к типу `void*` при присваивании или инициализации.

```
int *ip;
void *vp = ip;
double *dp;
vp = dp;
ip = (int *)vp; // не лучшая идея
```

`void*` преобразуется в указатель другого типа, если программист явно требует это при помощи преобразования. Такое явное преобразование рискованно, так как игнорируется проверка типов, гарантирующая, что указываемый объект интерпретируется надлежащим образом. Автоматические преобразования указателей реализованы для связанных классов. Они обсуждаются в Главе 5.

Массив - это последовательность подряд идущих элементов одного типа. Адреса элементов массива относительно других элементов могут быть вычислены с помощью арифметических операций, перегруженных для операндов типа указателей. Цикл, перебирающий элементы строки, демонстрирует использование арифметики для указателей.

```
for (char *p = s; *p; p++) {
    // и т.д.
```

В этом примере `s` - массив элементов `char`, представляющий строку. Указатель `p` изначально указывает на первый

элемент массива и затем увеличивается, последовательно указывая на элементы, пока среди них не встретится 0. Обычно строка заканчивается элементом 0.

Операторы инкремента ++ и декремента -- работают с указателями, представляющими адреса элементов массива. Инкремент изменяет указатель так, что он ссылается на следующий элемент массива, декремент переадресовывает указатель на предыдущий элемент.

Бинарные сложение + и вычитание - требуют, чтобы один операнд был указателем, а и другой - целым числом. При вычитании целым должен быть второй операнд. Здесь снова предполагается, что указатели являются адресами элементов массива. Результатом операций будут адреса других элементов. Например, следующее сложение приводит к тому, что `p` указывает на третий элемент после элемента, на который он указывал сначала.

```
p += 3;
```

Следующее вычитание устанавливает `p` на второй элемент до исходного.

```
p -= 2;
```

Вычитание также используется с двумя операндами указателями, являющимися адресами элементов одного массива. Результатом будет значение типа `int` - число элементов массива между указателями.

Массив определяется по модификатору типа []. В объявлении массива в скобках содержится число элементов массива, например:

```
char buffer[100];
```

объявляет массив из 100 элементов типа `char`. Имя `buffer` содержит адрес первого элемента массива. Это имя может быть использовано в операциях арифметики указателей для доступа к элементам массива. Следующий цикл обнуляет элементы `buffer`.

```
for (int i=0; i<100; i++)  
    *(buffer + i) = 0;
```

Выражение с указателями может быть использовано для доступа к элементам массива. В следующем примере `p` изначально указывает на адрес первого элемента `buffer`, а затем используется для копирования первых двадцати элементов в другой массив `name`.

```
char name[21];  
  
p = buffer;  
for (i = 0; i<20; i++)  
    name[i] = *p++;
```

Оператор индексации `[]` дает более короткое выражение для операций с указателями, используемых для доступа к элементам массива. Индексное выражение

```
name[i]
```

эквивалентно

```
*(name+i)
```

Оператор индексации может быть применен к любому указателю, а не только к именам массивов.

Следующая строка устанавливает элемент, находящийся перед `p` в значение `null`.

```
p[-1] = '\0';
```

Простое представление строки в C++ - это последовательность символьных значений в массиве `char`, завершающаяся элементом 0. Строчные литералы как раз и представляют собой такие массивы. Строчный литерал - это последовательность символов или `esc`-последовательности в двойных кавычках:

"Это строчный литерал\n."

Значение строчного литерала - указатель на первый элемент массива `char`, размер которого на единицу больше числа символов между кавычками. Элементы массива - символы, за которыми следует значение 0.

1.6. Ссылки.

Ссылочные типы устанавливают псевдонимы объектов. Они используются как типы параметров функций для передачи аргументов по ссылке.

Для описания ссылочного типа используется модификатор `&` аналогично модификатору `*` для указателей. Ссылка должна быть инициализирована. После инициализации использование ссылки дает тот же результат, что и прямое использование переименованного объекта. Ссылки в основном используются как параметры. Чтобы показать, как работают ссылки, сначала рассмотрим их непараметрические объявления.

Для установления псевдонима инициализатор должен быть именем объекта того типа, на который ссылаются.

```
int i;  
int &ir = i;
```

Эта строка устанавливает для `i` псевдоним `ir`. Присваивание `ir` значения и его использование даст тот же результат, что и присваивание значения и использование `i`.

```
ir = 3    // i получает значение 3  
int j;  
int *lp;  
j = i * ir; // j получает значение 9  
lp = &ir; // lp присваивается адрес i
```

Объект, псевдонимом которого является ссылка после инициализации, не может быть изменен в дальнейшем.

Если тип инициализированной ссылки не совпадает с типом объекта, создается анонимный объект, для которого ссылка является псевдонимом. Инициализатор преобразу-

ется и его значение используется для установки значения анонимного объекта.

```
double d;  
int &ir = d; // создан анонимный объект типа int  
ir = 3.0;    // d не изменилось!
```

Анонимный объект создается также, когда инициализатор не является объектом.

```
int &ir = 3; // анонимный объект получил  
           // значение 3
```

Параметры ссылочного типа в основном используются для того, чтобы функция могла устанавливать значения своим фактическим параметрам. В этом случае ссылки используются для установки псевдонимов аргументов функции.

```
void input(int &,int &,int &);  
int a,b,c;
```

```
input(a,b,c) //устанавливает значения аргументов
```

Параметры-указатели также могут быть использованы для изменения внешних по отношению к функции параметров, но тогда для манипулирования аргументами и параметрами необходимо использовать операции над адресами и указателями. Установка псевдонимов при помощи ссылочных параметров - это удобная альтернатива.

Ссылочные параметры также используются, чтобы избежать наложения значения аргументов на инициализированные параметры. Это особенно важно для параметров классов, где значения аргументов могут быть в действительности гораздо большими структурами данных.

```
class list {  
    //список,содержащий до 100 элементов типа int  
public:  
    int size;
```

```
int elements[100];
};

void output(list &); //не копирует значение list
```

В этом примере `output` будет использовать аргумент `list` напрямую, без копирования его значения, избегая накладных расходов по воспроизведению массива элементов. Создание анонимных объектов при инициализации ссылок делает возможными преобразования, необходимые для отождествления аргументов и ссылочных параметров. Например, мы можем изменить наши операторные функции для *complex* так, чтобы они не копировали свои аргументы, допуская преобразования для других числовых аргументов:

```
class complex {
    // и т.д.
    friend complex operator *(complex &,complex &);
};

complex V,Z;
// и т.д.

V = Z * 12;
```

Выражение `Z * 12` - это вызов операторной функции `*` из *complex* с аргументами `z` и анонимным объектом типа *complex* со значением 12. Эти объекты инициализируют ссылочные параметры, и наложение аргументов происходит без предварительных преобразований в выражениях смешанного типа.

1.7. Константы.

Квалификатор типа `const` означает, что объект этого типа не может изменять своего значения ни напрямую, ни через указатель на него. Как обычно подразумевается, это свойство делает `const` символьным обозначением значения.

Ссылочные параметры, используемые для того, чтобы избежать копирования аргументов, и не предназначенные

для изменения аргументы также можно объявить как `const`, чтобы это предотвратить:

```
complex operator *(const complex &,
                   const complex &);
```

Здесь `const` защищает фактические параметры от изменения внутри функции.

В объявлении указателя позиция `const` определяет, что указатель или указываемый объект не должен изменяться. Квалификатор перед модификатором указателя означает, что указываемый объект `const`:

```
const char *p = buffer; //указатель на константу
                        // типа char
p = name; // все нормально
*p = 'x' // ошибка!
```

Квалификатор после модификатора показывает, что сам указатель не может быть изменен:

```
char * const p = buffer; //постоянный указатель
                        // на char
*p = 'X'; // все нормально
p = name; // ошибка!
```

Адрес объекта, объявленного как `const`, не может быть присвоен не-`const`-указателю, так как такое присваивание может изменить константу через указатель.

```
const char space = ' ';
const char *p = &space; // все нормально
char *q = &space; // ошибка!
```

1.8. Упражнения.

Упр. 1-1. Напишите на C++ программу, определяющую сколько бит используется для представления

объектов типа `char`, `short`, `int` и `long`.

Упр. 1-2. Следующая программа выдает:

a 3 b 2 c 2

Объясните, почему.

```
#include <stdio.h>

int f(int i) { return ++i; }
int g(int &i) { return ++i; }
int h(char &i) { return ++i; }

main() {
    int a = 0, b = 0, c = 0;
    a += f(g(a));
    b += g(f(b));
    c += f(h(c));
    printf("a %d b %d c %d\n", a, b, c);
}
```

Упр. 1-3.+ Учитывая объявления

```
int a = 12;
int b;
int &c = b;
int d[5];
```

какие типы будут иметь следующие выражения?

```
a
b
*b
c
*c
d[2]
*d
**d
c[-1]
c-2
*(c-1)
&c
```

Упр. 1-4.+ Учитывая объявления

```
char c;  
const char cc = 'a';  
char *pc  
const char *pcc;  
char *const cpc = &c;  
const char *const cpcc = &cc;  
char *const *pcpc;
```

какие из присваиваний правильные и какие нет, и почему?

```
c = cc;  
cc = c;  
pcc = &c;  
pcc = &cc  
pc = &c;  
pc = &cc;  
pc = pcc;  
pc = cpc;  
pc = cpcc;  
cpc = pc;  
*cpc = *pc;  
pc = *pcpc;  
**pcpc = *pc;  
*pc = **pcpc;
```

Упр. 1-5. Объясните семантику каждой из следующих функций:

```
void  
swap1(int *a, int *b) {  
    int t = *a;  
    *a = *b;  
    *b = t;  
}  
void  
swap2(int &a, int &b) {  
    int t = a;  
    a = b;  
    b = t;  
}
```

```
void  
flop(int a, int b) {  
    int t;  
    t = a;  
    a = b;  
    b = t;  
}
```

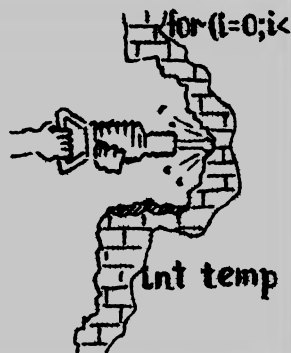
Упр. 1-6. Запустите следующую программу.

```
#include <stdio.h>  
  
int &  
f() {  
    int i = 1;  
    return i;  
}  
  
int  
g() {  
    int j = 2;  
    return j;  
}  
  
main() {  
    int &ri = f();  
    g();  
    printf("%d\n", ri);  
}
```

Что она выдаст в вашей системе? Объясните, почему.

Глава 2

ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ



Программа - это последовательность операций над структурами данных, которые реализуют алгоритм или процедуру решения задачи. Для большинства задач эта процедура достаточно длинная и сложная, в связи с этим программу сложно написать и дорого сопровождать, не используя методов управления ее размером и сложностью. Приемы процедурного программирования предлагают методы для разделения и структурирования программ, для облегчения их создания, понимания и сопровождения. Эти приемы концентрируются на организации последовательности операций в программе и большей частью игнорируют структуры данных, над которыми выполняются операции.

В C++ программа начинает выполняться с функции `main` после инициализации статических структур данных. Практически никогда `main` не содержит все операции программы, часть всегда передается другим функциям или встраивается в структуры данных, представленные в Главе 3.

В этой Главе мы используем только простые формы структур данных и раскрываем приемы процедурного программирования для разработки и организации функций в программе на C++.

2.1. Функции как модули.

Функция - это модуль, инкапсулирующий последовательность операций. Функции имеют параметры, поэтому их операции обобщены для применения к любым фактическим аргументам соответствующего типа. Входными данными для функции являются аргументы и глобальные объекты данных, используемые функцией. В результате получаем возвращаемое функцией значение, модификации, произведенные через указательные и ссылочные аргументы, и изменения глобальных данных.

Входные данные и результаты образуют интерфейс функционального модуля. Чтобы использовать функцию, пользователю необходимо понять этот интерфейс. Фактическая последовательность операций, реализующих функцию, скрыта, поэтому функция может рассматриваться как единая абстрактная операция. Разработка и реализация функций в программе может рассматриваться как построение операций, решающих данную задачу.

Программы должны не только удовлетворять технической цели корректного решения задачи, но и быть экономически обоснованными. Модульность функций можно использовать для придания им большей ясности и возможности повторного использования, и таким образом, помогает сократить расходы по ее реализации и сопровождению.

Функции можно легко понять, если они соответствуют абстрактным операциям, необходимым для решения задачи. Функцию и ее использование в программе можно в таком случае представлять в терминах задачи, а не в деталях реализации. Например, функция, получающая на входе несортированный список и переставляющая элементы в определенном порядке, может быть описана как абстрактная операция "*сортировка списка*". Одна и та же абстрактная операция может быть частью решений многих задач. Функция, реализующая операцию, также может быть использована во многих программах, если она создана как абстракция, не зависящая от контекста программы.

Функция, не использующая глобальные данные, параметризуется входными параметрами. Вместо того, чтобы

оперировать конкретными объектами в программе, функция является операцией над любыми аргументами соответствующего типа. Функцию можно затем использовать с различными параметрами не только в исходной программе, но и в других программах со структурами данных того же типа. Интерфейс понятен из объявления, а объекты данных, описанные в его реализации, можно понять из локальных объявлений функции. При параметризации входа и локализации описаний функция представляет собой тип самодокументированного модуля, который легко повторно использовать.

2.2. Функциональная декомпозиция.

Функциональная декомпозиция - это метод разделения большой программы на функции. Начиная с общего описания того, что должна делать программа, метод разделяет действие на несколько шагов или абстрактных операций.

Каждый шаг реализуется в программе как функция. Любая функция также может быть разделена на подшаги, реализованные функциями. На конечном уровне детализации шаги могут быть реализованы без применения функций.

Результатом этого метода конструирования является иерархия функций, в которой функции более высокого уровня предоставляют работу функциям нижнего уровня. Этот метод также известен как разработка сверху вниз, так как она начинается с высокоуровневого описания программы, опускаясь затем до все более мелких деталей реализации.

В качестве примера рассмотрим программу, сортирующую список слов. Сначала нам необходимо представить в программе список и слова. Структура данных для слов - это строка символов (заканчивающаяся нулем последовательность символов, доступных через указатели). Список представляется массивом указателей на строки. Этот массив имеет максимальный размер, определяющий сколько строк может быть введено в массив.

Мы дадим типу `char *` имя *String*, используя объявление `typedef`, затем создадим список и инициализируем его как

пустой.

```
typedef char *String,  
  
const int max = 100;  
String list[max];  
int size = 0;
```

Теперь мы готовы к разработке функций в нашей программе. Задача состоит в том, чтобы взять список и выдать отсортированный список. На высшем уровне, ее можно разделить на три операции:

```
прочитать исходный список  
отсортировать список  
распечатать отсортированный список
```

Мы получаем три функции, которые вызываются из функции `main` нашей программы:

```
int input(String *,int);  
void sort(String *,int);  
void output(String *,int);  
  
main() {  
    size = input(list,max);  
    sort(list,size);  
    output(list,size);  
}
```

`input` читает строки до заполнения списка, вставляет их в список и возвращает размер списка; `sort` упорядочивает элементы списка; и `output` печатает каждую строку списка.

Заметим, что массив и информация о размере будут параметрами функций. В этой программе функции используются один раз и могут получить доступ к глобальным данным напрямую, не передавая их как параметры. В этой программе польза от параметризации функций состоит в облегчении понимания и повторного использования функций в других программах.

Функции ввода, сортировки и вывода - простые, за исключением чтения и печати строк; для сортировки необходимо обеспечить сравнение строк. Мы передаем эти задачи функциям нижнего уровня и обсудим их позже.

```

Int readString(String &);
void printString(String);
Int lessthan(String,String);

Int
input(String *a,Int limit) {
    for (Int i=0; i<<limit; i++)
        if (!readString(a[i]))
            break;
    return i;
}

void
output(String *a,Int size) {
    for (Int i=0; i<<size; i++)
        printString(a[i]);
}

void
sort(String *a,Int n) {
    //используем сортировку методом пузырька
    Int changed;
    do {
        changed=0;
        for (Int i=0; i<<n-1; i++)
            if (lessthan(a[i+1],a[i])) {
                String temp=a[i];
                a[i]=a[i+1];
                a[i+1]=temp;
                changed=1;
            }
    } while(changed);
}

```

Если строка прочитана, readstring возвращает 1, иначе - 0. Функция lessthen возвращает 1, если первый аргумент лексически меньше второго и 0 - в противном случае.

В конечном итоге мы пришли к функциям, напрямую

работающими со строками. Необходимые нам функции есть в стандартных библиотеках.

```
#include <stdio.h>
#include <string.h>

int
readString(String &s) {
    // читает строку и копирует ее
    // добавляя место для заключительного null

    const bufsize=100;
    static char buffer[bufsize];

    if (scanf("%s",buffer) == EOF)
        return 0;
    s=new char[strlen(buffer)+1];
    if (!s) // new не удалось найти достаточно памяти
        return 0;
    strcpy(s,buffer);
    return 1;
}

void
printString(String s) {
    printf("%s",s);
}

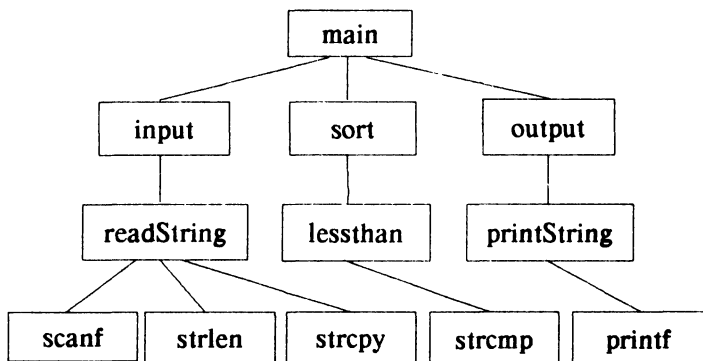
int
lessthan(String s1,String s2) {
    return strcmp(s1,s2)<0;
}
```

Функция `readstring` имеет ссылочный аргумент, так как она должна присваивать своему аргументу значение вновь прочитанной строки. Если вместо `String &` использовать аргумент типа `String`, присваивание `s` установит только его локальное значение, не изменяя значения фактического параметра функции. Ссылочный тип позволяет изменять фактический параметр внутри функции через псевдоним.

Функции `scanf` и `printf` производят ввод и вывод строки и объявлены в стандартном библиотечном файле `stdio.h`. Кро-

ме того, если достигнут конец файла, `scanf` возвращает EOF, а `strlen` содержит число символов в строке, исключая завершающий `'\0'`. Наконец, `strcpy` копирует одну строку в другую, а `strcmp` возвращает 0, если строки состоят из одних и тех же символов, или положительное (отрицательное) значение в зависимости от того, предшествует ли лексикографически первый аргумент второму или наоборот. Эти функции объявлены в `string.h`.

Разработка этой программы сверху-вниз представлена иерархической диаграммой:



2.3. Файловая организация.

Единицей компиляции в C++ является файл. Любое число файлов в C++ может быть откомпилировано и связано для функций и данных. Обычный способ доступа к внешним определениям в файле состоит в создании отдельного файла заголовка, содержащего внешние объявления и поддерживающего необходимые для корректного использования определения типов. Объявления заголовочного файла, таким образом, включаются в любой файл, использующий внешние функции или объекты. Функции и глобальные объекты, объявленные как `static`, внутренне связаны и на них нельзя ссылаться извне файла, в которых они объявлены. Глобаль-

ные объекты, объявленные как `const`, также внутренне связаны, если они явно не объявлены как `extern`.

Разделение большой программы на файлы может облегчить процессы развития, понимания и сопровождения программ. Например, над одной программой могут работать несколько программистов, не мешая друг другу; связанные функции и структуры могут группироваться для облегчения их чтения и понимания и локальные модификации могут иметь ограниченный внешний эффект.

Код, разделенный на файлы, легче перерабатывать для повторного использования в другой программе. Это обычный способ повторного использования кода, например, в библиотеках. Мы же показывали использование стандартных библиотек ввода/вывода и для работы со строками. Функции, объявленные в `stdio.h` и `string.h`, представлены в общедоступных файлах. Код пользователя подлинковывается к этим библиотекам для формирования полной программы.

Так как в нашем примере мы создали обобщенные функции, мы реорганизуем нашу программу сортировки списка и выделим функции в файл, который может быть использован другими программами.

Функцию `main` мы поместим в файл `main.c`. `.c` - это расширение имени файла, которое мы используем по договоренности для обозначения того, что он содержит текст программы на C++. Расширения имени файла различны в разных реализациях C++.

```
#include "strarray.h"
```

```
const int max=100;  
String llist[max];
```

```
int size=0;
```

```
main() {  
    size=input(list,max);  
    sort(list,size);  
    output(list,size);  
}
```

При компиляции main.cpp содержимое файла strarray.h вставляется в текст директивой `#include`. strarray.h содержит необходимые для функций объявления над строками-массивами:

```
typedef char *String;

int Input(String *,int);
void sort(String *,int);
void output(String *,int);
```

Определения функций, объявленных в strarray.h подставляются в strarray.c. При переупорядочивании наших функций, мы также можем избавиться от printing, так как она ничего не добавляет к операции, выполняемой printf.

```
#include <<stdio.h>
#include <<string.h>
#include "strarray.h"

static int
readString(String &s){
    // ничего нового
}

static int
lessthan(String s1,String s2){
    // ничего нового
}

int
Input(String *a,int limit){
    // ничего нового
}

void
output(String *a, int size){
    for (int i=0;i<<size;i++)
        printf("%s",a[i]);
}

void
sort(String *a,int n){
    // ничего нового
}
```

Вместе со стандартными заголовками этот определяет библиотечные функции, используемые в реализации; `strarray.c` включает интерфейсный заголовок `strarray.h`. Это сделано для получения необходимого определения типа *String* и для постоянного согласования описаний и реализаций. Если объявление в заголовке не согласуется с определениями функций, исходный файл не компилируется.

Следует отметить, что объявление вспомогательных функций `readString` и `lessThan` как `static` указывает, что они имеют внутреннюю линковку, и к ним невозможен доступ из других файлов. Так как их использование ограничено этим файлом, их можно изменить и даже удалить в соответствии с потребностями в модификации функций этого файла, не влияя на внешних пользователей.

2.4. Структурное программирование.

Использование приемов структурного программирования помогает сделать реализацию функций более понятной. Методы этой технологии включают использование управляющих структур, разделение функций на блоки, форматирование и сопровождение кода комментариями.

Управляющие операторы позволяют реализовывать ветвления и циклы так, что последовательность управления в функции можно легко проследить. Операторы придают управляющим структурам правильные, легко различимые формы. Управляющие операторы C++ - это ветвления: `if`, `if-else`, `switch`; и циклы: `for`, `while`, `do-while`. Есть также несколько операторов перехода: `break` - для выхода из циклов и операторов выбора; `continue` - для перехода к следующей итерации цикла; и "позорный" `goto`. Так как последний оператор позволяет перейти в любое место внутри функции, он (`goto`) может усложнить прослеживание и понимание операций. При любом использовании `goto` необходимо проверять, нельзя ли то же самое закодировать при помощи операторов ветвления или цикла.

Переходы могут вызвать проблемы не только потому, что затемняют последовательность выполнения. Нельзя перепрыгнуть через инициализацию внутрь объявления, ис-

пользуя `goto`:

```
goto label;  
int i=3; // Ошибка!  
label:  
i++;
```

так и в управляющей структуре:

```
switch(x){  
  int i=0; // Неверно!  
  case 1: {  
    int j=1; // Все нормально.  
    break;  
  }  
  case 2:  
    // и т.д.  
  default:  
    break;  
}
```

Единственный корректный способ перепрыгнуть через объявление с инициализатором - это полностью его пропустить.

Все функции содержат тело и объявления параметров должны содержаться в этом блоке. Блоки могут входить в другие блоки.

Чаще блоки используются для группировки операторов в управляющих структурах. Они также могут использоваться для локализации имен, которые используются ограничено. Например, в функции `sort` переменная `temp` объявлена внутри блока, в котором используется.

```
if (lessthan(a[i+1].a[i])) {  
  String temp=a[i];  
  // и т.д.  
}
```

Локальная переменная инициализируется, когда поток управления проходит ее точку объявления и прекращает

существование, когда управление покидает блок, немедленно закрывая объявление.

Если объявить `temp` во внешнем блоке, из контекста описания сложнее будет понять, зачем она нужна, и там нельзя ее полезно инициализировать. Помещение объявления внутрь блока позволяет также использовать общие, мнемонические имена в других контекстах.

Объявления являются операторами и могут появиться в любом возможном для оператора месте блока. Эта гибкость позволяет объявлять и инициализировать имена в точке их первого использования. Например, первой клаузой оператора цикла `for` является оператор, поэтому здесь можно объявить индекс цикла.

```
for (int i=0; i<<limit; i++) {  
    // и т.д.  
}  
return i;
```

Область действия имени распространяется от точки объявления до конца блока. В нашем примере `i` доступно и после завершения цикла. Она становится недоступной после конца блока, охватывающего оператор `for`.

Договоры о форматировании важны для улучшения читабельности функции. Наиболее важным для отображения управляющей структуры функции является использование отступов для выделения вложенности операторов и блоков. Также необходимо легко находить в тексте общие секции. Например, объявления, начало и конец функциональных блоков, метки должны быть явно видимы. Секции кода могут выделяться разными уровнями отступа, окружением пробелами и помещением в начало строки или блока. Для выделения секций можно использовать `/*` и комментарии.

Для отделения комментариев от кода в C++ есть два способа. Блок комментариев начинается с `/*` и заканчивается `*/`, а строчный комментарий начинается с `//` и продолжается до конца строки. Комментарии можно использовать для обзорного объяснения действий функций или сложных секций кода. Они также могут объяснять операторы с тонким

или скрытым смыслом. Например, необходимо выделять скрытую взаимосвязь операций и предположения, на которых основана реализация.

```
/* Глобальная переменная flag должна быть  
   установлена перед вызовом fuuc1, так как  
   она устанавливает контекст для  
   дальнейшего вызова fuuc2    */
```

```
//Подразумевается множество символов ASCII.
```

Комментарии также можно использовать для выделения неисправности. Например, комментарий типа

```
// !!!!!!!!!
```

отличает неправильность в программе, хотя и не очень помогает. Лучше сделать комментарий более информативным:

```
// Исправить ошибку округления.
```

Наверное, лучший способ комментирования функции - это выбор содержательных имен функции и переменных. В частности, имена функций должны отражать смысл абстрактной операции, которую реализует функция, как в предыдущем примере: `input`, `sort` и `output`. Если в программе использованы содержательные и мнемонические имена, необходимость в дополнительных комментариях возникает редко. Если не использовать мнемонические имена, вряд ли дополнительные комментарии сделают код более понятным.

Предшествующее обсуждение предполагает, что программист хочет помочь другим понимать его программы. Можно писать программу для решения задачи, не собираясь ее использовать в дальнейшем, или имея целью не дать другим изменить или использовать программу и умышленно делать ее непонятной. Тактика удаления комментариев и отступов, как и замена имен, часто используется для ус-

ложнения заимствования кода соперниками. В случаях, если это не явный акт враждебности, написание непрозрачного кода без комментариев - это плохая идея. Большинство программ, в конечном счете, требует сопровождения, независимо от того, приспособлены они для этого или нет. Программы, которые трудно понять, сложно модифицировать, не внося повреждения этими изменениями. Стоимость сопровождения будет увеличиваться или программу придется "рассыпать". Чтобы увеличить время полезной работы программ, программисты должны делать их понятными.

2.5. Перегружаемые и подставляемые функции.

Мы обсудили как использовать функции для реализации абстрактных операций, и как имена функций документируют операции. Это использование в дальнейшем было поддержано введением перегружаемых и подставляемых функций.

Перегрузка позволяет использовать одно имя для реализации разных функций, если реализация может быть распознана по типу или числу параметров. Таким образом, одну абстрактную операцию можно применять к операндам разного типа, не выдумывая искусственных имен для различения функций. Например, мы можем перегрузить `sort` для работы со списками двух типов.

```
typedef char *String;  
void sort(String *,int);  
void sort(int *,int);
```

В старых версиях C++ необходимо явное объявление перед перегрузкой функции:

```
overload sort;
```

Первая функция сортирует массив списков, вторая - массив целых чисел. Вызов функции

```
sort(list,size);
```

вызывает первую, если `list` имеет тип `String *`, или вторую, если тип - `int *`.

Подставляемые функции поддерживают использование маленьких функций, если из соображений эффективности программист вынужден использовать вызов функции для выполнения нескольких операторов, реализующих абстрактную операцию. Спецификатор `inline` в объявлении функций - это указатель компилятору на необходимость оптимизации вызова этой функции подстановкой тела функции.

Подстановка не изменяет семантики вызова функции. Любая функция может быть объявлена как `inline`, но оптимизация вызова функции может быть произведена не всегда, а в зависимости от возможностей компилятора и от того, как произведен вызов функции. Как и функция, объявленная `static`, подставляемая функция может быть использована только внутри файла, в котором определена, так как определение должно быть видимо из точки вызова для того, чтобы была возможна оптимизация. Если подставляемые функции надо использовать в разных файлах, их нужно включить в заголовочный файл.

В нашем примере оператор сравнения строк `lessthan` - естественный кандидат для превращения в `inline`-функцию, так как в нем всего лишь один оператор.

```
inline int
lessthan(String s1, String s2) {
    return strcmp(s1,s2)<0;
}
```

Ниже мы используем и перегрузку функций, и подставляемый `lessthan` в расширенной версии нашего примера сортировки списков, который сначала сортирует список строк, затем список чисел. Списки снова реализуются как массивы, но теперь есть два массива с разными типами элементов.

```
#include "arrays.h"
```

```
const int max=100;
```

```
String stringlist[max];
int Inlist[max];

int size=0;

main() {
    // Читает, сортирует и выдает список слов
    size=input(stringlist,max);
    sort(stringlist,size);
    output(stringlist,size);

    //Читает, сортирует и выдает список чисел
    size=input(Inlist,max);
    sort(Inlist,size);
    output(Inlist,size);
}
```

Заголовочный файл `arrays.h` содержит объявления перегружаемых функций.

```
typedef char* String;

overload input,output,sort;

void Input(String *,const int,int &);
void input(int *,const int,int &);

void output(String *,const int);
void output(int *,const int);

void sort(String *,const int);
void sort(int *,const int);
```

Функции представлены в `arrays.c`. Полная реализация обеих версий перегружаемых функций представлена для того, чтобы показать, что они во многом аналогичны. Это демонстрирует другую возможность повторного использования: заимствование кода и его редактирование. Иными словами, общий шаблон текста функций используется для реализации обеих функций, а для необходимых различий типов в тексте делаются небольшие изменения. Более хитрые методы копирования и редактирования кода будут об-

суждаться в Главе 4.

```
#include <<stdio.h>
#include <<string.h>
#include "arrays.h"

static int
readString(String &s) {
    // Ничего нового, см. выше
}

inline int
lessthan(String s1,String s2) {
    return strcmp(s1,s2)<0;
}

int
input(String *a,int limit) {
    for (int i=0;i<<limit;i++)
        if (!readString(a[i]))
            break;
    return i;
}

int
input(int *a,int limit) {
    for (int i=0;i<<limit;i++)
        if (scanf("%d",&a[i])==EOF)
            break;
    return i;
}

void
output(String *a,int size) {
    printf("\nlist:");
    for (int i=0;i<<size;i++)
        printf("%s",a[i]);
    printf("\n");
}

void
output(int *a,int size) {
    printf("\nlist:");
    for (int i=0;i<<size;i++)
        printf("%d",a[i]);
}
```

```
printf("\n");
}

void
sort(String *a,int n){
  int changed;
  do {
    changed=0;
    for (int i=0;i<n-1;i++)
      if (lessthan(a[i+1],a[i])) {
        String temp=a[i];
        a[i]=a[i+1];
        a[i+1]=temp;
        changed=1;
      }
  } while (changed);
}

void
sort(int *a,int n){
  int changed;
  do {
    changed=0;
    for (int i=0;i<n-1;i++)
      if (a[i+1]<a[i]) {
        int temp=a[i];
        a[i]=a[i+1];
        a[i+1]=temp;
        changed=1;
      }
  } while (changed);
}
```

При использовании перегружаемых функций необходимо учитывать следующее. Перегрузка концептуально унифицирует различные функции в программе, но это делается введением неоднозначности в имя функции. В примере с перегружаемой сортировкой не слишком сложно определить, какая функция вызвана

```
sort(list,size);
```

Так как нет автоматического преобразования типов ар-

гументов, тип, указанный в объявлении `list` явно определяет, какая из двух функций вызвана. Если есть многократно перегружаемые функции, и имеют место различные преобразования возможных типов аргументов, то идентификация функции может усложниться. Использование перегрузки для отражения концептуального единства должно быть сбалансировано с трудностями, вызванными двусмысленными именами, скрывающими, что же на самом деле делает программа.

2.6. Аргументы и возвращаемые значения.

Тип функции определяется типами параметров и типом возвращаемого результата. При вызове функция сначала идентифицируется по имени. Затем идентификация продолжается проверкой соответствия аргументов типу объявленных параметров или возможности его установления допустимым преобразованием типов. Выбор конкретной перегружаемой функции происходит при соответствии типа аргументов типу параметров.

Тип функций, имеющих произвольное число и тип аргументов, специфицируется в списке параметров.

```
void error(const char * ...);
```

В этом предложении языка `error` объявлена как функция, не возвращающая значения, первым аргументом которой является строка, и добавочно имеющая любое число аргументов любого типа. `const` указывает, что строку нельзя изменять через указатель, поэтому в вызове вполне можно использовать строчные литералы. Первый аргумент в вызове функции может содержать информацию, необходимую для интерпретации следующих аргументов.

Аргументы автоматически преобразуются в типы параметров вызванной функции. Если преобразование происходит в класс, сначала должны быть произведены предопределенные преобразования аргументов в тип, тре-

букет конструктором класса так, как это было показано в примере с классом `complex` в Главе 1. Если происходит преобразование из классового типа аргументов в неклассовый тип параметров, дополнительные предопределенные преобразования применяются после преобразований, определенных пользователем. Для аргументов, соответствующих спецификации эллипсисных параметров, целые типы преобразуются в `int` и типы с плавающей точкой преобразуются в `double`, но никакие другие преобразования не производятся.

Когда функция выполняется, формальные параметры в определении функции инициализируются фактическими аргументами в вызове функции. Значение, возвращаемое функцией, используется для инициализации результата. То, что аргументы и возвращаемые значения скорее инициализируются, чем присваиваются, особенно важно для `const`-типов и ссылочных типов, как и для классов с конструкторами, в которых инициализация значительно отличается от присваивания.

Формальные параметры функции - переменные, локализованные в блоке. После инициализации в вызове функции их можно использовать как любые другие переменные, включая использование в операторах присваивания и изменения их значений. Спецификатор `const` означает, что объявленный объект не должен изменять значение после инициализации. Исходное значение параметра `const`, полученное как аргумент, изменять нельзя. Такое объявление параметра может предохранить от ошибочного изменения значения того, чье значение должно быть постоянным. Например, в нашей сортировке списка размер содержимого списка не изменяется. Чтобы показать это, параметр следует объявить как `const`.

```
sort(String *,const int);
```

За исключением ссылочных, формальные параметры и фактические аргументы - это разные объекты. После инициализации значениями аргументов, параметры можно ис-

пользовать локально без всякого внешнего относительно функции эффекта. Как было сказано в Главе 1, ссылочные параметры не являются независимыми от фактических аргументов. Их инициализация устанавливает имя ссылки на псевдоним объекта-инициализатора. Ссылочный тип параметров допускает "вызов по ссылке", при котором присваивание параметру значения внутри функции приводит к изменению значения фактического аргумента. Для параметра `readstring` был использован ссылочный тип, так как через аргумент возвращается строка.

```
int readString(Sring &);
```

Удобным свойством C++, позволяющим гибко использовать функции, является наличие предопределенных инициализаторов аргументов. Значения аргументов по умолчанию можно задать в объявлении функции, при этом они подставляются автоматически в вызов функции, содержащей меньшее число аргументов, чем объявлено. Например, функция объявлена с тремя аргументами, два из которых инициализированы:

```
error(const char *msg, int level=0, int kill=0);
```

может быть вызвана с одним, двумя или тремя аргументами.

```
error("you goofed");  
// фактически вызывается error("you goofed",0,0);  
error("you screwed up",1);  
// фактически вызывается  
// error("you screwed up",1,0);  
error("you blew it!",3,1);  
// значения аргументов по умолчанию  
// не используются
```

Предопределенные аргументы часто полезны для минимизации эффектов изменений в программе при сопровождении. Существующая функция может быть изменена

добавлением аргументов без изменения уже существующих ее вызовов. Объявление с инициализатором аргументов по умолчанию для новых параметров позволяет старые вызовы не изменять. Это облегчает расширение возможностей функции.

В качестве примера изменим нашу функцию сортировки массива строк для того, чтобы выполнялась сортировка как по алфавиту, так и в обратном порядке. Сначала в заголовочном файле добавим в объявлении функции инициализатор нового аргумента.

```
void sort(String *,int,int descending=0);
```

Наличие имени подставляемого аргумента в объявлении функции необходимо только как комментарий, объясняющий читателю, почему здесь имеет место аргумент. После перекомпиляции заголовочных файлов третий аргумент, равный 0, автоматически будет добавлен ко всем вызовам функции сортировки массива строк с двумя аргументами.

Теперь адаптируем функцию сортировки для использования нового, третьего аргумента. Сначала добавим функцию `greaterthan` для сравнения двух строк. Затем изменим `sort` для использования переменной-указателя на функцию для вызова либо `lessthan`, либо `greaterthan`. Спецификатор `inline` не даст здесь оптимизирующего эффекта, так как функция вызывается через указатель.

```
inline int  
lessthan(String s1,String s2){  
    return strcmp(s1,s2)<0;  
}  
  
inline int  
greaterthan(String s1,String s2){  
    return strcmp(s1,s2)>0;  
}  
  
void  
sort(String *a,int n,int descending){
```

```

Int changed;
typedef Int (* Fptype)(String,String);

// объявляет и инициализирует указатель
// на функцию сравнения
Fptype compare=descending ?
&greaterthan : &lessthan;

do {
    changed=0;
    for (Int l=0;l<n-1;l++)
        If (compare(a[l+1],a[l])) {
            String temp=a[l];
            a[l]=a[l+1];
            a[l+1]=temp;
            changed=1;
        }
    } while (changed);
}

```

Оператор **?**, используемый для выбора функции сравнения - это условное выражение. Если первый операнд **descending** имеет ненулевое значение, второй операнд **greaterthan** выбирается и дает результат выражения. Если **descending** имеет значение ноль, то третий операнд, т. е. **&lessthan**, является результатом выражения.

Функция, таким образом, расширена без влияния на уже существующие ее использования; при этом потребовалась только перекомпиляция файлов, содержащих измененный заголовок. Новые вызовы **sort** могут использовать новые функциональные возможности, устанавливая третий аргумент.

```

#include "arrays.h"

const int max=100;
String stringlist[max];

Int size=0;
const Int descending=1;

main() {

```

```
// То же, что и ранее
size=input(stringlist,max);
sort(stringlist,size);
output(stringlist,size);

// Сейчас в порядке убывания ...
size=input(stringlist,max);
sort(stringlist,size,descending);
output(stringlist,size);
}
```

2.7. Упражнения.

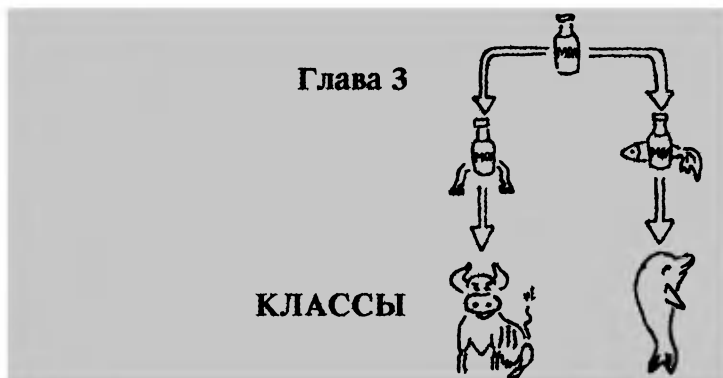
- Упр.2.1.** Измените программу сортировки строк, чтобы она могла работать с произвольным числом строк. Не изменяйте иерархических связей функций по сравнению с исходным решением. Является ли такое решение новым, или это другая версия того же решения?
- Упр. 2.2.** Придумайте другую декомпозицию для решения задачи сортировки строк, используя следующую концептуализацию:
пока читаем строку
вставляем строку в нужную позицию
печатаем строки.
Какое решение более эффективно? Какое решение лучше приспособлено для интерактивного использования? Какое решение лучше?
- Упр. 2.3.** Часто определенные функции могут быть для других функций в программе. Например, функция `error` может быть использована, если надо выдать сообщение об ошибке; функция `lookup` может использоваться любой функцией, которой необходим доступ к сохраненной информации. Каким образом метод функциональной декомпозиции, который приводит к иерархической разработке, мешает идентификации об-

щих вспомогательных функций?

Упр. 2.4.+ Определим знак как слово или осмысленную группу символов в тексте, окруженную разделительными символами, такими как пробел, табуляция или перевод строки. Напишите функцию, возвращающую адрес следующего знака, модифицируя значение своего аргумента так, чтобы он указывал на следующий знак. Введите дополнительный второй аргумент, указывающий, какие символы являются разделителями знаков. Используйте функцию `strlen` для написания версии функции, возвращающей длину своей строки аргументов.

Упр. 2.5. Напишите функцию
`int bsearch(int *array, int key, int num);`
производящую бинарный поиск по целому ключу в упорядоченном массиве целых. Перепишите `bsearch` для работы:

- а) с массивом `double` с ключом типа `double`;
- б) с массивом структур с целым ключом;
- в) с массивом структур и с ключевой строкой;
- г) с массивом указателей на структуры и ключом типа `int`;
- д) с массивом структур на диске, слишком больших, чтобы полностью считать их в память и ключом типа `int`.



Функции и структуры данных, которыми они оперируют, независимы. Функции, разработанные для применения к конкретной структуре данных, должны корректно изменять данные. В примере с сортировкой строк из Главы 2 функции зависят от структуры данных, представляющей строку как заканчивающийся нулем массив символов. Если одна из функций ошибочно перезапишет завершающий нуль, другие функции будут на испорченной структуре данных работать неправильно. Если изменить представление строки, введя в структуру вместо завершающего символа информацию о длине, для работы над этой новой структурой данных необходимо написать новые функции. В больших программах может быть сложно разобраться во взаимозависимостях множества функций и сложных структур данных.

Классы в C++ поддерживают несколько путей организации программ и управляющих зависимостей между структурами данных и функциями. Приемы программирования с использованием классов - это тема оставшейся части книги. Закладывая фундамент для дальнейшей дискуссии, эта глава начинает рассмотрение особенностей классов в C++.

3.1. Классовые типы.

Классы - это сложные структуры данных, определяемые пользователями. Они могут содержать как элементы дан-

ных, представляющих тип, так и функциональные элементы, реализующие операции над типом. Части класса могут быть скрыты или сделаны общедоступными при помощи деклараций `private` и `public`. Элементы в секции `private` доступны только через функциональные элементы этого класса или через другие функции, объявленные как дружественные (`friend`) классу. Используя этот механизм сокрытия информации, класс может инкапсулировать структуру данных так, что только специфицированные классом функции могут ее использовать. Мы представим новую реализацию примера с сортировкой строк с использованием классового типа *String* для представления строк. Признак класса *String* - это имя типа класса, и может использоваться также как имя, заданное при помощи `typedef`.

```
#include <stdio.h>
#include <string.h>

class String {
    char *str;
public:
    String() {str=new char[1]; *str=0;}
    String(char *);
    void print() {printf("%s",str);}
    friend int operator < (String s1,String s2)
        { return strcmp (s1.str,s2.str)<0;}
};
```

Структура данных, использованная для представления строки, не изменилась; это указатель на завершающийся нулем массив символов. Тем не менее, сейчас эта структура скрыта как элемент `str` в приватной части *String*. Только конструкторы *String* (элемент-функция `print` и дружественная функция `operator <`) имеет доступ к представлению строки. Все элементы-функции объявлены в секции `public` класса и, соответственно, общедоступны.

Имена первых двух функциональных элементов совпадают с именем класса, что идентифицирует их как конструкторы для класса. Когда создается *String*, конструктор используется для его инициализации. Первый конструктор

String не имеет аргументов и инициализирует *str* как указатель на пустую строку. Оператор *new* используется для выделения в памяти места под строку. Так как эта функция коротка, ее тело определяется внутри определения класса. Это простейший способ сделать элемент или *friend*-функцию *inline*-вида.

Второй конструктор только объявлен внутри класса, поэтому его определение должно быть дано где-то в другом месте.

```
String::String(char *s) {  
    str=new char[strlen(s)+1];  
    strcpy(str,s);  
}
```

Для элемента-функции, определяемого вне класса, имя функции должно быть квалифицировано, чтобы показать, что упоминаемая функция находится в области видимости класса. Здесь квалификатор области видимости *String::* указывает, что это определение элемента *String*.

Функции-элементы всегда находятся в области видимости их класса. Поэтому элементы объекта класса, с которыми оперируют элементы-функции, доступны без использования оператора доступа к элементам. Обратите внимание, что *String* и *print* обращаются к *str* без квалификации. Напротив, дружественная функция

```
friend int operator < (String s1,String s2)  
{ return strcmp(s1.str,s2.str)<0; }
```

должна использовать оператор доступа к элементу для получения *str* своих аргументов. Дружественный статус разрешает доступ к приватному элементу *str*, но не изменяет область видимости функции.

Операторные функции - это способ перегрузки предопределенных в языке операторов для применения к операндам типа классов. Здесь оператор *<* определяется для сравнения двух ссылочных элементов *String*. Инфиксный оператор *<* теперь можно использовать для двух операндов типа *String*.

Завершая пример с сортировкой строк, мы увидим, как можно использовать тип *String*. В функции, читающей строки в список, конструктор вызывается для создания объекта класса *String* из символов во входном буфере. Затем объект типа *String* присваивается слоту в массиве-списке.

```
void
Input(String *a,int llimit,int &i) {
    static char buffer[100];
    for (i=0;i<llimit;i++)
        if (scanf("%s",buffer) != EOF)
            break;
    else
        a[i]=String(buffer);
}
```

В процедуре, выдающей список, функция `print`, которая используется для выдачи *String*-ов, вызывается с каждым элементом списка с применением `.` (точки) - оператора доступа к элементу.

```
void
output(String *a,int size) {
    printf("\nlist:");
    for (int i=0;i<size;i++)
        a[i].print();
    printf("\n");
}
```

В функции, сортирующей список, `operator <` вызывается для сравнения элементов списка в инфиксной нотации, и эта процедура выглядит точно также, как сортировка списка целых чисел в Главе 2, за исключением типа элементов списка.

```
void
sort(String *a,int n) {
    int changed;
    do {
        changed=0;
        for (int i=0;i<n-1;i++)
```

```
    if (a[i+1]<a[i]) {  
        String temp=a[i];  
        a[i]=a[i+1];  
        a[i+1]=temp;  
        changed=1;  
    }  
} while(changed);  
}
```

И в завершение, функция `main` читает, сортирует и выводит список.

```
const int max=10;  
String list[max];  
  
int size=0;  
  
main() {  
    input(list,max,size);  
    sort(list,size);  
    output(list,size);  
}
```

Это выглядит точно также, как и вариант в Главе 2, использующий реализацию типа *String* как *char**. Конечно, есть разница в реализации примеров. В этом случае, для инициализации каждого элемента списка вызывается конструктор без аргументов. Массив списка изначально состоит из *String* ов, а не инициализирован нулями. Хотя для этой программы это не имеет значения, однако автоматическая инициализация объектов класса через конструкторы делает программы корректнее, так как структуры данных, инкапсулированные в объекте, с самого начала автоматически находятся в корректном состоянии.

3.2. Компоненты данных.

Определение классового типа, как в ситуации со *String* в предыдущем разделе, служит шаблоном для объектов классового типа. Когда создается объект классового типа, выделяется пространство для объекта, а также создаются и

инициализируются элементы данных как компоненты объекта. Также как и другие объекты данных в C++, объекты классового типа могут быть созданы в разных местах и существовать разные периоды времени. Внутрифайловые или локальные статические объекты создаются перед началом выполнения программы и существуют до ее окончания; параметры и локальные объекты создаются в случаях, когда выполнение программы достигнет точки их объявления и существуют до тех пор, пока блок, в котором они объявлены, существует; динамически создаваемые с помощью оператора `new` объекты существуют, пока не будут явно уничтожены оператором `delete`. Временные объекты создаются также для получения промежуточных результатов вычисления выражения и возвращаемых функциями значений.

Доступ к компонентам класса возможен при помощи операторов доступа `.` (точка) и `->`. Оператор `.` (точка) используется с классовыми объектами.

```
String s;  
s.str;
```

Оператор `->` используется с указателями на классовые объекты.

```
String *sp=new String;  
sp->str;
```

Оператор `new` создает объект, тип которого специфицируется его аргументом и возвращает указатель на новый объект.

Компоненты данных класса могут быть любого предопределенного в языке типа, предварительно определенного классового типа, указателями на классовый тип или ссылкой на классовый тип. Для указателей и ссылок на классовый тип не нужно, чтобы такой класс был определен, нужно только, чтобы имя класса было объявлено. Например:

```
class Node; // объявляется имя  
Node *np; // оно используется для объявления указателя
```

Используя указатель или ссылку на классовый тип как элемент класса, можно строить рекурсивные классовые структуры. Например, вершина бинарного дерева может содержать указатели на правого и левого потомка.

```
class Node {  
    Node *left, *right;  
    // и т.д.  
};
```

Вершины здесь являются указателями; *Node* не может содержать *Node* напрямую.

Каждый компонент данных в объекте класса занимает определенное место, необходимое для представления его типа. Между элементами классового объекта могут быть буферные промежутки, зависящие от необходимого для разных машин выравнивания.

Один из способов экономии места при использовании объекта - использование одного и того же пространства для размещения более чем одного компонента. Есть классы, элементы данных в которых размещаются не последовательно, но с перекрытием, имея начальный адрес, совпадающий с начальной позицией в объекте. Такой тип класса называется *union*. Спецификатор *union* заменяет *class* в объявлении типа. Элементы объединенного типа *union* все общедоступны (*public*).

```
union un_type {  
    int i;  
    double d;  
    char *p;  
};
```

Когда объект объединенного типа реализуется, его размер и выравнивание устанавливается так, что все элементы занимают в объекте одно и то же место. На элементы *union* можно сослаться, используя оператор доступа к элементу. Результатом доступа к элементу *union* служит тот факт, что одно и то же пространство памяти интерпретируется в соот-

ветствии с типом элемента.

```
un_type u;

u.l=1;    // в u запомнили значение типа int
u.d=3.1415; // перекрыли значением типа double
char c=*u.p // Опасность! М.б. неверное значение
           // указателя
```

Достаточно опасно запоминать элементы `union` как данные одного типа, а использовать как данные другого. Допустимое значение одного элемента может не быть допустимым для другого. Для корректного использования `union`-ов обычно необходимо обладать информацией, определяющей, какой элемент используется. `Union` можно сделать элементами классов, если другие элементы содержат информацию о том, как интерпретировать элементы `union`.

```
enum { ISINT,ISDOUBLE,ISCHARSTAR };

class Node {
    Node *left, *right;
public:
    int code;
    un_type info;
} *np;
```

В этом примере `code` в объекте *Node* определяет способ интерпретации элемента союза.

```
int l=0;double d=0;char *p=0;
switch(np->code){
    case ISCHARSTAR:
        p=np->info.p;
        break;
    case ISDOUBLE:
        d=np->info.d;
        break;
    case ISINT:
        l=np->info.l;
        break;
}
```

Так как *Node* содержит одновременно информацию только одного типа, используем союз для наложения объектов разных типов, экономя пространство, занимаемое объектом *Node*.

Совершенно аналогично элементы класса можно сделать перекрывающимися, не добавляя явного доступа к элементу союза, но вводя в класс анонимный союз. Анонимный союз не имеет признака и не объявляет имя элемента.

```
class Node {
    Node *left,*right;
public:
    int code;
    union {
        int i;
        double d;
        char *p;
    };
} *np;
```

Здесь *i*, *d* и *p* перекрываются, но к ним можно получить прямой доступ как к элементам *Node*.

```
int i=0;double d=0; char *p=0;

switch(np->code) {
    case ISCHARSTAR:
        p=np->p;
        break;
    case ISDOUBLE:
        d=np->d;
        break;
    case ISINT:
        i=np->i;
        break;
}
```

Другой способ экономии места в классовой объекте состоит в разделении сегмента целого размера на несколько элементов целого типа, занимающих определенное число бит. Битовые поля позволяют контролировать объем, используя

емый классовым объектом при условии, если известно, что конкретные целые элементы будут принимать значения только из ограниченного интервала. Битовое поле можно специфицировать, сопровождая объявление элемента данных двоеточием и размером поля.

```
class Node {  
    Node *left,*right;  
public:  
    unsigned int code: 2;  
    unsigned int is_leaf: 1;  
    unsigned int is_free: 1;  
};
```

В этом примере `code` занимает два бита в объекте *Node* и может принимать четыре значения. Элементы `is_leaf` и `is_free` занимают по одному биту и используются как двоичные флаги. Поля сгруппированы так, что они упаковываются в один сегмент объекта.

Программист может управлять размером класса с помощью битовых полей, группируя элементы для упаковки используемого объекта пространства. Результат такого управления зависит от реализации. Чтобы успешно использовать это управление для экономии памяти под классовый объект, программист должен знать размеры и выравнивание типов на используемой машине.

Элементы данных, объявленные как `static`, используются всеми объектами классового типа. Элемент, объявленный `static`, создается при определении класса и существует раньше любого объекта класса. На него можно ссылаться как на элемент данных или, не используя объект класса, с помощью оператора доступа::. Так как элементы данных, объявленные `static`, не зависят ни от какого конкретного объекта класса, можно брать их адреса и заводить указатели на них как на любой другой `static`-объект.

Как пример использования элемента данных `static`, предположим, что нам необходимо проследить, какое количество объектов класса *String* уже создано. Для подсчета `static` элемент *String* будет увеличиваться каждым конструктором

на 1.

```
class String {
    char *str;
    static int count;
public:
    String() { count++;str=new char; *str=0;}
    String(char *s) {
        count++;
        str=new char [strlen(s)+1];
        strcpy(str,s);
    }
    friend void report();
    // и т.д.
}
```

Для каждого объекта типа *String*, инициализируемого конструктором, один и тот же статический элемент `count` увеличивается на 1. Для определения количества созданных объектов типа *String* в любой точке программы, функция с соответствующим разрешением доступа может быть построена с использованием ссылки `String::count`, как показано ниже в `report`.

```
void report() {
    printf("Доклад об использовании String:");
    printf("%d String-ов создано\n",String::count);
}
```

Удобство использования классов состоит в том, что списки `enum` можно сгруппировать так, что символьные значения, определяемые ими, находятся в области видимости класса. Сгруппированные значения `enum` доступны как элементы класса или с использованием оператора видимости.

```
class Node {
    Node *left,*right;
public:
    enum { JSINT,ISDOUBLE,ISCHARSTAR };
    int code;
    union {
```

```
    int i;  
    double d;  
    char *p;  
};  
} *np;
```

В этом примере значения `enum` локальны по отношению к классу. Они не конфликтуют с именами в разных областях видимости, и доступ к ним можно защитить, поместив их в приватную часть. На значения элементов `enum` класса можно ссылаться также, как и на статические элементы.

```
switch(np->code) {  
    case Node::ISCHARSTAR:  
        p=np->p;  
        break;  
    case Node::ISDOUBLE:  
        d=np->d;  
        break;  
    case Node::ISINT:  
        i=np->i;  
        break;  
}
```

3.3. Функциональные компоненты.

Функция, объявленная внутри определения класса и не специализированная как `friend`, является функциональной компонентой класса. Функциональная компонента (элемент функция) может быть объявлена внутри класса; в этом случае она безоговорочно является `inline`. Для вызова элементов функций используются операторы доступа `.` и `->`. Оператор `.` (точка) используется с классовыми объектами.

```
String s;  
s.print();
```

Оператор `->` используется с указателями на классовые объекты.

```
String *sp=new String;  
sp->print();
```

Как и для простых функций, для функциональных компонент осуществляется проверка вызова при компиляции. Для идентификации функции используется тип классового объекта; фактические аргументы сопоставляются с типом параметров в объявлении функции.

Элементы-функции оперируют объектами того классового типа, с которыми они вызываются. Указатель на этот объект является скрытым аргументом всех функциональных компонент, на них можно напрямую сослаться в определении функции как на `this`. В функциональной компоненте не нужно использовать `this`. Тем не менее, `this` косвенно используется для ссылок на элементы.

```
class String {  
    char *str;  
    public:  
    void print() { printf("%s",str); }  
};
```

Выше ссылка на элемент `str` в `print` представляет собой то же самое, что и `this->str`. При вызове `print` содержимому `this` присваивается адрес объекта, использованного в вызове. Например, `s.print()` устанавливает `this` в значение `&s`, поэтому внутри `print` `str` представляет собой то же, что и `(&s)->str` или `s.str`. В вызове `sp->print()`, `this` равно `sp` и доступный элемент определен как `sp->str`.

Все функциональные элементы логически помещаются внутри области видимости их класса. Если определение элемента функции находится вне описания класса, область видимости функции надо указывать при помощи оператора видимости с именем класса.

```
class X {  
    int f();  
};  
  
int X::f() { /* и т.д. */ }
```

Вложенность элементов-функций внутри класса ведет к

различиям в правилах доступа для функциональных компонент и функций, не являющихся элементами. В функциональной компоненте объявление идентификатора сначала рассматривается в пределах видимости блока, затем класса и наконец, файла. Таким образом, объявление элемента могут перекрывать объявление в файле. В следующем примере идентификатор `x` в функции `X::f()` относится к элементу, а не к переменной файла `x`.

```
class X {  
    int x;  
    int f();  
};  
  
int x;  
  
int X::f() { return x; } // возвращает this->x
```

Для доступа к объявлению идентификатора файла, перекрытому локальным объявлением или объявлением в классе, можно использовать оператор видимости. Для изменения нашего примера так, чтобы `X::f()` возвращала значение переменной файла, а не элемента, используем `::` для индикации, что `x` является переменной файла.

```
int X::f() { return ::x; } // возвращает x из файла
```

Области видимости класса - это подобие блока вокруг функциональной компоненты. Объявление функции внутри блока может перекрыть объявление компоненты точно так же, как объявление внутреннего блока может перекрыть объявление внешнего. В следующем случае `X::f()` возвращает значение локальной переменной.

```
int X::f() { int x=3; return x; }
```

Используя оператор видимости, можно изменить пример так, чтобы возвращался элемент `X::x` или переменная файла `::x`.

3.4. Операторные функции.

Предопределенные в языке операторы могут быть перегружены для работы с операндами классового типа. Это делается при помощи введения операторной функции, которая берет всего один аргумент классового типа. Объявления и определения операторных функций синтаксически такие же, как и для других функций, за исключением имени, имеющего форму `operator x`, где `x` - символ перегружаемого оператора. Определяемые пользователем операторы можно вызывать в инфиксном синтаксисе - обычном для символа `x`. Операнды выражения являются аргументами для вызова функции.

Так как синтаксис их вызова не меняется, операторные функции должны иметь то же число операндов, что и в предопределенной версии оператора. Перегружаемые операторы имеют тот же приоритет, что и соответствующие встроенные.

Взаимосвязи предопределенных операторов, такие как эквивалентность $a+=b$ и $a=a+b$ или $a[b]$ и $*(a+b)$, или $(&p)->x$ и $p.x$ не сохраняются для определенных пользователем операторов, если только операторные функции не реализованы таким образом. Результат операторной функции может быть абсолютно не связан с результатом предопределенных версий. В частности, определенные пользователем `++` и `--` не могут вести себя полностью аналогично предопределенным версиям. Нет способа определить другие версии этих операторов в префиксной и постфиксной формах, так как один и тот же результат будет выработан независимо от того, как используются операторы, разработанные пользователем.

Предположим, мы добавим операторы к классу *complex* из Главы 1.

```
class complex {  
    // и т.д.  
    complex operator ++();  
    complex operator +-(complex);  
};
```

Одна и та же версия ++ будет использована для префиксной и постфиксной формы, вырабатывая один результат.

```
complex c1,c2;  
c2=c1++; // то же, что и c2 = ++c1;
```

Оператор += можно определить так, что

```
c1 += c2;
```

эквивалентно

```
c1 = c1 + c2;
```

а можно и иначе, хотя, возможно. Это и будет выглядеть странным. Так как класс *complex* должен работать как числовой тип, лучше реализовывать += так, чтобы он действовал аналогично встроенным арифметическим операторам.

Операторные функции могут быть, а могут и не быть элементами класса, за исключением операторов (), [] и ->, которые могут быть только функциональными компонентами. Если операторная функция является элементом класса, неявный аргумент *this* является ее первым операндом. В классе *String operator <*, который реализован как *friend*, т. е.

```
class String {  
    char *str;  
public:  
    friend int operator < (String s1,String s2)  
    { return strcmp(s1.str,s2.str)<0; }  
    // и т.д.  
};
```

может также быть реализован как элемент.

```
class String {  
    char *str;  
public:  
    int operator < (String s2)  
    { return strcmp(str,s2.str)<0; }  
    // и т.д.  
};
```

Если операторная функция является элементом, первый операнд должен всегда соответствовать типу класса, преобразования типа при этом не допускаются. Операторные функции часто предполагаются работающими аналогично предопределенным в языке версиям, в которых преобразования применяются к каждому операнду. В связи с этим операторные функции часто реализуются как **friend-функции**.

Операторные функции - это функции со специфическими именами, оперирующие аргументами классового типа. Они не обязательно должны вызываться в инфиксной нотации, можно вызывать их и как любую другую функцию. Для функций, не являющихся элементами класса, все аргументы передаются в списке аргументов.

```
class String {  
    friend int operator < (String, String);  
    // и т.д.  
};  
  
void sort(String *a, int n) {  
    // и т.д.  
    if (operator < (a[i+1], a[i])) {  
        // и т.д.
```

Для функциональных компонент используется оператор доступа к элементу.

```
class String {  
    int operator < (String);  
    // и т.д.  
};  
  
void sort(String *a, int n) {  
    // и т.д.  
    if (a[i+1].operator < (a[i])) {  
        // и т.п.
```

3.5. Защита доступа и дружественные функции.

Элементы класса могут быть как `public` (общедоступными), так и `private` (доступ к элементам и дружественным функциям защищен). Доступность элементов определяется их объявлением в секции определений класса, озаглавленной меткой `public` или `private`. Эти метки могут появляться в определении класса любое число раз и в любом порядке. Первая секция определения класса начинается как `private`, пока метка не укажет другой уровень защиты.

Есть и третий уровень защиты в C++, играющий роль при использовании наследования классов. Элемент может специфицироваться как `protected`, точно также как `public` или `private`. Использование защищенных элементов обсуждается в Главе 5.

Функции, объявленные дружественными в определении класса, не являются элементами класса, но имеют разрешение на доступ к приватным элементам объектов классового типа. Один класс также может быть объявлен дружественным другому, указывая, что все функциональные компоненты дружественного класса дружественны.

```
class Y;  
class X {  
    friend Y;  
    int i;  
    void f();  
};  
  
class Y {  
    int f1(X&);  
    void f2(X&);  
    // и т.д.  
};
```

В этом примере приватные элементы объектов типа `X`, такие, как элементы `i` и `j()` аргументов `X&` доступны внутри `Y::f1` и `Y::f2`. Операторы доступа к элементам обязательно должны использоваться дружественными функциями, так

как только элементы класса имеют `this`.

Мы уже упоминали, что в союзах элементы общедоступны, если не указано противное. Структуры - это другой тип классов, которые отличаются от классов предопределенным уровнем доступа к элементам. В определении структуры вместо `class` используется `struct`. Первая секция структуры будет общедоступной, пока при помощи метки не будет указан другой уровень защиты.

```
struct String {  
    String();  
    String( char * );  
    void print();  
    friend int operator < (String s1,String s2);  
    // и т.д.  
private:  
    char *str;  
};
```

В этой версии *String* все функциональные компоненты общедоступны, тогда как элемент данных `str` - приватный.

3.6. Инициализация и преобразования.

Если класс содержит конструкторы, то конструктор всегда используется для инициализации объектов классового типа при их создании. В примере с сортировкой строк в этой главе, конструктор без аргументов инициализирует массив *String*. Для классовых объектов - не массивов, аргументы конструктора могут использоваться для инициализации объектов в точке объявления.

```
String s1( "hi" );
```

```
String s2 = "hi";
```

В объявлениях выше используются обе формы инициализаторов как аргументы конструктора для инициализации объектов `s1` и `s2` типа *String*.

Аргументы конструктора также могут использоваться, когда объекты создаются оператором `new`.

```
String *sp;
```

```
sp = new String( "hello" );
```

Здесь `sp` указывает на *String*, инициализированную конструктором с аргументом "hello". Если аргументы не используются при объявлении или создании объекта класса, используется конструктор без аргументов или с предопределенными аргументами.

Классовые объекты, являющиеся компонентами других классовых объектов, также инициализируются конструктором. Аргументы для конструктора могут быть даны в списке инициализации элементов для конструктора объемлющего класса. Начальные значения для любого элемента (а не только аргументы конструктора) могут задаваться в списке инициализации элементов конструктора. Элементы, требующие инициализации, такие как константы или элементы ссылочного типа, должны иметь свои инициализаторы в списке. В определении конструктора список инициализации элементов отделяется от списка аргументов конструктора двоеточием. Список инициализации содержит список имен элементов, каждый из которых сопровождается заключенным в скобки списком аргументов конструктора или начальным значением.

```
enum { A, B, C };
```

```
class Node {  
public:  
    int code;  
    String str;  
  
    Node(int, char *);  
};
```

```
Node::Node(int c, char *s)  
    : code( c ), str( s ) { }
```

Конструктор *Node* в этом примере инициализирует элемент *code* значением аргумента *c* и передает второй аргумент конструктору *String* для инициализации элемента *str* из *Node*. С учетом данного определения *Node*, фрагмент программы

```
Node *np = new Node(A, "hello");
```

```
np->str.print();
```

печатает

```
hello
```

Может показаться, что использование списка инициализации элементов можно заменить присваиваниями в теле конструктора.

```
Node::Node(int c, char *s) {  
    code = c;  
    str = s;  
}
```

Для целого элемента *code* нет большой разницы между первым присваиванием в этой новой версии конструктора *Node* и указанием начального значения в списке инициализации. Элемент *str* имеет классовый тип *String*, имеющий конструкторы:

```
class String {  
    char *str;  
public:  
    String() { str=new char; *str = 0; }  
    String( char *s) {  
        str = new char[ strlen(s)+1 ];  
        strcpy(str,s);  
    }  
    // и т.д.  
}
```

Когда бы не создавался объект *String*, он всегда инициализируется. Так как не указан никакой инициализатор, для инициализации *str* при его создании используется предопределенный конструктор *String*. Присваивание внутри тела конструктора заменяет начальное значение *str* на результат преобразования в правой части присваивания. Будут произведены два вызова конструкторов *String*: один для инициализации *str*, другой - для преобразования *s* перед присваиванием. Присваивание, таким образом, это не то же самое, что инициализация. Аргументы конструкторов и инициализирующие элементы должны быть указаны в списке инициализации.

Конструктор создает значение классового объекта из его аргументов, и поэтому преобразует аргументы в классовый тип. Конструкторы - это не только инициализаторы, но и операторы преобразования. Это показано в программе сортировки строк, в которой указатель на массив символов преобразуется в *String* с использованием преобразования в функциональном стиле и присваивается элементу массива *String*.

```
a[i] = String(buffer);
```

Когда конструктор принимает один аргумент, он также может быть вызван для преобразования с использованием оператора преобразования в форме приведения типов.

```
a[i] = (String) buffer;
```

Преобразования по типу вызова функции и приведением типов вообще-то взаимозаменяемы, но преобразование при помощи конструктора с многими аргументами требует вызова в функциональном стиле, а спецификации преобразования синтаксически сложного типа требуют следующего приведения типов:

```
x=X(i,j);    // преобразует i и j в X  
fp=(int(*)0)g; // преобразует g в указатель на функцию
```

Операнды выражений и аргументы вызова функции автоматически преобразуются в нужный тип, если доступны необходимые предопределенные и/или определенные пользователем операции преобразования. Самое большее одно предопределенное и одно определенное пользователем преобразование производится автоматически. Двусмысленный выбор возможных преобразований является ошибкой. Для вызова преобразования, которое не может быть произведено автоматически необходима явная операция преобразования.

Так как преобразования могут происходить автоматически, явная операция преобразования входного буфера `char*` в `String` в примере с сортировкой строк не нужна. Наличие конструктора `String(char*)` позволяет определенное пользователем преобразование из типа в правой части в тип левой части. Присваивание, заботящееся об автоматическом преобразовании

```
a[l] = buffer;
```

работает точно также, как присваивание с явным преобразованием. В обоих случаях для преобразования правой части используется конструктор `String`.

Конструкторы производят преобразования в тип класса. Можно производить преобразования из классового типа с помощью операторных функций преобразования. Эти функции должны быть элементами преобразуемого класса. Они имеют имена в форме `operator T`, где `T` - имя или спецификация типа, являющегося результатом преобразования.

Добавим к классу `String` оператор обратного преобразования `String` в `char *`:

```
class String {  
    char *str;  
public:  
    operator char*();  
    // и т.д.  
};
```

```
String::operator char*() {  
    char *p = new char[strlen(str)+1];  
    strcpy(p,str);  
    return p;  
}
```

Необходимо учитывать, что `operator char*` создает копию массива символов, на который указывает `str` и возвращает указатель на этот новый массив. Простой возврат значения `str` производит необходимое преобразование, но это дает возможность доступа к структуре данных, защищенных *String*, оставляя ее открытой для неправильного использования. Создание копии оставляет массив, на который указывает элемент `str`, доступным только через объект типа *String* и функции, имеющие доступ к *String*.

Следующий пример демонстрирует использование оператора преобразования и показывает, каким образом результат `char*` может быть использован без влияния на исходный *String*. Мы также ввели в наш пример новый элемент `~String()`. Это деструктор для класса *String*. Он выполняется, как только *String* покидает область видимости. Деструктор используется для освобождения места, занимаемого массивом символов, созданного конструктором *String*. Необходимо учитывать, что предопределенный конструктор изменен. Оператор `delete` освобождает память, захваченную `new` и может применяться только к объекту, созданному с помощью `new`. Оператор `delete` вызывает деструктор для объекта класса перед тем, как освободить память.

```
#include <stdio.h>  
#include <string.h>  
  
class String {  
    char *str;  
public:  
    String() { str=new char; *str=0; }  
    String( char * );  
    void print() { printf("%s",str); }  
    operator char*();  
};
```

```
~String() { delete str; }
// и т.д.
};

String::String( char *s) {
    str = new char[strlen(s)+1];
    strcpy(str,s);
}

String::operator char*() {
    char *p = new char[strlen(str)+1];
    strcpy(p,str);
    return p;
}

main() {
    String *sp = new String( "hello world?" );
    sp->print();
    char *cp;
    cp=(char *)*sp;
    cp[11]='!';
    printf("\n %s \n",cp);
    sp->print();
    delete sp;
    cp[0]='H';
    cp[6]='W';
    printf("\n %s \n",cp);
}
```

В этой программе явное преобразование *String* в *char**

```
cp=(char *)*sp;
```

не является необходимым, так как преобразование может быть произведено автоматически. Это преобразование включено только для того, чтобы отметить место использования **operator char***. Программа выдает:

```
hello world?
hello world!
hello world?
Hello World!
```

3.7. Указатели на компоненты класса.

Элементы класса, отличные от элементов данных `static`, являются компонентами класса. Адреса компонентов класса являются "смещениями" относительно конкретного объекта. Относительные адреса компонентов класса имеют тип указателя на элемент класса.

Модификатор типа для индикации указателя на элемент в объявлении записывается как `X::*`, где `X` - имя класса. Для разыменования используются операторы `.*` и `->.*`. Как и связанные с ними операторы доступа к элементам, операторы разыменования надо использовать с левым операндом классового типа.

В следующих маленьких примерах мы объявляем, создаем и используем указатели на элемент простого класса.

```
class Node {  
    public:  
        int code;  
        int num;  
        void print();  
        void report();  
};
```

Сначала указатель `pi` на элемент данных *Node* типа `int`

```
int Node::*pi;  
  
Node n,*np= new Node;  
int i,j;  
  
pi=&Node::code;  
i=n.*pi; // достаем n.code  
j=np->*pi; // достаем np->code  
  
pi=&Node::num;  
i=n.*pi; // достаем n.num  
j=np->*pi; // достаем np->num
```

объявляется как указатель на элемент `int` из *Node*. Он создан для указания на `Node::code` и `Node::num` и использу-

ется для доступа к этим элементам объектов типа *Node*.

Спецификация типа указателя на функциональный элемент - это нечто сложное, поэтому мы используем `typedef` для присвоения типу указателя на функциональный элемент *Node* имени `pftype`, не используя аргументов и возвращая `void`. Затем объявим `pf` как указатель этого типа.

```
typedef void (Node::*PftypeX);
Pftype pf;

pf=&Node::print;
(n.*pf); // вызывает n.print()
(np->*pf); // вызывает np->print()

pf=&Node::report;
(n.*pf); // вызывает n.report()
(np->*pf); // вызывает np->report()
```

Указатель `pf` использован для вызова сначала `Node::print`, а затем `Node::report` для разных объектов *Node*. При использовании `n.*pf` и `np->*pf` следует указывать скобки для получения нужной компоновки операторов разыменования относительно оператора вызова. Вызов функциональных компонент таким образом - это основное использование указателей на элементы.

Для демонстрации использования указателя на функциональный элемент мы расширим функцию сортировки строк для расположения списка по возрастанию или убыванию в зависимости от значения дополнительного третьего параметра. Используются варианты операторов-элементов для сравнения значений *String*.

```
class String {
    // скрытая реализация
public:
    // и т.д.
    int operator < (String s);
    int operator > (String s);
};
```

Функция `sort` объявляет переменную `compare` как указатель на элемент-функцию `String`, имеющую аргумент типа `String` и возвращающую `int`. Поскольку переменная `compare` представляет собой указатель на тип функций, одну из которых мы хотим выбрать, то он устанавливается на `String::operator >` или на `String::operator <`.

```
void sort(String *a, int n, int descending) {
    int changed;

    typedef int (String::*Ftype)(String);
    Ftype compare = descending ?
        String::operator > :
        String::operator <;
    do {
        changed=0;
        for (int i=0; i<n-1; i++)
            if ((a[i+1].*compare)(a[i])) {
                // поменять местами элементы массива
            }
    } while (changed);
}
```

Вызов функции через указатель,

`(a[i+1].*compare)(a[i])`

- это тоже, что и инфиксный вызов операторной функции

`a[i+1] > a[i]`

или

`a[i+1] < a[i]`

в зависимости от значения `compare`.

3.8. Упражнения.

- Упр.3.1.+ Напишите функцию, печатающую битовое представление своего аргумента типа `double`.
- Упр.3.2.+ Напишите класс хэш-таблицы, запоминающий и возвращающий записи с ключом-строкой символов. Постройте общедоступные функциональные элементы для вставки и удаления записей из хэш-таблицы. Спрячьте дета-

ли реализации в приватную часть класса.

- Упр.3.3.** Реализуйте тип хэш-таблицы из предыдущего упражнения с использованием структуры бинарного дерева без изменения объявлений функций вставки, перекодировки и удаления. Получим ли мы в результате удовлетворительные хэш-таблицы?
- Упр.3.4.+** Покажите, как отслеживание входа и выхода блока во время выполнения можно реализовать с помощью конструкторов и деструкторов.

Глава 4

АБСТРАКЦИЯ ДАННЫХ



Абстрактный тип данных - это инкапсулированный тип данных, доступ к которому осуществляется только через интерфейс, скрывающий детали реализации. Свойства абстрактного типа данных определяются его интерфейсом, а не его внутренней структурой или реализацией. Один и тот же абстрактный тип данных, таким образом, можно реализовывать по-разному, не влияя на фрагменты кода, его использующие. Именно в этом состоит смысл абстрактности типа данных: свойства типа определяются интерфейсом, а от деталей реализации абстрагируются.

Для абстракции данных в C++ используются классы, скрывающие реализацию типа в приватной части своего определения и предлагающие интерфейс из общедоступных операций. В этой главе мы представляем несколько примеров реализации абстрактных типов данных с помощью классов C++ и обсуждаем вопросы разработки классов для абстракции данных.

4.1. Комплексные числа.

Класс комплексных чисел, представленный в Главе 1, является хорошим примером абстрактного типа данных.

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

```
public:
    friend complex operator + (complex,complex);
    // ...
    friend complex operator / (complex,complex);
    complex (double=0.0, double=0.0);
};
```

Здесь мы решили представлять комплексное число его координатами в прямоугольной системе на комплексной плоскости.

В будущем, тем не менее, мы можем решить изменить реализацию, чтобы использовать полярную систему координат. Представление, при помощи которого обеспечивается реализация комплексных чисел, скрыто от общего доступа в приватной части определения класса; поэтому мы можем произвести это изменение, не влияя на программы пользователя. Общедоступный интерфейс определяет свойства типа *complex*. Пока этот интерфейс поддерживается, никакие изменения реализации не влияют на программы пользователя.

```
class complex {
    double theta;
    double r;
public:
    friend
    complex operator* (complex a,complex b) {
        complex result;
        result.r=a.r*b.r;
        result.theta=a.theta+b.theta;
        return result;
    }
    // ...
};
```

Так как абстрактный интерфейс определяет для пользователя семантику типа, основной задачей при создании абстрактного типа данных является разработка интерфейса. Эта разработка должна учитывать абстрактные свойства типа, а не просто предохранять пользователя от изменения реализации. В этом разница между защитой, описанной в

Главе 3 и абстракцией данных. Интерфейс, защищающий реализацию типа от несанкционированного доступа, не скрывая при этом его структуры, провоцирует создание программы, использующей его с установлением зависимостей от реализации.

Например, если интерфейс нашего типа *complex* открывает, что реализация - пара *double*, пользователи *complex* могут написать программу, предполагающую, что комплексное число - пара прямоугольных координат.

```
class complex {  
    // ...  
    double first*() { return re; }  
    double second() { return im; }  
};  
extern complex a;  
complex b(a.first(),a.second());
```

Если *complex* реализован как пара координат, то *a* равно *b*. Если же мы изменим реализацию, вводя полярные координаты, *complex* все также представляется парой чисел типа *double*, но *a* уже не обязательно равно *b*.

Для комплексных чисел дополнительная семантика типа встраивается в множество арифметических операций и преобразований из других и в другие числовые типы.

Комплексные числа можно складывать, вычитать, перемножать и делить, поэтому эти свойства реализуются перегрузкой операторов *+*, *-*, *** и */* для поддержки операндов и выработки результатов типа *complex*. Совсем необязательно для реализации этого интерфейса перегружать операторы; "обыкновенные" функции с подходящим разрешением доступа тоже можно использовать для реализации абстрактного интерфейса.

```
class complex {  
    // ...  
public:  
    friend complex add(complex,complex);  
    // ...  
    friend complex div(complex,complex);
```

```
complex(double=0,double=0);
};
```

Применение таких функций для абстрактного типа данных, тем не менее, не приводит к естественному расширению числовых типов и операций, представляемых языком. Сравните:

```
Z = add(add(R,mul(mul(j,omega),L)),
        div(1,mul(mul(j,omega),C)));
```

с намного более читабельным выражением для импеданса цепи переменного тока из Главы 1.

```
Z = R + j * omega * L + 1/(j * omega * C);
```

Предопределенные арифметические операторы предлагают интуитивный интерфейс для класса *complex*, но перегрузка операторов может быть осуществлена неправильно. Так как семантика перегруженного оператора определяется реализатором, можно определить + для вычитания и - для сложения, но при отсутствии злого умысла это маловероятно.

Более частой ошибкой при перегрузке операторов является их неправильное использование. Например, другое свойство комплексных чисел, которое может быть отражено в абстрактном интерфейсе - это экспоненциальность. Так как в C++ нет оператора возведения в степень, для реализации последнего необходимо использовать один из существующих операторов, как, например, ^ (исключающее или), который не имеет смысла для комплексных чисел. Хотя это и возможно, однако такое использование перегрузки операторов вызовет ошибки и неправильное прочтение. Причина - несоответствующий возведению в степень приоритет ^. Пользователь класса *complex* может записать выражение $-1+e^{i\pi}$ как $-1+e^{(i*\pi)}$, предполагая, что оператор возведения в степень более сильный, чем сложение, как это сделано в большинстве языков со встроенным экспоненциальным оператором. Вспомним, что перегрузка не изменяет суще-

ствующих приоритетов и ассоциативности операторов, поэтому выражение $-1 + e^{(i \cdot \pi)}$ интерпретируется как $(1 + e)^{i\pi}$, так как $^$ имеет приоритет ниже $+$. В этом случае лучше заменить перегрузку оператора более прозаическим, но и более понятным использованием неоператорной функции.

```
-1 + pow(e, i * pi);
```

Перегрузка операторов может использоваться только, если существующий приоритет и семантика оператора поддерживает интуитивное понимание его нового использования.

Как мы отмечали в Главе 3, к приватным частям определения класса имеют доступ два типа функций: функции, являющиеся элементами класса, и функции, специально объявленные как дружественные. Почему же мы реализовали арифметические операции над комплексными числами как дружественные функции, а не как элементы класса?

```
class complex {
    // ...
public:
    // операторы-элементы
    complex operator +(complex); // бинарный
    complex operator -(complex); // бинарный
    complex operator -(); // унарный
    // ...
};
```

Причина существует в способе взаимодействия комплексных чисел с другими арифметическими типами в смешанных выражениях.

Конструктор для класса *complex* (который, как и операторные функции, является частью интерфейса) играет двойную роль. Кроме инициализации каждого объекта типа *complex*, он также специфицирует преобразование значения типа *double* в значение типа *complex*. Существуют предопределенные преобразования других арифметических

типов в `double`; поэтому конструктор также определяет преобразование в `complex` других предопределенных арифметических типов. Конструктор вызывается, если необходимо произвести преобразование при присваивании или инициализации объекта `complex` предопределенным арифметическим типом.

```
complex x = 12.34; // complex(12.34,0)
x = 12; // complex((double)12,0)
```

В этом случае конструктор, если необходимо, вызывается для преобразования при инициализации формальных аргументов функции фактическими аргументами вызова.

```
complex add(complex,complex);
complex a,b;
double c,d;
// ...
add(a,b);
add(a,d); // add(a,complex(d));
add(c,b); // add(complex(c),b);
```

Операторная функция не сильно отличается от неоператорной, и может быть вызвана и как инфиксный оператор, и как неоператорная функция.

```
complex operator +(complex,complex);
a + b;
operator +(a,b); // то же, что и выше
a + d;
operator +(a,d);
c+b
operator +(c,b);
```

Попробуем написать ту же последовательность выражений с использованием операторной функции-элемента класса.

```
a + b;
a.operator +(b); // отлично...
a + d;
```

```
a.operator +(d); // отлично...  
c + b;          // ошибка!  
c.operator +(b); // ошибка!
```

Затруднения со сложением `b` и `c` в приведенном примере состоит в том, что мы пытаемся вызвать операторную функцию элемент `c`, однако `c` - не классового типа и не имеет элементов! И в записи `c+b`, и как `c.operator+(b)`, выражение не имеет смысла. Если мы реализуем комплексные операторные функции как элементы класса, пользователи нашего типа никогда не смогут написать выражение, в котором первым операндом комплексного оператора будет некомплексный элемент, не используя при этом явного преобразования.

Реализация операций с комплексными числами перегрузкой существующих операторов и поддержка инициализации и преобразований с помощью конструктора, позволяет нам расширить систему арифметических типов C++ за счет включения комплексных чисел, которые можно также легко и естественно использовать, как и встроенные арифметические типы.

4.2. Строки.

В качестве следующего примера абстракции данных рассмотрим модифицированную версию типа данных *String* из Главы 3.

```
class String_rep {  
    char *str;  
    int refs;  
    String_rep( char s);  
    friend class String;  
};  
  
class String {  
    String_rep *r;  
public:  
    friend String operator + (String,String);  
    friend int operator  (String,String);  
    // другие операторы...
```

```
String &operator = (String);
operator char *();
String(char * = "");
String(String &);
~String();
};
```

В этой версии класса *String* мы решили расширить фактические строки символов настолько, насколько это возможно, и создали класс *String_rep*, объединяющий символьную строку и счетчик обращений. Класс *String* теперь ссылается на эту структуру данных вместо того, чтобы ссылаться напрямую на строку символов.

Первый конструктор очень прост; он специфицирует способ инициализации *String* символьной строкой (и, как и конструктор для класса *complex*, определяет преобразование символьной строки в *String*).

```
String::String(char *s) {
    r = new String_rep(s);
}
```

Класс *String_rep* сам позаботится о своей инициализации.

```
String_rep::String_rep(char *s) {
    str = new char[strlen(s)+1];
    strcpy(str,s);
    refs = 1;
}
```

Когда один *String* инициализируется другим, вместо создания второй копии символьной строки, мы ссылаемся на существующее представление и увеличиваем его счетчик обращений. Мы определяем такую семантику конструктором, получающим аргумент *String&*.

```
String::String(String &init) {
    r = init.r;
    r-refs++;
}
```

Семантика присваивания аналогична, но мы должны позаботиться о том, чтобы *String_rep*, на который ссылается левый операнд (цель) присваивания, получил свое новое значение. Счетчик обращений *String_rep* уменьшается на 1, потому, что присвоенный *String* на него больше не ссылается. Если не осталось ссылок, *String_rep* удаляется.

```
String &  
String::operator =(String str) {  
    if (!--r-refs)  
        delete r;  
    r = str.r;  
    r-refs++;  
    return *this;  
}
```

Для симуляции поведения встроенного оператора присваивания, оператор присваивания *String* не только изменяет объект, которому происходит присваивание, но и возвращает значение объекта. Указатель *this* используется для доступа к объекту, чтобы вернуть его значение.

Следует иметь в виду существующую разницу в семантике присваивания и инициализации для *String* и комплексных чисел. В случае нашей реализации комплексных чисел, предопределенная семантика присваивания и инициализации является достаточной, и представление одного комплексного числа покомпонентно копируется в другое. Для правильной реализации *String*, счетчик обращений в *String_rep*, на который ссылается *String*, необходимо при присваивании модифицировать. Поэтому для класса *String* необходимо реализовать *operator =*.

В общем случае, если необходимо для абстрактного типа данных реализовать присваивание, неплохо реализовать также инициализацию, и наоборот. В типе *String* предоставление одного без другого влечет такие же трудности, как и отсутствие того и другого. Вместе эти операции определяют, как объекты данного классового типа должны копироваться во всех ситуациях. Более подробно семантику копирования мы рассмотрим в Главе 7.

Мы также должны позаботиться об установке счетчиков обращений для *String*-ов, которые удаляются или выходят из области видимости. Для этого определим деструктор.

```
String::~~String() {  
    if (!--r-refs)  
        delete r;  
}
```

Как и в случае комплексных чисел, мы организуем перегружаемые операторы и конструктор так, чтобы они хорошо сочетались с существующей в C++ системой типов. Один из конструкторов реализует преобразование символьных строк в *String*-и, а операция конкатенации реализуется перепреопределением оператора `+` как дружественного для реализации комплексных арифметических операций.

```
extern char *home_dir, *path, *file;  
String home = home_dir;  
String fpath = home + "/" + path + "/" + file;
```

Для завершения связи типа *String* с существующей системой типов, реализуем оператор, определяющий преобразования из *String* в `char*`. В этой версии оператора преобразования, в отличие от предложенной в Главе 3, мы выбрали более эффективный, но потенциально более опасный подход с возвращением адреса строки символов вместо адреса копии строки.

```
String::operator char*() {  
    return r-str;  
}  
// ...  
char *newfile = fpath; // fpath.operator char *()
```

Это преобразование по сути реализует функцию, обратную конструктору. Конструкторы класса *String* определяют как создавать форму *String* из `char*` или другого *String*, тогда как оператор преобразования `operator char*` определяет как

создать `char*` из *String*. Эта возможность важна для полноты связи с существующей системой типов C++; она предоставляется для связи с существующими программами для объектов типа `char*`.

```
extern FILE *fopen(const char *,const char*);
FILE *fp = fopen(fpath,"r+");
```

Как и конструктор, оператор преобразования при необходимости вызывается в выражениях и инициализаторах.

В отличие от перегружаемого оператора `+` для конкатенации, и в отличие от операторов комплексной арифметики, мы реализовали присваивание для *String* как функциональную компоненту класса. Причина в том, что то, что порождают элементы-функции с учетом автоматического преобразования аргументов, есть в точности то, что нужно для присваивания; мы не хотим применять преобразования к левому аргументу присваивания.

Если `operator=` реализован как дружественная функция, будет возможным присваивать *String*-и символьным строкам, так как `char*` в левой части оператора присваивания будет автоматически преобразован в *String* конструктором *String*, принимающим аргумент `char *`.

```
"/usr/bin" = pathname
```

Вызов инфиксного оператора, который мы использовали - это только принятая запись, и он полностью эквивалентен вызову функции.

```
operator ("/usr/bin",pathname);
```

Поэтому, даже если мы, возможно, не хотим позволить присваивания символьной строке типа `"/usr/bin"`, это тем не менее корректно, если `operator=` не является элементом класса *String*. Если сделать `=` функциональным элементом, присваивание, приведенное выше, будет иметь тот же смысл, что и

```
"/usr/bin".operator =(pathname);
```

то есть никакого. Реализация присваивания *String* как функциональной компоненты предотвращает применение таких преобразований к левой части оператора присваивания. Более разумная семантика присваивания или инициализации *char** элементами элементов типа *String* уже производится оператором преобразования.

Правильное использование операторных функций, конструкторов и операторов преобразования позволяет нам разрабатывать абстрактные типы данных, соответствующие системе предопределенных типов C++ и расширяющие ее.

4.3. Упорядоченные последовательности.

Абстракция данных дарит две основных, одинаково важных, возможности тем, кто ее использует. Во-первых, она упрощает семантику использования типа, ограничивая операции теми, которые представлены в общедоступном интерфейсе. Пользователи типа не должны бороться с семантическими особенностями, зависящими от реализации и не имеющими значения для абстрактной функции типа. Например, пользователи типа *String* не должны заботиться об обновлении счетчиков обращений при присваивании или о переполнении буферов при конкатенациях. Эта "вторичная" семантика - забота скрытой реализации и ответственность за это возлагается на реализатора типа. По этой же причине реализатор волен изменить реализацию, не боясь повлиять на программы пользователя до тех пор, пока сохраняется семантика. Также важна предоставляемая абстракцией данных возможность сделать язык программирования ближе к конкретной проблемной области. Возможность определения типов данных *complex* и *String* дает использующему эти типы возможность писать понятные, компактные программы, базирующиеся на комплексной арифметике и манипулировании строками, в результате чего язык C++ расширяется, сочетая возможности думать и программировать в этих проблемных областях. Это использование абстракции данных для поддержки кон-

цептуальной абстракции для разработки программ наиболее важно.

Комплексные числа и строки - это типы данных с широкой областью использования и на самом деле являются встроенными типами во многих языках программирования. Хорошо разработанные абстрактные типы данных, тем не менее, могут дать аналогичные возможности и в более специализированных прикладных областях.

Например, для некоторых приложений может потребоваться тип с семантикой упорядоченной последовательности целых. Используя ту же технику, что и для *complex* и *String*, мы определим общедоступный интерфейс для такого типа.

```
typedef int ETYPE;

class sorted_collection {
public:
    sorted_collection();
    void insert(ETYPE);
    void apply(void (*)(ETYPE));
};
```

Этот класс представляет простую концепцию и поэтому имеет простой интерфейс: есть операции создания *sorted_collection*, вставки нового целого в последовательность, и применения функции к каждому элементу последовательности в определенном порядке.

После разработки интерфейса пользователи типа могут начать разработку и программирование прикладных программ.

```
#include "collection.h"

printInt() {
    // считывает, сортирует и печатает целые числа
    extern void print(int);
    extern int read(int &);
    sorted_collection sc;
    int i;
```

```
while (read(l))
    sc.insert(l);
sc.apply(print);
}
```

Пока пользователи работают с интерфейсом *sorted_collection*, мы можем откомпилировать его первую реализацию.

```
typedef int ETYPE;

class sorted_collection {
    ETYPE ary[100];
    int free;
public:
    sorted_collection() { free = 0; }
    void insert(ETYPE);
    void apply(void (*)(ETYPE));
};

void
sorted_collection::apply(void (*)(ETYPE)) {
    for (int i=0; i!=free; i++)
        f(ary[i]);
}

void
sorted_collection::insert(ETYPE el) {
    for (int i=free++; i && ary[i-1] < el; i--)
        ary[i] = ary[i-1];
    ary[i] = el;
}
```

Эта реализация, понятно, не слишком хороша. Алгоритм вставки неэффективен для больших последовательностей. Эта неэффективность, правда, вряд ли вызовет проблемы, ибо, как ранее упоминалось, большая последовательность целых перескочит массив для последовательности фиксированного размера и разрушит программу. Единственный позитивный момент этой реализации - это то, что ее можно написать за несколько минут. Этого достаточно, так как теперь пользователи типа могут компилировать и начинать

отлаживать свои прикладные программы в то время, пока мы работаем над получением лучшей реализации. Быстрые реализации, похожие на эту, также полезны на начальных этапах разработки программного проекта, когда общедоступный интерфейс абстрактного типа данных еще не зафиксирован. Таким образом, пользователи могут экспериментировать с типом и модифицировать его интерфейс, не опасаясь свести на нет дорогостоящую программу.

Изменим теперь нашу реализацию.

```
typedef int ETYPE;

class tree {
    ETYPE el;
    tree *lchild, *rchild;
    tree(ETYPE l) { el = l; lchild = rchild = 0; }
    void insert(ETYPE);
    void apply(void (*)(ETYPE));
    friend sorted_collection;
};

void
tree::insert(ETYPE l) {
    if (l)
        if (lchild)
            lchild->insert(l);
        else
            lchild = new tree(l);
    else
        if (rchild)
            rchild->insert(l);
        else
            rchild = new tree(l);
}

void
tree::apply(void (*)(ETYPE)) {
    if (lchild)
        lchild->apply(f);
    f(el);
    if (rchild)
        rchild->apply(f);
}
```

```
class sorted_collection {  
    tree *root;  
public:  
    sorted_collection() { root=0; }  
    void insert(ETYPE el) {  
        if (root)  
            root->insert(el);  
        else  
            root = new tree(el);  
    }  
    void apply(void (*f)(ETYPE))  
    { if (root) root->apply(f); }  
};
```

Наша вторая реализация лучше первой, так как может оперировать большими последовательностями и вставка элементов реализована гораздо более эффективно. Если пользователи не возмутятся одним из ограничений первой реализации (и "бомбой"), они не заметят разницы между реализациями. Каждая реализация - это другое представление одного и того же абстрактного типа; при этом абстрактная семантика не меняется от одной реализации к другой.

Это не значит, что эта "вторичная" или зависящая от реализации семантика не влияет значительно на поведение программ, использующих тип. Наша начальная реализация *sorted_collection* использует неявное допущение, что ни одна программа пользователя не попытается вставить в данную последовательность более 100 целых. Так как это ограничение не представлено в абстрактном интерфейсе, пользователи типа не знают об этом и могут разрушить защиту.

Более хитрыми являются ситуации, в которых программы обходят зависящую от реализации семантику, не являющуюся явной частью абстрактной семантики типа. Например, мы можем создать тип данных последовательность (чьи элементы не обязаны быть упорядоченными) из упорядоченной последовательности.

```
typedef sorted_collection collection;
```

К несчастью, пользователь типа может написать программу, учитывающую факт, что последовательность оказалась упорядоченной. Позже мы можем реализовать *collection* для большей эффективности вставок по сравнению с *sorted_collection* по-другому, без упорядочения, но при этом эффективно повреждая программы, зависящие от конкретной семантики реализации.

Поэтому часто полезно рассматривать общедоступный интерфейс абстрактного типа данных как договоренность между реализатором и пользователем типа. От реализатора требуется предоставление корректной абстрактной семантики, специфицированной интерфейсом, не делая никаких дополнительных предположений, а пользователи типа должны учитывать только явно представленную в общем интерфейсе семантику.

4.4. Общность.

Упорядоченная последовательность целых - полезный для конкретных приложений тип данных, но таковыми являются и упорядоченные последовательности *double*, *String* и собственно упорядоченные последовательности. Ясно, что абстракция, с которой мы оперируем, - это упорядоченная последовательность вообще, а не упорядоченная последовательность объектов какого-то типа. Хотелось бы параметризовать нашу реализацию *sorted_collection* и затем конкретизировать или создавать ее варианты для элементов конкретных типов. Таким образом, единственная параметризованная реализация *sorted_collection* может служить общим представлением семантики упорядоченной последовательности. Каждая версия произведет новый тип упорядоченной последовательности для конкретного типа элемента. Пользователи затем смогут объявлять и использовать объекты этих конкретизированных типов.

К сожалению, язык C++ не поддерживает пока концепцию родовых или параметризованных типов. Тем не менее, можно получить большинство потенциальных возможностей параметризуемых типов с использованием других черт языка. Рассмотрим нашу вторую реализацию

sorted_collection. Мы фактически реализовали тип для работы с *ETYPE*-элементами, где *ETYPE* определен при помощи `typedef` как `int`. Какие особенности *ETYPE* использует наша реализация? Просматривая программу, мы находим, что для *ETYPE* должны быть определены операторы `=` и `,` и должна быть возможность инициализации одного *ETYPE* другим (чтобы инициализировать формальный аргумент *ETYPE* указателя на функцию в `sorted_collection::apply`).

Просто изменив `typedef` для определения некоторого другого типа *ETYPE* с этими свойствами, мы получим в результате версию *sorted_collection* для данного *ETYPE*.

```
typedef String ETYPE;
class sorted_collection {
    // ...
};
printString() {
    // сортирует и печатает вводимые предложения,
    // удаляя нежелательный текст
    extern void print(char *);
    extern void censor(char *);
    extern char *readstr();
    sorted_collection list;
    char *s;
    while (s=readstr())
        list.insert(s);
    list.apply(censor);
    list.apply(print);
}
```

(Обратите внимание, как работают описанные ранее в этой главе преобразования из *String* в *char** и обратно).

Эта схема работает, если нам необходима упорядоченная последовательность не более, чем одного типа, но ее нельзя использовать для нескольких реализаций. *ETYPE* не может представлять два типа одновременно! Единственное, что может помочь, это копирование реализации *sorted_collection* и редактирование каждой копии для получения разных версий реализации для элементов различных типов. Обычно для редактирования используется препроцессор.

Один из стандартных заголовочных файлов C++ `generic.h`, содержит препроцессор макроопределений для проведения этих операций редактирования. Макросы предназначены для связывания имен и для объявления и определения родовых типов. Одна проблема при определении различных примеров родового типа по единому образцу состоит в создании уникальных имен для каждого конкретного типа. Мы используем макрос `name2` из `generic.h` для решения этой задачи.

```
#define sorted_collection(ETYPE) \
    name2(ETYPE,sorted_collection)
#define tree(ETYPE) name2(ETYPE,tree)
```

Здесь мы создали уникальные имена для наших классовых типов при помощи связывания параметра типа с именами родовых типов. Например, текст `tree(String)` будет с помощью макроса `tree` преобразован в `Stringtree`. Реализация макроса `name2` проста, но может отличаться у разных препроцессоров. Для препроцессора ANSI C, `name2` может быть реализовано оператором конкатенации.

```
#define name2(a,b) a##b
```

Для конкретизации новой версии родового типа `generic.h` предлагает макрос `declare`.

```
#define declare(a,t) name2(a,declare)(t)
```

Пользователь, создавая тип "упорядоченная последовательность *String*-ов", может использовать его следующим образом:

```
declare(sorted_collection,String);
```

После расширения получим:

```
sorted_collectiondeclare(String);
```

`sorted_collectiondeclare` - это еще один макрос. Как поставщики родового типа `sorted_collection`, мы должны определить этот макрос.

```
#define sorted_collectiondeclare(ETYPE) \
class tree(ETYPE) { \
    ETYPE el; \
    tree(ETYPE) *l,*r; \
    tree(ETYPE)(ETYPE i) { el=i;l=r=0; } \
    void insert(ETYPE); \
    void apply(void (*)(ETYPE)); \
    friend sorted_collection(ETYPE); \
}; \
class sorted_collection(ETYPE) { \
    tree(ETYPE) *root; \
public: \
    sorted_collection(ETYPE) { root=0; } \
    void insert(ETYPE el) \
        { if (root) root->insert(el); } \
    void apply(void (*)(ETYPE)) \
        { root->apply(f); } \
};
```

Когда макрос `sorted_collectiondeclare` расширяется своим аргументом-типом, аргумент замещает *ETYPE* и макросы, определенные нами ранее, на `sorted_collection` и `tree`; по необходимости вызываются для создания уникальных имен классов для родовых `sorted_collection` и `tree`. Пользователи теперь могут порождать версии нашего родового класса упорядоченной последовательности и объявлять объекты конкретизированных типов.

```
#include generic.h
#include "collection.h"
#include "string.h"

declare(sorted_collection,int);
declare(sorted_collection,String);

printint() {
    // ...
    sorted_collection(int) sc;
```

```
int i;
while (read(i))
    sc.insert(i);
sc.apply(print);
}

printStats() {
    // ...
    sorted_collection(String) list;
    char *s;
    while (s=readstr())
        list.insert(s);
    list.apply(censor);
    list.apply(print);
}
```

В добавление к определениям класса для каждой конкретизированной версии *sorted_collection*, сгенерированной однажды для каждого файла, в котором они используются, соответствующие определения функциональных компонент *tree::insert* и *tree::apply* должны быть сгенерированы ровно один раз в программе для каждого конкретизированного типа. Для этого мы поступим также, как и с определениями классов, однако нужно убедиться, что мы определяем эти функции в программе только однажды для каждого типа-параметра. Чтобы упростить определение таких параллельных операций, *generic.h* предоставляет макрос *implement*, функция которого идентична *declare*.

Поступая аналогично, можно определить и использовать родовые типы, требующие больше одного параметра-типа для конкретизации. Помните, что *generic.h* предоставляет *name3*, *declare2* и прочие макросы для создания и использования таких типов.

4.5. Абстракция управления.

Функция *apply* класса *sorted_collection* фактически является композицией двух различных концепций: отслеживание любой структуры данных, использованной для хранения элементов *sorted_collection*, и применение функции к элементу последовательности. Обе эти концепции,

как и их композиция (как в функции `apply`), принадлежат к общему классу абстракций, которые мы называем абстракциями управления. Отслеживание скрытого представления структуры данных без всяких сопровождающих операций используется, конечно, не часто. Если отслеживание можно разделить на стадии, например, каждая из которых дает некоторое интересное значение, тогда пользователи типа могут получать значения, связанные некоторым образом с типом так, что предохраняется приватность скрытой реализации. Управляющая абстракция такого типа называется итератор.

Обратимся к реализации типа список. У списка есть голова, хвост и последовательность элементов между ними. Мы хотим разработать механизм доступа к значениям элементов списка по порядку.

```
struct node {  
    node *next;  
    ETYPE el;  
    node(ETYPE l,node *n) { el=l;next=n; }  
};
```

```
class list {  
    node *hd;  
public:  
    list(node *n=0) { hd=n; }  
    list(list &seq) { hd=seq.hd; }  
    void Insert(ETYPE l)  
        { hd = new node(l,hd); }  
    friend ETYPE head(list seq)  
        { return seq.hd-el; }  
    friend list tail(list seq)  
        { return list(seq.hd-next); }  
    friend int lsempy(list seq)  
        { return seq.hd == 0; }  
};  
void  
print_list(list seq) {  
    extern void print(ETYPE);  
    for (list s=seq; lsempy(s); s=tail(s))  
        print(head(s));  
}
```

Это полезная абстракция, если вы привыкли рассматривать списки рекурсивно, то есть список - пустой, либо же это элемент, за которым следует список. В нашем случае мы хотим рассматривать списки итеративно, то есть как последовательность элементов списка, а не как рекурсивно определенную последовательность списков. Хотя приведенная выше абстракция эффективна, она позволяет получить доступ к элементам только последовательно, не поддерживая наш способ рассмотрения списков, поэтому мы пробуем другую версию:

```
class list {
    ETYPE el;
    list *link;
public:
    list(ETYPE l, list *next)
        { el=l; link=next; }
    list *next() { return link; }
    ETYPE value() { return el; }
};

void
print_list(list *seq) {
    extern void print(ETYPE);
    for (list *p=seq; p; p=p-next())
        print(p-value());
}
```

Эта вторая реализация, конечно, итеративна по своей природе, но мы как-то потеряли целостность нашей абстракции, а принимаем во внимание только элементы списка.

Вот другая версия:

```
class node {
    node *next;
    ETYPE el;
    node(ETYPE, node *);
    friend list;
    friend iter;
};

class list {
```

```

    node *hd;
public:
    llist();
    void Insert(ETYPE);
    friend Iter;
};
class Iter {
    node *current;
public:
    Iter(list);
    ETYPE *operator() ();
};
void
print_list(list llist) {
    extern void print(ETYPE);
    Iter next=llst;
    ETYPE *p;
    while (p=next()) print(*p);
}

```

Здесь мы создали тип итератор, связанный с типом список, точно также включающий в себя абстракцию управления (последовательный просмотр элементов списка), как и тип список включает абстракцию структуры данных списка. Для перебора элементов списка мы создали итерационный объект и встроили его в списочный объект в инициализации.

```

Iter::Iter(list llist) {
    current=llist.hd;
}

```

Итерационный объект сам по себе сохраняет состояние итерации от обращения к обращению.

```

ETYPE *
Iter::operator() () {
    if (current)
        node *tmp=current;
        current=current-next;
        return &tmp-el;
    }
    return 0;
}

```

Этот итератор имеет единственную операцию: он получает следующий элемент списка. В связи с этим мы решили использовать перегружаемый оператор `()` для выражения этой операции, но мы также могли перегрузить `++` или использовать неоператорную элемент-функцию, или даже дружественную функцию, не являющуюся элементом.

```
class Iter {  
    // ...  
public:  
    // ...  
    ETYPE *operator ()(); // next()  
    ETYPE *operator ++(); // next++ или ++next  
    ETYPE *next();       // next.next()  
    friend ETYPE *nxt(iter); // nxt(next)  
};
```

Основная идея состоит в том, что итераторный объект имеет концепцию сохранения от обращения к обращению. Эта точка зрения на список - как раз то, что нам нужно. Мы сохранили концепцию списка как "вещи в себе", но имеем возможность естественным образом проследовать по элементам списка. Итератор в результате, - это расширение класса список, предоставляющее абстрактную операцию поточного управления. Это создание абстрактных типов поточного управления, использующего вместе с абстрактными типами данных мощный механизм для работы с абстрактными типами со сложной внутренней структурой в реализационно-независимом стиле.

Итератор для нашего класса список очень простой; с каждым вызовом он дает следующий элемент списка до тех пор, пока элементов не останётся. Мы, конечно же, могли реализовать более сложную семантику. Например, создание итератора для списка может иметь побочный эффект блокировки списка, так что пока итерация не завершилась, новые элементы не могут быть вставлены. Мы могли определить более общее продвижение по списку. Например, мы могли добавить в класс *iter* сканирующую функцию, возвращающую следующий элемент списка, удовлетворяющий

как аргумент предикатной функции,

```
class employee {
    // ...
public:
    int ismgr();
    int istrue() { return 1; }
};

typedef employee ETYPE;

class list {
    // ...
};

class iter {
    // ...
public:
    ETYPE *operator ();
    ETYPE *operator ()(int (ETYPE::*)());
};

void
clean_up(list alist,int all) {
    extern void print(employee);
    extrn void fire(employee);
    iter next = alist;
    employee *e;
    while (e=next())
        print(*e);
    int (employee::*predicate)()
        = all ? employee::istrue : employee::ismgr;
    iter next_victim = alist;
    while (e = next_victim(predicate))
        fire(*e);
}
```

или мы могли добавить возможность возврата к предыдущему элементу, или к голове списка. Необходимо помнить, что итераторный объект может быть присвоен другому итераторному объекту или передаваться функции как аргумент, и что для одного списка может быть несколько одновременно активных итераторных объектов. Для одного

и того же абстрактного типа, (как и нескольких объектов одного итераторного типа), можно создать несколько типов итераторов. Тип данных дерево может определять различные итераторы для обхода слева, справа, сначала вширь и сначала вглубь.

Две центральные идеи абстракции управления такие же, как и уже обсужденные нами ранее в этой главе для абстракции данных вообще. Во-первых, программист, использующий абстракцию управления, не интересуется деталями ее реализации. В добавок к сокращению объема информации, которую пользователь должен охватить прежде, чем он сможет написать простую структуру поточного управления, реализатор типа может изменить его реализацию (и реализацию связанных с типом абстракций управления), не влияя на код пользователя.

Во-вторых, абстракция управления приближает язык программирования к аспектам управления задачи точно также, как абстракция данных делает это для аспектов типа. Это имеет особенное значение для более сложных ситуаций, в которых концепция того, что происходит, может быть затеряна в деталях реализации.

Некоторые приемы так очевидны, что их можно не заметить, как, например, возможность поддержки элементов класса только для чтения. В некоторых ситуациях реализатор может хотеть представить окно в приватной части реализации типа, или реализатор может хотеть, чтобы пользователи типа проэкзаменовали часть общедоступного интерфейса, не изменяя ее. Эта возможность может быть реализована подставляемыми (inline) функциями.

```
class node {
    node *n;
    ETYPE val;
public:
    // ...
    node *next() { return n; }
    ETYPE value() { return val; }
};
// ...
for (node *p=head; p; p=p-next())
    print (p-value());
```

В этом случае реализация - это абстракция, но мы не хотим, чтобы пользователь типа *Node* мог изменять структуру списка или значения элементов.

Другой простой прием управления представляется аппликаторами. Аппликатор - это функция, применяющая один из своих аргументов к другим.

```
void apply (char *s, void(*f)( char *)) {  
    f(s);  
}
```

Это простой аппликатор, применяющий свой аргумент функцию (указатель) к своему аргументу - символьной строке.

```
String pathname="/usr/cmd";  
extern void exec(char *);  
extern char *cmd;  
apply(pathname+"/"+cmd,exec);
```

Конечно, мы могли реализовать это более эффективно, вызывая функцию напрямую, зачем же нужны аппликаторы?

Мы уже видим, как используются аппликаторы в классе *sorted_collection*, хотя и не в явной форме. Функциональный элемент *apply* - это композиция отслеживания и аппликатора, позволяющая пользователям *sorted_collection* применять функцию к каждому элементу последовательности. Другие использования аппликаторов мы описываем в Главе 8.

Понятно, что нет явной разделительной линии между абстракциями данных и управления или между абстрактными и неабстрактными типами данных. Она и не нужна. Парадигма абстракции данных - это не жесткий формализм, определяющий "как надо писать программное обеспечение". Скорее есть возможность развязать ваше воображение при решении, какими типами данных оперирует ваша задача. Целые, комплексные числа, строки, списки, коммуникационные сети, телефоны и грамматики - все

имеют (во всяком случае в том, что касается программы) абстрактные свойства, которые могут быть приведены в общедоступном интерфейсе к абстрактному типу данных. Абстракция данных также располагает к созданию новых реализаций этих типов и новых управляющих структур, чтобы сделать работу с ними естественной.

4.6. Упражнения.

- Упр. 4.1. Измените реализацию класса *complex* для использования полярных координат вместо пар (действительное, мнимое). Как это изменение повлияет на код пользователя?
- Упр. 4.2. Разработайте комплексный тип, аналогичный классу *complex*, но использующий *float* вместо *double* для представления. Сделайте его совместимым с существующим классом *complex* и предопределенными арифметическими типами так, чтобы короткие комплексные числа можно было использовать в смешанных выражениях с комплексными числами или другими арифметическими типами. Можете ли вы сделать полярное представление из предыдущего упражнения способным работать с неполярным представлением?
- Упр. 4.3. Используйте наследование для реализации обычных и коротких комплексных чисел из общего родового класса.
- Упр. 4.4. Рассмотрите абстрактный тип данных дата со следующим интерфейсом:

```
class Date {  
public:  
    const char *str();  
    Date(char *);  
    Date(int,int,int);  
    Date &operator ++();  
    Date &operator --();  
};
```

Date иницируется или строчным представлением даты, как например, 17 июня 1775 года, или тремя целыми, представляющими день, месяц и год: 17, 6, 1775. Оператор преобразования возвращает указатель на представление даты строкой символов. Операторы ++ и -- соответственно увеличивают и уменьшают дату на один день. Предложите три различные реализации абстрактного типа данных, используя одно целое, три целых и символьную строку для реализации внутренней формы даты. Каковы сравнительные достоинства каждой реализации? В чем преимущества использования единого абстрактного типа данных для всех трех реализаций?

Упр. 4.5. Большинство вычислений, связанных с физическими системами, не составлены исключительно из скалярных операндов, но также содержат операнды со связанными физическими единицами. Например, вычисления с элементами в омах, фарадах и т.д. Полезной проверкой при проведении таких вычислений является поддержка алгебраических операций над типами физических единиц, как и над численными значениями операндов. Вычисленный тип физической единицы должен соответствовать ожидаемому типу результата. Разработайте целый, с плавающей запятой и комплексный типы данных, которые содержат связанные физические единицы. Перегрузите арифметические операторы для вычисления арифметического результата и физической единицы результата. Предложите преобразования между типами и предопределенными и комплексными арифметическими типами.

Упр. 4.6. Разработайте тип данных "ориентированный граф" и рассмотрите следующие вопросы: скрыто ли представление? является ли интерфейс достаточно маленьким и ясным, как это возможно? И, с другой стороны, достаточно ли он

богат, чтобы позволить пользователю типа писать полезные процедуры?

- Упр. 4.7. Напишите итератор для абстрактного типа данных граф из Упр. 4.6., который рассматривает последовательность узлов для множества деревьев при просмотре графа вглубь. (Намек: обратите особое внимание на то, как вы записываете состояние итерации. Сможете ли вы пройти очень большие графы?)
- Упр. 4.8. Разработайте тип данных "автомат с конечным числом состояний". Что представляет из себя множество инициализаторов для этого типа (функциональный указатель на функцию, возвращающую описание, входной файл описаний). Какие ошибки могут быть выявлены при конструировании типа? Как можно использовать такой тип? Может ли единственное представление служить и для абстракции ориентированного графа, и для такого автомата? Почему и в чем относительные преимущества единственного и двух разных представлений?
- Упр. 4.9. Подумайте о полезном типе данных, не являющемся встроенным ни в одном из языков программирования, и реализуйте его как абстрактный тип данных в C++. Напишите программу, использующую этот тип данных.
- Упр. 4.10. Представьте, что вы разрабатываете портативный компилятор. Как вы отделите общие части компилятора от машинно-зависимых частей? Можете ли вы представить интерфейс этих машинно-зависимых частей процесса компиляции как абстрактный тип данных?
- Упр. 4.11. C++ не проверяет автоматически во время выполнения ситуацию выхода индекса за границы

массива. Разработайте родовой тип массива, выполняющий проверку границ.

- Упр. 4.12. Перепишите тип *String* без счетчиков обращений, не изменяя абстрактный интерфейс. Является ли конструктор *String(String&)* все еще необходимым? А *operator=*?
- Упр. 4.13. Разработайте тип данных "гистограмма" перегрузкой *operator[]* для индексации с плавающими аргументами.
- Упр. 4.14.+ Разработайте тип "ассоциативный массив для словаря" (выражающий соответствие *String-String*). Сделайте тип родовым, снабжая параметрами типа индекс и элемент массива. Проиллюстрируйте примером версию типа, индексирующего словарь *String*-ами. Какие еще полезные типы могут быть получены из этого родового ассоциативного списка?
- Упр. 4.15. Реализуйте общий итератор списка, который имеет операции возвращения следующего элемента, предыдущего элемента и головы списка. Реализуйте этот итератор, не изменяя реализации класса *list*.
- Упр. 4.16. Разработайте двухсвязный список, реализация которого использует только один связанный указатель. Включите в вашу разработку итератор, который может проходить список в любом направлении.

Глава 5



НАСЛЕДОВАНИЕ

Абстракция данных - эффективный метод расширения предопределенной системы типов, если можно определить одну, ясно определенную концепцию (например, типы *complex*, *String* и *sorted_collection* из предыдущей главы). Мы иногда можем иметь абстрактный тип данных, который почти, но не полностью тот, что нам нужен, или совокупность типов с одинаковой реализацией или значением, не являющихся идентичными. В таких случаях наследование - это полезный прием для усиления абстракции данных.

Наследование в C++ - это механизм для построения классовых типов из других классовых типов путем определения нового класса как специализации или расширения существующего класса. Эта глава раскрывает механику и использование наследования для кодирования, разработки и абстрагирования.

5.1. Базовые и производные классы.

Класс может наследовать черты функционирования другого класса как производный от него. Класс, выведенный из исходного или базового класса, может добавить или приспособить черты базового для получения специализации или расширения базового типа, или просто повторного использования реализации базового класса.

Например, мы можем рассматривать двусвязный список

как односвязный список с добавочными указателем и операцией доступа к предыдущему элементу так же, как и к следующему. Мы начинаем с класса *список* из предыдущей главы.

```
class list {  
    ETYPE el;  
    list *llink;  
public:  
    list(ETYPE, list *);  
    list *next();  
    ETYPE value();  
};
```

Затем мы используем производный класс для определения двусвязного списка с использованием класса *list* в качестве исходного.

```
class list2 : public list {  
    list2 *llink;  
public:  
    list2(ETYPE, list2 *, list2 *);  
    list2 *previous();  
};
```

Запись

```
class list2 : public list
```

определяет *list2* как новый класс, выведенный из класса *list*. Мы говорим также, что класс *list* является базовым для класса *list2*. *list2* наследует все элементы *list* (данные и функции) и добавляет свои: связь с предыдущим элементом списка, функцию для доступа к этой связи и конструктор.

Реализация функционального элемента *previous* аналогична функции *next* класса *list*.

```
list2 *  
list2::previous() {  
    return llink; }
```

Класс *list2* содержит два различных элемента с именем *link*: исходная связь вперед, унаследованная от базового класса, и обратная связь, объявленная в теле *list2*. *link*, возвращаемый *previous*, - это *link* из *list2*, вследствие влияния наследования на видимость классов.

Производные классы образуют иерархию областей видимости. Область видимости производного класса содержится внутри области видимости его базового блока, во многом аналогично внутреннему блоку функции внутри охватывающего блока. Поэтому когда мы ссылаемся на *link* в элементе-функции *list2*, компилятор сначала будет искать *link*, принадлежавший классу *list2*, и проверит область видимости базового класса только тогда, когда это имя не найдено в производном классе. В случае, если необходимо получить доступ к элементу базового класса, скрытого элементом производного класса, запись *base::member* позволяет специфицировать просмотр в поисках этого имени, начиная с области видимости *base*.

Функция *whatsbefore* получает доступ к наследуемым и обычным элементам совершенно аналогично.

```
list2 *
whatsbefore(list2 *lst, ETYPE e) {
    for (list2 *p = lst; p; p = (list2 *)p->next())
        if (p->value() == e)
            break;
    if (p)
        return p->previous();
    else
        return 0;
}
```

Вложенность областей видимости объясняет причину, почему работает этот код. Только вызов *previous* дает доступ к элементу, явно объявленному в *list2*. Вызовы *next* и *value* ссылаются на функциональные элементы расширенного базового класса.

Напротив, мы можем снабдить *list2* полным набором функциональных элементов для передвижения по списку и

доступа к значению элементов.

```
class list2 : public list {  
    list2 *link;  
public:  
    list *next();  
    list2 *previous();  
    ETYPE value();  
};
```

Мы пришли к определенным проблемам. Рассмотрим реализацию `list2::next`.

```
list *  
list2::next() {  
    return list::link; // ошибка!  
}
```

Здесь мы пытаемся получить значения обратной связи из базового класса. Это ошибка, так как `list::link` - приватный. Если производный класс не объявлен как дружественный, он не имеет никаких специальных привилегий доступа к приватным элементам своего базового класса. Единственная возможность получить это значение - использовать общедоступный интерфейс *list*.

```
return list::next();
```

который будет работать, но вряд ли представляется ценной функцией, которая ничего не делает.

Напоминаем, что определение класса *list2* начинается с

```
class list2 : public list
```

Ключевое слово **public** в этом контексте определяет, что общедоступные элементы *list* будут общедоступными и для пользователей *list2*. Если использовано ключевое слово **private**, или если не использованы ни **private**, ни **public**, то общедоступные элементы *list* будут приватными при досту-

пе через *list2*. Например, в функции *whatsbefore*, все ссылки на *value* и *next* будут ошибочными ссылками на приватные элементы. Помните, что элементы и дружественные функции *list2* имеют доступ к общедоступным элементам *list*, несмотря на то, является ли он общедоступным или приватным базовым классом.

Хорошее разделение доступа к общедоступным элементам данных класса может быть достигнуто объявлениями общедоступных базовых элементов в производном классе. Объявление общедоступного базового элемента имеет вид

```
Base::member;
```

где *Base* - имя базового класса, а *member* - общедоступный элемент этого базового класса. Объявление должно находиться в общедоступной части определения производного класса.

```
class otherlist {  
public:  
    otherlist *forw;  
    ETYPE el;  
};  
  
class otherlist2 : private otherlist {  
public:  
    otherlist::forw;  
    otherlist2 *back;  
};
```

Хотя *otherlist* - приватный базовый класс *otherlist2*, объявление в базовом классе общедоступного *forw* позволяет получать доступ к нему как к общедоступному методу через *otherlist2*. Так как *otherlist::el* не объявлен как общедоступный базовый элемент, при доступе через *otherlist2* он будет приватным.

Конструкторы производных классов могут включать явную инициализацию базового класса в списке инициализации элементов. Конструктор для *list2* использует список инициализации элементов для инициализации своего базо-

вого класса, как будто он инициализирует элемент. Инициализация базового класса полностью аналогична инициализации элемента класса, описанной в Главе 3, за исключением того, что базовый класс всегда (явно или неявно) инициализируется раньше любого элемента, даже если инициализатор элемента появляется до инициализатора базового класса в списке инициализации. Если у базового класса нет конструктора, его не нужно инициализировать, а если его конструктор можно вызвать без аргументов, его не нужно инициализировать явно.

```
list2::list2(ETYPE e,list2 *fl,list2 *bl)
: list(e,fl) {
link = bl;
}
```

Так же, как и для инициализации элементов, инициализация базового класса может быть реализована любым возможным инициализатором объекта базового классового типа, а не только инициализатором конструктора. Например, мы можем определить второй конструктор *list2*, инициализирующий его базовый класс копированием значения существующего объекта *list*.

```
extern list &exlist;

list2::list2(list2 *bl) : list(exlist) {
link = bl;
}
```

Аргументы, передаваемые конструктору базового класса в списке инициализации элементов исходного конструктора *list2*, имеют типы *ETYPE* и *list2** соответственно, тогда как типы, требуемые конструктором *list* - это *ETYPE* и *list**. Почему не является ошибкой инициализация *list** как *list2**? Причина в том, что объект класса *list2* является одновременно объектом класса *list*. Мы определили это, когда выводили один класс из другого. Кроме того, если класс *Base* - общедоступный базовый класс другого класса *Derived*, то

существует предопределенное преобразование из *Derived* в *Base*, из указателя на *Derived* в указатель на *Base*, и из ссылки на *Derived* в ссылку на *Base*. Мы говорим, что во многих случаях *Derived* - это то же, что и *Base*. Эти преобразования не существуют, если *Base* - приватный базовый класс для *Derived*.

Концепция "то же, что и" - мощный механизм абстракции, так как она позволяет во многих контекстах пользоваться производным классом как базовым. Например, так как *list* - общедоступный базовый класс *list2*, мы можем вызвать функцию `print_list` из предыдущей главы как к списку из компонентов типа *list*, так и к списку из компонентов типа *list2*.

```
typedef int ETYPE;
void
print_list(list *lst) {
    extern void print(int);
    for (list *p=lst; p; p=p->next() )
        print(p->value());
}

main() {
    list *lp = new list2(1,0);
    lp = new list(2,lp);
    lp = new list(3,lp);
    print_list(lp);

    list2 *l2p = new list2(1,0,0);
    l2p = new list2(2,l2p,0);
    l2p = new list2(3,l2p,0);
    print_list(l2p);
}
```

В некоторых случаях производный класс может что-то добавлять или модифицировать поведение своего базового класса. Например, в предыдущей главе мы использовали тип данных *String* для создания полного имени файла и его открытия. Мы можем инкапсулировать это поведение в класс, производный по отношению к *String*.

Абстрактные операции, которые мы хотим выполнять

над именами - это создание, конкатенация и сравнение строк, представляющих имена, а также операция открытия файлов, на которые они ссылаются. Многие из этих операций предлагаются типом *String*, и имя файла может быть рассмотрено и как расширение, и как особый случай *String*. Когда мы говорим "имя файла - это *String* со следующими дополнительными свойствами"... , мы думаем о расширении *String*. Когда мы говорим: "Имя файла - это *String*, который..." , мы рассматриваем специальный случай *String*. Какую бы из этих концептуализаций мы не использовали, мы выразим ее наследованием классов.

```
class Pathname : public String {
public:
    friend
    Pathname &operator +(Pathname &,Pathname &);
    FILE *open();
};
```

В этом случае мы не добавляем элементы данных к расширению *Pathname* класса *String*, мы лишь добавим поведение. В этом смысле *Pathname* - это просто *String*, рассмотренный с другой точки зрения.

Мы наследуем от *String* абстрактные операции создания, уничтожения, сравнения и так далее без изменений. Для конкатенации же необходимо разделять две последовательности имен директорий при помощи слэша "\".

```
Pathname &
operator +(Pathname &p1,Pathname &p2) {
    return (String &p1 + "/" + (String &p2);
}
```

Pathname::open пытается открыть файл, на который ссылается представление полного имени при помощи *String*.

```
FILE *
Pathname::open() {
    return fopen(*this,"r+");
}
```

Текущий аргумент типа `Pathname` для `fopen` преобразуется в `char*` при помощи оператора `char*`, унаследованного из класса *String*.

```
extern char *home_dir, *path, *file;  
Pathname home = home_dir;  
Pathname file = home + path + file;  
FILE *fp = file->open();
```

Этот пример представляет одну из центральных идей при использовании производных классов: унаследовать основное поведение класса от базового и добавить к поведению базового класса что-то необходимое. Наследование, используемое таким образом, представляет эффективный прием для распространения и повторного использования кода.

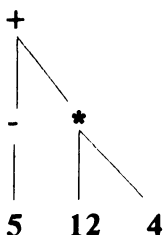
5.2. Иерархии классов.

Многие задачи не так легко моделировать единственным типом, их более естественно представить как совокупность связанных типов. Например, абстрактное синтаксическое дерево для компилятора может представлять несколько конструкций языка программирования. Каждый из этих типов вершин имеет "ядро" общих с другими типами свойств (те свойства, которые делают их элементами дерева), но каждый тип вершины имеет дополнительные свойства, отличающие его от других. Использование только абстракции данных для представления всех таких типов дает нам возможность создать либо единый тип, объединяющий свойства всех остальных, либо множество различных типов, не отражающее их общность. Наилучшим подходом будет использование производных классов для создания множества типов, связанных наследованием.

Мы решили эту проблему при помощи создания иерархии типов. Базовые классы в иерархии представляют общую структуру и функционирование, а производные классы, наоборот, предлагают специализированные версии своих базовых.

Предположим, мы создаем абстрактное синтаксическое

дерево для программы калькулятора. Каждая внутренняя вершина дерева представляет операцию, которую необходимо выполнить, а каждый лист представляет значение. Например, выражение $5+12*4$ может быть представлено как при условии, что операторы $+$, $*$ и унарный $-$ имеют в языке нашего калькулятора такой же приоритет, как в C++. Мы начнем с определения общего типа вершины, кото-



рый служит базовым типом для других типов вершин.

```
class Node {
public:
    enum {
        PLUS,
        TIMES,
        UMINUS,
        INT
    };
    const int code;
    Node(int c) : code(c) {}
    int eval();
};
```

Node содержит код, определяющий текущий тип вершины: $+$, $*$, унарный $-$ или целое, конструктор и функциональный элемент для вычисления абстрактного синтаксического дерева.

Мы используем производные классы для создания отдельных типов вершин из *Node* для операторов $*$, $+$ и унарного $-$ и для целых значений листьев дерева. Конструкторы

производных классов явно вызывают конструктор базового класса и предоставляют соответствующий код вершины.

```
class Plus : public Node {
public:
    Node *left,*right;
    Plus(Node *l,Node *r) : Node(PLUS)
    { left = l;right = r; }
};

class Times : public Node {
public:
    Node *left,*right;
    Times(Node *l,Node *r) : Node(TIMES)
    { left = l;right = r; }
};

class Int : public Node {
public:
    int value;
    Int(int v) : Node(INT)
    { value = v; }
};
```

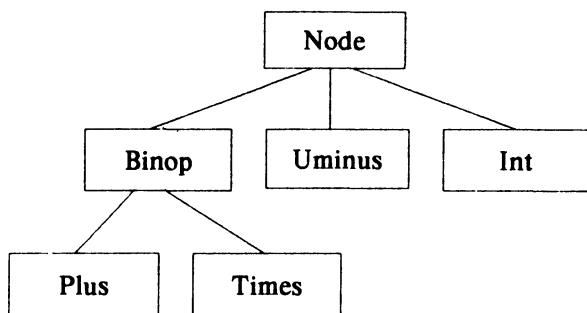
Это решение не слишком плохое, хотя для двух бинарных операций многое дублируется. Если к языку нашего калькулятора добавить другие бинарные операции, это дублирование становится неуклюжим и может превратиться в потенциальный источник ошибок. Лучше ввести еще один уровень наследования, отражающий общность бинарных операций.

```
class Binop : public Node {
public:
    Node *left,*right;
    Binop(int c,Node *l,Node *r) : Node(c)
    { left = l;right = r; }
};

class Plus : public Binop {
public:
    Plus(Node *l,Node *r) : Binop(PLUS,l,r) {}
};
```

```
class Times : public Binop {  
public:  
    Times(Node *l, Node *r) : Binop(TIMES, l, r) {}  
};
```

Результирующая иерархия классов показывает, как типы вершин связаны наследованием.



Таким образом, *Plus* - это *Binop* и *Node*, но не *Uminus*. *Int* - это *Node*, но не *Binop* и так далее. Если мы при создании иерархии будем внимательны, эти наследственные связи между типами будут отражать наше интуитивное представление о концепциях, которые они определяют в проблемной области. Наследование является методом концептуализации, как таковым является и метод расширения и обобщения программ.

```
Int  
Node::eval() {  
    switch(code) {  
        case INT:  
            return ((Int *)this)->value;  
        case UMINUS:  
            return -((Uminus *)this)->operand->eval();  
        case PLUS:  
            return ((Binop *)this)->left->eval()  
                + ((Binop *)this)->right->eval();
```

```
case TIMES:
    return((Binop *)this->left->eval()
        * ((Binop *)this->right->eval());
default:
    error();
    return 0;
}
}
```

Здесь мы реализовали `Node::eval` выбором `Node::code` и рекурсивным вычислением поддеревьев. Чтобы вызвать правильную элемент-функцию `eval`, мы должны привести указатель `this` к соответствующему типу производного объекта, как указано в коде для `Node`. Так как любой объект класса, производного по отношению к `Node`, также имеет тип `Node`, то указатель на `Node` - это и указатель на объект производного класса, определяемый как `Node::code`. Приводя указатель `this` к типу соответствующего указателя на производный класс, мы получаем возможность доступа к элементам производного класса и можем производить вычисления.

Это не слишком плохо в нашем ограниченном случае, но это вряд ли можно назвать естественным решением задачи. Легко видеть, как такой подход разваливается в более сложных ситуациях, когда необходимо идентифицировать правильную программу `eval` перед вызовом.

Несмотря на то, что наша конструкция работает - мы можем строить и вычислять абстрактные синтаксические деревья - в нашей реализации есть существенный дефект. Мы реализовали иерархию типов вершин, но при использовании явного кода вершин `class Node` в основном определяет, какие типы вершин могут быть выведены из нее.

Проблема очевидна, когда мы добавляем новый тип вершины в нашу иерархию. Например, если мы хотим добавить оператор деления, не так просто объявить класс, производный для `Binop`. Мы должны также изменить реализацию `Node::eval` (и, вероятно, добавить новый элемент в перечисление типов операций класса `Node`). Если существующая иерархия вершин была частью библиотеки, наши пользова-

тели вынуждены будут копировать и редактировать ее для получения своих собственных версий. Действуя так, они будут вынуждены либо вносить изменения в библиотеку в своей версии, либо не использовать стандартную библиотеку. В обоих случаях, наша цель - создание библиотеки - не будет достигнута.

Лучше будет инкапсулировать специфическую информацию вершины внутри соответствующей вершины, не записывая никакой специфической информации в типы вершин, находящихся в иерархии выше. Чтобы делать это максимально эффективно, мы должны иметь возможность более обобщенно различать типы вершин в ходе выполнения программы, чем это сделано в текущей реализации `Node::eval`, где мы использовали явное приведение типов. Мы сделаем это при помощи виртуальных функций.

5.3. Виртуальные функции.

Оставим ненадолго наш пример с абстрактным синтаксическим деревом и рассмотрим более простой. Предположим, мы работаем с набором типов фруктов, связанных наследованием.

```
class fruit {
public:
    char *identify() { return "fruit"; }
};

class apple : public fruit {
public:
    char *identify() { return "apple"; }
};

class orange : public fruit {
public:
    char *identify() { return "orange"; }
};
```

Нам необходимо создать список разнородных "фруктовых" объектов, в котором каждый элемент идентифицируется отдельно. Мы используем односвязный список,

описанный ранее в этой главе.

```
typedef fruit *ETYPE;
#include "list.h"

void
print_list(list *lst) {
    extern void print(char *);
    while (lst) {
        print (lst->value()->identify());
        lst = lst->next();
    }
}

main() {
    list *lst = new list(new fruit,0);
    lst = new list(new apple,lst);
    lst = new list(new orange,lst);
    print_list(lst);
}
```

Эта программа печатает слово "фрукт" трижды, даже когда список содержит *orange*, *apple* и *fruit*. Это поведение корректно, так как выражение `lst->value()` имеет тип *fruit**, и `fruit::identify` возвращает указатель на строку символов "fruit". Естественно, элемент-функция вызывается в зависимости от типа указателя или ссылки, использованной для доступа к ней, а не от текущего типа объекта, на который ссылаются указатель или ссылка.

```
fruit *fp = new apple;
fp->identify(); // возвращает "fruit"
((apple *)fp)->identify(); // возвращает "apple"
apple *ap = (apple *)fp;
ap->identify(); // возвращает "apple"
```

Сравните это с реализацией `Node::eval` в этой главе ранее.

Для нашей программы идентификации фруктов мы хотим наоборот, т.е. чтобы соответствующая идентифицирующая функция определялась типом текущего объекта, а не

типом ссылки или указателя, использованных для доступа к нему. Для этого мы определяем виртуальные функции.

```
class fruit {  
    public;  
    virtual char *identify() { return "fruit"; }  
};
```

Виртуальные функции позволяют производным классам предлагать альтернативные версии функций базового класса. В объявлении `fruit::identify` как виртуальной функции, мы утверждаем, что классы, выведенные из *fruit*, могут иметь собственные версии `identify`, и что эти функции будут вызываться на основе текущих типов объектов. Так, *apple* или *orange* будут иметь собственные версии виртуальной функции, вызываемой для них, даже если она рассматривается как *fruit*.

С добавлением ключевого слова `virtual` в объявление `fruit::identify`, наша программа теперь работает так, как мы хотели, печатая "orange", "apple", "fruit".

Обратите внимание, что `apple::identify` и `orange::identify` являются виртуальными, хотя мы их такими явно не объявили (как будто мы это сделали). Правило для определения, когда функция виртуальная, простое: функция является виртуальной, если она объявлена виртуальной, или если есть функция из базового класса с той же сигнатурой, которая является виртуальной. Сигнатура функции состоит из ее имени и типов формальных аргументов. Например, если объявление `apple::identify`, было

```
char *identify(int=0);
```

то оно не будет виртуальным, так как тогда оно будет иметь сигнатуру, отличную от `fruit::identify`. Выходом нашей программы в этом случае будет "orange", "fruit", "fruit". Если сигнатура функционального элемента производного класса совпадает с сигнатурой виртуального элемента базового класса, тогда возвращаемый тип тоже должен соответствовать типу результата функции базового класса. Это

гарантирует, что динамическое (в ходе выполнения) связывание имен, производимое виртуальными функциями, сохраняет тип.

Комбинация связи "то же, что и" производного класса со своим общедоступным базовым и динамического связывания имен виртуальных функций - очень полезный метод абстракции. Например, наша функция `print_list` работает только со списками указателей *fruit*. Мы можем построить иерархию типов произвольной сложности и глубины, базирываясь на классе *fruit*, не изменяя реализации `print_list`. Например, мы можем расширить иерархию, добавив тип *banana* - производного от *fruit*, или создать новый уровень, выводя *macintosh* и *delicious* из *apple*. Наследование, использованное таким образом, позволяет нам выделить общность из группы связанных типов и писать общие программы на базе нашей абстракции. Специфические детали типа инкапсулированы внутри типа.

Используя эти концепции, мы можем теперь заново разработать реализацию нашего абстрактного синтаксического дерева при помощи виртуальных функций.

```
class Node {
public:
    Node() {}
    virtual ~Node() {}
    virtual int eval() { error(); return 0; }
};

class Binop : public Node {
public:
    Node *left,*right;
    ~Binop() { delete left; delete right; }
    Binop(Node *l,Node *r) { left = l;right = r; }
};

class Plus : public Binop {
public:
    Plus(Node *l,Node *r) : Binop(l,r) {}
    int eval() { return left->eval() + right->eval(); }
};
```

```
class Times : public Binop {
public:
    Times(Node *l, Node *r) : Binop(l, r) {}
    int eval() { return left->eval() * right->eval(); }
};

class Uminus : public Node {
    Node *operand;
public:
    Uminus(Node *o) { operand = o; }
    ~Uminus() { delete operand; }
    int eval() { return -operand->eval(); }
};

class Int : public Node {
    int value;
public:
    Int(int v) { value = v; }
    int eval() { return value; }
};
```

eval объявлена в *Node* как виртуальная, поэтому *eval* в *Plus*, *Times*, *Uminus* и *Int* также являются виртуальными. *Binop* не объявляет функции *eval*, поэтому объект типа *Binop* вызывает *Node::eval*, унаследованную от базового класса *Node*. Теперь мы можем создавать и вычислять абстрактные синтаксические деревья.

```
Int
limited_use() {
    // вычисляет -5+12*4
    Node *np =
        new Plus(
            new Uminus(
                new Int(5),
            ),
            new Times(
                new Int(12),
                new Int(4)
            )
        );
    int result = np->eval();
    delete np;
    return result;
}
```

Виртуальный вызов `eval` вызывает `Plus::eval`, так как вершина *Plus* - в корне абстрактного синтаксического дерева, на которое указывает `pr`. `Plus::eval` вызывает виртуальные функции `eval` для своих левого и правого поддеревьев, что в этом случае приводит к вызову `Uminus::eval` и `Times::eval` соответственно, и так далее. На каждом шаге текущая функция для вызова определяется типом вершины в корне вычисляемого поддерева.

Класс *Node* также объявляет виртуальный деструктор. Хотя деструктор нельзя вызвать явно как элемент-функцию, его можно вызвать в виртуальном стиле. Строка

```
delete pr,
```

вызывает виртуальный деструктор для корня абстрактного синтаксического дерева, на который указывает `pr`. Это вершина типа *Plus*, которая не определяет деструктора, поэтому вызывается унаследованный деструктор `Binop::~~Binop`. `Binop::~~Binop` вызывает виртуальные деструкторы для своих левого и правого поддеревьев и так далее. В результате единственное удаляющее выражение освобождает целое абстрактное синтаксическое дерево.

Необходимо дополнительно учитывать, что конструкторы не могут быть виртуальными. Конструктор создает объект, тогда как виртуальная функция требует, чтобы объект уже существовал и по нему определяется, какая функция будет вызвана.

Использование виртуальных функций позволяет нам инкапсулировать специфические операции для вершины внутри объявления спецификатора типа вершины. Теперь мы можем добавлять новые типы вершин в иерархию, не влияя на существующие программы.

```
class Div : public Binop {
public:
    Div(Node *l, Node *r) : Binop(l, r) {}
    int eval() { return left->eval() / right->eval(); }
};
```

Объявив класс *Div*, мы можем создавать и вычислять объекты типа *Div* как вершины абстрактного синтаксического дерева, не меняя реализаций других типов вершин.

5.4. Защищенные элементы.

`Node::eval` играет важную роль. Это, с одной стороны, "корневая" виртуальная функция, которая заставляет все остальные элементы `eval` с той же сигнатурой быть виртуальными, и с другой - нечто вроде стоп-черты. Если мы забыли объявить `Div::eval`, любая попытка вызова `eval` для вершины типа *Div* приведет к вызову `Node::eval` и к ошибке. Никакие объекты *Binop* или *Node* не должны создаваться или вычисляться, а только объекты производных по отношению к ним классов. Отсекающая функция в корне иерархии - эффективный путь отлавливания ошибок такого типа, но, к сожалению, ошибка не проявляется до момента выполнения, когда такая вершина вычисляется.

Предпочтительнее (и безопаснее) оградить пользователей иерархии вершин от создания объектов этих типов. Один из путей - сделать конструкторы для *Node* и *Binop* приватными. К сожалению, это также не позволит использовать конструкторы производным классам.

```
class Node {
Node() {}
public:
// ...
};
class Int : public Node {
Int value;
public:
Int(Int v) {
// Ошибка! Неявный вызов приватного Node::Node
// ...
}
```

Аналогичная проблема возникает с элементами `left` и `right` класса *Binop*. Нам бы хотелось, чтобы они были приватными, чтобы предохранить их от неправильного употребления неаккуратными пользователями иерархии вершин, но они должны быть общедоступны для того, чтобы

производные от *Binop* классы (*Plus*, *Times* и *Div*) могли их использовать.

Мы решаем эту проблему при помощи защищенных элементов класса. Последовательность защищенных элементов вводится в определение класса при помощи **protected**: точно также, как общедоступные элементы предваряются **public**:, а приватные - **private**:. Защищенный элемент похож на приватный, за исключением того, что он доступен элементам и дружественным функциям классов, производных по отношению к классу, в котором он объявлен. Аналогично общедоступным элементам защищенные элементы базового класса являются защищенными в производных классах в случаях, если база общедоступна, и приватными - в противном случае.

```
class Node {
protected:
    Node() {}
public:
    virtual ~Node() {}
    virtual int eval();
};

class Binop: public Node {
protected:
    Node *left,*right;
    Binop(Node *l,Node *r)
        { left = l; right = r; }
    ~Binop() { delete left; delete right; }
};
```

Теперь только элементы и дружественные функции *Node*, *Binop* и производных от них классов могут вызывать их конструкторы, поэтому другие фрагменты программы не могут размещать вершины этих типов. Кроме того, *Binop::left* и *Binop::right* будут доступны только для элементов и дружественных функций производных для них бинарных операторов.

5.5. Наследование как инструмент проектирования.

Наследование - это метод проектирования, поскольку он позволяет абстрагировать проблему в более общем виде в базовом классе в корне иерархического дерева, а также моделировать и писать программы, работающие только с абстракцией. Специальные случаи (которые сами могут быть абстракциями еще более специальных случаев) могут выражаться производными по отношению к базовому классами.

Например, в случае с абстрактным синтаксическим деревом, мы начали с общей концепции вершины и успешно расширили иерархию типов вершин при помощи специализации. Но исходная абстракция, тем не менее, осталась. В программе типа

```
void  
print_value(Node *np) {  
    extern void print(int);  
    print(np->eval());  
}
```

мы работаем только с *Node*, и сложность специализированных типов, выведенных из *Node*, скрыта от программ общего назначения.

Использование наследования, как инструмента проектирования, аналогично пошаговому уточнению в парадигме функциональной декомпозиции, описанной в Главе 2. Пошаговое уточнение разделяет процедурные аспекты задачи на иерархию процедур, тогда как наследование преобразует аспекты типов задачи в иерархию типов.

Хотя результат развития программы пошаговой детализацией - строгая иерархия процедур, к этому результату не обязательно приходят напрямую разработкой сверху вниз. Иногда сначала пишутся низкоуровневые процедуры, чтобы посмотреть, как их реализация влияет на структуру процедур более высокого уровня.

Аналогичным образом, в сложной программе, работающей со многими различными типами, может быть не ясно,

как типы связаны друг с другом, если они вообще связаны. Иногда явную общность можно понять только после программирования реализаций нескольких, предположительно разных, типов. Наследование можно тогда использовать для расширения общих частей их интерфейса и реализации. Часто эта общность - повод для более глубокой абстракции, и "открытая" иерархия наследования становится рабочей абстракцией проектирования.

Рассмотрим в качестве метапримера реализацию символической таблицы идентификаторов для компилятора C++. В интересах краткости мы частично упростим задачу, но общая идея решения может быть (и была) использована для создания таблицы идентификаторов для компилятора C++.

Рассмотрим основы концепции видимости в C++. Области видимости в C++, по сути, блочно-структурированные. То есть, имя определено от точки своего первого появления до конца блока, в котором оно определено. Появление имени во внутреннем блоке скрывает все имена с тем же идентификатором, определенные во внешних блоках. В целях управления видимостью файл также может рассматриваться как блок.

Функциональные элементы и наследование усложняют эту простую блочную структуру. Область видимости функционального элемента - это область видимости класса, элементом которого он является.

```
int i;  
class C {  
    int i;  
    f() { i=1; }  
};
```

Функциональный элемент `f` ссылается на элемент класса `i`, а не на глобальное `i`. Следующий фрагмент эквивалентен предыдущему:

```
int i;  
class C {  
    int i;  
    inline f().
```

```
};
```

```
C::f() { i=1; }
```

Область видимости функционального элемента - его класс, независимо от того, находится ли он текстуально внутри класса.

Граница видимости производного класса - его базовый класс. Например:

```
class B { public: int i; };
int i;
void g() {
    int i;
    class D1 : public B {
        f() { i=1; }
    };
}
class D2 : public B {};
```

Классы *B*, *D1* и *D2* формируют иерархию классов, в которой класс *B* - область видимости для *D1* и *D2*. В частности, необходимо заметить, что область видимости *D1* не совпадает с областью видимости функции *g*. Так, функциональный элемент *f* из *D1* связывается с элементом базового класса *i*, а не с локальным или файловым *i*.

В качестве первого подхода к разработке таблицы идентификаторов рассмотрим представление различных уровней видимости (глобальный уровень, класс, функция) разными таблицами вместо поддержки монолитной структуры. Мы представим таблицы каждого типа отдельными классами с функциональными элементами для просмотра и вставки имен. Кроме того, так как область видимости производного класса помещается в области видимости его базового класса, тип таблицы для классов имеет функциональный элемент, возвращающий указатель на таблицу его базового класса.

```
class Gtab {
public:
    name *insert( char * );
```

```
    name *lookup( char * );  
};  
  
class Ctab {  
public:  
    name *Insert( char * );  
    name *lookup( char * );  
    Ctab *base_class();  
};  
  
class Ftab {  
public:  
    name *Insert( char * );  
    name *lookup( char * );  
};
```

В каждом случае аргументом функции вставки или поиска является указатель на идентификатор, а возвращают они имя в таблице идентификаторов. Для этого предположим, что `name` - структура с указателем на ее идентификатор и, возможно, содержащая другую информацию.

Разделив нашу задачу на три различных случая, мы можем реализовать семантику каждой таблицы наиболее подходящим для каждого случая образом. Например, мы можем разработать эвристику, показывающую, что таблицу для классов лучше организовать в виде списка, глобальную - в виде простой хэш таблицы, а таблицу для функций - как более сложную хэш таблицу, которая содержит информацию о видимости блоков внутри функции. Единственная глобальная таблица может иметь следующую структуру:

```
class Gtab {  
    static name *t[256];  
    Int hash( char * );  
public:  
    name *insert( char * );  
    name *lookup( char * );  
} gtable;
```

В этой точке мы имеем полное представление о данной области видимости. Мы используем правила, приведенные

в начале этого примера для поиска объемлющего контекста. Мы начинаем с объявления трех "текущих контекстов", по одному на каждый тип таблицы идентификаторов.

```
Gtab *gptr; // указывает на глобальную таблицу
Ctab *cptr; // указывает на текущую таблицу классов
Ftab *fptr; // указывает на текущую таблицу функций
```

Тем не менее, хотя этот подход и работает, он порождает программы, в которых простая вставка и просмотр очень сложны.

```
name *
lookup(char *id) {
    name *n;
    if (fptr)
        if (n = fptr->lookup(id))
            return n;
    for (Ctab *cp = cptr; cp; cp = cp->base_class())
        if (n=cp->lookup(id))
            return n;
    return gptr->lookup(id);
}
```

```
name *
insert(char *id) {
    if (fptr)
        return fptr->insert(id);
    if (cptr)
        return cptr->insert(id);
    return gptr->insert(id);
}
```

По этой схеме мы должны всегда различать специфические типы областей видимости, даже в ситуации, когда мы производим общие операции.

Альтернативный подход состоит в выделении общности всех этих трех таблиц идентификаторов и использовании наследования для выявления общности.

```
class Tab {
protected:
    Tab *parent;
```

```
public:
    virtual name *insert(char *);
    virtual name *look(char *);
};

class Gtab : public Tab { /* ... */ };
class Ctab : public Tab { /* ... */ };
class Ftab : public Tab { /* ... */ };
```

Здесь мы представили объемлющий контекст явно как указатель на тип *Tab* в корне иерархии таблиц идентификаторов. Так как *Tab* - общедоступный базовый класс для *Gtab*, *Ctab* и *Ftab*, то *Tab::parent* может ссылаться на любой объект - символьную таблицу. Для наблюдения за текущим контекстом мы используем глобальную переменную типа *Tab** с именем *curr_tab* для указания на текущую таблицу.

Таким образом мы получили работоспособный остов для таблицы идентификаторов C++. В любой момент времени состояние контекста состоит из множества индивидуальных таблиц для функций и классов и глобальной таблицы. Связь между этими таблицами представляется указателями на их родителя.

Когда появляется новый контекст, например, в начале тела класса, мы создаем новую таблицу класса и иницилируем ее ссылку на соответствующий объемлющий контекст. Например, если класс является производным, он появляется в области видимости своего базового класса, поэтому *parent* ссылается на таблицу базового класса. Если класс не производный, то родительской для него будет глобальная таблица (независимо от того, входит ли он лексически в функцию или другой класс). Переписанная с использованием иерархии таблиц и возможности использования виртуальных функций, программа общего просмотра становится заметно проще.

```
name *
lookup(char *id) {
    name *n;
    for (Tab *t=curr_tab; t; t=t->parent)
        if (n=t->look(id))
```

```
    return n;  
    return 0;  
}
```

Аналогично упрощаются вставки и

```
curr_tab->Insert(id);
```

вставляет имя с идентификатором *id* в текущую таблицу, независимо от того, является ли она глобальной, классовой или предназначена для функции.

Выразив связь трех наших исходных таблиц идентификаторов в виде специальных случаев общего типа таблицы, мы не только упростим использование типов, но в пользу развития абстракции таблицы идентификаторов мы упростили наше представление о типах.

5.6. Наследование для расширения интерфейса.

В некоторых примерах этой главы производные классы являются расширением или специализацией базовых, которые сами по себе вполне полезны. *list2* выведен из *list* и *Pathname* - из *String*, но и *list*, и *String* - отличные функционирующие типы. С другой стороны, базовый тип *Tab* нашей иерархии таблиц идентификаторов и *Node* нашей иерархии вершин синтаксического дерева не являются полностью функциональными, и требуют, чтобы их определение было дополнено наследованием классов. Поэтому *Tab* и *Node* формируют некий частичный тип, наполовину реализацию, наполовину шаблон, который дополняется в производных классах.

Мы можем расширить эту концепцию и определить базовый класс, служащий только шаблоном для производных классов, для того, чтобы гарантировать, что производный тип удовлетворяет конкретному интерфейсу. Здесь мы относимся к производному классу не как к расширению или специализации базового, но как к реализации интерфейса, специализируемого базовым классом.

Рассмотрим задачу специфицирования интерфейса между ядром и драйвером устройства в операционной системе (ОС) UNIX. Для выражения системного вызова на открытие, закрытие, чтение, запись и т.д. данному устройству, ядро индексирует двумерную таблицу переключателей устройства с главным номером устройства и кодом, соответствующим типу системного вызова (открыть, закрыть и т.д.). Главный номер устройства определяет вид устройства (диск или терминал и т.п.), а младший номер устройства используется для определения конкретного устройства определенного типа (из нескольких дисков или терминалов). Таблица содержит адреса функций, соответствующих устройству; операционная пара используется для индексации таблицы.

Ядро выполняет системный вызов, обращаясь к соответствующей программе косвенно через таблицу переключателей устройства.

Эта таблица определяет интерфейс между ядром UNIX и программами драйвера устройства. В этой схеме есть недостатки. Во-первых, размеры и содержание таблицы фиксированы после построения (компиляции) ядра и добавление нового типа устройства требует перестройки ядра. Во-вторых, нет гарантии, что функции, адреса которых содержатся в таблице, имеют нужные типы. В третьих, информация, относящаяся к специфическому типу устройств, "рассыпана" в нескольких местах и не помогает нам рассматривать устройство в целом.

Альтернативный способ реализации интерфейса состоит в определении типа *драйвера устройства*, инкапсулирующего поведение обобщенного устройства.

```
class Device {  
public:  
    virtual int open(char *,int,int);  
    virtual int close(int);  
    virtual int read(int,char *,unsigned);  
    virtual int write(int,char *,unsigned);  
    virtual int ioctl(int,int ...);  
    Device();  
    ~Device();  
};
```

Функциональные элементы этого обобщенного драйвера реализуют стандартное поведение.

```
int
Device::open(char *path, int oflag, int mode) {
    return nulldev();
}
```

Теперь ядро может работать со всеми устройствами через этот обобщенный тип *драйвера* точно так же, как мы выразили все типы вершин абстрактного синтаксического дерева через обобщенный тип *Node* и получили доступ ко всем таблицам идентификаторов через общий тип *Tab*.

Мы создаем тип для драйвера конкретного вида устройств при помощи наследования из этого шаблона.

```
class Terminal : public Device {
public:
    int open(char *, int, int);
    int close(int);
    int read(int, char *, unsigned);
    int write(int, char *, unsigned);
    int ioctl(int, int ...);
    Terminal();
    ~Terminal();
};
```

Любая часть интерфейса, не пересопределенная явно в производном типе устройства, по умолчанию совпадает с программой в базовом типе.

```
class Mem_mapped_io : public Device {
public:
    int read(int, char *, unsigned);
    int write(int, char *, unsigned);
    int ioctl(int, int ...);
};
```

Иерархия драйверов устройств заменяет таблицу переключателей устройства. Мы можем рассматривать конкретные производные типы устройств как замену главных номеров устройств, а динамическое связывание, производи-

мое виртуальными функциями, как аналог таблицы коммутации. Для замены младших номеров устройств мы создаем объект *устройство* для каждого физического устройства данного типа. Например, если к каналу подключены пять дисководов, существует пять соответствующих дисковых объектов под управлением ядра; при каждом входе пользователя в систему создается соответствующий терминальный объект, который уничтожается при выходе.

5.7. Множественное наследование.

Производный класс может иметь любое число базовых классов. Использование двух или более классов называется множественным наследованием. Хотя использование множественного наследования встречается реже, чем простого, множественное наследование полезно для создания классов, комбинирующих поведение двух или более других классов.

Например, мы можем захотеть создать тип, отслеживающий данное условие и отражающий его статус на экране. Предположим, у нас есть библиотека растровой графики с абстрактным типом данных *циферблат*, отражающим изменяющееся значение.

```
class Dial {  
    // несущественные детали реализации  
protected:  
    double value;  
    Dial(char *,double,double);  
    ~Dial(),  
};
```

Конструктор *Dial* отражает связь с меткой и с диапазоном измерений между значениями второго и третьего аргументов. После создания объект *Dial* отображает текущее значение *Dial::value*.

Нам также доступен тип *модель*.

```
class Sampler {  
    // детали реализации
```

```
protected:
double freq;
virtual void sample();
Sampler(double);
~Sampler();
};
```

Объект типа *Sampler* вызывает свою виртуальную функцию *sample* каждые *Sampler::freq* секунд.

Наш монитор представляет собой одновременно *Dial* и *Sampler*. Мы выразим это с использованием множественного наследования для получения черт обоих классов.

```
class Monitor : public Sampler, public Dial {
    void sample() { value = get_value(); }
protected:
    virtual double get_value();
    Monitor(char *lab,double l,double h,double f)
        : Dial(lab,l,h),Sampler(f) {}
};
```

Запись

```
class Monitor:public Sampler, public Dial
```

объявляет *Monitor* как новый классовый тип, производный и от *Dial*, и от *Sampler*. Использование нескольких базовых классов - естественное расширение случая с наследованием от одного базового. Любое число имен классов может быть представлено через запятую в списке базовых классов, но ни одно из имен не должно появляться в одном списке дважды.

Конструктор *Monitor* использует список инициализации элементов для явного вызова конструкторов классов. Порядок инициализации определяется списком инициализации элементов, при этом любые неявные инициализации базовых классов вызываются в том порядке, в котором базовые классы появляются в списке в определении классов. Напоминаем, что базовые классы всегда инициализируются раньше элементов, даже если имя элемента появляется

раньше имени базового класса в списке инициализации.

Область видимости класса *Monitor* "помещается" внутри области видимости всех его базовых классов. Поэтому *Monitor::sample* - виртуальная функция, переопределяющая *Sampler::sample*. Ссылка на *value* в *Monitor::sample* является ссылкой на *Dial::value*.

Это "множественное помещение" может быть источником двусмысленностей, которые не возникают при простом наследовании. Для иллюстрации некоторых из этих проблем вернемся к иерархии фруктов и подключим иерархию деревьев.

```
class fruit {
public:
    virtual char *identify() { return "fruit"; }
};

class tree {
public:
    virtual char *identify() { return "tree"; }
};
```

Некоторые типы являются одновременно *fruit* и *tree*.

```
class apple : public fruit, public tree {};
apple *ap = new apple;
ap->identify(); // ошибка!
```

Этот фрагмент не компилируется, так как вызов функции двусмысленен. Так как контекст *apple* находится в области видимости как *fruit*, так и *tree*, вызов может относиться и к *fruit::identify*, и к *tree::identify*. Мы можем сделать вызов однозначным, выявив базовый класс, на который мы ссылаемся при помощи оператора видимости

```
ap->fruit::identify();
```

или приведения типов:

```
((fruit *) ap)->identify();
```

Лучшим решением будет определение функции *identify* для класса *apple* с нужным поведением.

```
class apple : public fruit, public tree {  
    public:  
        char *identify() { return "apple"; }  
};  
// ...  
ap->identify();
```

Все обычные связи между производным классом и каждым из его базовых при множественном наследовании сохраняются такими же, как и в случае единичного наследования. Например, `apple::identify` - виртуальная функция, переопределяющая и `fruit::identify`, и `tree::identify`. Кроме того, так как и *fruit*, и *tree* являются общедоступными базовыми классами *apple*, существуют предопределенные преобразования из *apple* и в *fruit*, и в *tree*.

```
apple a;  
tree &t=a; // да, apple это tree  
fruit *f=&a; // да, apple это fruit  
*f=t; // ошибка! tree это не fruit  
a=t; // ошибка! fruit это не apple
```

Сравните это с преобразованиями, описанными ранее для иерархии вершин абстрактного синтаксического дерева.

Вернемся к классу *Monitor*. Используя множественное наследование, мы комбинируем две различные концепции *Dial* и *Sampler* для создания нового типа. Такое использование множественного наследования для комбинирования типов очень важно.

Теперь мы можем выводить специализированные типы из класса *Monitor*. Тип, выведенный из *Monitor*, должен представить инициализирующие значения, которые передаются через конструктор *Monitor* классам *Dial* и *Sampler*, и определить функцию, предоставляющую значения для мониторинга.

```
class Mem_usage : public Monitor {  
    char *start;  
    char *max;  
    public:
```

```
double get_value() {
    return (current_top()-start)/(max-start);
}
Mem_usage() : Monitor("Memory Usage",0,100,0.1),
    start(current_top()),max(get_limit()) {}
};
```

Объект *Mem_usage* отслеживает процент доступной памяти для процесса, в котором он появился. Виртуальная функция *get_value* возвращает отношение доступной памяти с момента создания объекта *Mem_usage*. Системная функция, названная здесь *current_top*, выдает верхний адрес памяти процесса для вычисления в *get_value*. Конструктор отмечает шкалу монитора, устанавливает интервал отображаемых значений от 0.0 до 100.0 и устанавливает дискретность, равную десять раз в секунду. Обратите внимание, что *current_top* снова используется для записи начального верхнего адреса процесса. Другая системная функция, названная *get_limit*, выдает максимальный адрес, до которого может увеличиваться память процесса.

В качестве другого (гораздо менее серьезного) примера, предположим, что мы хотим отслеживать скорость мыши, связанной с нашим терминалом. Пусть нам доступны обычные типы растровой графики (возможно, из той же библиотеки, в которой определен тип *Dial*), объекты типа *Point* (позиции на экране в декартовой системе координат) и *Mouse* (объект, представляющий мышь).

```
class Mouse_velocity : public Monitor {
    Point prev;
public:
    double get_value() {
        extern const int PIXELS_PER_INCH;
        double value=dist(Mouse.abs_pos,prev)
            / (freq * PIXELS_PER)INCH);
        prev = Mouse.abs_pos;
        return value;
    }
    Mouse_velocity() :
        Monitor("Velocity",0,120,.01) {
        prev = Mouse.abs_pos; } }
```

Объект *Mouse_velocity* отражает текущую скорость мыши в дюймах в секунду. Обратите внимание, что виртуальная функция *get_value* обращается к *freq*, элементу *Sampler*, который отличается на два уровня наследования.

5.8. Виртуальные базовые классы.

Тип *Monitor* использует множественное наследование для комбинации двух полностью разных типов. Множественное наследование также можно использовать и для комбинации более близко связанных классов. Например, мы можем захотеть создать новый тип устройства, имеющий свойства двух существующих типов устройств.

```
class Monitored_device :  
    public Device, public Monitor {  
    // ...  
};
```

```
class Network_device :  
    public Device, public Protocol {  
    // ...  
};
```

Monitored_device отслеживает темп передачи данных через устройство, а *Network_device* - интерфейс к драйверу платы сети, реализующий данный протокол сети. Мы хотим создать новый тип устройства, наблюдающий за темпом работы сети. Мы можем сконструировать новый тип устройства "с самого начала",

```
class Monitored_network_device :  
    public Device, public Monitore, public Protocol {  
    // ...  
};
```

но мы не сможем тогда использовать наш новый тип как *Network_device* или как *Monitored_device*, так как тогда между этими типами не будет наследственной связи. Альтернативный подход - выведение из двух существующих

устройств - вызовет появление в иерархии двух базовых классов *Device*.

```
class Monitored_network_device :  
    public Monitored_device, public Network_device {  
    // ...  
};
```

В некоторых случаях это именно то, что требовалось, но это не то, что мы хотим от нашего нового типа устройства. Во-первых, простая попытка использования нового типа как устройства приведет к неопределенным ошибкам, и нам нужно будет каждый раз уточнять конкретный *Device*, на который мы ссылаемся.

```
Monitored_network_device iso;  
iso.open("/dev/iso", O_RDWR, 0);  
    // ошибка! двусмысленно  
iso.Device::open("/dev/iso", O_RDWR, 0);  
    // ошибка! все еще двусмысленно...  
iso.Network_device::open("/dev/iso", O_RDWR, 0);  
    // OK, устройство типа Network_device  
iso.Monitored_device::open("/dev/iso", O_RDWR, 0);  
    // OK, устройство типа Monitored_device
```

Более общая проблема состоит в том, что если даже мы работаем с одним физическим устройством, наш тип содержит два различных его представления. Это может вызвать трудности. Например, так как конструктор *Device* будет активизирован дважды для каждого объекта класса *Monitored_network_device*, ядро операционной системы, работающее с устройством только через интерфейс, предлагаемый типом *Device*, может заключить, что имеются два устройства там, где в действительности существует только одно.

Мы хотим создать класс, выведенный и из *Monitored_device*, и из *Network_device*, но представляющий единственный *Device*. Мы осуществим это с помощью виртуальных базовых классов.

```
class Monitored_device :  
    public virtual Device, public Monitor {  
    // ...  
};  
  
class Network_device :  
    public virtual Device, public Protocol {  
    // ...  
};  
class Monitored_network_device :  
    public Monitored_device, public Network_device {  
    // ...  
};
```

Объявляя базовый класс виртуальным, мы говорим, что хотим, чтобы его представление разделялось с любым другим виртуальным появлением этого базового класса в объекте. В классе *Monitored_network_device* *Device* появляется только один раз, так как все упоминания *Device* объявлены виртуальными. Так как существует единственный *Device*, неопределенные ссылки на его элементы не являются более двусмысленными.

```
iso.open("/dev/iso", O_RDWR, 0);
```

Точно также, конструктор *Device* вызывается для объекта *Monitored_network_device* только один раз, так как необходимо инициализировать только один *Device*.

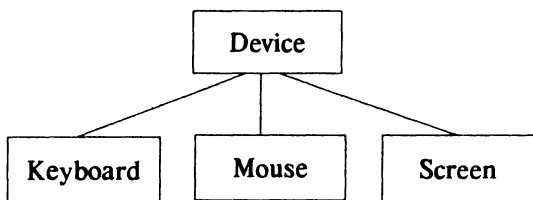
При использовании виртуальных базовых классов существуют некоторые ограничения. Во-первых, базовый класс, являющийся виртуальным, не может явно инициализироваться упоминанием в списке инициализации конструктора. Так как класс, имеющий конструктор, должен инициализироваться, то класс, используемый как виртуальный базовый, должен либо не иметь конструктора, либо иметь конструктор, который может вызываться без аргументов. Любая (неявная) инициализация виртуальных базовых классов выполняется раньше любого другого базового, включая те, которые явно инициализируются в списке инициализации элементов. Второе ограничение ка-

сается приведения типов: запрещается приводить указатель на виртуальный базовый класс к типу указателя на класс, прямо или косвенно выведенный из него.

5.9. Упражнения.

- Упр.5.1. Рассмотрим реализацию класса *complex* парой координат и в полярной системе из предыдущей главы. Могут ли они быть реализованы как различные типы, связанные наследованием и используемые вместе? Является ли это подходящим использованием наследования?
- Упр. 5.2. Рассмотрим типы данных *граф* и *абстрактный автомат с конечным числом состояний* из упражнений предыдущей главы. Может ли один быть выведен из другого? Могут ли они иметь общий базовый класс? Как вы можете охарактеризовать каждое из этих использований наследования: как только совместное использование кода или между графом и конечным автоматом есть концептуальная общность?
- Упр. 5.3.+ Добавьте типы вершин *идентификатор* и *присваивание* в иерархию вершин абстрактного синтаксического дерева для калькулятора. Нужно ли вам для этого менять реализацию других типов вершин? Используйте получившуюся иерархию для написания полной интерактивной программы-калькулятора.
- Упр. 5.4. Что произойдет в реализации *operator +* (конкатенация) для типа *Pathname*, если мы не приведем типы двух аргументов *Pathname&* к типу *String&*? (Намек: Какая версия *operator +* вызывается при наличии приведения, а какая - без приведения?)
- Упр. 5.5. Пусть существует библиотека интерфейса с ап-

паратурой для терминальных устройств со следующей иерархией наследования:



Предположим, мы не можем модифицировать библиотеку и вынуждены добавить типы *планшет* и *световое перо* в иерархию. Куда и как вы их добавите? Пусть Вы можете модифицировать библиотеку. Как вы добавите эти типы теперь? Как бы вы перестроили иерархию для упрощения введения новых типов в будущем?

- Упр. 5.6. Покажите, как наследование можно использовать для пакетирования и предоставления множества констант другим классам.
- Упр. 5.7. Используйте тип *Monitor* для создания классовых типов для мониторинга а) скорости печати, б) загрузки системы, в) числа пользователей, вошедших в систему, +г) числа размещенных вершин абстрактного синтаксического дерева и д) относительного процента соотношения вершин-бинарных операторов к унарным.
- Упр. 5.8. Покажите, как наследование можно использовать для представления различных "точек зрения" на данный базовый класс. Разработайте запись для базы данных о служащих и разработайте производные классы, представляющие следующие точки зрения а) для общего использования, б) для начисления зарплаты, в) для наблюдения и г) для ФБР.

Глава 6

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



Приемы процедурного программирования концентрируются на алгоритмах, используемых для решения задачи. При этом не обращается внимание на структуры данных, задействованные функциями, как на отдельные части организации программы. В противоположность этому, объектно-ориентированное программирование (ООП) концентрируется на сути задачи, для которой пишется программа: элементы программы разрабатываются в соответствии с объектами в описании задачи. Общий подход ООП состоит в определении набора объектных типов. Объектные типы - это модули, интегрирующие структуры данных, которые представляют элементы задачи, с операциями, необходимыми для выработки решения. Как только эти объектные типы определены, создаются образцы объектов для конкретной задачи и вызываются операции для произведения обработки.

В C++ классы служат для представления объектных типов, а функциональные элементы предоставляют средства для встраивания операций в тип.

6.1. Проектирование в терминах объектов.

Объектно-ориентированное проектирование программ -

это расширение использования абстракции данных. Абстрактные типы данных, или объектные типы, не только скрывают структуру данных, но инкапсулируют и все функционирование как операции над объектами. Суть объектно-ориентированного проектирования состоит в нахождении наиболее подходящих объектных типов.

Перед обсуждением некоторых принципов проектирования объектных типов, рассмотрим две объектно-ориентированных программы, сортирующие список целых чисел. Чтобы сконцентрироваться на проектировании, мы не будем показывать деталей реализаций. Первая программа использует абстрактный тип данных *Intlist*, содержащий операторы для сортировки и выдачи списка. Целые числа считываются в список при создании объекта *Intlist*.

```
class IntList {  
    // и т.д.  
public:  
    IntList();  
    sort();  
    write();  
};  
  
main() {  
    IntList *ilist = new IntList();  
    ilist->sort();  
    ilist->write();  
}
```

Другая программа использует абстрактный тип данных *Sortedintlist*, который читает целые числа в процессе создания отсортированного списка. *Sortedintlist* содержит оператор для выдачи списка.

```
class Sortedintlist {  
    // и т.д.  
public:  
    Sortedintlist();  
    wrlte();  
};
```

```
main() {  
    SortedIntlist *slist = new SortedIntlist();  
    slist->write();  
}
```

Какой из этих типов лучше спроектирован, или они оба одинаково хороши?

Хороший способ проектирования объектных типов состоит в поиске существительных и глаголов в описании решаемой задачи. Существительные при проектировании программы становятся объектами, а глаголы становятся операциями. Используя это правило для задачи "сортировка списка целых", разработка программы должна включать тип "список целых", имеющий операцию "сортировать". Программа *Intlist* ближе к "естественному" решению, на которое намекает постановка задачи, тогда как вторая программа вводит специализированный тип, не являющийся необходимым для решения.

Тип отсортированного списка может пригодиться для различных задач, особенно как специальный случай списка. Например, в программе, выполняющей множество разных обработок списков, тип отсортированного списка может пригодиться для оптимизации поиска или объединения списков. Тип сжатого списка, представляющий списки с неповторяющимися элементами, также может быть частью хорошей программы обработки списков.

Это выявляет другой аспект объектно-ориентированного проектирования - использование наследования для создания специализированных версий базовых объектных типов. Вместо создания совершенно другого объектного типа для каждой программы можно строить новые типы из общих базовых типов. В нашей гипотетической программе обработки списков мы используем *List* как базовый класс и получим *Sortedlist* и *Compressedlist* как производные от него. Операция *sort* может быть объявлена как виртуальная функция в базовом классе. Версия *sort* в *Sortedlist* не будет делать ничего. Для *Compressedlist* специальная версия *sort* не будет нужна.

Способ определения ситуации, когда для специализации

типов можно использовать наследование при объектно ориентированном проектировании состоит в поиске прилагательных в описании программируемой задачи.

Для задачи обработки списков "отсортированный список" и "сжатый список" определяют, где можно использовать производные типы. В примере с таблицей идентификаторов из Главы 5, "таблица функций", "глобальная таблица" и "таблица классов" в конечном итоге представлены как классы, производные от класса "таблица".

Описание задачи не всегда подходящим образом определяет наилучшие объектные типы для реализации программы. Нахождение абстракции, общей для нескольких объектов, так что у них может быть одна реализация базового типа, не обязательно является прямой задачей. Проектирование программы - это зачастую процесс проб и ошибок, требующий несколько попыток точного описания задачи, идентификации абстракций и предварительных реализаций. Это не ново и не является особенностью объектно-ориентированного проектирования.

Новое в ООП - это способ рассмотрения программ как реализаций объектных типов, а не как реализаций алгоритмов. Этот концептуальный сдвиг иногда трудно освоить программисту. Абстрактные концепции и сущности, не являющиеся реальными вещами, может быть особенно сложно распознать как кандидатов в объектные типы.

Несложно осознать объектные типы, отвечающие знакомым нам физическим объектам реального мира. Например, устройства типа драйверов диска или ленты являются физическими объектами. Устройства - один из объектов в проблемной области ядра операционной системы. В Главе 5 мы представили класс, служащий общим интерфейсом драйверов устройств.

```
class Device {  
public:  
    virtual int open(char *,int,int);  
    virtual int close(int);  
    virtual int read(int,char *,unsigned);  
    virtual int write(int,char *,unsigned);
```

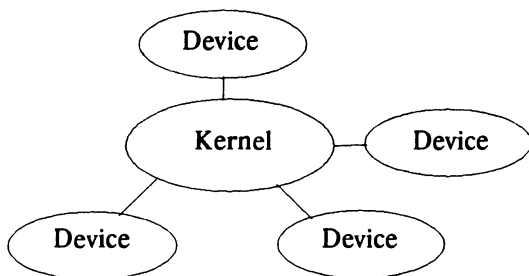
```
int ioctl(Int,Int ...);  
Device();  
~Device();  
};
```

Этот класс также может быть рассмотрен как базовый объектный тип для всех разнообразных устройств при объектно ориентированном проектировании операционной системы. Используя наследование, можно создать вариации типов для представления дисковых и ленточных устройств.

```
class Disk : public Device {  
public:  
    Int open(char *,Int,Int);  
    Int close(Int);  
    Int read(Int,char *,unsigned);  
    Int write(Int,char *,unsigned);  
    Int ioctl(Int,Int ...);  
    Disk();  
    ~Disk();  
};
```

```
class Tape: public Device {  
public:  
    // и т.п.  
};
```

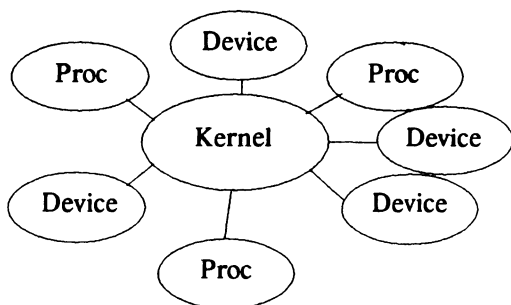
Хотя существует ряд различных типов устройств, ядро рассматривает их как несколько идентичных объектов устройств.



Многие программы никогда не работают напрямую с интуитивно понятными представлениями физических объектов. Необходимые в этих случаях объектные типы являются частью абстрактной проблемной области. В операционной системе процесс может быть объектом, даже если "process" понимается как деятельность. Типичное ядро операционной системы может представлять процесс разбросанными структурами данных и функциями, манипулирующими различными структурами. В объектно-ориентированном проектировании эти структуры данных и функции упаковываются в класс для создания объектного типа "процесс". Интерфейс объектного типа - это множество функций, с помощью которых ядро управляет процессами. В приведенном выше примере `class Proc` предлагает функции для помещения процесса в состояние дремлющего ожидания конкретного события с конкретным приоритетом, пробуждения дремлющего процесса, сохранения и восстановления состояния работающего процесса и для отправки сообщения процессу в форме номера сообщения. Конструктор реализован так, что новый `Proc` может быть создан из другого, как при порождении параллельного процесса. Деструктор удаляет завершенный процесс из системы. Операции подкачки процессов вне и внутри системы не поддерживаются интерфейсом примера, так как это предполагаемая часть управления памятью скрытых элементов объекта `Proc`.

```
class Proc {
public:
    int sleep(Sleepq *,int);
    void wakeup();
    int save_runstate();
    void resume_runstate();
    int send_msg(int);
    Proc(Proc &);
    ~Proc();
};
```

Процессы-объекты теперь можно добавить в область рассмотрения ядра:



Другой аспект объектно-ориентированного проектирования состоит в определении родового типа. Так, в программе обработки списков могут быть "списки целых", "списки строк" и "списки записей о работающих". Все эти списки с различными типами элементов могут быть построены как сущности родового типа список. Ключ к познанию родового типа состоит в том, что это вместилище других типов. Вероятно не стоит строить родовой тип, если только программист не использует более одного примера типа, или тип не будет повторно использоваться в других программах.

Для программиста, использовавшего для декомпозиции проектирование в терминах функций, необходима некоторая практика при переходе к проектированию в объектном стиле. Если программист вступил на путь объектно-ориентированного мышления, он должен быть внимателен, чтобы не перестараться. Как в предыдущем примере, создающем сортированный список вместо сортировки списка, обычно можно создать объектный тип для любой цели. Множество сфабрикованных объектных типов усложняет проектирование программы. Хорошее объектно ориентированное проектирование ясно соответствует задаче, которую должна решать программа, упрощая понимание, реализацию и сопровождение программы.

6.2. Объектные типы как модули.

В ООП элементы задачи отвечают элементам проектиро-

вания, которые являются объектными типами, формирующими программные модули. Одно из преимуществ этого метода построения программ состоит в том, что есть концептуальное единство всех фаз создания программы, и получившиеся модули объектного типа легко повторно использовать.

Концептуальная структура программы сохраняется не только от описания задачи до реализации, но и при совершенствовании разработки. После идентификации объектных типов проектирование уточняется добавлением деталей этих типов. Это контрастирует с пошаговым уточнением функции сверху вниз, при котором проект более высокого уровня преобразуется в другую структуру. Имея полную структуру проекта до того, как известны все детали, мы получаем возможность быстрого прототипирования. Прототипные реализации объектных типов могут быть получены помещением небольшого числа операций и скрытием незаконченных деталей, позволяя проверить осуществимость проекта на ранних стадиях. Возможность быстрого построения прототипов сокращает расходы на проектирование методом проб и ошибок.

Рассмотрим программу обработки слов, манипулирующую строками. Так как потребность в объектах типа строка при разработке определена, *class String* можно будет быстро реализовать с использованием простых структур данных и существующих библиотечных программ, как в нашем примере с *class String*, приведенном в Главе 3.

Эта простая версия *String* может быть использована при прототипировании для точного определения, какие операции над строками требуются. Добавление операций в прототипный *class String* происходит параллельно с уточнением деталей в разработке программы. Пока доказывается осуществимость разработки, прототипную версию может заменить более сложная реализация *String*, например, как представленная в Главе 4.

Поскольку объектно-ориентированное проектирование не является проектированием сверху вниз, полученные программные модули более независимы и, поэтому, их лег-

че повторно использовать. Модуль не включается в иерархию проекта конкретной программы, и поэтому, не зависит от конкретной структуры программы. Модуль объектного типа является реализацией элемента конкретной проблемной области, вместо того, чтобы служить подфункцией одного решения задачи. В этой связи модуль может использоваться в любой программе в той же или смежной проблемной области. Например, *String* может использоваться в любой программе, которой необходим тип для представления слов или текста.

Другой аспект повторного использования модуля состоит в легкости расширения объектных типов наследованием. Исходный модуль остается нетронутым, тогда как новое функционирование добавляется в производный тип. Модули, являющиеся общими объектными типами, могут быть легко повторно использованы для различных вариантов задачи. Специализированный тип *Pathname*, выведенный из *String* в Главе 5 - пример того, как общий тип может быть адаптирован для конкретных целей при помощи наследования.

6.3. Динамический объектно-ориентированный стиль.

ООП связано со стилем программирования, который заключается в создании объектов во время выполнения программы и динамическом связывании операций над объектами. Этот стиль программирования восходит к интерпретирующим объектно-ориентированным языкам и системам типа *Flavors* и *Smalltalk*. Эти системы были разработаны для гибкости во время выполнения и показали себя как действенные инструменты прототипирования, непосредственного решения задач и имитационного моделирования.

Можно писать объектно-ориентированные программы на C++, состоящие полностью из создания образцов объектов во время выполнения и имеющие динамическое связывание операций над объектами с виртуальными функциями. Создание объектов и вызов объектных операций делается в том месте текста программы, которое компилируется. Хотя вы-

зовы виртуальных функций динамически связаны, они проходят статическую проверку типов. То, чего не хватает C++ в смысле интерактивной гибкости, вызвано либо сильной проверкой типов на этапе компиляции, либо сделано для эффективного выполнения программ.

Динамическое ООП часто используется для моделирования. Для моделирования различных сущностей в задаче реализуются объектные типы. Образцы объектов в этом случае задействуются в роли моделей. Объекты взаимодействуют, вызывая операции один над другим и создавая новые объекты, включающиеся в моделирование. Остаток этой главы посвящен программе, моделирующей движение самолетов в аэропорту.

В модели аэропорта используется библиотека задач C++ для того, чтобы разные действия работали как сопрограммы. Сопрограммы в моделировании - это конструкторы объектов класса *task* или производных от него классов. Библиотека задач реализует диспетчеризацию без вытеснений (*nonpreemptive scheduling*), поэтому каждая задача должна себя контролировать, чтобы дать возможность другим задачам работать. В этом моделировании выполнение задачи управляется функциональным элементом *task*,

```
void task::delay(int)
```

который приостанавливает выполнение задачи на данное число моделируемых единиц времени. Глобальная переменная библиотеки *clock* отслеживает время. Функция

```
void task::cancel(int)
```

заканчивает выполнение задачи.

Библиотека также предоставляет классы, реализующие очереди. Классовые типы *qhead* и *qtail* представлены в библиотеке отдельно, потому что они должны иметь возможность функционировать независимо для различных задач, ставящих и выбирающих объекты из очереди. Функции, которые мы используем для работы с очередью, это

```
qtail *qhead::tail()
```

для получения *qtail*, который сцепляется с конкретным *qhead*,

```
Int qtail::put(object *)
```

который добавляет объект в хвост очереди и

```
object *qhead::get()
```

который удаляет объект с головы очереди. Функции для проверки, является ли очередь заполненной или пустой, это

```
Int qtail::rdspace()
```

возвращающая объем, оставшийся для добавления объектов в очередь и

```
Int qhead::rdcount()
```

дающая число объектов, находящихся в очереди. Классовый тип *object* - это базовый тип, из которого должны быть выведены все элементы очереди. Так как *task* и другие классы из библиотеки задач имеют корневой базовый тип *object*, объекты этих типов можно помещать в очереди.

Библиотека задач также содержит генераторы случайных чисел. Мы используем *class urand* для получения случайных чисел, равномерно распределенных по данному интервалу, через функцию

```
Int urand::draw()
```

Функция *main* создает объект-задачу *Airport*, дает ей возможность поработать период моделируемого времени, затем приостанавливает моделирование. Как только *Airport* создан, *main* сама становится задачей, которая выполняется как сопрограмма с другими задачами. Системный указатель

задачи *thistask* необходим для выполнения *main* как сопрограммы.

```
#include "Airport.h"

main() {
    Airport *ap = new Airport;
    thistask->delay(500);
    delete ap;
    thistask->resultis(0);
}
```

Заголовочный файл аэропорта включает заголовок библиотеки задачи. Объектные типы для моделирования строятся из типов библиотеки задач.

```
/* Это файл Airport.h */
#include "task.h"

class Plane : object {
    static int fltcount;
    long start;
    int fltno;
public:
    Plane() { fltno = ++fltcount; }
    long howlong() { return clock-start; }
    void set() { start=clock; }
    int flt() { return fltno; }
};

class PlaneQ {
    qhead *head;
    qtail *tail;
public:
    PlaneQ() { head = new qhead(ZMODE,50);
        tail = head->tail(); }
    void put(Plane *p) { p->set();
        tail->put((object*)p); }
    Plane *get() { return (Plane *)head->get();}
    int lsroom() { return tail->rdspace(); }
    int notempty() { return head->rdcount(); }
};
```

```
class AirControl : public task {
    PlaneQ *landing, *inalr;
public:
    AirControl(PlaneQ *, PlaneQ *);
};

class GroundControl : public task {
    PlaneQ *takingoff, *onground;
public:
    GroundControl(PlaneQ*, PlaneQ*);
};

class Airport : public task {
    PlaneQ *takeingoff, *landing, *inalr, *onground;
    AirControl *acontrol;
    GroundControl *gcontrol;
public:
    Airport();
    ~Airport();
};
```

Объектные типы в этом примере: *Plane* - для самолетов, *PlaneQ* - для очередей самолетов, ожидающих возможности использования аэродрома, *AirControl* - для управления самолетами в зоне аэропорта, *GroundControl* - для управления самолетами на земле и *Airport*, состоящий из очередей самолетов и контроллеров для управления движением в воздухе и на земле. *Airport* сам управляет своим выполнением.

Plane - класс, производный от *object* - базового типа, с которым работает система задач. Реализация *Plane* следит за временем полета и задержкой, используя переменную *clock* из библиотеки задач, которая измеряет модельное время.

PlaneQ инкапсулирует *qhead* и *qtail*, реализующие очередь ожидающих самолетов. *PlaneQ* устроена так, что она содержит пятьдесят самолетов, и *get* от пустой очереди возвращает указатель *null* (Очереди можно создать и так, что *get* от пустой очереди откладывает вызывающую задачу).

AirControl, *GroundControl* и *Airport* - это задачи (tasks). Когда создаются образцы этих типов, их конструкторы выполняются как сопрограммы.

Конструктор *Airport* создает очереди самолетов, в кото-

рых они ожидают обслуживания, а затем - управляющие задачи, обслуживающие очереди. После этого происходит бесконечный цикл до тех пор, пока задача *Airport* не аннулируется. Аэропорт обслуживает один садящийся и один взлетающий самолет за каждые десять тактов модельного времени. Если нет самолетов, уже ожидающих, берутся самолеты из очередей. Сообщения о движении самолета делаются с использованием `printf`. Израсходование самолетами топлива в ожидании посадки не слишком элегантно моделируется аварийным приземлением, удаляющим самолет из модели.

```
Airport::Airport() {  
  
    takingoff = new PlaneQ;  
    Inair = new PlaneQ;  
    landing = new PlaneQ;  
    onground = new PlaneQ;  
  
    acontrol = new AirControl(landing, Inair);  
    gcontrol = new GroundControl(takingoff, onground);  
  
    Plane *tp=0, *lp=0; // ожидающие самолеты  
    int maxwait=30;  
  
    for (;;) {  
        delay(10);  
        if (!lp) lp=landing->get();  
        if (!tp) tp=takingoff->get();  
        if (lp) {  
            if (onground->isroom()) {  
                printf("рейс %d совершает посадку \n",  
                    lp->flt());  
                onground->put(lp);  
                lp=0;  
            }  
            else if (lp->howlong() >> maxwait) {  
                printf("рейс %d разбился!\n",  
                    lp->flt());  
                delete lp;  
                lp=0;  
            } else  
                printf("рейс %d ,посадка задерживается \n",
```

```
        lp->flt());
    }
    if (tp) {
        if (lnair->lsroom()) {
            printf("рейс %d взлетает off\n",
                tp->flt());
            lnair->put(tp);
            tp=0;
        } else
            printf("рейс %d ,взлет откладывается \n",
                lp->flt());
    }
}
```

Деструктор *Airport* прекращает задачу *AirControl* тем, что в список ожидающих посадку не поступает больше самолетов. Это продолжается до тех пор, пока все ожидающие самолеты не приземлятся, затем прекращаются оставшиеся задачи, моделирующие аэропорт.

```
Airport::~~Airport() {
    acontrol->cancel(0);
    while (landing->notempty())
        thistask->delay(10);
    gcontrol->cancel(0);
    printf("Аэропорт закрыт\n");
    cancel(0);
}
```

AirControl обрабатывает запросы от самолетов на посадку с интервалом от одного до тридцати тактов. Случайный интервал между прибытиями реализуется с использованием *urand* - тип генератора случайных чисел, предлагаемый библиотекой задач. В нашем случае, когда моделируется единственный аэропорт, самолеты создаются для постановки в очередь на приземление или пересоздаются из уже взлетевших из аэропорта.

```
AirControl::AirControl(PlaneQ *landing,
    PlaneQ *lnair) {
```

```

urand n(1,30);

for (;;) {
    delay(n.draw());

    Plane *p = lna1r->get();
    if (!p) {
        p = new Plane;
    }
    if (landing->lsroom()) {
        printf("рейс %d запрашивает посадку\n", p->flt());
        landing->put(p);
    } else {
        printf("рейс %d направлен на другой аэродром\n",
            p->flt());
        delete p;
    }
}
}
}

```

GroundControl использует для обслуживания самолета на земле от десяти до тридцати тактов, а затем самолет может снова взлетать.

```

GroundControl::GroundControl(PlaneQ *takingoff,
    PlaneQ *onground) {

    urand n(10,30);
    Plane *p=0;

    for (;;) {
        delay(n.draw());
        if (!p) p=onground->get();
        if (p) {
            if (takingoff->lsroom()) {
                printf("рейс %d отправляется\n", p->flt());
                takingoff->put(p);
                p=0;
            } else
                printf("рейс %d задерживается\n", p->flt());
        }
    }
}
}
}

```

Вот результат работы программы, моделирующей аэропорт:

```
рейс 5 запрашивает посадку
рейс 82 разбился!
рейс 84 запрашивает посадку
рейс 9 посадка задерживается
рейс 48 отправляется
рейс 9 совершает посадку
рейс 48 взлетает
рейс 48 запрашивает посадку
рейс 17 отправляется
рейс 83 совершает посадку
рейс 17 взлетает
рейс 17 запрашивает посадку
рейс 5 разбился!
рейс 84 разбился!
рейс 32 отправляется
рейс 48 совершает посадку
рейс 32 взлетает
рейс 17 разбился!
рейс 60 отправляется
Аэропорт закрыт
```

При нашем моделировании возникло множество аварийных приземлений. Это вызвано, по-видимому, не перегрузкой приземляющимися самолетами, а временем ожидания и перегрузкой наземных очередей. Можно испробовать различные значения для наибольшей задержки посадки, ожидания наземного обслуживания, интервала между прибытиями и размера очередей и рассмотреть их влияние на появление аварий, задержки и переадресованные полеты.

6.4. Упражнения.

- Упр.6.1.** Улучшите обработку аварийных посадок в модели аэропорта.
- Упр.6.2.** Реализуйте абстрактный тип данных *Intlist*, использованный в примере в первой части этой главы. Реализуйте тип *Sortedlist* как производ-

ный от *Intlist*.

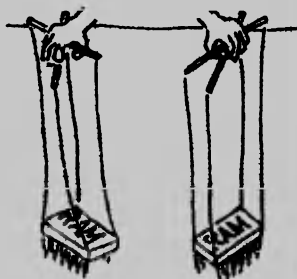
- Упр.6.3.** Спроектируйте заново модули воздушного и наземного управления в модели аэропорта как специализированные версии управляющего типа.
- Упр.6.4.+** Расширьте модель аэропорта так, чтобы она включала управление полетами в государственном масштабе между различными аэропортами.
- Упр.6.5.** Многие приложения концептуализируются как переход из начального состояния через последовательность состояний в соответствии со входными данными. В каждом состоянии выполняется некоторое действие. Автоматизированные банковские программы, системы бронирования авиабилетов, лексические анализаторы компиляторов могут быть концептуализированы таким образом. Обычно переходы из состояния в состояние кодируются таблицей, индексируемой текущим состоянием и целым кодом, соответствующим входу. Для каждого нового состояния выполняется соответствующее ему действие.

В чем преимущества и потенциальные трудности этого подхода?

Альтернативный подход к этой концептуализации состоит в создании объекта "состояние", инкапсулирующего всю семантику для данного состояния, включая следующее состояние и необходимые действия.

В чем преимущества объектно-ориентированного подхода в этой ситуации? В чем недостатки?

Разработайте автоматизированную банковскую систему и реализуйте ее с использованием обоих подходов.



УПРАВЛЕНИЕ ПАМЯТЬЮ

В книгах по программированию тема управления памятью часто считается не слишком важной или же областью, которой занимается язык или операционная система. Соответствующий контроль над управлением памятью, тем не менее, существует для корректности программы и может быть важен для ее эффективности.

Неэффективное выделение и освобождение памяти может сильно снизить эффективность программы, порождая слишком много системных вызовов для получения дополнительной памяти, фрагментируя доступную память и вызывая "метания", при которых страницы непрерывно подкачиваются и откачиваются из памяти, или (в определенных аппаратных средах) полностью выходя за пределы доступной памяти. Прямолинейные попытки избавиться от таких неэффективностей могут привести к некорректному управлению памятью, как, например, преждевременное освобождение или случайное переименование блока памяти.

Требования к управлению памятью определяются задачей. В некоторых случаях все потребности в памяти статически определены, в других - они почти полностью динамические. Часто имеется специфическая информация о задаче, которую можно учитывать при построении эффективной схемы управления памятью. Хотя C++ поддерживает общее управление памятью по умолчанию, он также дает

возможность приспособливать управление памятью для данной проблемной области, класса или приложения.

7.1. Управление памятью при помощи конструкторов и деструкторов.

Мы рассматривали конструкторы как средство для размещения новых объектов, как инициализаторы уже существующих объектов и как род предопределенного пользователем преобразования. Это просто три различных взгляда на одну и ту же операцию, создающую объект данного типа из последовательности значений.

В отличие от функциональных элементов, конструкторы имеют скрытый механизм, который явно не приводится программистом. Мы уже писали о некоторых его свойствах, таких, как автоматическая инициализация базовых классов и элементов, имеющих конструкторы, которые можно вызывать без аргументов. Рассмотрим некоторые конструкторы детально и посмотрим, что происходит за кулисами.

```
complex::complex( double r=0.0, double i=0.0) {  
    re = r;  
    im = i;  
};
```

Конструктор класса *complex* прямолинеен. Во-первых, он убеждается, что есть память для комплексного числа, а затем он выполняет тело конструктора (представленное программистом) для инициализации памяти. Каждый конструктор начинается с некоторого неявно вставленного кода, проверяющего значение *this*, чтобы определить, существует ли объект для работы. Если *this* - пустой, то объекта нет и конструктор вызывает *operator new* для получения памяти под объект.

Два случая, в которых неявный аргумент *this* не *null* - это для статических и автоматических объектов. Статический объект (со спецификаторами *static* либо *extern*) существует всегда. Автоматический объект получает память автоматически при входе в функцию, в которой он определен.

```

complex zero;    // статическая
complex zilch() {
    complex a(1,2); // автоматическая
    return a * zero;
}

```

Память для *zero* существует всегда, а память под *a* выделяется при входе в *zilch*. В обоих случаях `operator new` не вызывается в `complex::complex`. Но во фрагменте типа

```
complex *cp = new complex;
```

значение `this` при входе в `complex::complex` равно `null` и вызывается `operator new`.

Другой случай, когда `this` - не `null` при входе в конструктор, - это инициализация базового класса и элементов, так как память для этого будет уже выделена конструктором производного или содержащегося класса. Например, при размещении одного из производных типов вершин абстрактного синтаксического дерева в Главе 5, мы вызвали конструктор производного класса, который выделяет память (при необходимости) и вызывает конструктор базового класса. Фрагмент

```
Node *np = new Int(5);
```

вызывает конструктор для класса *Int* с пустым (`null`) значением `this`.

```

Int::Int( int v ) {
    value = v;
}

```

Сначала конструктор выделяет память для объекта *Int*, вызывая `operator new` и присваивая полученный адрес `this`. Затем неявно вызывается конструктор базового класса `Node::Node`. Так как `this` - не `null` при вызове конструктора базового класса, то `operator new` в `Node::Node` не вызывается.

Тот же результат имеет место и для многоуровневой иерархии классов. Когда мы пишем

```
extern Node *x, *y;  
Node *np = new Plus(x,y);
```

мы вызываем

```
Plus::Plus( Node *l, Node *r)  
: Binop(l,r) {}
```

выделяющий память для объекта *Plus* и явно вызывающий конструктор базового класса *Binop::Binop* с инициализированным указателем *this* и указателями на левое и правое поддеревья. Конструктор *Binop* неявно вызывает конструктор *Node*. Так как память под объект *Plus* уже была выделена (в том числе и для частей *Plus - Node* и *Binop*), конструкторы *Node* и *Binop* не вызывают *operator new*.

Как и конструкторы, деструкторы имеют скрытый механизм вызова деструкторов элемента и базового класса и *operator delete* для освобождения памяти. Точно так же, как конструкторы классов выделяют память для элементов и базовых классов, деструктор класса освобождает память своих элементов и базовых классов. Вернемся к типу вершины унарный минус из иерархии вершин Главы 5.

```
class Uminus : public Node {  
    Node *operand;  
public:  
    Uminus(Node *o) { operand = o; }  
    ~Uminus() { delete operand; }  
    // ...  
};
```

Конструктор *Uminus* сначала выделяет память, потом вызывает конструктор базового класса *Node*. После возврата из конструктора базового класса, он выполняет свое тело, инициализируя *operand*. Деструктор действует, по сути, в обратной последовательности. Сначала он выполняет свое

тело, удаляя operand (явным вызовом деструктора), потом вызывает деструктор базового класса *Node*. После возврата из деструктора базового класса *Uminus::~~Uminus* вызывает **operator delete** для освобождения памяти класса. Аналогично вызову **operator new**, **operator delete** не вызывается деструкторами базового класса или элемента, а также если уничтожаемый объект статический или автоматический. Последовательность вызовов деструкторов базового класса или элемента и вызов **operator delete** вставляется перед каждым **return** в теле деструктора, включая неявный **return** в конце блока функции деструктора.

7.2. Операторы new и delete.

Операторы **new** и **delete** - предопределенные библиотечные функции, управляющие свободной памятью или "кучей".

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

Обратите внимание, что **operator new** возвращает блок свободной памяти размером в столько байт, сколько определяется его аргументом, а **operator delete** освобождает блок памяти, предварительно выделенный с помощью **operator new**. Кроме того, **operator new** возвращает пустой указатель, если нельзя выделить требуемый объем памяти.

```
f0{
    int *lp = new int[1000000];
    if (!lp) {
        error("исчерпали память!");
        exit(1);
    }
}
```

Для увеличения контроля над выделением памяти внешний указатель `_new_handler` может быть установлен на функцию, которую надо вызывать, если **operator new** потерпел неудачу. До тех пор, пока **operator new** не сможет выде-

лить требуемую память или `_new_handler` не `null`, `operator new` повторно вызывает функцию, на которую ссылается `_new_handler`.

```
extern void (*_new_handler)();

void
give_up() {
    error("исчерпали память!");
    exit(1);
}

void
increase_limit() {
    // запрашивает память у ОС
    extern long ulimit(int,long);
    ulimit(3,0);
    _new_hendler = give_up;
}

f() {
    _new_handler = increase_limit;
    int *ip = new int[1000000];
}
```

Теперь, если `operator new` не сможет выделить требуемый объем памяти, он будет вызывать `increase_limit` (неявно, через `_new_handler`) для увеличения объема памяти, доступного программе. Если это не удастся или последующие вызовы `operator new` безрезультатны, `give_up` печатает сообщение и прекращает выполнение программы.

Это очень гибкий механизм, и "стандартные" библиотечные операторы `new` и `delete` подходят для большинства задач. Мы можем также перегрузить стандартные версии, предлагая собственные `operator new` и `operator delete`. Эти определенные пользователем версии используются вместо стандартных.

Например, `operator new` не гарантирует, что выделяемая память инициализирована нулями, поэтому мы можем захотеть предоставить версии `new` и `delete`, гарантирующие обнуление памяти.

```

extern void *
operator new(long sz) {
    // calloc возвращает обнуленную память
    extern char *calloc(unsigned,unsigned);
    return calloc(1,sz);
}

extern void
operator delete(void *p) {
    extern void free(char *);
    free((char *)p);
}

```

Запомните, что установка `_new_handler` больше не будет влиять на поведение `operator new` (если мы явно это не запрограммируем). `_new_handler` - это просто часть реализации стандартного библиотечного `operator new`.

При замене стандартных программ управления памятью мы можем быть более честолобивы. Например, если наши программы производят много выделений небольших фрагментов памяти и редко освобождают память сразу после выделения, мы можем разработать эффективную схему управления памятью, учитывающую эту модель использования.

```

extern void(*_new_handler)() = 0;

extern void *
operator new(long nbytes) {
    extern char *calloc(unsigned,unsigned);
    static const long BSIZ = 4*1024;
    static char ibuf[BSIZ];
    static char *start = ibuf;
    static char *end = ibuf + BSIZ;
    static const long ALIGN = sizeof(double);

    if (nbytes & (ALIGN-1))
        nbytes += ALIGN - nbytes % ALIGN;

    if (end - start < nbytes) {
        long bufsiz = BSIZ > nbytes ? BSIZ : nbytes;
        while (!(start = calloc(1,bufsiz)))
            if (_new_handler)

```

```
    _new_handler();  
    else  
        return 0;  
    end = start + bufsiz;  
}  
start += nbytes;  
return start - nbytes;  
}
```

Эта версия `operator new` сначала округляет запрашиваемый объем памяти для обеспечения выравнивания (следующего требования), потом проверяет, достаточно ли имеется свободного места в текущем буфере для выполнения запроса. Для удобства пусть два указателя определяют первое свободное место в текущем буфере и конец буфера. Если в текущем буфере достаточно места для удовлетворения запроса, указатель, определяющий следующее свободное место, обновляется и возвращается адрес выделенной памяти.

Если в текущем буфере недостаточно места для удовлетворения запроса, тогда у операционной системы запрашивается новый буфер минимального размера, но достаточно большой для выполнения запроса. Если запрос нового буфера неудачен, применяется стандартный механизм `_new_handler`, дающий пользователю возможность сделать некоторое восстановление.

Запрашивая память у операционной системы нечасто, большими блоками, эта версия `operator new` будет работать лучше, чем стандартная версия, для программ, выдающих частые запросы на небольшие объемы памяти. Так как начальный буфер размещен статически, запросов к операционной системе может вообще не поступить.

Так как наши программы редко освобождают память, удаления можно игнорировать. Так как `operator delete` вообще ничего не делает, наиболее эффективно реализовать его подставляемой функцией.

```
inline void  
operator delete(void *) {}
```

Важный аспект, присущий нашим версиям `operator new` и `operator delete` - это то, что они имеют тот же интерфейс пользователя, что и стандартные `new` и `delete`. Частично это обязательно: `operator new` должен быть типа `void*(long)`, а `operator delete` должен быть типа `void(void*)`. Мы также сохраним действия по ошибке `operator new`, то есть возврат `null` при неудаче. Мы можем затем инкапсулировать эти программы в файл, предохраняя пользователя от использования специфических особенностей реализации. Поддерживание стандартного интерфейса и скрытие деталей реализации дает те же возможности, что и абстрактный тип данных (фактически, это и есть абстрактный тип данных), и мы можем менять реализацию управления памятью, не влияя на программы пользователя.

7.3. Управление памятью для массивов.

Стандартная библиотека C++ определяет две функции, управляющие памятью для массивов классовых объектов: `_vec_new` и `_vec_delete`.

```
void *_vec_new(void *,int,int,void *);
void _vec_delete(void *,int,int,void *,int);
```

Эти функции используются не для всех размещений и удалений массивов, а только для тех, которые требуют инициализацию конструктором или уничтожения. Так,

```
int *ip = new int[3][4][5];
```

вызывает в конечном итоге `operator new`, запрашивая память для 60-ти `int`, а

```
struct Pair {
    void *first,*second;
};
Pair *list = new Pair[12];
```

требует пространства, достаточного для 12-ти *Pair*. Если все элементы массива требуют инициализации конструктором, как в

```
struct Pair2 {  
    void *first,*second;  
    Pair2() { first = second = 0; }  
};  
Pair@ *list2 = new Pair2[12];
```

то сначала вызывается `_vec_new` для выделения пространства под массив, а потом конструктор для каждого элемента.

Аналогично, `_vec_delete` вызывается, если тип элементов удаляемого массива имеет деструктор. В этом случае необходимо явно передавать размер массива.

```
struct Strlist {  
    Strlist *next;  
    String str;  
    Strlist();  
    ~Strlist();  
};  
Strlist *slist = new Strlist[100]; //вызывает _vec_new  
delete [100] slist; //вызывает _vec_delete
```

Если информация о размере пропущена, компилятор сгенерирует вызов `operator delete` для удаления одного *Strlist*. В случае удалений, не приводящих к вызову `_vec_delete`, информация о размере игнорируется.

```
int *lp = new int[1000000];  
delete [10] lp; // информация о размере  
                // игнорируется,  
                // к счастью...  
delete [12] llist2; // информация о размере игнорируется
```

Как для `operator new`, так и для `operator delete`, мы можем предложить собственные версии `_vec_new` и `_vec_delete`. Аргументами `_vec_new` являются последовательно, адрес "раз-

мещаемого" массива, число элементов в массиве, размер (в байтах) элемента массива и адрес инициализирующего конструктора. Первый аргумент используется аналогично `this` в конструкторе, так как `_vec_new` и `_vec_delete` используются и для инициализации и уничтожения статических и автоматических массивов.

```
static Strlist hashtable[512];
```

Первый аргумент `_vec_new` - адрес массива для статических и автоматических массивов и нуль в противном случае. Аргументы `_vec_delete` те же, что и для `_vec_new`, кроме того, что есть добавочный аргумент, сигнализирующий о необходимости освобождения памяти. Для статических и автоматических массивов флаг равен нулю и память не освобождается; для других - флаг равен единице и память освобождается.

Интерфейс `_vec_new` определяет его строго ограниченное использование. Так как единственная информация, представляемая `_vec_new` - это адрес конструктора, то невозможно вызвать `_vec_new` с конструктором, имеющим аргументы (даже предопределенные). В существующих реализациях C++ нельзя `_vec_new` вызывать с конструктором для класса, имеющего виртуальные базовые классы.

```
complex cary[10]; //ошибка:  
Monitored_network_device boards[3]; //ошибка!
```

В первом объявлении неявная инициализация конструктором использует предопределенные аргументы. Во втором объявлении некоторые из базовых классов виртуальны. Так как область допустимых инициализаторов для массивов классовых объектов этим ограничивается, имеется хороший, в общем-то, совет - избегать массивов из классовых объектов за исключением наиболее простых классовых типов.

7.4. Специфические new и delete для классов.

Рассмотренные нами реализации new и delete были общего назначения. Как и стандартные библиотечные версии, их предполагалось использовать для всех типов во всех случаях. Как мы уже рассмотрели ранее, для значительного ускорения управления памятью может быть задействована специфическая информация о типе или реализации. Для использования специализированных сведений C++ предлагает возможность определения специфически-классовых версий operator new и operator delete.

Например, класс *complex* имеет очень маленькую и простую реализацию, и, кроме того, не требует обнуления памяти. Мы можем значительно ускорить управление памятью при помощи специфически-классовых операторов new и delete, учитывающих конкретные детали реализации.

```
class complex {  
    double re,im;  
public:  
    void *operator new(long);  
    void operator delete(void *);  
    // ...  
};
```

Эти операторы вызываются при любом размещении или удалении объекта типа *complex*, независимо от того, выполняется размещение и удаление в контексте класса *complex* или нет.

```
#include <complex.h>  
  
main() {  
    complex *cp = new complex(12),  
    int *ip = new int;  
    *ip = 12;  
    complex cgross = *cp * *ip,  
}
```

Обратите внимание, что `sr` ссылается на память, выделенную `complex::operator new`, а `ip` ссылается на память, выделенную `::operator new`. Кроме того `sgross` - автоматический и не требует вызова для инициализации какого либо `operator new`.

Мы реализуем управление памятью для `complex`, размещающая элементы статического массива. Освобожденные элементы содержатся в списке свободных. Реализация `complex::operator new` сначала просматривает список свободных в поисках предварительно размещенного и освобожденного элемента. Если такого нет, но есть элементы массива, которые еще не размещались, размещается один из них. Иначе `operator new` терпит неудачу и возвращает `null`.

```
static const int MAX = 100;

static class rep {
    static rep *free;
    static int num_used;
    union {
        double store[2];
        rep *next;
    };
    friend complex;
} mem[MAX];

void *
complex::operator new(long) {
    if (rep::free) {
        rep *tmp = rep::free;
        rep::free = rep::free->next;
        return tmp->store;
    }
    else if (rep::num_used < MAX)
        return mem[rep::num_used++].store;
    else
        return 0;
}
```

Реализация `complex::operator delete` как раз и добавляет освобожденный элемент массива в голову списка свободных элементов.

```
void
complex::operator delete(void *p) {
    ((rep *)p)->next = rep::free;
    rep::free = (rep *)p;
}
```

Помните, что *complex::operator new* не только проверяет значение аргумента, но всегда дает память достаточного размера для объекта типа *complex*. Неявно предполагается, что не будет классов, производных от класса *complex*. Это неплохое предположение, но в случае его невыполнения мы можем попасть в неприятное положение.

```
class point3 : public complex {
    // точка в трехмерном пространстве
    double z;
public:
    // ...
};
point3 *p = new point3;
```

Так как *point3* наследует *new* и *delete* вместе с другими элементами класса *complex*, вызов конструктора для *point3* вызывает *complex::operator new* и возвращает память, достаточную для *complex*, но недостаточно большую для *point3*.

Мы можем продвинуться в разработке специфически классовых *new* и *delete* дальше учета свойств типа; мы можем также разработать схему распределения памяти, зависящую от свойств программы в целом.

Рассмотрим программу-калькулятор, для которой мы разработали иерархию в Главе 5. Пусть наша программа калькулятор строит только одно абстрактное синтаксическое дерево, вычисляет его и затем удаляет все дерево. Учитывая это, мы можем разработать более эффективное управление памятью.

```
classNode {
protected:
    Node() {}
```

```
void *operator new(long);  
void operator delete(void *);  
// ...  
};
```

Node::operator new (и *Node::operator delete*) унаследованы классами, выведенными из *Node*, поэтому мы должны или предоставить отдельный *operator new* для каждого производного класса, или сделать *Node::operator new* достаточно общим для работы со всеми типами вершин. Здесь мы реализуем последнее.

```
static const int SZ = 1000;  
  
static struct buf {  
    buf *next;  
    char mem[SZ];  
} b,*bp = &b;  
  
static char *memp = b.mem;  
  
void *  
Node::operator new(long sz) {  
    if (memp+sz > &bp->mem[SZ])  
        if (bp->next) {  
            bp = bp->next;  
            memp = bp->mem;  
        }  
    else if (bp = bp->next = new buf)  
        memp = bp->mem;  
    else  
        return 0;  
    char *r = memp;  
    memp += sz;  
    return r;  
}
```

Мы начнем со статически размещенного буфера, размер которого мы предполагаем достаточным для большинства запусков программы и по необходимости будем выделять дополнительные буферы из свободной памяти.

Помните, что использование *new* для выделения памяти

под дополнительные буферы имеет в виду общий **operator new**, так же, как и для буфера, у которого тип не *Node* и для типа, не производного от него.

operator delete определяет тот факт, что любое использование **delete** предназначено для удаления целого абстрактного синтаксического дерева. Мы игнорируем аргумент и переустанавливаем указатели управления памятью в их начальные значения. Мы должны помнить обо всех добавочных буферах, которые мы, возможно, разместили для предотвращения повторного размещения. Мы избавляемся от деструкторов в *Node* и в производных от него классах, так как они излишни.

```
void
Node::operator delete(void *){
    bp = &b;
    memp = b.mem;
}
```

Эта версия **new** и **delete** работает гораздо быстрее стандартной; она, кроме того, тесно связана с нашим способом использования абстрактных синтаксических деревьев в конкретной программе.

Как и для функциональных элементов, возможно, хотя и редко бывает разумным, присваивание **this** в конструкторе. Делая так, программист снимает задачу выделения памяти классовому объекту. Если есть явное присваивание **this** в теле конструктора, то неявное обращение к **operator new** не производится, и все явные и неявные инициализации базовых классов и элементов после каждого такого присваивания повторяются.

Присваивание **this** в конструкторе опасно. Рассмотрим выделение памяти в конструкторе элемента списка, рассмотренного в Главах 4 и 5. Для иллюстрации мы вынесли инициализации элементов из тела конструктора в список инициализации.

```
list::list(ETYPE e, list *lst) : el(e), link(lst) {
    if (!this)
        this = my_malloc(sizeof(list));
}
```

Этот фрагмент не будет работать для статических и автоматических объектов. Инициализации в списке инициализации элементов выполняются только после присваиваний *this*, но для статических и автоматических объектов присваивание не выполняется, так как *this* при входе в конструктор имеет непустое значение (не *null*).

Даже если мы ожидаем, что статические и автоматические объекты типа *list* создаваться не будут, проблемы остаются, так как *list* может служить базовым классом или элементом другого класса.

```
list2::list2(ETYPE e, list2 *fl, list2 *bl)
    : list(e, fl), link(bl) {}
```

В этом фрагменте есть две проблемы. Первая состоит в том, что так как значение *this* непустое для инициализации базового класса, *list::el* и *list::link* не будут инициализированы. Вторая - память для объекта *list2*, включая часть *list*, представляется *operator new*, а не *my_alloc*. Будет ли это проблемой или нет, зависит от программы, но предварительно мы имели основания для реализации альтернативной процедуры выделения памяти для *list*.

Первую проблему мы можем уладить, вводя кажущиеся ненужным присваивания *this* в *list::list*, чтобы значение *this* на входе не влияло на инициализацию.

```
if (this)
    this = this; // статический, автоматический,
                // базовый или элемент
else
    this = my_alloc(sizeof(list));
```

Это не только неестественный и располагающий к ошибкам путь; он также дважды вызывает вставку неявного иници-

циализирующего кода в тело конструктора.

Аналогично можно получить контроль над освобождением памяти в деструкторах присваиванием значения `this`. Если значение `this` перед `return` пустое, то деструкторы элементов и базовых классов не вызываются и не производится обращение к `operator delete`.

В общем случае предпочтительнее использовать специфически-классовые версии `operator new` и `operator delete`, чем присваивание `this`.

7.5. Оператор `->`.

Управление памятью связано не только с выделением и освобождением памяти, но и с доступом к ней. Программы часто должны содержать фрагменты, которые выполняют некоторые действия либо до, либо после доступа к памяти.

сделать что-либо;

получить память;

сделать что-то еще.

Так как эти фрагменты не вставляются автоматически, их можно пропустить неумышленно или для "оптимизации".

```
class info {  
    void f0;  
    // ...  
};  
  
info *p; // ВСЕГДА проверяйте, что p - не null!!!  
  
// ...  
  
void  
process_info() {  
    // p обычно не null..  
    p->f0;  
    // ...  
}
```

Разумная оптимизация в `process_info` ведет к катастрофе. Чтобы помочь в преодолении трудностей такого типа, C++

позволяет перегружать оператор `->`, давая возможность создавать "умные" указатели, инкапсулирующие семантики проверки и доступа. Как `operator ()` и `operator []`, `operator ->` должен быть функциональным элементом. Он должен быть объявлен без аргументов и возвращать либо классовый тип, для которого определен `operator ->`, либо указатель на классовый тип.

Имейте в виду, что хотя предопределенный оператор `->` бинарный, определяемый пользователем `operator ->` объявляется как унарный. Идея состоит в том, что определенный пользователем `operator ->` вырабатывает, прямо или косвенно, указатель на объект классического типа. Этот указатель затем используется с предопределенным оператором `->` для доступа к элементу класса.

```
class InfoP {
    static Info null_info;
    Info *p;
public:
    Info *operator ->()
    { return p ? p : &null_info; }
    InfoP(Info *ip)
    { p = ip; }
    InfoP &operator = (InfoP ip)
    { return *this = ip; }
};
```

Это одно из решений предыдущей проблемы; `InfoP` - это "умный" тип указателя, никогда не возвращающий пустое значение для использования в ссылке. Кроме того, он защищает любые операции над указателями, типа `++`, `+` и т.п., которые могут привести к ошибочному адресу.

```
InfoP p;
// ...
void
process_info() {
    p_.f(); // безопасно
    // ...
}
```

`p -> f` сначала вызывает `infoP::operator ->` для `p`, вырабатывающий значение типа `info*`, которое затем используется для доступа к элементу `info::f`.

`operator ->` можно определить так, что он будет возвращать классовый тип, определяющий `operator ->`.

```
class infowarn {
    infoP &p;
public:
    infowarn(infoP &ip) : p(ip) {}
    infoP &operator ->() {
        warn("получаем информацию!");
        return p;
    }
};
```

Объект `infowarn` при использовании ведет себя как объект `infoP`, за исключением того, что он печатает сообщение перед выполнением надежного доступа.

```
infowarn wp = p;
// ...
wp->f();
```

Обращение `wp->f` сначала вызывает `infowarn::operator ->`, печатающий сообщение и возвращающий `infoP`. Затем вызывается `infoP::operator ->`, возвращающий `info*`, используемый для доступа к `info::f`.

В качестве более реалистичного примера использования "умных" указателей, предположим, что есть файл из записей на диске, и мы хотим иметь возможность писать программы, использующие эти записи так, как если бы они были элементами массива в памяти, не связываясь с системно-зависимыми функциями чтения и записи на диск. Так как записей много, мы не можем просто считать их все в память, но в любом случае относительно мало записей просматривается любой данной программой и еще меньше изменяется. Мы не можем предоставить возможность чтения и записи большого числа дисковых записей.

Для решения этой задачи мы предоставим два типа. Пер-

вый представляет файл записей, а второй - "умный" указатель на запись файла.

```
struct rec { // дисковая запись
    int key;
    int value;
    // ...
};
```

```
class File {
    FILE *fp;
    sary *hd;
    rec *operator [](int);
public:
    File(char *);
    ~File();
    friend Ptr;
};
```

Пользователь создает файловый объект, предлагая составное имя файла, который надо открыть. Тип FILE и программы для открытия, закрытия, чтения и записи файлов являются системно-зависимыми. Здесь мы используем функции stdio.

```
File::File(char *fname) {
    fp = fopen(fname, "r+");
    hd = 0;
}
```

Так как мы предполагаем считывать и записывать относительно мало записей, мы реализуем *File* как разреженный массив. Наш разреженный массив - это неупорядоченный связный список индексов, пар записей.

```
class sary {
    sary *next;
    int index;
    rec *p;
    sary(int, sary *, FILE *);
    friend File;
};
```

Итак, *File* состоит из системно-зависимого дескриптора файла и разреженного массива записей файла. *File::operator []* создает элементы разреженного массива как только есть попытка индексировать элемент, не присутствующий в массиве.

```
rec *
File::operator [] (int l) {
    for (sary *sp = hd; sp; sp = sp->next)
        if (sp->Index == l)
            break;
    if (!sp)
        sp = hd = new sary(l,hd,fp);
    return sp->p;
}
```

Конструктор элемента разреженного массива создает элемент и считывает соответствующую запись с диска. Кроме того, он делает копию записи для дальнейшего сравнения. Запись на диске не обновляется, если она не была изменена.

```
sary::sary(int l, sary *n, FILE *fp) {
    const int sz = sizeof(rec);
    Index=l;
    next=n;
    p=(rec *) new char[2*sz];
    fseek(fp,l*sz,0);
    fread((char *)p,sz,1,fp);
    *(p+1) = *p;
}
```

Деструктор для *File* обновляет те записи на диске, которые были изменены, удаляет разреженный массив и закрывает файл.

```
File::~~File() {
    const int sz = sizeof(rec);
    sary *sp = hd;
    while (hd) {
        if (memcmp(sp->p[0],sp->p[1],sz) != 0) {
```

```

        fseek(fp,sp->Index*sz,0);
        fwrite((char *)sp->sz,1,fp);
    }
    hd = hd->next;
    delete sp;
}
fclose(fp);
}

```

Основное общение пользователя с файлом из записей базы данных происходит через тип "умного" указателя.

```

class Ptr {
    Int Index;
    File *fp;
public:
    rec *operator->();
    rec &operator[](Int);
    rec &operator *();
    Ptr(File *f,Int I=0) : fp(f),index(I) {}
    friend Ptr operator +(Ptr,Int);
    friend Ptr operator +(Int,Ptr);
    friend Int operator -(Ptr,Ptr);
    friend Ptr operator -(Ptr,Int);
    Ptr &operator ++();
    Ptr &operator --();
    Ptr &operator ++(Int);
    Ptr &operator --(Int);

    friend Int operator ==(Ptr,Ptr);
    friend Int operator !=(Ptr,Ptr);
    friend Int operator <(Ptr,Ptr);
    friend Int operator <=(Ptr,Ptr);
    friend Int operator >(Ptr,Ptr);
    friend Int operator >=(Ptr,Ptr);
};

```

Ptr реализован как указатель на *File* и, учитывая, что *File* - массив записей, на текущий индекс *File*. Мы предоставляем все обычные операции арифметики указателей.

```

Ptr &
Ptr::operator ++() {

```

```
    index++;  
    return *this;  
}  
  
int  
operator — (Ptr a,Ptr b) {  
    return a.index — b.index;  
}
```

Реализации других операторов аналогичны. Арифметические операции над указателями выполняются над индексом файловой записи, и никакие ссылки на конкретный *File* не производятся, пока не появится необходимость.

Текущая работа со ссылкой выполняется *File::operator []*, и наш "умный" указатель возвращает элемент массива *File*, соответствующий *Ptr::index*.

```
rec *  
Ptr::operator ->() {  
    return (*fp)[index];  
}
```

Для предопределенных типов указателей в C++ есть соответствие между операторами: \rightarrow , $*$ и $[]$ типа $p \rightarrow m$ эквивалентно $(*p).m$, или $*(p+i)$ эквивалентно $p[i]$. Как и для других эквивалентностей, существующих в предопределенных частях языка, например, между $++$ и $+= 1$, если мы хотим, чтобы аналогичные эквивалентности сохранились для наших определенных пользователем операций, их надо явно определить.

```
rec &  
Ptr::operator *() {  
    return *(*fp)[index];  
}  
  
rec &  
Ptr::operator [] (int i) {  
    return *(*fp)[index+i];  
}
```

Теперь мы можем писать программы. Пусть у нас есть упорядоченный массив записей базы данных на диске, и мы хотим найти и изменить значение одной записи.

```
void
alter(int key,char *file,int num_recs,int val) {
    File f = file;
    Ptr mid = f;
    Ptr low(f,0),high(f,num_recs-1);
    while (low <= high) {
        mid = low + (high-low)/2;
        if (mid->key < key)
            high = mid - 1;
        else if (mid->key > key)
            low = mid + 1;
        else {
            mid->value = val;
            break;
        }
    }
}
```

Обратите внимание, что `alter` открывает файл, создает разреженный массив записей файла объявлением `f` и создает три "умных" указателя на `f`, `low`, `mid` и `high`, которые используются для бинарного поиска файловых записей. Если запись, соответствующая аргументу `key` найдена, ее значение изменяется. Деструктор для `f` активизируется перед возвратом для очистки разреженного массива файловых записей и обновления дисковых записей для измененных элементов.

7.6. X(X&).

Конструктор, который можно вызвать с аргументом его собственного классового типа, определяет, как инициализировать объект этого класса. Так как такой конструктор обычно пишется как воспринимающий единственный аргумент типа ссылки на класс, элементом которого он является, то эта концепция записывается как `X(X&)` (произносится "X от ссылки X").

Мы уже рассматривали использования `X(X&)`. Тип *String*

из Главы 4 использует конструктор *String(String&)* для проверки, что когда бы ни копировалась строка (при инициализации и присваивании, соответственно), поддерживаются корректные счетчики обращений в соответствующих объектах *String_ref*. Определение конструктора *X(X&)* для класса констатирует, что операция копирования объектов этого классового типа должна внимательно контролироваться.

Как было сказано в Главе 3, инициализации возникают не только как часть объявления или в списке инициализации элементов конструктора, но и в двух других контекстах: инициализация формальных аргументов функции фактическими и инициализация возвращаемых функцией значения (в возвращаемых выражениях).

Рассмотрим инициализацию формального аргумента фактическим. В некоторых случаях для преобразования фактического аргумента в значение допустимого для формального аргумента типа вызывается конструктор или оператор преобразования. В других случаях вызываемому необходимо создать временный объект, в котором создается значение перед копированием его в формальный аргумент.

```
void f(complex);  
f(12);
```

При преобразовании целого 12 в значение *complex* в вызываемом контексте создается временный объект *complex*, который инициализируется конструктором класса *complex*. Значение временного объекта копируется в формальный аргумент. Поэтому формальный аргумент *f* инициализируется в контексте, вызываемом *f*, а не в самом *f*. Вот другой пример, использующий типы вершин абстрактного синтаксического дерева из Главы 5.

```
Int  
incr(Int I) {  
    return I.eval() + 1; // опасность!  
}
```

Этот фрагмент проблематичен по нескольким причинам.

Во первых, так как тип возвращаемого выражения (*int*) не соответствует возвращаемому типу функции *Int*, компилятор создает временный объект *Int*, инициализирует его конструктором *Int*, и возвращает значение временного объекта. Класс *Int* имеет также деструктор (унаследованный от класса *Node*), поэтому временный объект *Int* уничтожается до возвращения *incr*. Аналогично, формальный аргумент *i* уничтожается до возвращения. Если вызов *incr* генерирует временный объект для инициализации формального аргумента, то и этот временный объект будет уничтожен в вызывающем контексте.

```
extern Int val;
Int x = incr(val); // создает временный
```

Поэтому при работе с формальным аргументом происходят два вызова деструктора и только одно обращение к конструктору: вызов конструктора и деструктора для временного объекта в вызывающем контексте и (только) деструктора для формального аргумента в *incr*.

Фактически, это не создает проблемы для пользователей иерархии абстрактного синтаксического дерева, так как программы, на ней базирующиеся, не используют типы вершин для аргументов и возвращаемых значений, а используют только указатели и ссылки на типы вершин.

```
Int *
Incr(Int &i) {
    return new Int(i.eval() + 1);
```

В этой версии *incr* в теле функции не активизируют деструктор. Это может быть серьезной проблемой для пользователей класса *String*, так как несбалансированная генерация обращений к конструктору и деструктору повредит счетчику обращений в объектах *String_rep*.

По этой причине классовые типы, определяющие конструктор $X(X\&)$, ведут себя по-другому, нежели прочие типы при использовании в качестве аргументов и возвращаемых значений. Если класс *X* определяет $X(X\&)$, то формальные

аргументы типа *X* используются в целях инициализации и деструкции так, как если бы они были типа *X&*, и как *X* во всех других контекстах. Поэтому для аргументов *X*, как и для *X&*, память для формального аргумента фактически выделяется в вызывающем контексте, а не в вызываемой функции.

Аналогично, функция, объявленная как возвращающая значение *X*, фактически инициализирует память для объекта *X* в вызывающем контексте вместо возврата объекта *X*.

При переносе памяти для формальных аргументов и возвращаемых значений в вызывающий контекст деструкторы в вызываемой функции для этих объектов не активизируются и поддерживается однозначное соответствие между созданием и уничтожением.

```
String  
incr(String s) {  
    return s + "1";  
}
```

Для формального параметра *s* деструктор не активизируется и оператор возврата инициализирует память в вызывавшем объекте. Этот объект инициализирует и уничтожает формальный аргумент и уничтожает возвращаемое значение, инициализированное `return`.

7.7. Семантика неявной копии.

Мы уже видели, что для управления семантикой копирования для объектов класса *X* необходимо определять и элемент `operator=(X&)`. (Можно также использовать `operator=(X&,X&)`, не являющийся элементом, но этот метод неудовлетворителен по причинам, изложенным в примере со *String* в Главе 4). Объекты класса *X* могут служить, однако, элементами и базовыми классами других классовых типов и та же семантика копирования, что и для объектов *X*, не внедренных в другие объекты, должна применяться для элементов и базовых классов типа *X*.

```

class Text {
    Text *next;
    String phrase;
public:
    Text(Text &);
    Text &operator =(Text &);
    // ...
};

```

Класс *Text* определяет семантику копирования с помощью конструктора $X(X\&)$ и элемента-оператора присваивания. Они реализованы так, что сохраняют семантику копирования элемента *String*.

```

Text::Text(Text &t)
: phrase(t.phrase) {
    // ...
}

Text &
Text::operator =(Text &t) {
    phrase = t.phrase;
    // ...
}

```

В обоих случаях мы явно реализуем корректную инициализацию или присваивание элемента *String*.

Даже если классовый тип не определяет явно свою семантику копирования, непредопределенная семантика требуется, если семантика копирования определена для элемента или базового класса.

```

class Empl {
    long ssn;
    String name;
    // ...
};

```

Чтобы во всех случаях соблюдалась корректная семантика копирования, C++ при необходимости определяет ее неявно. Если какой-то элемент или базовый класс данного

класса определяет $X(X\&)$ или элемент $\text{operator}=(X\&)$ и для этого класса семантика копирования не определена явно, то она определяется неявно компилятором. Для класса X определяется (неявно)

```
X::X(const X &);  
const X &X::operator =(const X &);
```

Семантика этого фрагмента состоит в вызове соответствующего $X(X\&)$ или элемента $\text{operator}=(X\&)$ для каждого элемента или базового класса, который этого требует, и простом копировании оставшихся элементов класса.

Поэтому определение класса *Empl* включает неявные определения

```
Empl::Empl(const Empl &);  
и  
const Empl &Empl::operator =(const Empl &);
```

Это гарантирует, что для элемента *String* из *Empl* реализуется надлежащая семантика копирования и фрагмент, вызывающий копирование объектов *Empl*, не влияет на корректность элемента *name* из *String*.

```
extern Empl a;  
Empl b = a; // Empl::Empl  
a = b; // Empl::operator =
```

7.8. Упражнения.

- Упр. 7.1. Реализуйте `operator new` и `operator delete`, заменяющие стандартные версии и возвращающие обнуленную память, которые имеют интерфейс, аналогичный стандартному, включая определение и использование `_new_handler`.
- Упр. 7.2.+ Реализуйте стандартные библиотечные версии `_vec_new` и `_vec_delete`.
- Упр. 7.3. Объясните, почему в добавок к другим своим

недостаткам, первая реализация *sorted_collection* в Главе 4 как массива фиксированного размера не подходит в качестве базиса для родового типа. (Подсказка: рассмотрите создание образца *sorted_collection*, содержащего тип *complex*).

Упр. 7.4. Обобщите реализацию операторов *new* и *delete* для класса *complex* так, чтобы они обрабатывали запросы памяти произвольного размера.

Упр. 7.5. Чем отличается семантика двух следующих функций при использовании специализированных версий операторов *new* и *delete* для класса *Node*?

```
void
limited1() {
    Node *np =
        new Plus (
            new Int(1),
            new Int(2)
        );
    int result = np->eval();
    delete np;
    return result;
}
```

```
void
limited2() {
    Node *np =
        new Plus (
            new Int(1),
            new Int(2)
        );
    int result = np->eval();
    Node *p
    delete p;
    return result;
}
```

Является ли это поведение ошибочным, или это особенность нашей реализации операторов *new* и *delete*?

- Упр. 7.6. В нашей реализации *complex::operator new* есть неявная зависимость от скрытого представления типа *complex*. В чем она состоит? Модифицируйте реализацию *complex::operator new* для автоматической самонастройки на изменения в скрытом представлении класса *complex*.
- Упр. 7.7.+ Разработайте общий тип "умного" указателя, гарантирующий указывание только на свободную память (heap).

Глава 8



БИБЛИОТЕКИ

Традиционно библиотека программ - это набор функций, или функций и структур данных специфического назначения. Например, существуют библиотеки математических функций, доступа к услугам операционной системы, описания структур данных стандартных файловых форматов и стандартные программы ввода и вывода.

Библиотеки такого типа обеспечивают три важных выгоды. Во-первых, они предлагают код общего назначения из единого источника. Повторное использование кода сокращает время разработки, кодирования и тестирования программ и увеличивает вероятность того, что программа правильна. Во вторых, стандартная библиотека - это также и стандартный интерфейс. Если зависящие от машины и оборудования черты программы инкапсулированы внутри библиотеки со стандартным интерфейсом, задача стыковки такой программы с различным оборудованием значительно упрощается. В-третьих, библиотека, развивающаяся для поддержки конкретной проблемной области, уменьшает сложность разработки и кодирования задач в реализации и представляет концепции более высокого уровня в библиотечном интерфейсе. Например, пользователи стандартной библиотеки ввода/вывода не должны учитывать детали буферизации и использования аппаратных устройств, но могут разрабатывать и писать с использованием концепций

более высокого уровня типа "установка", "чтение" и "запись".

В этой главе мы обсуждаем, каким образом парадигмы программирования и поддерживаемые C++ языковые особенности, описанные в предыдущих главах, позволяют нам расширить традиционную концепцию библиотеки для облегчения повторного использования кода, расширения интерфейса и концептуальной поддержки проектирования и кодирования.

8.1. Доступ к существующим библиотекам.

Как уже обсуждалось в Главе 0, методы программирования и парадигмы разработки развиваются прежде чем язык, делающих программирование с их использованием естественным. Нас поэтому не должно удивлять, что существующие библиотеки, написанные на более старых языках программирования, используют современные методы проектирования. Так как эти методы и парадигмы разработки нельзя выразить явно в синтаксисе языка, на котором они запрограммированы, интерфейс библиотеки может оказаться усложненным или невыразительным. В этих случаях мы можем создать на C++ "конверт" или альтернативный интерфейс библиотеки, упрощающий существующий и выявляющий концепции его разработки.

Например, рассмотрим функции ввода/вывода, предлагаемые библиотекой `stdio.h`.

```
FILE *fopen(const char * const char *);  
int fclose(FILE *);  
int fflush(FILE *);  
int fprintf(FILE *,const char * ...);  
int fscanf(FILE *,const char * ...);
```

Эти функции можно рассматривать как набор операций над файловыми объектами. Обратите внимание, что `fopen` играет роль конструктора, `fclose` - деструктора, а другие - выглядят в точности как функциональные элементы, в том

числе фактически имеют неявный первый аргумент `this`.

Мы можем определить альтернативный интерфейс для выявления нашей абстракции.

```
class File {
    FILE * const f;
public:
    File(const char *path, const char *mode = "w")
        : f(fopen(path, mode)) {}
    ~File() { fclose(f); }
    int flush() { return fflush(f); }
    int printf(const char * ...);
    int scanf(const char * . .);
};
```

Реализации `File::printf` и `File::scanf` изменены тем, что последний аргумент опущен. Мы хотим, чтобы формальные аргументы в определениях функциональных элементов `printf` и `scanf` использовались как фактические аргументы вызова функций `fprintf` и `fscanf` из `stdio`. Как говорят нам опущенные параметры, мы не можем определить число аргументов до выполнения программы. Есть несколько путей работы в такой ситуации, большинство из которых зависит от среды программирования. Здесь, в нашей минимальной реализации `File::printf` мы используем метод работы с переменным числом параметров, предлагаемый определениями в стандартном заголовочном файле `stdarg.h`. Объект `ap`, имеющий тип `va_list`, в нашем примере содержит информацию о способе доступа к аргументам. Обратите внимание, что `va_start` инициализирует `ap`, используя последний аргумент перед эллипсисом (в нашем примере `fmt`). Кроме того, `va_arg` возвращает последовательно аргументы из списка аргументов, интерпретированного в соответствии с типом, представляемым вторым аргументом `va_arg`. Наконец, `va_end` реализует нормальное прекращение использования переменных аргументов.

```
int
File::printf(const char *fmt ...) {
    va_list ap;
```

```
va_start(ap,fmt);

register int c;
while (c= *fmt++) {
    if (c=='%') {
        switch( c=*fmt++) {
            case 'c': // символ
                fprintf(f,"%c",va_arg(ap,int));
                break;
            case 'd': // десятичное целое
                fprintf(f,"%d",va_arg(ap,int));
                break;
            case 'g': // с плавающей точкой
                fprintf(f,"%g",va_arg(ap,double));
                break;
            case 's': // строка символов
                fprintf(f,"%s",va_arg(ap,char *));
                break;
            // и т.д. ...
        }
    }
    else
        fprintf(f,"%c",c);
}

va_end(ap);
}
```

Теперь мы можем создавать и использовать объекты типа *File* для stdio. Обратите внимание, что мы изменили имя функционального элемента для вывода с `fprintf` на `printf`, и имя функционального элемента для ввода с `fscanf` на `scanf`, чтобы они совпадали с именами, использованными для ввода и вывода из предопределенных файлов `stdout` и `stdin`. Обращение к этим программам может рассматриваться как ввод/вывод в предопределенные объекты типа *File*.

```
void
tee(char *fn){
    File f = fn;
    char c;
    while (scanf("%c",&c) != EOF) {
        f.printf("%c",c);
```

```
    printf("%c",c);  
}  
}
```

8.2. Проблемно-ориентированные языки.

Существуют специализированные языки программирования, предназначенные для моделирования, обработки строк, численных расчетов и т.п. с синтаксисом и системой типов, приспособленными к специфической проблемной области. Эти языки существуют, поскольку привычный язык упрощает проектирование, кодирование и сопровождение специализированных прикладных программ по сравнению с языком более общего назначения. Язык SNOBOL позволяет своим пользователям писать компактные программы обработки строк, а APL - компактные программы, работающие с матрицами, так как эти типы явно поддерживаются языками. Пользователи этих языков могут свободно разрабатывать и программировать в концепциях, с которыми работают их программы, а не с использованием представлений низкого уровня.

Большинство функциональных возможностей проблемно ориентированных языков может быть реализовано в C++ при помощи библиотек классов. Библиотека комплексной арифметики определяет тип данных комплексное число, перегружает существующие арифметические операторы для реализации семантики комплексной арифметики и определяет преобразования в и из существующих арифметических типов для интеграции комплексного типа в предопределенную систему типов. По сути дела, библиотека комплексной арифметики приспособливает язык C++ к конкретной проблемной области. Аналогично, библиотека String расширяет C++ для работы со строками символов так, как если бы *String* был встроенным типом.

Слишком сложно и дорого создавать новый язык программирования, приспособленный к проблемной области, но библиотеки классов реализовать сравнительно просто. Поэтому имеет смысл разрабатывать проблемно-ориенти-

рованную языковую библиотеку для области, в которой пишется сравнительно мало программ, или даже для одной программы. Например, если мы работаем с программой растровой графики, предпочтительнее рассматривать позицию на экране как точку, а не как пару прямоугольных координат.

```
class Point {
    short x,y;
public:
    Point(int u,int v) : x(u),y(v) {}
    Point operator -()
        { return Point(-x,-y); }
    Point operator +(Point p)
        { return Point(x+p.x,y+p.y); }
    Point operator -(Point p)
        { return Point(x-p.x,y-p.y); }
    Point operator *(int i)
        { return Point(x*i,y*i); }
    Point operator /(int i)
        { return Point(x/i,y/i); }
    Point operator %(int i)
        { return Point(x%i,y%i); }
    Point operator &(int i)
        { return Point(x&i,y&i); }
    Point operator +=(Point p)
        { return Point(x+=p.x,y+=p.y); }
    Point operator -(Point p)
        { return Point(x-=p.x,y-=p.y); }
    int operator =(Point p)
        { return x=p.x && y=p.y; }
    int operator !=(Point p)
        { return x!=p.x && y!=p.y; }
    int operator >(Point p)
        { return x>p.x && y>p.y; }
    int operator <-(Point p)
        { return x<-p.x && y<-p.y; }
    int operator <(Point p)
        { return x<p.x && y<p.y; }
    int operator >(Point p)
        { return x>p.x && y>p.y; }
};
```

Теперь мы можем программировать так, как если бы в C++ был предопределенный тип *Point*. Например, прямоугольник можно определять в терминах точек и операции над прямоугольниками как операции над точками.

```
class Rectangle {
    Point origin, corner;
public:
    Rectangle(Point p, Point q)
        : origin(p), corner(q) {}
    Rectangle operator +( Point p)
        { return Rectangle(origin+p, corner+p); }
    Rectangle operator -( Point p)
        { return Rectangle(origin-p, corner-p); }
    Rectangle translate(Point p)
        { return Rectangle(p, corner+(p-origin)); }
    inline int operator <(Rectangle);
    inline int operator <=(Rectangle);
    Point center()
        { return (origin + corner)/2; }
    friend int operator <(Point, Rectangle);
    friend int operator <=(Point, Rectangle);
};
```

```
inline int
operator <(Point p, Rectangle r) {
    return p>r.origin
        && p<(r.corner - Point(1,1));
}
```

```
inline int
operator <=(Point p, Rectangle r) {
    return p>=r.origin
        && p<=(r.corner - Point(1,1));
}
```

```
inline int
operator >(Point p, Rectangle r) {
    return !(p<=r);
}
```

```
inline int
Rectangle::operator <(Rectangle r) {
    return origin<r && corner<r;
```

```
}  
  
inline int  
Rectangle::operator <=(Rectangle r){  
    return origin<=r && corner<=r;  
}
```

Прямоугольники можно затем использовать для определения окон и так далее. В каждый момент мы создаем классы, позволяющие нам разрабатывать и программировать в концепциях, с которыми мы имеем дело, а не с представлениями более низкого уровня.

8.3. Расширяемые библиотеки.

Все библиотеки расширяемы в том смысле, что их можно использовать для создания других библиотек. Мы называем конкретные библиотеки расширяемыми, если они явно спроектированы для расширения пользователями путем предоставления каркаса для расширения.

Рассмотрим стандартную библиотеку C++ для потокового ввода/вывода. Библиотека `streamio` предоставляет тип ориентированный ввод/вывод, который можно расширить для использования определенных пользователем типов. Напротив, возможности ввода/вывода, предлагаемые библиотекой `stdio` не безразличны к типу.

```
#include <stdio.h>  
  
extern int age;  
extern String name;  
printf("Возраст %s - %d\n",age,name); // авария!
```

Этот фрагмент - "правильный" C++, но при выполнении выработает "мусор", или, возможно, сделает из программы "бомбу". Аргументы `age` и `name` неправильно упорядочены для форматной строки, но ошибка не обнаружится, так как для компилятора нет доступной информации о типах, говорящей, что форматная строка требует сначала `char*`, а потом `int`. Но, даже если мы подадим аргументы в нужном

порядке, оператор печати не будет работать. Функция `printf` ожидает, что `%s` в форматной строке соответствует аргумент `char*`, а мы предоставляем аргумент типа `String`. При отсутствии информации о типе компилятор C++ не знает, что нужно применять неявное преобразование из `String` в `char*` (с использованием `String::operator char*`). Мы можем предоставить необходимую информацию о типе

```
inline int
age_print(char *f, char *n, int a) {
    return printf(f, n, a);
}
// ...
age_print("Возраст %s - %d\n", name, age);
```

но этот подход ведет к специальной обработке каждой печати.

Лучший подход предлагается библиотекой `streamio`. Помните, что `streamio` перегружает операторы `<<` и `>>` для предоставления безразличного к типу вывода и ввода соответственно. Оператор печати, представленный выше, может быть записан как

```
cout << "Возраст" << name << " - " << age << "\n";
```

Следующий фрагмент - упрощенная версия некоторых фрагментов библиотеки `streamio`. Сначала мы определяем класс выходной поток, включая операторы сдвига влево или "вставщики" для вставки значения в выходной поток.

```
typedef FILE *FP;
class ostream {
    FP f;
public:
    ostream(FP fl) : F(fl) {}
    operator FP() { return f; }
    ostream &operator <<(char *);
    ostream &operator <<(int);
    ostream &operator <<(long);
    ostream &operator <<(double);
};

ostream cout = stdout;
```

Для нашей упрощенной версии мы используем библиотеку `stdio` для реализации семантики нашего оператора сдвига.

```
ostream &
ostream::operator <<(char *s) {
    fprintf(f, "%s", s);
    return *this;
}
```

Определения других образцов `ostream::operator <<` аналогичны. Обратите внимание, что каждый оператор сдвига возвращает (ссылку на) объект `ostream`, к которому он был применен. Это позволяет нам вставлять различные операции вывода в единственное (не через запятую) выражение. Как большинство операторов C++, оператор сдвига левоассоциативен, поэтому выражение

```
cout << "Возраст" << name << " - " << age << "\n";
```

рассматривается как

```
((cout << "Возраст") << name) << " - " << age << "\n";
```

Так как объект `cout` типа `ostream` возвращается каждым `operator <<`, это эквивалентно

```
cout << "Возраст";
cout << name;
cout << " - ";
cout << age;
cout << "\n";
```

но гораздо проще читается.

Входной поток можно создать аналогичным образом, определяя операторы сдвига вправо или "извлекатели" для выбора значений из входного потока.

```

class istream {
    FP f;
public:
    istream(FP fl) : f(fl) {}
    operator FP() { return f; }
    istream &operator >>(char *);
    istream &operator >>(int &);
    istream &operator >>(long &);
    istream &operator >>(double &);
    int gchar(); // берет один char
};

istream &
istream::operator >>(long &l) {
    fscanf(f, "%ld", &l);
    return *this;
}

istream cin = stdin;

cout << " Ваш возраст и имя: ";
int age;
char nm[32];
cin >> age >> nm;

```

Классы `ostream` и `istream` формируют базис расширяемой библиотеки ввода/вывода, определяя операции ввода/вывода для предопределенных типов C++ и определяя (примером) синтаксис операций для ввода и вывода. Этот скелет может быть расширен для типов, определенных пользователем, заданием дополнительных перегруженных операторов сдвига. Например, мы можем захотеть производить потоковый ввод/вывод комплексных чисел.

```

class complex {
    double re,im;
public:
    friend ostream &operator <<(ostream &,complex &);
    friend istream &operator >>(istream &,complex &);
    // ...
};

```

Объявления их дружественными нужны в этом случае

для того, чтобы дать операторам сдвига доступ к приватному представлению комплексного числа. Операторы сдвига для ввода/вывода `complex` определяются в терминах операторов сдвига для предопределенных типов точно так же, как `complex` определяется через предопределенные типы.

```
ostream &
operator <<(ostream &out, complex &c) {
    out << "(" << c.re << ", " << c.im << ")";
    return out;
}

istream &
operator >>(istream &in, complex &c) {
    in >> ws; // См. Упр. 8-2.
    if (in.gchar() != '(') error();
    in >> c.re;
    in >> ws;
    if (in.gchar() != ',') error();
    in >> c.im;
    in >> ws;
    if (in.gchar() != ')') error();
    return in;
}
```

Обратите внимание, что оператор `<<` выводит комплексное число в формате, подходящем для оператора ввода `>>`.

А как насчет вывода более сложного типа, определенного пользователем, вроде абстрактного синтаксического дерева? Мы хотим иметь возможность писать фрагмент типа

```
extern Node *np;
cout << "Выражение : " << np;
```

для печати инфиксного представления в круглых скобках дерева, на корень которого ссылается `np`.

Мы решим эту задачу, добавив сначала виртуальную возможность печати в иерархию вершин абстрактного синтаксического дерева.

```

class Node {
    // ...
    virtual void print(ostream &) { error(); }
};

class Binop : public Node {
    // ...
    virtual void print_op(ostream &) { error(); }
    void print(ostream &out) {
        out << "(";
        left->print(out);
        print_op(out);
        right->print(out);
        out << ")";
    }
};

class Plus : public Binop {
    // ...
    void print_op(ostream &out) { out << "+"; }
};

class Int : public Node {
    // ...
    void print(ostream &out) { out << value; }
};
// ...

```

Операция, выполняемая виртуальными программами `print`, аналогичны операциям `eval`, а виртуальные деструкторы уже присутствуют в иерархии вершин абстрактного синтаксического дерева. Программы рекурсивно печатают абстрактное синтаксическое дерево, вызывая соответствующую программу `print`, основываясь на типе вершин в корне каждого поддерева.

Все, что осталось, это присоединить эти программы `print` к существующей библиотеке `streamio`.

```

ostream &
operator <<(ostream &out, Node *np) {
    np->print(out);
    return out;
}

```

В предыдущем примере мы запрашивали ввод у пользователя, а затем ожидали ответа.

```
cout << "Ваш возраст и имя: ";
```

К сожалению, это может работать не так, как мы хотели. Если выходной поток `cout` буферизируется, послание пользователю может бесконечно ждать заполнения выводного буфера перед тем, как его написать. Для получения корректного поведения мы должны заполнять буфер `cout` для уверенности, что послание будет действительно выдано.

```
cout << "Ваш возраст и имя: ";\nflush(cout);
```

Управляющие сигналы такого рода, перемежаемые операциями ввода/вывода ухудшают читабельность кода и затрудняют понимание логики программы читателем и программистом. Лучше написать так:

```
cout << "Ваш возраст и имя: " << flush;
```

Мы сделаем это при помощи манипуляторов. Манипулятор - это значение в выражении, вызывающее побочный эффект, не связанный явно с основным назначением выражения. Обычно эти значения - адреса функций.

```
ostream &flush(ostream &);
```

Чтобы можно было использовать такие адреса функций в выражениях вывода, мы определим аппликатор, применяющий свой аргумент-функцию (указатель) к выходному потоку и возвращает выходной поток.

```
typedef ostream &(*MANIP)(ostream&);\ninline ostream &\noperator <<(ostream &out, MANIP f) {\n    return f(out);\n}
```

Этот аппликатор позволяет любой манипулятор типа *MANIP* использовать в выходном выражении. Для манипуляторов других типов можно определить другие аппликаторы.

```
typedef int (*MANIP2)(FILE *);

inline ostream &
operator <<(ostream &out,MANIP2 f){
    f(out);
    return out;
}

extern int close (FILE *);

cout << " Так долго ..." << flush << close;
```

Часто полезно предоставить параметризованные манипуляторы. Например, мы можем захотеть установить заданные значения различным характеристикам передачи данных выходного потока.

```
enum {odd,even,none};
enum {slow=1200,lagom=4800,fast=9600 };

cout << speed(fast)<<parity(none)<<"login: "<<flush;
```

В этом случае значениями манипулятора являются классовые объекты, содержащие пары величин: адрес функции и целое значение аргумента.

```
struct act_rec {
    void (*fp)(FILE *,int);
    int arg;
    act_rec(void (*f)(FILE *,int),int a)
        : fp(f),arg(a) {}
};
```

Обратите внимание, что *speed* и *parity* - функции, возвращающие *act_rec*.

```
inline act_rec  
speed (int baudr) {  
    extern void set_speed(FILE *,int);  
    return act_rec(set_speed, baudr);  
}
```

Наконец, мы определяем аппликатор, применяющий *act_rec* к *ostream*.

```
inline ostream &  
operator <<(ostream &out, act_rec m){  
    m.fp(out, m.arg);  
    return out;  
}
```

8.4. Настраиваемые библиотеки.

Часто библиотеки предлагают услуги, близкие к требуемым в программе, не будучи в точности подходящими. В этих случаях может помочь тот факт, что библиотека спроектирована как настраиваемая.

Рассмотрим иерархию абстрактного синтаксического дерева из Главы 5. Эта библиотека предлагает общую абстракцию бинарного оператора, поэтому просто расширить библиотеку с помощью наследования для работы с дополнительными бинарными операциями.

```
class And : public Binop {  
public:  
    And(Node *l,Node *r) : Binop(l,r) {}  
    int eval() { return left->eval() & right->eval();}  
};
```

Все, что нам нужно сделать для добавления нового бинарного оператора - это расширить общую абстракцию, добавив спецификацию конкретного добавляемого оператора.

Библиотека не содержит абстракцию унарного оператора, а только специальный унарный минус. Добавление нового унарного оператора требует от пользователя библиотеки большой дополнительной работы, включая оп-

ределение общих свойств унарных операторов (как, например, семантику деструктора) вместе со свойствами специфического добавляемого унарного оператора.

```
class Uplus : public Node {  
    Node *operand;  
public:  
    Uplus(Node *o) { operand = o; }  
    ~Uplus() { delete operand; }  
    int eval() { return operand->eval(); }  
};
```

Большая часть этого определения класса была бы не нужна, если бы в библиотеке имелась общая абстракция унарного оператора. Отсутствие такой абстракции не только заставляет пользователей библиотеки писать больше кода, чем им понадобилось бы в противном случае, но и требует от них большего знания деталей реализации библиотеки. Для библиотек со сложной реализацией маловероятно, что наивный пользователь поймет все детали правильно. В общем случае необходимо предусматривать, как данная библиотека может использоваться или приспособливаться. При этом все уровни абстракции должны быть явными.

Другое использование наследования для приспособления состоит в создании альтернативных версий существующих типов. Например, мы можем захотеть иметь версию *ostream*, которую можно инициализировать составным именем файла и которая запрещает копирование, чтобы избежать случайного переименования файлов.

```
class ostr : public ostream {  
public:  
    ostr(char *path) : ostream(fopen(path, "w")) {}  
    ~ostr() { fclose(FP(*this)); }  
private:  
    ostr(ostr &);  
    void operator =(ostr &);  
};
```

Класс *ostr* расширяет операторы `<<` и `FP` своего базового

класса и, так как он фактически ostream (то есть ostream - общедоступный базовый класс *ostr*), он может использоваться там же, где и ostream. Мы сделали $X(X\&)$ и $operator =$ приватными операциями для предотвращения копирования объектов типа *ostr*.

Мы можем также использовать наследование для слияния и настройки нескольких классов.

```
class ostream : public istream, public ostream {  
    public:  
    ostream(FILE *fp) : istream(fp), ostream(fp) {}  
};
```

iostream может использоваться для ввода и вывода, так как он наследует свойства *istream* и *ostream*. Для приспособления объектов типа *iostream* для произвольного ввода/вывода мы можем создать манипулятор *seek* аналогично ранее описанным манипуляторам *speed* и *parity*.

```
extern ostream cfile;  
extern char first, fifth, replacement;  
cfile >> first >> seek(5) >> fifth;  
cfile << replacement;
```

8.5. Упражнения.

Упр. 8.1.+ Напишите манипулятор и аппликатор для класса *complex* для округления сложений:

```
extern complex a,b,c,d,e;  
d = a + b + round + c + round;
```

Напишите аналогичный манипулятор, выполняющий ту же операцию для (предопределенных) целых выражений.

Упр. 8.2. Напишите манипулятор и аппликатор *istream*, уничтожающий (пропускающий) пробел: `cin >> ws`.

Упр. 8.3. Мы рассмотрели манипуляторы, являющиеся адресами функций и парами значений. Приду-

майте примеры других типов значений, которые могут использоваться как манипуляторы, и напишите программы, применяющие их в выражениях типа *istream*, *ostream*, *String* и *complex*.

- Упр. 8.4. Напишите "вставщик" для типа *sorted_collection* из Главы 4. Не изменяйте реализацию и интерфейс *sorted_collection*.
- Упр. 8.5. Напишите извлекатель для абстрактных синтаксических деревьев, соответствующий вставщику, описанному в этой главе. Не предполагайте, что при вводе имеются все скобки. (Подсказка: в чем разница между синтаксическим анализатором для калькулятора из Главы 5 и этим извлекателем?).
- Упр. 8.6. Создайте библиотеку, делающую C++ прикладным языком для векторной и матричной арифметики.
- Упр. 8.7. Разработайте библиотеку алгебры реляционной базы данных. Реализуйте два интерфейса операторов базы данных: один с использованием инфиксных операторов, а другой с использованием синтаксиса вызовов функций. (Подсказка: как вы представите характеристики для поисков абстрактными синтаксическими деревьями? В чем разница между программным обеспечением базы данных и интерпретатором языка программирования?). Разработайте управляющие структуры (типа операторов) для расширения вашей реляционной алгебры до реляционного исчисления.
- Упр. 8.8. Используя определения *istream* и *ostream* в этой главе, объясните семантику двух следующих выражений:

```
cout << 12;  
cin >> 12;
```

Используя определение `iostream` в этой главе, объясните, почему выражение `cfile >> first << replacement` не будет откомпилировано. Переделайте реализацию `iostream` так, чтобы смешанные выражения типа этого были допустимы.

- Упр. 8.9.** Является ли выбор `>>` и `<<` для операторов вставки и выделения `streamio` хорошим? Почему да или нет? Какой другой оператор(ы) подойдет? Почему `->` - плохой выбор? Почему `=` - ужасный выбор?
- Упр. 8.10.** Покажите, как использование наследования для приспособления настройки классовых типов может быть применено для исправления ошибок библиотеки без изменения исходного текста библиотеки. Используйте этот метод для обмана зависящего от среды программирования порядка вычислений операндов бинарных вершин абстрактного синтаксического дерева в функции `eval`, как отмечено в решении Упр. 5.3.
- Упр. 8.11.+** Часто библиотеки необходимо инициализировать перед использованием. Разработайте схему, которая гарантирует, что библиотека инициализирована (однажды) в любой программе, в которую она включена.
- Упр. 8.12.** Некоторые настраиваемые библиотеки разработаны как "скелеты программ", то есть пустые программы, настраиваемые для получения семейства действительно программ. Вспомним Упр. 6.5. Разработайте скелет программы для приложений, моделируемых последовательностью переходов из состояния в

состояние. Реализуйте систему бронирования авиабилетов, лексический анализатор для C++ и абстрактный тип данных *автомат с конечным числом состояний* из Упр. 4.8., используя наследование для настройки Вашего скелета программ.

Приложение

РЕШЕННЫЕ УПРАЖНЕНИЯ.

Упр. 1.3.

```
a // int
b // int *
*b // int
c // int *
*c // int
d[2] // int *
*d // int *
**d // int
c[-2] // int
c-2 // int *
*(c-2) // int
&c // int **
```

Упр. 1.4.

```
c=c; // правильно
c=c; // неправильно
rcc=&c; // правильно
rcc=&cc; // правильно
rc=&c; // правильно
rc=&cc; // неправильно
rc=rcc; // неправильно
rc=rc; // правильно
rc=rc; // неправильно
rc=rc; // неправильно
*rc=*rc; // правильно
rc=*rc; // правильно
**rc=**rc; // правильно
*rc=**rc; // правильно
```

Упр. 2.4.

```
char *
get_token(char *&s, char *ws = "\\t\\n"){
    char *p;
    do
        for (p=ws; *p && *s != *p; p++);
    while (*p ? *s++ : 0);
    char *ret = s;
```

```

do
    for (p=ws;*p && *s != *p; p++);
    while (*p ? 0 : *s ? s++ : 0);
    return ret;
}

int strlen(char *s) {
    char *t = get_token(s, "");
    return s-t;
}

```

Упр. 3.1.

Для получения битового представления аргумента **double** используется союз.

```

void
bitprint(double val) {
    static int nbbitsinchar = 0;
    if (!nbbitsinchar) {
        // вычисляем число бит в char
        for (unsigned c=1; c; c<=1)
            nbbitsinchar++;
    }
    union {
        double d;
        unsigned char a[sizeof(double)];
    };
    d=val;
    for (int i=0; i<sizeof(double); i++) {
        unsigned c = a[i];
        for (int j=nbbitsinchar; j; j--)
            if (c & (1<j))
                printf("1");
            else
                printf("0");
    }
}

```

Упр. 3.2.

Класс хэш-таблицы *Htab*, можно реализовать с использованием списка записей, хэшируемых в одну точку, как показано ниже:

```
class Rec {
    // детали
public:
    char *key;
};

class Clist { // конфликтный список хэш-таблицы
    Rec *r;
    Clist *next;
    friend Htab;
};

class Htab { // тип хэш-таблица
    Clist **tab; // указатель на таблицу
    int hash(char *);
    int size; // размер таблицы
public:
    Rec *insert(Rec *);
    Rec *lookup(char *);
    Rec *remove(char *);
    Htab(int);
    ~Htab();
};

Htab::Htab(int sz)
    : size(sz), tab(new Clist *[size]) {}

Htab::~~Htab() {
    for (int i=0; i<size; i++) {
        Clist *p=tab[i], *q;
        while (p) {
            q = p;
            p = p->next;
            delete q;
        }
    }
    delete [size] tab;
}

Rec *
Htab::lookup(char *s) {
    // Возвращает указатель на запись, 0 при неудаче.
    for (Clist *p = tab[hash(s)]; p; p=p->next)
        if (strcmp(s, p->r->key) == 0)
            return p->r;
}
```

```
    return 0;
}

Rec *
Htab::Insert(Rec *r) {
    // Возвращает указатель на вставленную запись,
    // 0 при неудаче
    If (lookup(r->key))
        return 0;
    int i=hash(r->key);
    Clist *p = new Clist;
    p->r = r;
    p->next = tab[i];
    tab[i] = p;
    return r;
}

Rec *
Htab::remove( char *s) {
    // Возвращает указатель на удаленную запись,
    // 0 при неудаче
    int i = hash(s);
    Clist *p = tab[i], *q=0;
    while (p && strcmp(p->r->key,s) ) {
        q = p;
        p = p->next;
    }
    If (p) {
        If (q)
            q->next = p->next;
        else
            tab[i] = p->next;
        Rec *r = p->r;
        delete p;
        return r;
    }
    else
        return 0;
}
```

Htab::hash может быть реализована аналогично хэш-функции в Разделе 3 Главы 1.

Упр. 3.4.

Мы определим трассировочный классовый тип, инициализируемый именем функции или блока, в котором он объявлен.

```
class trace {
    char * const msg;
public:
    trace(char *m) : msg(m) {
        fprintf(stderr, "входим в %s \n", msg);
    }
    ~trace() {
        fprintf(stderr, "покидаем %s \n", msg);
    }
};
```

Объявление объекта *trace* вызывает конструктор для печати сообщения "входим в ...", когда объект входит в область видимости. Когда объект покидает область видимости, деструктор печатает сообщение "покидаем ...". Объясните, как трассировочный объект работает в следующем фрагменте:

```
void
buggy_func(int arg) {
    trace a="buggy_func";
    if (arg) {
        trace b="then part";
        // ...
        return;
    }
    else {
        trace c="else part";
        // ...
    }
}
```

Упр. 4.14.

Ассоциативный массив устанавливает соответствие между значениями индексов и элементами массива. В соответствии с этим определением, стандартный массив Си++ ассоциативен - он сопоставляет целым элементы массива. Другой пример ассоциативного массива - таблица иденти-

фикаторов компилятора, так как она сопоставляет идентификаторы и их атрибуты. Наш ассоциативный массив должен устанавливать соответствие между *String*-ами и *String*-ами, поэтому элементы и индексы имеют тип *String*.

```
typedef String I;
typedef String E;
```

Для реализации соответствия между *String* и *String* мы создаем класс ассоциативный массив и перезагружаем оператор [], чтобы индексировать наш ассоциативный массив как обыкновенный.

```
class Pair {
    I Index;
    E el;
    Pair *next;
    friend Aary;
};
class Aary {
    Pair *elems;
public:
    Aary() { elems=0; }
    E &operator[] (I);
};
```

В этой реализации наш массив - это список индексов пар элементов. Обратите внимание, что оператор [] выполняет линейный поиск по списку при каждой индексации ассоциативного массива. Это подойдет для массивов с небольшим числом индексов, но для больших массивов требуется более эффективный метод поиска (хэш-таблица, сбалансированное дерево и т.п.)

```
E &
Aary::operator[] (I i) {
    for (Pair *p = elems; p; p = p->next)
        if (p->Index == i)
            return p->el;
    p = new Pair;
    p->Index = i;
    p->next = elems;
    elems = p;
    return p->el;
}
```

Обратите внимание, что оператор [] создает элемент, если он еще не существует. Теперь мы можем объявить и

использовать наш тип словаря.

```
Aary dict;
dict["cat"] = "meow";
dict["dog"] = "moo";
dict["cow"] = "woof";
dict["sheep"] = "baa";
dict["dog"] = dict["cow"];
dict["cow"] = "moo";
```

Наша реализация навязывает очень мало ограничений на типы индекса и элемента. Тип индекса должен иметь определенные операторы `==` и `=`, а тип элемента должен иметь оператор `=`. Кроме того, оба типа должны иметь возможность объявления без инициализатора или иметь определенный инициализатор.

Мы позаботимся об этих ограничениях и сделаем наш тип родовым, используя макросы, определенные в библиотеке `generic.h`.

```
#include <generic.h>
#define Pair(I,E) name3(I,E,Pair)
#define Aary(I,E) name3(I,E,Aary)
#define Aarydeclare2(I,E) \
class Pair(I,E) { \
    I index; \
    E el; \
    Pair(I,E) *next; \
    friend Aary(I,E); \
}; I
class Aary(I,E) { \
    Pair(I,E) *elems; \
public: I
    Aary(I,E) () { elems = 0; } \
    E &operator[](I); \
}; \
static E & \
Aary(I,E)::operator[](I i) { \
    for (Pair(I,E) *pp = elems; p; p=p->next) \
        if (p->index == i) \
            return p->el; \
    p = new Pair(I,E); \
    p->index = i; \
    p->next = elems; \
    elems = p; \
```

```
    return p->el; \
}
```

Теперь мы можем получать образцы нашего родового типа и объявлять объекты конкретизированного типа.

Можно начать с конкретизации нашего словаря.

```
declare2(Aary,String,String);
```

Можно использовать макрос препроцессора Aary, объявленный выше для генерации имени типа нашего словаря при каждом его использовании

```
Aary(String,String) dict;
```

или можем дать типу имя при помощи typedef:

```
typedef Aary(String,String) Dictionary;
Dictionary dict;
```

Мы поможем создать тип *ассоциативный массив*, устанавливающий соответствие между *String*-ами и *Dictionary*, конкретизируя другую версию нашего родового ассоциативного массива.

```
declare2(Aary,String,Dictionary);
typedef Aary(String,Dictionary) Library;
Library lib;
lib["barnyard"] = dict;
```

Другие конкретизации могут включать таблицу идентификаторов компилятора:

```
declare2(Aary,String,NameInfo);
typedef Aary(String,NameInfo) Symtab;
```

или очень большой разреженный массив с целыми индексами.

```
declare2(Aary,long,Type);
typedef Aary(long,Type) Large_ary;
Large_ary a;
Type x,y,z;
a[0]=x;
a[1000000000]=y;
a[2000000000]=z;
```

Упр. 4.16.

Мы используем старый трюк с исключаящим ИЛИ для кодирования битов для указателей на предыдущий и последующий одним указателем. Мы можем получить адрес

предыдущего элемента из кода, если у нас есть код последующего и наоборот.

```
encoded = previos ^ next;
next = encoded ^ previous;
previous = encoded ^ next;
```

Для прохождения списка в обоих направлениях все, что нам нужно - это незашифрованные адреса головы и хвоста списка.

```
typedef int ETYPE;
typedef int BITS;

class list_el {
    list_el *ptr;
    ETYPE el;
    friend list;
    friend iter;
    friend void paste(list_el * &, list_el * &, ETYPE);
};

class list {
    list_el *hd, *tl;
public:
    list() { hd=tl=0; }
    friend list &operator +(list &, ETYPE);
    friend list &operator -(ETYPE, list &);
    list_el *head() { return hd; }
    list_el *tail() { return tl; }
    friend iter;
};
```

Мы перезагружаем оператор = для добавления элемента в голову списка и оператор + для добавления элемента в конец списка.

```
list &
operator +(list &lst, ETYPE e) {
    paste(lst.tl, lst.hd, e);
    return lst;
}

list &
operator -(ETYPE e, list &lst) {
    paste(lst.hd, lst.tl, e);
    return lst;
}
```

```

void
paste(list_el *&front, list_el *&back, ETYPE e) {
    list_el *tmp = new list_el;
    tmp->el = e;
    if (!front)
        back = front = tmp;
    else if (back == front) {
        back->ptr = front = tmp;
        tmp->ptr = back;
    }
    else {
        tmp->ptr = front;
        front->ptr =
            (list_el *) (BITS(front->ptr) ^ BITS(tmp));
        front = tmp;
    }
}

```

Итераторный объект инициализируется концом списка, который надо пройти.

```

class Iter {
    list_el *prev, *curr;
public:
    Iter(list_el *start) { prev=0; curr=start; }
    ETYPE *operator()();
};

ETYPE *
Iter::operator()() {
    if (!curr)
        return 0;
    list_el *tmp = curr;
    curr = (list_el *) (BITS(curr->ptr) ^ BITS(prev));
    prev = tmp;
    return &tmp->el;
}

```

Теперь мы можем создавать списки и проходить их в любом направлении.

```

typedef int ETYPE;
#include "list.h"
main() {
    int *ip;
    list lnts;
    1 - 2 - 3 - lnts + 4 + 5 + 6;
}

```

```

    iter forw = ints.head();
    while (ip = forw())
        printf("%d", *ip);

    iter backw = ints.tail();
    while (ip = backw())
        printf("%d", *ip);
}

```

Хотя целые в списке находятся по возрастанию, они добавляются в список в следующем порядке: 4, 5, 6, 3, 2, 1. Почему? Что вы думаете о нашем использовании оператора = для добавления элементов в голову списка? Хорошая ли это идея?

Один из важных аспектов этого решения задачи состоит в том, что закодированное представление прямых и обратных связей скрыто от пользователей списка. Когда мы позже задумаемся и решим использовать отдельные указатели вперед и назад, мы сможем сделать изменение, не влияя на программу, использующую список.

Упр. 5.3.

Сначала мы определим тип вершины для идентификатора и таблицы идентификаторов для сопоставления идентификаторам значения. Так как в языке нашего калькулятора нет объявлений, мы включаем имя в таблицу идентификаторов, когда его идентификатор используется первый раз.

```

class Nament {
    char *name;
    int value;
    Nament *next;
    Nament(char *nm, nament *n) {
        name = strcpy(nnew char[strlen(nm) + 1], nm);
        value = 0;
        next = n;
    }
    friend Id;
};

```

```

class Id : public Node {
    static Nament *symtab;
    Nament *entry;
    Nament *look(char *);

```

```

public:
  Id(char *nm) { entry = look(nm); }
  int set(int i) { return entry->value = i; }
  int eval() { return entry->value; }
}

Nament *
Id::look(char *nm) {
  for (Nament *p = symtab; p; p=p->next)
    if (strcmp(p->name,nm) == 0)
      return p;
  return symtab = new Nament(nm,symtab);
}

```

Затем определим тип вершины *присваивание*.

```

class Assign : public Binop {
public:
  Assign(Id *t, Node *e) : Binop(t,e) {}
  int eval()
  { return ((Id *)left)->set(right->eval); }
};

```

Обратите внимание, что мы должны применить приведение типов к указателю *left*, чтобы вызвать *Id::set* в *Assign::eval*. Почему? Мы можем избежать приведения типов, объявив *set* как виртуальный функциональный элемент в *Node*. Почему это плохая идея?

Обычно нам не нужно изменять реализации других типов вершин абстрактного синтаксического дерева, чтобы приспособить оператор присваивания. Мы делаем неявное предположение в реализации других бинарных операторов, что порядок вычислений не имеет значения. Рассмотрим реализацию *Plus::eval*.

```

int
Plus::eval() {
  return left->eval() + right->eval();
}

```

Напомним, что Си++ не определяет полностью порядок вычисления выражений. Потому мы не можем сказать, правое или левое дерево в сложении будет вычисляться раньше. Это вообще-то не имеет значения, если вычисляемое выражение не имеет побочных эффектов, но операторы присваивания такой эффект имеют. Рассмотрим результат вычисления правого поддерева сложения раньше левого в

следующем выражении: $(id=12)+id$. Более короткой реализацией *Plus::eval* будет

```
Int
Plus::eval() {
    Int l = left->eval();
    return l + right->eval();
}
```

Реализация интерактивного калькулятора, использующая нашу иерархию абстрактного синтаксического дерева, приведена ниже.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "Nodes.h"
```

```
static Int token; // текущая лексема
static char lexeme[81]; // атрибут для ID и INT
enum ( ID = 257, INT, EOLN, BAD );
```

```
Node *E(), *T(), *F();
```

```
/*
```

Сканер: возвращает следующую лексему входного потока. атрибуты для целых констант и идентификаторов сбрасываются в глобальную переменную "lexeme".

```
*/
```

```
Int
scan() {
    Int c;
    while(1)
        switch (c=getchar()) {
            case '+': case '-':
            case '*': case '/':
            case '(': case ')':
            case '=':
                return c;
            case ' ': case '\t':
                continue;
            case '\n':
                return EOLN;
            default:
                if (isdigit(c)) {
                    char *s = lexeme;
                    do
```

```

        *s++ = c;
        while (isdigit(c = getchar()));
        *s = '\0';
        ungetc(c, stdin);
        return INT;
    }
    if (isalpha(c)) {
        char *s = lexeme;
        do
            *s++ = c;
        while (isalnum(c = getchar()));
        *s = '\0';
        ungetc(c, stdin);
        return ID;
    }
    return BAD;
}
}
}

```

```

void
error() {
    printf("ОШИБКА!!!\n");
    exit(255);
}

```

```

/*
    разбор простого выражения:

    E -> T {(+|-)T}
    T -> F {(*/|/)F}
    F -> ID | INT | (E) | ID = E | -F
*/

```

```

Node *
E() {
    Node *root = T();
    while (1)
        switch(token) {
            case '+':
                token = scan();
                root = new Plus(root, T());
                break;
            case '-':
                token = scan();
                root = new Minus(root, T());

```

```

        break;
    default:
        return root;
    }
}

Node *
F0 {
    Node *root;
    switch(token) {
    case ID:
        root = new Id(lexeme);
        token = scan();
        if (token == '-') {
            token = scan();
            root = new Assign((Id *)root, E0 );
        }
        return root;
    case INT:
        root = new Int(atol(lexeme));
        token = scan();
        return root;
    case '(':
        token = scan();
        root = E0;
        if (token != ')')
            error();
        token = scan();
        return root;
    case '-':
        token = scan();
        return new Uminus(F0);
    default:
        error();
    }
}

```

```

Node *
T0 {
    Node *root = F0;
    while(1)
        switch(token) {
        case '*':
            token = scan();
            root = new Times(root, F0);

```

```

        break;
    case '/':
        token = scan();
        root = new Div(root,F());
        break;
    default:
        return root;
    }
}

/*
Драйвер: инициализация,цикл до ошибки.
*/
main() {
    Node *root;
    while(1) {
        token = scan();
        if ((root==E()) && token == EOLN) {
            printf("%d\n",root->eval());
            delete root;
        }
        else
            error();
    }
}

```

Упр. 5.7.г).

Сначала модифицируем конструктор и деструктор для *Node* (корневого класса нашей иерархии абстрактного синтаксического дерева) для отслеживания числа размещенных вершин. Достаточно модифицировать только конструктор и деструктор для *Node*, так как при этом для каждого вызова класса, выведенного из *Node*, они будут вызываться.

```

class Node {
protected:
    Node() { num_nodes++; }
public:
    ~Node() { num_nodes--; }
    virtual int eval();
    static long num_nodes;
}

```

Теперь мы можем вывести новый тип монитора, отслеживающий число вершин.

```
class Nalloc : public Monitor {
    long max_nodes;
public:
    double get_value() {
        return Node::num_nodes > max_nodes
            ? max_nodes
            : Node::num_nodes;
    }
    Nalloc(long max = 1000) :
        Monitor("Вершины",0,max,0.5),
        max_nodes(max) {}
};
```

Упр. 6.4.

Моделирование с центральным управлением многими аэропортами:

```
#include "task.h"

class Airport;

class Plane : object {
    static int fltcount;
    long start;
    int fltno;
    Airport *origin;
    Airport *destination;
public:
    Plane(Airport *o,Airport *d) {
        fltno = ++fltcount;
        origin = o;
        destination = d;
    }
    long howlong() { return clock-start; }
    void set() { start = clock; }
    int flt() { return fltno; }
    Airport *to() { return destination; }
    Airport *from() { return origin; }
    void reschedule();
};

class PlaneQ {
    qhead *head;
```

```

    qtail *tail;
public:
    PlaneQ() { head = new qhead(ZMODE,50);
        tail = head->tail(); }
    void put(Plane *p) { p->set();
        tail->put((object*)p); }
    Plane *get() { return (Plane *)head->get(); }
    int isroom() { return tail->rdspace(); }
    int notempty() { return head->rdcount(); }
};

class AirControl : public task {
    PlaneQ *landing, *inair, *incoming, *outgoing;
    PlaneQ *incircling, *outcircling;
    int ok_take_off;
public:
    AirControl(PlaneQ*,PlaneQ*,PlaneQ*,PlaneQ*);
    ~AirControl();
    int tok() { return ok_take_off; }
};

class GroundControl : public task {
    PlaneQ *takingoff, *onground;
public:
    GroundControl(PlaneQ*,PlaneQ*);
};

class Airport : public task {
    PlaneQ *takingoff, *landing, *inair,
        *onground, *incoming, *outgoing;
    AirControl *acontrol;
    GroundControl *gcontrol;
    const int index;
    char * const nm;
public:
    Airport(PlaneQ *,PlaneQ *,int char*);
    ~Airport();
    char *name() { return nm; }
    friend CentralControl;
};

class CentralControl : public task {
    static nairports;
    static Airport **airports;
    static urand *acity;

```

```

public:
    CentralControl();
    ~CentralControl();
    friend Airport *pickacity() {
        return CentralControl::
            airports[CentralControl::acity->draw()]; }
};

Airport::Airport(PlaneQ *In,PlaneQ *out,
    Int l, char *Id) :
    Incoming(In),outgoing(out), Index(l),nm(Id) {

    takingoff = new PlaneQ;
    Inair = new PlaneQ;
    landing = new PlaneQ;
    onground = new PlaneQ;
    acontrol = new AirControl(landing,Inair,
        Incoming,outgoing);
    gcontrol = new GroundControl(takingoff,onground);
    Plane *tp=0,*lp=0; // ожидание самолетов
    Int maxwait = 30;
    for (;;) {
        delay(10);
        If (!lp) lp = landing->get();
        If (!tp && acontrol->tok()) tp = takingoff->get();
        If (lp) {
            If (onground->Isroom() ) {
                printf("рейс %d из %s приземляется в %s\n",
                    lp->flt(),lp->from()->name(),name());
                onground->put(lp);
                lp = 0;
            }
            else if (lp->howlong() > maxwait) {
                printf("рейс %d разбился в %s\n",
                    lp->flt(),name());
                delete lp;
                lp = 0;
            }
            else
                printf("рейс %d посадка в %s отложена\n",
                    lp->flt(),name());
        }
        If (tp) {
            If (Inair->Isroom() ) {
                printf("рейс %d из %s в %s взлетел\n",
                    tp->flt(),name(),tp->to()->name());
            }
        }
    }
}

```

```

        Inair->put(tp);
        tp = 0;
    } else
        printf("рейс %d взлет из %s откладывается\n",
            Ip->flt(), name());
    }
} // end for(;;)
}

```

```

AirControl::AirControl(PlaneQ *land, PlaneQ *Ina,
    PlaneQ *Income, PlaneQ *outgo) :
    landing(land), Inair(Ina),
    Incoming(Income), outgoing(outgo) {
    Incircling = new PlaneQ;
    outcircling = new PlaneQ;
    Plane *Ip=0, *op=0;
    ok_take_off = 1;
    for (;;) {
        delay(10);
        while(Inair->notempty() && outcircling->isroom())
            outcircling->put(Inair->get());
        while (Incoming->notempty() && Incircling->isroom())
            Incircling->put(Incoming->get());
        if (!Ip) Ip = Incircling->get();
        if (!op) op = outcircling->get();
        if (Ip) {
            if (landing->isroom()) {
                printf("рейс %d приземляется в %s\n",
                    Ip->flt(), Ip->to()->name());
                landing->put(Ip);
                Ip = 0;
            }
            else {
                printf("рейс %d ожидает посадки в %s\n",
                    Ip->flt(), Ip->to()->name());
            }
        }
        if (op) {
            if (outgoing->isroom()) {
                printf("рейс %d покидает воздушное пространство %s\n",
                    op->flt(), op->from()->name());
                outgoing->put(op);
                ok_take_off=1;
                op = 0;
            }
        }
    }
}

```

```

        else {
            printf("рейс %d ожидает посадку в %s\n",
                op->flt(), op->from()->name());
            ok_take_off = 0;
        }
    }
} // end for(;;)
}

```

```
CentralControl::CentralControl() {
```

```

    const int n = 9;
    nairports = n;
    acity = new urand(0, n-1);
    static char *nms[n] = {
        "New York", "Newark", "Chicago",
        "Denver", "Columbus", "Austin",
        "Dallas", "Los Angeles", "Portland"
    };
    airports = new Airport*[n];
    for (int i=0; i<n; i++) {
        airports[i] =
            new Airport(new PlaneQ, new PlaneQ, i, nms[i]);
    }

```

```

    // направить несколько самолетов в каждый аэропорт
    urand nplanes(20, 40);
    for (i=0; i<n; i++) {
        Airport *port = airports[i];
        for (int m = nplanes.draw(); m; m--)
            port->onground->put(new Plane(0, port));
    }

```

```

    // прямое воздушное движение между аэропортами
    // новые рейсы отправляются из пустых аэропортов
    for (;;) {
        delay(10);
        for (i=0; i<n; i++) {
            Airport *port = airports[i];
            Plane *p = port->outgoing->get();
            if (p)
                if (p->to()->incoming->isroom())
                    p->to()->incoming->put(p);
            else {
                printf("рейс %d разбился".

```

```

        p->flt());
        delete p;
    }
    if (!port->onground->notempty()) {
        port->onground->put(new Plane(0,port));
    }
}
} // end for(;;)
}

```

```

AirControl::~AirControl() {
    while (Incircling->notempty()
        && Incoming->notempty())
        thistask->delay(10);
    Plane *p;
    while (p=outcircling->get() )
        Incircling->put(p);
    while (Incircling->notempty())
        thistask->delay(10);
    cancel(0);
}

```

```

GroundControl::GroundControl(PlaneQ *takingoff,
    PlaneQ *onground) {
    urand n(10,30);
    Plane *p=0;
    for (;;) {
        delay (n.draw());
        if (!p) p=onground->get();
        if (p) {
            if (takingoff->isroom()) {
                p->reschedule(); // устанавливает новое направление
                printf("рейс %d покидает аэропорт %s\n",
                    p->flt(),p->from()->name() );
                takingoff->put(p);
                p=0;
            } else
                printf("рейс %d задерживается в аэропорту %s\n",
                    p->flt(),p->from->name());
        }
    } // end for(;;)
}

```

```

void
Plane::reschedule() {

```

```

    origin = destination;
    destination = pickacity();
    start = clock;
}

Airport::~Airport() {
    gcontrol->cancel(0);
    delete acontrol;
    while (landing->notempty())
        thistask->delay(10);
    printf("%s Аэропорт закрыт\n",name());
    cancel(0);
}

CentralControl::~CentralControl() {
    for (int i=0;i<nairports;i++) {
        delete airports[i];
    }
    printf("Центральное управление закрыто\n");
    cancel(0);
}

main() {
    CentralControl *ccp = new CentralControl;
    thistask->delay(5000);
    delete ccp;
    thistask->resultis(0);
}

```

Упр. 7.2. Вот одна реализация.

```

typedef void (*CTOR)(void *);
typedef void (*DTOR)(void *,int);

void *
_vec_new(void *vthis,int n,int sz,void *f) {
    if (vthis == 0)
        vthis = new char[n*sz];
    register char *p = (char *) vthis;
    if (f)
        for (register int i=0;i<n;i++)
            (*CTOR(f))(p+i*sz);
    return p;
}

```

```

void
_vec_delete(void *vthis,int n,int sz,void *f,int del)
if (vthis==0)
    return;
if (f) {
    register char* p=(char *) vthis;
    for (register int l=0;l < n;l++)
        (*DTOR(f))(p+l*sz,0);
}
if (del)
    delete vthis;
}

```

Упр. 7.7.

Мы используем макросы из generic.h. для обобщенного типа указателя, определяемого типом объекта, на который он ссылается:

```

#define Hptr(ETYPE) name@(ETYPE,Hptr)

#define Hptrdeclare(ETYPE) \
class Hptr(ETYPE) { \
    ETYPE *p; \
public: \
    Hptr(ETYPE)(ETYPE *v=0) { p=v; } \
    operator ETYPE *() { return p; } \
    ETYPE &operator *() { \
        check(p); \
        return *p; \
    } \
    ETYPE &operator[](long i) { \
        check(p+i); \
        return p[i]; \
    } \
    Hptr(ETYPE) &operator ++() { \
        p++; \
        return *this; \
    } \
    // и т.д. ...
};

```

Наш родовой указатель реализован как обычный указатель на тип, которым конкретизирован родовой указатель. Мы предоставляем все обычные арифметические операции указателями (хотя мы выше показали только ++), выполняя

операцию над обычным указателем в представлении. Операции * и [] проверяют адрес указателя перед разыменованием для уверенности, что он ссылается на свободную память.

Реализация программы, выполняющей эту проверку, сильно зависит от аппаратуры, на которой используется наш указательный тип. Под управлением операционной среды UNIX для поиска текущих границ свободной памяти можно использовать системный вызов `sbrk`.

```
static char *llimit;
class Hptr_init {
    // см. Упр 8-11.
public:
    Hptr_init() { llimit = sbrk(0); }
};
static Hptr_init init;

void
check(void *p){
    if (p<llimit || p>=sbrk(0))
        error();
}
```

Большой эффективности и независимости от аппаратуры можно достичь, переопределяя операторы `new` и `delete`. В этом случае `new` и `delete` могут поддерживать список начальных и конечных адресов блоков свободной памяти, избегая использования `check` для выполнения системного вызова при разыменовании умного указателя.

Наш тип указателя, определенный выше, может быть конкретизирован любым типом, поэтому мы можем получить безопасные указатели на свободную память для предопределенных типов, вроде `char` и `double`, как и для классовых типов.

```
Hptrdeclare(char);
typedef Hptr(char) cHptr;

Hptrdeclare(double);
typedef Hptr(double) dHptr;

struct Pair {
    int i,j;
```

```
};  
Hptrdeclare(Pair);  
typedef Hptr(Pair) PHptr;  
PHptr PP;
```

Наш родовой указатель не предоставляет оператор `->`, так как `->` нельзя использовать с указателями на неклассовые типы. Поэтому для доступа к элементу *Pair* с помощью `PHptr` мы должны использовать запись `(*PP)i` вместо `PP->i`. Мы можем расширить наш тип родowego указателя оператором `->`, но тогда его уже нельзя будет конкретизировать неклассовыми типами.

Предлагая решение этого упражнения, мы допустили некоторые вольности в интерпретации постановки задачи. Нам надо было разработать родовой указатель, гарантирующий указывание только на свободную память, а мы реализовали указатель, который может только ссылаться на свободную память. Указатель может вообще содержать любой адрес; адрес, содержащийся в указателе, проверяется только непосредственно перед его разыменованием при помощи `+` или `[]`.

Это может быть недостатком, так как мы предоставили оператор преобразования, позволяющий преобразование в соответствующий обычный тип указателя.

```
char *strcpy(char *,char *);  
cHptr s;  
char *t;  
strcpy(s,t);
```

Это преобразование может позволить использовать адрес, содержащийся в указателе, даже если тот не указывает на свободную память.

Можно решить эту проблему, избавившись от оператора преобразования, но это уменьшит применимость нашего типа, так как мы не сможем больше использовать объекты этого типа как фактические аргументы и операнды существующих функций и операторов, объявленных для работы с типом обычного указателя. Наш обобщенный тип указателя нельзя будет интегрировать в предопределенную систему типов. Мы можем гарантировать, что значение указателя всегда ссылается на свободную память, проверяя значение

после каждой инициализации, присваивания и арифметической операции, а не при разыменовании. Это может (иногда) предотвратить использование нашего типа указателя в любом алгоритме, пытающемся сослаться за конец структуры данных, если эта структура данных находится в конце свободной памяти.

```
int  
strlen2(cHptr s) {  
    // возвращает длину строки включая '\0'  
    cHptr t = s;  
    while (*t++);  
    return t-s;  
}
```

Так как `t` в конце концов содержит адрес за концом строки символов, на которую ссылается `s`, это вполне может быть адрес "несвободной" памяти.

Можно ли сказать, какой из этих подходов правильный? Есть ли ситуации, в которых каждый из этих подходов предпочтительнее других?

Упр. 8.1.

Предполагаем, что у нас есть функция, округляющая комплексное значение:

```
complex round(complex);
```

Все, что нам нужно сделать - это предложить соответствующий аппликатор.

```
inline complex  
operator +(complex val, complex (*manip)(complex)) {  
    return manip(val);  
}
```

Предоставление тех же возможностей для целых выражений требует другого подхода. Напомним, что определяемый пользователем оператор должен содержать хотя бы один аргумент: классового типа. Так как `int` не является классовым типом, нам требуется, чтобы другой аргумент `operator +` имел классový тип. Вот один из путей:

```
int int_round(int);
```

```
class ROUND {};  
ROUND lround;
```

```
inline int
operator +(int ival, ROUND dummy) {
    return int_round(ival);
}
```

Упр. 8.4.

Так как *sorted_collection* уже имеет способ применения функции к каждому элементу последовательности в нужном порядке, то мы это используем. К сожалению, функция *apply* может иметь аргумент только типа *void(*) (ETYPE)*, поэтому мы должны продумать способ передачи *ostream* обрабатывающей функции.

```
static ostream *oarg;

void
print(ETYPE e) {
    *oarg << e;
    *oarg << " ";
}

ostream &
operator <<(ostream &out, sorted_collection &sc) {
    out << "{";
    oarg = &out;
    apply(print);
    out << "}";
    return out;
}
```

Какие потенциальные проблемы с этим решением вы видите? Как бы вы их решили, если бы могли изменять интерфейс *sorted_collection*?

Упр. 8.11.

Вот одно из решений: мы создаем инициализирующий тип, конструктор которого инициализирует библиотеку. Мы объявляем статический объект этого типа в библиотечном источнике (не в заголовочном файле!).

```
class lib_init {
public:
    lib_init() {
        // текст инициализатора библиотеки
    }
};
```

```
~lib_init() {  
    // текст деструктора  
}  
};
```

```
static lib_init Init;
```

Неявный вызов конструктора для статического инициализирующего объекта инициализирует библиотеку и начинает выполнение. Напротив, вызов деструктора выполнит все необходимые чистки перед завершением выполнения.

ОГЛАВЛЕНИЕ.

Предисловие переводчика.	6
Предисловие.	9
Глава 0. Введение.	11
0.1. Язык Си++.	12
0.2. Парадигмы программирования.	13
0.3. Организация книги.	14
Глава 1. Типы данных и операции.	17
1.1. Числовые типы.	17
1.2. Скалярные типы. Операторы сравнения и логические операторы.	24
1.3. Неабстрактные операции.	27
1.4. Типы, определяемые пользователем.	28
1.5. Указатели и массивы.	32
1.6. Ссылки.	37
1.7. Константы.	39
1.8. Упражнения.	40
Глава 2. Процедурное программирование.	44
2.1. Функции как модули.	45
2.2. Функциональная декомпозиция.	46
2.3. Файловая организация.	50
2.4. Структурное программирование.	53
2.5. Перезагружаемые и подставляемые функции.	57
2.6. Аргументы и возвращаемые значения.	62
2.7. Упражнения.	67
Глава 3. Классы.	69
3.1. Классовые типы.	69
3.2. Ксмпоненты данных.	73
3.3. Функциональные компоненты.	80
3.4. Операторные функции.	83
3.5. Защита доступа и дружественные функции.	86
3.6. Инициализация и преобразования.	87
3.7. Указатели на компоненты класса.	94
3.8. Упражнения.	96
Глава 4. Абстракция данных.	98
4.1. Комплексные числа.	98
4.2. Строки.	104
4.3. Упорядоченные выборки.	109

4.4. Общность.	114
4.5. Абстракция управления.	118
4.6. Упражнения.	126
Глава 5. Наследование.	130
5.1. Базовые и производные классы.	130
5.2. Иерархии классов.	138
5.3. Виртуальные функции.	143
5.4. Защищенные компоненты.	149
5.5. Наследование как инструмент проектирования.	151
5.6. Наследование для расширения интерфейса.	157
5.7. Множественное наследование.	160
5.8. Виртуальные базовые классы.	165
5.9. Упражнения.	168
Глава 6. Объектно-ориентированное программирование.	170
6.1. Проектирование в терминах объектов.	170
6.2. Объектные типы как модули.	176
6.3. Динамический объектно-ориентированный стиль.	178
6.4. Упражнения.	186
Глава 7. Управление памятью.	188
7.1. Управление памятью с помощью конструкторов и деструкторов.	189
7.2. Операторы New и Delete.	192
7.3. Управление памятью для массивов.	196
7.4. New и Delete, ориентированные на классы.	199
7.5. Оператор ->.	205
7.6. X (X&).	212
7.7. Семантика неявной копии.	215
7.8. Упражнения.	217
Глава 8. Библиотеки.	220
8.1. Доступ к существующим библиотекам.	221
8.2. Языки, ориентированные на приложения.	224
8.3. Расширяемые библиотеки.	227
8.4. Настраиваемые библиотеки.	235
8.5. Упражнения.	237
Приложение. Решения упражнений.	241

Учебное издание

**Стефан Дьохарст, Кэти Старк.
Программирование на C++**

Ответственный за выпуск

В. И. Антоненко

Художественный редактор

И. В. Хижняк

Технический редактор

Ю. Н. Артеменко

Корректор

Ю. Н. Артеменко

Сдано в набор 1.06.93. Подписано к печати 8.06.93. Формат 84x108¹/32.
Бумага офсетная. Гарнитура "Таймс". Печать офсетная. Усл. печ. л. 13,03
Усл. краскоот. 13,45. Уч.-изд. л. 14,28. Тираж 40000 экз. Зак. № 3-206
Цена договорная

Издательство Научно-исследовательская проектная фирма «ДиаСофт»
Адрес: 252055, Киев-55, а/я 100.
Тел/факс: (044) 441-9766.

Полиграфкомбинат «Украина»
Адрес: 254119, Киев-119, ул. Дегтяревская, 38—44.

