Министерство экономического развития и торговли Российской Федерации

Министерство образования и науки Российской Федерации

ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ – ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

Нижегородский Филиал

Основы объектно-ориентированного программирования в примерах на C++

Рекомендовано Ученым советом Нижегородского филиала Государственного университета-Высшей школы экономики в качестве учебного пособия для студентов направления "Бизнес-информатика"

Нижний Новгород 2005

УДК 681.3.06
ББК 32.973.26
Д 30

В.М.Дёмкин. Основы объектно-ориентированного программирования в примерах на C++: Учебное пособие. – Н.Новгород: НФ ГУ-ВШЭ, 2005. – 148 с.

ISBN 5-901956-02-8

Учебное пособие предназначено для самостоятельного изучения одного из разделов курса “Информатика” – объектно-ориентированного программирования на языке C++. Обсуждаются вопросы проектирования пользователем абстрактных типов данных с помощью структур, объединений и классов. Пособие содержит как учебные примеры в виде программных реализаций, так и отдельные фрагменты кода, которые все вместе иллюстрируют принципы объектно-ориентированной парадигмы программирования. Программный код апробирован на современных компиляторах платформ Windows и Linux.

Библиогр.: 9 назв.

Рецензенты: д-р физ.-мат. наук, проф. Е.М.Громов
д-р физ.-мат. наук, проф. С.Н.Митяков

ББК 32.973.26

ISBN 5-901956-02-8

Государственный Университет-Высшая Школа Экономики Нижегородский филиал
Н.Новгород, ул. Б.Печерская, 25

Подп. к печ. 25.03.2005. Формат 60x84/16. Бумага офсетная.
Печать офсетная. Усл. п. л.: 8,6. Тираж 200 экз. Заказ № 154.

Отпечатано ООО «Растр НН» г. Н.Новгород, ул. Белинского, 61
ИД № 05407 от 20.07.2001

Предисловие

Курс “Информатика” является одним из базовых курсов блока дисциплин естественнонаучного направления, изучаемый студентами высшей школы всех форм обучения. К числу одних из основных разделов курса “Информатика”, предназначенных для профессиональной подготовки специалистов на стадии изучения языков, поддерживающих императивную и объектно-ориентированную парадигмы программирования, относятся ставшие уже классическими как по форме, так и по содержанию следующие дисциплины:

- Основы алгоритмизации и императивного программирования
  языки программирования – Fortran, Turbo Pascal, C и C++;
- Основы объектно-ориентированного программирования
  языки программирования – Turbo Pascal, Object Pascal, C++ и Java;
- Основы обобщенного программирования
  языки программирования – C++ и Java.

Имея многолетний опыт разработки программного обеспечения широкого ряда уникальных систем автоматизации физических экспериментов, опыт преподавания курса “Информатика” студентам различных форм обучения и специальностей НГТУ и Нижегородского филиала ГУ-ВШЭ, а также опыт проведения студенческих олимпиад по программированию и подготовки студенческих команд НГТУ к командным чемпионатам мира по программированию ACM (Association for Computing Machinery), автор может смело утверждать, что для формирования у студентов младших курсов целостного взгляда на информатику требуется немало усилий. Реальным помощником в этом могут служить отдельные учебные пособия и практикумы по основным разделам курса “Информатика”, связанные единой формой изложения. Прежде всего, сама идея издания подобной литературы связана со стремлением автора рассматривать информатику под углом зрения проектирования алгоритмов и абстрактных типов данных, придерживаясь рамок требуемой для решения задачи парадигмы программирования (например, императивной, объектно-ориентированной или обобщенной) при помощи таких популярных языков программирования для персональных компьютеров, как Fortran, Basic, Turbo Pascal, C, C++ и Java.

Одно из первых таких учебных пособий, предназначенных для самостоятельного изучения основ алгоритмизации и императивного программирования в примерах на Turbo Pascal – “Основы алгоритмизации и императивного программирования” – уже вышло в свет, и автор надеется, что сквозь призму личного опыта он смог донести до каждого заинтересованного читателя характерные этапы процесса обучения навыкам алгоритмизации и императивного программирования.

Автор также надеется, что предлагаемое вниманию читателя следующее учебное пособие, написанное по материалам курса лекций и лабораторного практикума учебной дисциплины “Методы программирования” для студентов НГТУ специальности и направления “Прикладная математика и информатика”, а также для студентов Нижегородского филиала ГУ-ВШЭ направления “Бизнес-информатика”, окажет посильную помощь теперь и в самостоятельном изучении основ объектно-ориентированного программирования на C++.

Введение

Парадигмы программирования и C++

Язык программирования C++ является одним из представителей семейства так называемых гибридных языков, он поддерживает четыре парадигмы программирования: императивную (от слов ***imperative programming***), называемой также процедурно-ориентированной (от слов ***procedure-oriented programming***) или просто процедурной (от слов ***procedural programming***), модульную (от слов ***modular programming***), объектно-ориентированную (от слов ***object-oriented programming***) и обобщенную (от слов ***generic programming***).

C++ был создан Бьярном Страуструпом (Bjarne Stroustrup) в начале 1980-х годов. Перед Страуструпом стояли тогда две основные задачи: во-первых, сделать C++ совместимым со стандартным языком C, во-вторых, расширить C конструкциями объектно-ориентированного программирования, основанными на концепции классов из языка Simula 67. Язык C был создан Дэннисом Ритчи (Dennis M.Ritchie) как язык системного программирования в начале 1970-х годов. История C связана с разработкой операционной системы Unix для мини-компьютера DEC PDP-11.

C++ развивался не только благодаря идеям, взятых из других языков программирования, но и в соответствии с запросами и опытом большого числа пользователей различного уровня, использующих C++ в самых разнообразных прикладных областях. В августе 1998 года был ратифицирован стандарт языка C++ (ISO/IEC 14882). Стандартный C++ и его стандартная библиотека до сих пор остаются одним из основных инструментов для разработки приложений, предназначенных для широкого диапазона использования.

В основе ***императивной парадигмы программирования*** лежит классическая фон-неймановская модель вычислений, разделяющая абстракцию состояния и поведения. Программа, написанная на императивном языке, рассматривается как процесс изменения состояния путем последовательного выполнения отдельных команд (императив), т.е. императивная программа явно указывает способ получения желаемого результата при помощи определенной процедуры, не определяя при этом ожидаемых свойств результата.

Императивный подход наиболее естественен для человека, общающегося с компьютером, поэтому класс императивных языков до сих пор остается самым распространенным классом языков программирования. Первые императивные языки программирования высокого уровня Fortran I (1956 г.), Algol-58 (1958 г.), Fortran II (1958 г.), Cobol (1960 г.) и Algol-60 (1960 г.) оказали весьма существенное влияние на дальнейшее развитие науки программирования. Так, начиная с середины 1960-х годов, во многом благодаря теории и практике императивного программирования стали осознавать роль подпрограмм – своеобразных частей императивной программы – как важного промежуточного звена между решаемой задачей и компьютером, что в конечном итоге привело к становлению принципа процедурного структурирования программ, где подпрограммы рассматривались как абстрактные программные функции. С той поры наряду с понятием “императивный” вошло в обиход и понятие “процедурный” или “процедурно-ориентированный”.

Под процедурой понимается часть программы, которая выполняет некоторые определенные действия над данными, в общем случае определяемыми параметрами. Процедура может быть вызвана из любого места программы (или другой процедуры), и при каждом вызове процедуры ей могут быть переданы различные параметры. Одной из разновидностей процедур являются функции, которые также выполняют некоторые определенные действия над данными, но результат всегда передают в программу (или процедуру, или функцию) в виде значения функции посредством своего имени. Стандартный C++ предоставляет пользователю широкий выбор средств для передачи аргументов вызова функциям и возврата значений из функций. Процедуры в C++ – это функции, которые не возвращают значения.

Процедуры и функции представляют собой относительно самостоятельные фрагменты программы, самостоятельность процедур и функций позволяет локализовать в них все детали программной реализации какого-либо алгоритма. Проектирование программы и ее составных частей при этом основывается на применении методов нисходящего проектирования программ – алгоритм программы представляется в виде последовательности относительно крупных подпрограмм, которые, в свою очередь, представляются в виде последовательности менее крупных подпрограмм и т.д., тем самым реализуется принцип “от сложного – к простому”.

Типичными представителями императивных языков программирования, которые проектировались по мере развития идей процедурного структурирования программ, начиная с 1950-х и до середины 1970-х годов, являются, например, такие языки, как Fortran, Algol, Cobol, Basic, Pascal, Modula.

В основе ***модульной парадигмы программирования*** лежит принцип разбиения всей программы на раздельно компилируемые компоненты, называемых модулями, каждый из которых представляет собой набор связанных процедур вместе с данными, которые обрабатываются только этими процедурами. Физическим представлением каждого модуля является файл. Модули после этапа раздельной компиляции могут участвовать в сборке программы в виде почти независимых частей, что позволяет использовать их как мощный инструмент для разработки библиотек прикладных программ. Модульная парадигма программирования также известна как “принцип сокрытия данных”, который легко расширяется до понятия “сокрытие информации”, что позволяет скрыть от пользователя не только детали реализации каждого модуля, но и исполняемый код процедур.

Дальнейшим развитием идей модульного программирования стал отказ от использования только единственной глобальной области видимости при взаимодействии отдельных частей программы, что привело к созданию механизма логической группировки связанных процедур и данных в пространства имен. Этот механизм группировки позволяет при сборке программы из отдельных частей, например, избегать конфликта имен, не засорять глобальное пространство имен, существенно сократить издержки как на этапе компоновки программы, так и во время ее выполнения. Например, такой подход уже реализован в стандартном C++.

Типичными представителями языков программирования, поддерживающих модульную парадигму программирования, являются, например, такие языки, как Ada, Modula-2, Modula-3, Turbo Pascal, C и C++.

Объектно-ориентированное программирование – это основная методология программирования 1990-х годов и начала XXI века. Она представляет собой продукт более 35 лет практики и опыта, которые восходят к использованию языка Simula 67, затем Smalltalk и не так давно Objective C, Object Pascal, Turbo Pascal, CLOS, а теперь, например, C++ и Java.

В основе объектно-ориентированной парадигмы программирования лежат три принципа:

- инкапсуляция (от слова ***encapsulation*** или ***incapsulation***) – объединение данных и процедур для работы с этими данными в единое целое – объект, который является экземпляром класса;
- наследование (от слова ***inheritance***) – средство получения новых классов из существующих путем добавления в них новых данных и процедур или модифицирования наследуемых данных и процедур;
- полиморфизм (от слова ***polymorphism***) – создание общего интерфейса для группы близких по смыслу действий, выполнение каждого из которых определяется типом данных.

Класс выступает в роли абстрактного типа данных, определяемого пользователем, и отражает базовые концепции предметной области. Объекты типов, определяемых пользователем, называют либо переменными классов, либо экземплярами классов. Каждый объект содержит необходимую информацию, свою для каждого типа. Классы обеспечивают сокрытие информации, гарантированную инициализацию данных, неявное преобразование определяемых пользователем типов, динамическое определение типа, контроль пользователя над управлением памятью, механизм перегрузки стандартных операторов языка – и это далеко не полный перечень того, что классы могут обеспечить.

Преимущество объектно-ориентированной парадигмы программирования по сравнению с императивной в полной мере проявляется при разработке только сложных программных систем, которые проектируются по принципу восходящего проектирования, т.е. “от простого – к сложному”. Императивное программирование наиболее пригодно для реализации небольших по сложности задач, где очень важна скорость исполнения на современных компьютерах. Отметим при этом, что следование принципам императивной парадигмы программирования вовсе не отрицает равноправного использования стандартной библиотеки C++ наряду с традиционной библиотекой C.

Обобщенное программирование – это методология программирования, основанная на принципе параметризации алгоритма, что позволяет ему работать с множеством подходящих типов и структур данных. Такое обобщение алгоритма, позволяющее выразить его независимо от деталей представления, реализуется посредством объявления этого алгоритма как шаблона (от слова ***template***). Механизм шаблонов в C++ допускает использование типа в качестве параметра при определении семейства функций или классов. С помощью шаблонов стандартной библиотеки и шаблонов, определяемых пользователем, создаются обобщенные спецификации для функций и классов – ***родовые*** или ***параметризованные*** функции (от слов ***generic functions***) и классы (от слов ***generic classes***).

Шаблон функции или класса определяет общий набор операций, которые будут применяться к данным различных типов. Тип данных, над которыми должны выполняться эти операции, передается конкретной функции или классу в виде параметра на стадии компиляции, т.е. одна и та же функция или один и тот же класс могут использоваться с несколькими различными типами данных без явного переписывания версии для каждого конкретного типа данных.

Реализация шаблона функции является механизмом генерации исполнимого кода, а реализация шаблона класса – соответственно механизмом генерации необходимых типов на основе спецификации пользователя. Вместе с такой выразительной мощью возникает и некоторая сложность, большая часть которой относится к разнообразию контекстов определения и использования шаблонов.

Объектно-ориентированное программирование и C++

Объектно-ориентированное программирование (ООП) – это такой стиль программирования, который фиксирует поведение реального мира таким способом, при котором детали его реализации скрыты. Если подобное удается, то это позволяет тому, кто занимается решением проблемы, вести рассуждения в терминах предметной области. ООП позволяет разложить проблему на связанные между собой задачи, каждая проблема становится самостоятельным объектом, содержащим свои собственные данные и код.

ООП – это взгляд на программирование, сосредоточенный на данных, в котором данные и поведение (код) жестко связаны. Данные и поведение представлены в виде ***классов***, экземпляры которых – ***объекты***. Класс – это то, о чем можно мыслить как об отдельном понятии, а объект некоторого класса – это то, о чем можно мыслить как об отдельной сущности.

Совокупность принципов проектирования, разработки и реализации программ, которая базируется на абстракции данных, предусматривает создание новых типов данных, с наибольшей полнотой отображающих особенности решаемой задачи. Одновременно с данными для каждого типа определяется набор операций, удобных для обработки этих данных. Таким образом, создаваемые пользователем ***абстрактные типы данных*** (АТД), составленные из множества значений и коллекции операций для работы с этими значениями, могут обеспечить представление понятий предметной области решаемой задачи.

C++ предоставляет пользователю возможность определять собственные типы данных и операции над ними с помощью структур, объединений и классов.

Инкапсуляция

Новая конструкция C++ – класс – обеспечивает механизм инкапсуляции для реализации АТД. Инкапсуляция – это возможность скрыть внутренние детали при описании общего интерфейса АТД с целью защиты его от внешнего вмешательства или неправильного использования. Инкапсуляция включает как детали внутренней реализации специфического типа, так и доступные извне операции и функции, которые могут оперировать объектами этого типа. Детали реализации могут делать недоступным для пользователя код, который использует тип.

Например, стек – одна из разновидностей динамических структур данных – может быть реализован как массив фиксированной длины, доступ к которому осуществляется посредством специального индекса, называемого ***top*** (верхушка стека), при этом общедоступные операции будут включать вталкивание данных в стек (***push***) и выталкивание данных из стека (***pop***). Операции ***push*** и ***pop*** в терминологии ООП называются ***методами***. Изменение внутренней реализации в таком линейном списке не будет влиять на то, как будут извне использоваться операции ***push*** и ***pop***. Реализация стека в данном случае скрыта от его пользователей. АТД, такой как стек, является описанием идеального общего поведения типа. Пользователь этого типа понимает, что операции ***push*** и ***pop*** приводят к определенному общему поведению. Конкретная реализация АТД имеет также свои ограничения, например, после большого числа операций ***push*** область стека переполняется. Эти ограничения действуют на общее поведение.

Классы позволяют пользователю управлять видимостью того, что лежит в основе их реализации. То, что в классе определено как ***public*** (общий, общедоступный), является видимым, т.е. доступным извне класса, а то, что определено как ***private*** (частный, собственный), – скрытым, т.е. изолированным внутри класса. Сокрытие данных – это один из основных принципов абстракции данных. Для обеспечения инкапсуляции C++ использует объявления ***struct***, ***union*** и ***class*** в соединении с ключевыми словами ***private***, ***protected*** и ***public***, определяющими уровень доступа.

Общедоступные компоненты класса образуют открытый интерфейс класса с любыми частями программы. Расширить этот интерфейс позволяют друзья класса – ***дружественные функции*** (***функции-друзья***) и ***классы*** (***классы-друзья***).

В заключение отметим, что терминология ООП все еще находится под сильным влиянием программирования на языке Smalltalk, где новые термины, например, такие как ***сообщение*** и ***метод***, заменили традиционные термины ***вызов функции*** и ***функция-член***. В стандартном C++ при описании внутренней структуры классов используются традиционные термины – члены-данные и функции-члены.

Наследование

Объекты являются переменными класса, т.е. переменными определяемого пользователем типа. ООП позволяет легко создавать и использовать АТД. Для удобства создания нового типа из уже существующих типов, определяемых пользователем, ООП использует механизм наследования. Наследование – это процесс, посредством которого один объект может приобретать основные свойства другого объекта и добавлять к ним черты, характерные только для него.

Наследование – это средство получения новых классов, называемых ***производными классами***, из существующих, называемых ***базовыми классами***. При этом повторно используется уже имеющийся код. Производный класс образуется из базового путем добавления или изменения кода. Посредством наследования может быть создана ***иерархия*** родственных типов (иерархия классов), которые совместно используют код и общий интерфейс. Наследование – это первооснова ООП.

Различают одиночное или простое наследование, когда имеется один базовый класс, и множественное наследование, когда базовых классов более одного.

Наряду с классической иерархией как средством последовательного построения классов особый интерес для пользователей представляют иерархии ***абстрактных классов***, которые предоставляют ясные интерфейсы, не загроможденных деталями реализации и при этом не требующих значительных накладных расходов.

Объекты в ООП отвечают за свое поведение. Создатель АТД должен подключить код для любого поведения, которое сможет "понять" объект. Наличие объекта, отвечающего за свое поведение, облегчает пользователю не только задачу проектирования, но и кодирования этого объекта.

Классы имеют специальные функции-члены – ***конструкторы*** (конструирование значений создаваемых объектов и захват каких-либо ресурсов) и ***деструкторы*** (освобождение памяти и захваченных ресурсов при уничтожении объектов).

Полиморфизм

С концепцией наследования неразрывно связана одна из важнейших концепций ООП – ***полиморфизм***, смысл которой заключается в том, что одни и те же сообщения можно посылать как объектам базового класса, так и объектам всех производных классов. Принцип полиморфизма является естественным следствием существования принципа наследования – наследование без изменения набора свойств не имеет смысла. Целью полиморфизма, применительно к ООП, является использование одного имени для задания общих для класса действий. В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов", что означает возможность создать общий интерфейс для группы близких по смыслу действий, при этом выполнение конкретного действия зависит от типа данных.

Полиморфизм – это механизм, обеспечивающий многократное использование кода. Полиморфизм может применяться и к функциям, и к операторам: в C++ имя функции или оператор перегружаемы – ***статический полиморфизм***. Например, когда одно и то же имя функции используется для множества различных действий, то это называется ***перегрузкой функции***. Фактически в С ограниченно применяется полиморфизм, например, при выполнении арифметических операций. Так, например, для операции деления: если аргументы целого типа, то используется целочисленное деление; если один или оба аргумента вещественного типа, то используется деление с плавающей точкой, т.е. поведение операции деления определяется во время компиляции программы типом данных. Этот механизм применим и к другим – определяемым пользователем – типам данных, в C++ это называется ***перегрузкой оператора***.

При иерархическом наследовании широко используются также ***виртуальные функции-члены***. Виртуальные функции производных классов во время выполнения программы могут замещать соответствующие виртуальные функции своих базовых классов – ***динамический полиморфизм***. Виртуальные функции динамически вызываются в зависимости от типа объекта и, как правило, должны иметь различные реализации. C++ также имеет ***параметрический полиморфизм***, когда один и тот же код используется относительно различных типов, где тип – это параметр тела кода (обработка родовых указателей и шаблонов).

Структуры и объединения – абстрактные типы данных

Структуры

Из базовых типов языка C++ пользователь может конструировать производные типы, среди которых особое место принадлежит таким структурированным типам, как структуры, объединения и классы. Такие типы часто называют абстрактными типами данных или типами, определяемыми пользователем. Абстрактные типы данных служат для представления абстрактных сущностей предметной области решаемой задачи, и каждый из них, в свою очередь, может стать базовым.

Вообще говоря, структуры или объединения являются своего рода упрощенными формами класса, в дальнейшем при изложении классов будет сказано об отличиях, которые как раз приводят к такому упрощению. Здесь же и далее будем говорить о структурах или объединениях как о таких формах класса, в котором будут присутствовать лишь его компонентные данные, т.е. это будут всего лишь просто структуры и объединения данных, что соответствует их представлению в языке C.

Структура – это объединенное в единое целое множество поименованных объектов в общем случае разных типов (переменных, массивов, указателей, структур и т.д.), называемых ***элементами*** или ***членами*** (по аналогии с классами). Все элементы структуры размещаются в памяти подряд и сообща занимают столько места, сколько отведено структуре в целом. Однако размер объекта структурного типа не обязательно равен сумме размеров его элементов. Такое может происходить на многих компьютерах благодаря различиям в реализации механизма выравнивания объектов определенных типов по аппаратно определяемым границам.

Определение структурного типа:

struct имя_структурного_типа {
    имя_типа имя_элемента_структуры_1;
    …
    имя_типа имя_элемента_структуры_n;
};

Элементы структуры могут быть как базовых, так и производных типов, имена элементов структуры должны быть уникальными внутри определяемой структуры.

Определив структурный тип, можно объявлять конкретные структуры:

имя_структурного_типа имя_структуры_1, … , имя_структуры_k;

Кроме того, что структура является упрощенной формой класса, все члены которого по умолчанию являются открытыми, заметим также, что в силу причин, уходящих корнями глубоко в предысторию языка C, разрешается объявлять структуру и не структуру с одинаковыми именами в одной области видимости. Для разрешения этой неоднозначности доступ к структуре в таких случаях должен осуществляться с использованием ключевого слова ***struct***. Сходным образом для разрешения подобной неоднозначности в качестве префикса можно использовать и ключевые слова ***class***, ***union*** и ***enum***. Однако, по совету Страуструпа, лучше избегать такой перегрузки имен.

Обсудим теперь некоторые детали, связанные как со свойствами элементов определяемой структуры, так и с особенностями так называемого опережающего объявления структурного типа:

- Элемент определяемой структуры не может иметь тот же самый тип, что и определяемый структурный тип, т.е. такое рекурсивное определение будет ошибочным, так как компилятор не может определить размер структуры.
- Элементом определяемой структуры может быть указатель на структуру определяемого типа:

struct имя_структурного_типа {
  имя_структурного_типа* имя_указателя_на_структуру;
  …
};

Одним из традиционных примеров использования указателей на структуру определяемого типа является реализация таких АТД, как двухсвязные списки:

```
struct List {
  List* predecessor;   // предыдущий элемент списка
  int data;            // поле данных элемента списка
  List* successor;     // следующий элемент списка
};
```

- Элементом определяемой структуры может быть структура ранее определенного типа:

struct имя_структурного_типа_1 {
  имя_типа имя_элемента_структуры_1;
  …
  имя_типа имя_элемента_структуры_n;
};
struct имя_структурного_типа_2 {
  имя_структурного_типа_1 имя_структуры;
  …
};

- Если в определении структурного типа в качестве элемента необходимо использовать указатель на структуру другого типа, который, например, в этой же единице трансляции будет определен позже или был (может, и вообще не был) определен в другой единице компиляции, то предварительно для этого структурного типа следует дать его опережающее объявление:

struct имя_структурного_типа_1;
struct имя_структурного_типа_2 {
  имя_структурного_типа_1* имя_указателя_на_структуру_1;
  …
};
struct имя_структурного_типа_1 {
  имя_типа имя_элемента_структуры_1;
  …
  имя_типа имя_элемента_структуры_n;
};

Такое определение структурного типа возможно, так как объявление указателя на структуру не требует сведений о размере структуры.

- Чтобы два (или более) объекта структурного типа могли ссылаться друг на друга, можно так же вначале дать опережающее объявление структурного типа:

struct имя_структурного_типа_1;
struct имя_структурного_типа_2 **{**
имя_структурного_типа_1* имя_указателя_на_структуру_1;
…
};
struct имя_структурного_типа_1 **{**
имя_структурного_типа_2* имя_указателя_на_структуру_2;
…
};

Состав каждой структуры определяет ее структурный тип. Определение структурного типа может быть совмещено с объявлением конкретных структур этого типа:

struct имя_структурного_типа **{**
имя_типа имя_элемента_структуры_1;
…
имя_типа имя_элемента_структуры_n;
} ***имя_структуры_1, … , имя_структуры_k;***

Можно не определять именованный структурный тип, а непосредственно объявить конкретные структуры одновременно с определением их компонентного состава:

struct **{**
имя_типа имя_элемента_структуры_1;
…
имя_типа имя_элемента_структуры_n;
} ***имя_структуры_1, … , имя_структуры_k;***

По умолчанию статические элементы структур инициализируются нулевым значением соответствующего типа. При определении структур также возможна их инициализация. В этом случае для структур, как и для массивов, используется синтаксис списков инициализации, принятый в языке C:

имя_структурного_типа имя_структуры **= {**
инициализатор_1,
…
инициализатор_n
};

Например, после определения структурного типа ***Node*** выполним инициализацию элементов структуры ***a***:

```
struct Node {
  char* name;     // имя элемента узла
  int data;       // поле данных элемента узла
  Node* next;     // следующий узел
};
Node a = { "Dolly", 16, 0 };   // здесь 0 или (Node*)0
```

Как будет сказано далее, для инициализации элементов структуры – как одной из форм класса – можно также воспользоваться и его конструктором.

Объект структурного типа, как и любой другой объект, можно инициализировать существующим объектом этого же типа, например, структурой ***a***:

```
Node* pointer = new Node(a);
Node b = a;
Node c(a);
```

Здесь содержимое структуры ***a*** будет скопировано элемент за элементом в новый объект, память для которого будет выделена компилятором либо из кучи с помощью оператора ***new***, либо обычным способом.

Для доступа к элементам объявленной структуры, например, можно использовать оператор **.** (операция выбора члена):

имя_структуры .имя_элемента_структуры

Такую конструкцию иногда называют уточненным именем.

Например, обратимся к первому элементу ***name*** структуры ***a***:

```
cout << a.name << endl;     // Dolly
```

Если элементом структуры, в свою очередь, является тоже структура (называемая членом-объектом), то уточненное имя для ее элемента будет таким:

имя_структуры .имя_элемента_структуры-члена .имя_элемента_структуры

Так как имя структурного типа обладает всеми правами типа, то после определения структурного типа разрешено определять указатели на структуры:

имя_структурного_типа* имя_указателя_на_структуру;

При этом, как правило, определение указателя совмещают с его инициализацией:

имя_структурного_типа* имя_указателя_на_структуру =
&имя_структуры;

Структурный тип задает размеры структуры и тем самым определяет, на какую величину (в байтах) изменится значение указателя на структуру, например, в случае массива структур, если к нему прибавить (или вычесть) ***1***.

После определения указателя доступ к элементам объявленной структуры возможен теперь еще двумя способами:

либо при помощи оператора **->** (операция выбора члена)

имя_указателя_на_структуру->имя_элемента_структуры

либо при помощи операторов * и **.** (операции разыменования и выбора члена)

(имя_указателя_на_структуру*).*имя_элемента_структуры***

Например, инициализацию объекта структурного типа ***Node***, память для которого выделена из кучи с помощью оператора ***new***, можно выполнить и так:

```
Node* pointer = new Node;
pointer->name = "Roman";
pointer->data = 24;
pointer->next = 0;          // здесь 0 или (Node*)0
```

Как и для других объектов, для структур могут быть определены ссылки:

имя_структурного_типа& имя_ссылки_на_структуру = имя_структуры;

После такого определения ссылка ***имя_ссылки_на_структуру*** есть синоним имени структуры ***имя_структуры***.

Объекты структурного типа можно присваивать, передавать в качестве аргументов и возвращать в качестве значений функций.

Например, для функции с одним параметром и типом возвращаемого значения ***void*** информацию о структуре можно передавать:

либо непосредственно

void имя_функции(имя_структурного_типа имя_структуры);

либо через указатель

void имя_функции(имя_структурного_типа* имя_указателя_на_структуру);

либо с помощью ссылки

void имя_функции(имя_структурного_типа& имя_ссылки_на_структуру);

Заметим, что непосредственная передача информации о структуре в виде ее копии (передача функции аргумента вызова по значению) заметно снижает эффективность программы из-за дублирования этого объекта в памяти. Применение указателей на структуру полезно не только для динамических структур данных, но, например, и для таких статических АТД, как массивы структур. Применение же ссылки на объект структурного типа позволяет избегать дублирования этого объекта в памяти.

Например, функция с пустым списком параметров может возвращать:

структуру

имя_структурного_типа имя_функции();

указатель на структуру

имя_структурного_типа* имя_функции();

ссылку на структуру

имя_структурного_типа& имя_функции();

В качестве примеров разработки структурных типов пользователя рассмотрим реализации трех АТД – стек (динамическая структура данных, организованная по принципу ***LIFO*** – ***Last In First Out*** – “последним пришел, первым ушел”), комплексное число (вещественная и мнимая части) и двумерный массив объектов в свободной памяти (динамический массив). Каждый такой АТД составлен из множества значений и коллекции операций для работы с этими значениями.

Стек реализован как одномерный символьный массив фиксированной длины, доступ к которому осуществляется посредством индекса ***top*** (вершина стека). Коллекция операций состоит из стандартных операций работы со стеком.

Например, представим стек для 10 элементов:

```
// Пример 1
// C++ Абстрактный тип данных - стек
#include <iostream>
using namespace std;
const int stackSize = 10;
const int stackEmpty = -1;
const int stackFull = stackSize - 1;
struct Stack {
  char buffer[stackSize];
  int top;
};
```

```
// Привести стек в исходное состояние
void reset(Stack* stack)
{
  stack->top = stackEmpty;
}
// Вталкивание данных в стек
void push(char symbol, Stack* stack)
{
  stack->buffer[++stack->top] = symbol;
}
// Выталкивание данных из стека
char pop(Stack* stack)
{
  return stack->buffer[stack->top--];
}
// Доступ к вершине стека
char top(Stack* stack)
{
  return stack->buffer[stack->top];
}
// Проверить состояние стека - "пустой"
bool empty(const Stack* stack)
{
  return (stack->top == stackEmpty);
}
// Проверить состояние стека - "заполнен"
bool full(const Stack* stack)
{
  return (stack->top == stackFull);
}
int main()
{
  Stack stack;
  char line[] = "Hello, Hello!";
  int i = 0;
  cout << line << endl;
  // Привести стек в исходное состояние
  reset(&stack);
  // Вталкивание символов С-строки в стек
  while (line[i])
    if (!full(&stack))
      push(line[i++], &stack);
    else ++i;
  // Выталкивание символов из стека
  while (!empty(&stack)) cout << pop(&stack);
  cout << endl;
  return 0;
}
```

Результат работы программы:

```
Hello, Hello!
leH ,olleH
```

Вещественная и мнимая части комплексного числа определены как ***double***. Коллекция операций состоит из четырех операций комплексной арифметики (сложение, вычитание, умножение и деление), а также операций инициализации и визуализации комплексных чисел.

Например, вычислим значение выражения $\frac{(-1+5i)^2(3-4i)}{1+3i}+\frac{10+7i}{5i}$:

```
// Пример 2
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  double re;
  double im;
};
// Инициализация комплексного числа
void define(Complex& c, double r = 0.0, double i = 0.0)
{
  c.re = r;
  c.im = i;
}
// Сложение комплексных чисел
Complex add(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}
// Вычитание комплексных чисел
Complex subtract(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re - b.re;
  temporary.im = a.im - b.im;
  return temporary;
}
// Умножение комплексных чисел
Complex multiply(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re * b.re - a.im * b.im;
```

```
  temporary.im = a.re * b.im + b.re * a.im;
  return temporary;
}
// Деление комплексных чисел
Complex divide(Complex a, Complex b)
{
  Complex temporary;
  double divider = b.re * b.re + b.im * b.im;
  temporary.re = (a.re * b.re + a.im * b.im) / divider;
  temporary.im = (b.re * a.im - a.re * b.im) / divider;
  return temporary;
}
// Визуализация комплексного числа
void print(Complex c)
{
  cout << '(' << c.re << ", " << c.im << ')' << endl;
}
int main()
{
  Complex x1, y1, z1, x2, z2;
  // Инициализация операндов выражения - комплексных чисел
  define(x1, -1, 5);              // x1 = -1 + 5i
  define(y1, 3, -4);              // y1 = 3 - 4i
  define(z1, 1, 3);               // z1 = 1 + 3i
  define(x2, 10, 7);              // x2 = 10 + 7i
  define(z2, 0, 5);               // z2 = 5i
  // Визуализация значения выражения - комплексное число
  print(add(divide(multiply(multiply(x1, x1), y1), z1),
        divide(x2, z2)));
  return 0;
}
```

Результат работы программы:

```
(10, 38.2)
```

Динамический двумерный массив определяется значениями своих размерностей: ***columnSize*** (размер столбца) – число строк – и ***rowSize*** (размер строки) – число столбцов массива. Указатель ***base*** – указатель на указатель на ***double***, указатель ***base*** содержит начальный адрес массива указателей (***0, 1, …, columnSize–1***), а каждый указатель этого массива – начальный адрес строки, состоящей из элементов ***double*** (***0, 1, …, rowSize–1***). Коллекция операций состоит из операций выделения и освобождения свободной памяти, а также поиска максимального элемента массива.

Здесь, как и далее, выделение свободной памяти динамическим массивам и соответственно ее освобождение будет реализовано только при помощи операторов ***new*[]** и ***delete*[]**.

Например, представим двумерный массив из 5×5 элементов:

```
// Пример 3
// C++ Абстрактный тип данных – динамический двумерный массив
#include <iostream>
#include <iomanip>
using namespace std;
struct Array2D {
  double** base;
  int columnSize;
  int rowSize;
};
// Выделение свободной памяти для массива
void allocate(int columnSize, int rowSize, Array2D& a)
{
  a.base = new double*[columnSize];
  for (int i = 0; i < columnSize; ++i)
    a.base[i] = new double[rowSize];
  a.columnSize = columnSize;
  a.rowSize = rowSize;
}
// Освобождение выделенной для массива памяти
void deallocate(Array2D& a)
{
  for (int i = 0; i < a.columnSize; ++i)
    delete[] a.base[i];
  delete[] a.base;
}
// Поиск максимального элемента массива
double find_maximum(Array2D& a)
{
  double maximum = a.base[0][0];
  for (int i = 0; i < a.columnSize; ++i)
    for (int j = 0; j < a.rowSize; ++j)
      if (a.base[i][j] > maximum) maximum = a.base[i][j];
  return maximum;
}
int main()
{
  Array2D a;
  // Выделение свободной памяти для массива
  allocate(5, 5, a);
  // Инициализация и визуализация элементов массива
  for (int i = 0; i < a.columnSize; ++i)
  {
    for (int j = 0; j < a.rowSize; ++j)
    {
      a.base[i][j] = (i + 1) * (j + 1);
      cout << setw(4) << (i + 1) * (j + 1);
    }
```

```
    cout << endl;
  }
  // Визуализация максимального элемента массива
  cout << "maximum = " << find_maximum(a) << endl;
  // Освобождение выделенной для массива памяти
  deallocate(a);
  return 0;
}
```

Результат работы программы:

```
   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25
maximum = 25
```

Объединения

Объединение – это производный структурированный тип, синтаксис которого подобен структурам, за исключением того, что ключевое слово ***union*** заменяет ключевое слово ***struct***. Элементы объединения разделяют общий участок памяти, и их значения перекрываются, т.е. все элементы объединения имеют один начальный адрес. Объединение позволяет в разное время использовать одну и ту же область памяти для хранения объектов различных типов и размеров.

Естественно, в данный момент времени объединение может хранить значение только одного из элементов. Размеры элементов объединения соответствуют размерам своих типов, размер объединения определяется размером наибольшего элемента. Инициализатор объединения – это значение, тип которого соответствует типу первого элемента объединения.

Фактически объединение – это структура, все элементы которой имеют нулевое смещение относительно ее базового адреса и размер которой позволяет поместиться в ней самому большому ее элементу, а выравнивание этой структуры удовлетворяет всем типам объединений.

Как и для структур, для объединений может быть определен тип объединения:

union имя_типа_объединения **{**
  имя_типа имя_элемента_объединения_1;
  …
  имя_типа имя_элемента_объединения_n;
};

Как и в случае структур, определив тип объединения, можно объявлять конкретные объединения, их массивы, а также указатели и ссылки на объединения.

Для обращения к элементу объединения, как и в случае структур, можно использовать либо уточненное имя, либо указатель вместе с оператором выбора члена или операторами разыменования и выбора члена.

Например, определим тип объединения ***ShortDoubleChar*** и объединение ***u***, выполнив при этом его инициализацию:

```
union ShortDoubleChar {
  short s;
  double d;
  char c[2];
} u = { 5 };
```

Применение оператора ***sizeof*** для получения сведений о размере типа ***ShortDoubleChar*** и объекта этого типа ***u*** приведет к следующим результатам:

```
sizeof(ShortDoubleChar) = sizeof u = 8
```

Как видим, размер объединения ***u*** определяется размером наибольшего его элемента, а инициализировать это объединение можно только значением типа ***short***. Значение одного из указанных здесь трех типов теперь может быть присвоено переменной ***u*** и далее использовано в выражениях, где это правомерно.

Заметим, что многие применения объединений являются чистой оптимизацией, которые часто основываются на непереносимых способах распределения памяти, поэтому следует с осторожностью относиться к такому способу экономии памяти. Применение объединения таким образом, что его значение всегда читается при помощи элемента, через который оно было записано – это также чистая оптимизация. Однако не всегда легко гарантировать, что объединение будет использоваться только таким образом. Чтобы избежать здесь возможных ошибок, Страуструп рекомендует инкапсулировать объединение, чтобы гарантировалось соответствие между типом элемента и способом доступа к нему.

Как и для структур, можно не определять именованный тип объединения, а непосредственно объявить конкретные объединения одновременно с определением их компонентного состава.

Основное достоинство объединений – возможность разных трактовок одного и того же содержимого (кода) участка памяти, т.е. возможность доступа к одному и тому же участку памяти с помощью объектов разных типов. Заметим, что во многих других языках программирования высокого уровня данное языковое средство называется ***записью с вариантами*** (от слов ***variant record***).

Например, для типа объединения ***ShortChar***, элементами которого являются переменная целого типа ***short s*** и беззнаковый символьный массив ***unsigned char c[2]***, представим различные варианты доступа к элементам объединений ***union1***, ***union2*** и ***union3***:

```
// Пример 4
// C++ Абстрактный тип данных - объединение short и char
#include <iostream>
using namespace std;
union ShortChar {
  short s;
  unsigned char c[2];
} union1 = { 16961 }, union2 = { 17475 };
ShortChar union3;
```

```
short byte0, byte1;
int main()
{
  cout << union1.s << endl;
  cout << (short)union1.c[0] << ' ' << union1.c[0] << endl;
  cout << (short)union1.c[1] << ' ' << union1.c[1] << endl;
  cout << union2.s << endl;
  cout << (short)union2.c[0] << ' ' << union2.c[0] << endl;
  cout << (short)union2.c[1] << ' ' << union2.c[1] << endl;
  union1.c[0] = 'C';
  union1.c[1] = 'D';
  cout << union1.s << endl;
  cout << (short)union1.c[0] << ' ' << union1.c[0] << endl;
  cout << (short)union1.c[1] << ' ' << union1.c[1] << endl;
  union3.s = 19788;
  byte0 = (short)union3.c[0];
  byte1 = (short)union3.c[1];
  cout << union3.s << endl;
  cout << byte0 << ' ' << union3.c[0] << ' '
       << oct << byte0 << ' ' << hex << byte0 << endl;
  cout << dec << byte1 << ' ' << union3.c[1] << ' '
       << oct << byte1 << ' ' << hex << byte1 << endl;
  return 0;
}
```

Результат работы программы:

```
16961
65 A
66 B
17475
67 C
68 D
17475
67 C
68 D
19788
76 L 114 4c
77 M 115 4d
```

Зачастую объединения являются элементами структур или массивов. Например, комбинированное использование структуры и объединения позволяет объявлять так называемые переменные с изменяемой структурой. Рассмотрим пример, в котором информация о геометрических фигурах будет представлена на основе именно такого использования структуры и объединения. Общая информация о фигурах может включать, например, такие элементы, как площадь и периметр, поэтому их следует

представить в виде элементов структуры. Поскольку информация о геометрических размерах фигур может оказаться различной в зависимости от их формы, то ее можно представить в виде элементов объединения.

Приведем определение структурного типа ***Figure*** для создания переменных с изменяемой структурой, где в качестве фигур выбраны круг, прямоугольник и треугольник:

```
struct Figure {
  double area;        // общие элементы
  double perimeter;
  int type;           // метка активного элемента
  union {
    double radius;    // круг
    double a[2];      // прямоугольник
    double b[3];      // треугольник
  } anyFigure;
} figure1, figure2, figure3;
```

Значение переменной ***type*** используется для указания, какой из элементов объединения ***anyFigure*** в данный момент является активным. Объект структурного типа ***Figure*** называется переменной структурой, потому что ее элементы меняются в зависимости от значения метки активного элемента.

Обычно переменная ***type*** – это переменная типа перечисление. Так, для указанных фигур элементами перечисления, например, могут быть выбраны именованные константы ***circle*** (круг), ***rectangle*** (прямоугольник) и ***triangle*** (треугольник):

```
enum figureStatus { circle, rectangle, triangle };
```

Особого разговора заслуживают объявления неименованных типов объединений или так называемых анонимных объединений. Основное достоинство анонимных объединений – это экономия памяти, так как доступ к элементам таких объединений реализуется как к обычным переменным программы. Различие между элементами анонимных объединений и обычными переменными лишь в том, что присваивание значения какому-либо элементу объединения приводит к утрате значений других его элементов. Элементы анонимного объединения находятся в той же области видимости, что и само объединение.

Например, объявим в локальной области видимости главной функции ***main*()** анонимное объединение, элементы которого будут встроенных типов ***short*** и ***long***:

```
// Пример 5
// C++ Абстрактный тип данных - анонимное объединение
#include <iostream>
using namespace std;
int main()
{
  union {
    short shortInt;
    long longInt;
  };
```

```
  shortInt = 1;
  cout << "short\t" << shortInt << "\tlong\t" << longInt << endl;
  longInt = 32768;
  cout << "short\t" << shortInt << "\tlong\t" << longInt << endl;
  return 0;
}
```

Результат работы программы:

```
short   1       long    134479873
short   -32768  long    32768
```

Как видим, присваивание значения одному из элементов анонимного объединения приводит к утрате значения другого его элемента.

И в заключение отметим здесь, что типом элемента объединения не может быть класс с конструктором, деструктором или операцией копирования. В противном случае, как станет ясно из дальнейшего изложения, было бы невозможно предохранить этот член-объект от разрушения или гарантировать, что будет вызван нужный деструктор, когда объединение выйдет за пределы области видимости.

Класс – абстрактный тип данных
Класс как расширение понятия структуры

Класс – это производный структурированный тип, определяемый пользователем на основе уже существующих базовых типов. Механизм классов позволяет создавать типы в полном соответствии с принципами абстракции данных, т.е. класс задает некоторую структурированную совокупность типизированных данных и позволяет определить набор операций над этими данными.

Класс можно определить с помощью инструкции объявления:

ключ_класса имя_класса **{**
  компоненты_класса
};

где ***ключ_класса*** – одно из ключевых слов ***class***, ***struct***, ***union***; ***имя_класса*** – произвольно выбираемый идентификатор; ***компоненты_класса*** – объявления типизированных данных и принадлежащих классу функций. Компонентами класса могут быть данные, функции, вложенные классы, перечисления, битовые поля, дружественные функции, дружественные классы и имена типов.

Все компоненты класса в английском языке обозначаются термином ***member*** (член, элемент). Так, принадлежащие классу функции называют ***member functions*** (функции-члены), а данные класса – ***data members*** (члены-данные). На русском языке принадлежащие классу функции называют либо ***методами класса*** (в терминологии объектно-ориентированного программирования), либо ***функциями-членами*** или ***компонентными функциями***; данные класса называют либо ***членами*** (члены и члены-объекты), либо ***компонентными данными*** или ***компонентами данных***, либо ***элементами данных***.

Слово “метод” чаще используется при рассмотрении внешних связей классов, а словосочетание “функция-член” или “компонентная функция” – при описании внутренней структуры классов.

Например, определим класс ***Complex***, используя ключ класса ***struct*** (в этом случае все компоненты класса по умолчанию будут общедоступными):

```
// Пример 6
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Инициализация комплексного числа
  void define(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
```

```
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)
  {
    re = c.re;
    im = c.im;
  }
};
int main()
{
  Complex x1, x2, y;
  x1.define(-1, 5);
  x2.define(10, 7);
  y.define();
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  y.print(y);                            // (0, 0)
  x1.print(x1.add(x1, x2));              // (9, 12)
  y.assign(y.add(x1, x2));
  y.print(y);                            // (9, 12)
  return 0;
}
```

Общедоступные компоненты класса (компонентные данные и методы класса) образуют открытый интерфейс объектов класса. Как правило, общедоступные компонентные данные объявляют в начале класса.

Класс (как и его частные случаи – структура или объединение) обладает всеми правами типа, следовательно, можно определять объекты класса, а также определять новые производные типы, используя класс как базовый тип.

Для определения объекта класса здесь используется инструкция объявления:

имя_класса имя_объекта;

Например, определим объекты ***x1*** и ***x2*** класса ***Complex***:

```
Complex x1, x2;
```

Как будет сказано далее, наряду с указанным способом существуют и другие, позволяющие определять объекты класса.

В определяемые объекты входят все данные, соответствующие компонентным данным класса (кроме статических данных). Компонентные функции позволяют обрабатывать данные конкретных объектов класса. Но в отличие от компонентных данных компонентные функции не тиражируются при создании конкретных объектов класса, т.е. место в памяти выделяется только для компонентных данных создаваемого объекта класса. Например, применение оператора ***sizeof*** для получения сведений о размере типа ***Complex*** и объектов этого типа – ***x1*** и ***x2*** – приведет к следующим результатам:

```
sizeof(Complex) = sizeof x1 = sizeof x2 = 16
```

Для доступа к компонентам класса, например, можно использовать ***имя_класса*** в качестве квалификатора:

квалификация без указания имени объекта (полная квалификация)

имя_класса*::*имя_компонента_класса

квалификация с указанием имени объекта

имя_объекта*.*имя_класса*::*имя_компонента_класса

Квалификатор ***имя_класса*** вместе с оператором разрешения области видимости **::** могут быть опущены, тогда для доступа к компонентам объекта класса, например, можно использовать уточненные имена:

имя_объекта*.*имя_компонента_данных

***имя_объекта*.*имя_компонентной_функции*(*список_аргументов_вызова*)**

При этом возможности работы с компонентными данными те же самые, как и в случае структур. Например, можно явно присвоить значения компонентным данным объекта ***x1*** класса ***Complex***:

```
x1.re = -1;     // x1.re = -1.0
x1.im = 5;      // x1.im = 5.0
```

Уточненное имя компонентной функции обеспечивает ее вызов для обработки данных именно того объекта класса, имя которого было использовано в уточненном имени этой функции. Например, для объекта ***x1*** класса ***Complex*** вызов компонентной функции ***define*()** позволяет определять значения его компонентных данных:

```
x1.define();         // x1.re = 0.0,  x1.im = 0.0
x1.define(-1, 5);    // x1.re = -1.0, x1.im = 5.0
```

Другой способ доступа к компонентам объекта класса, как и в случае структур или объединений, предусматривает использование указателя на объект класса:

либо при помощи оператора **->** (операция выбора члена)

имя_указателя_на_объект_класса*->*имя_компонента_класса

либо при помощи операторов * и **.** (операции разыменования и выбора члена)

(имя_указателя_на_объект_класса*).*имя_компонента_класса***

Заметим здесь, что если для структур и объединений как простейших форм класса равноправное использование оператора **->** или операторов * и **.** было оправданным, то для классов благодаря разрешению перегрузки оператора **->** (и соответственно запрету перегрузки оператора **.**) предпочтение отдается именно первой конструкции.

Для доступа к компонентным данным используются конструкции:

имя_указателя_на_объект_класса*->*имя_компонента_данных

(имя_указателя_на_объект_класса*).*имя_компонента_данных***

Для доступа к компонентным функциям используются конструкции:

имя_указателя_на_объект_класса*->*имя_компонентной_функции
(*список_аргументов_вызова*)

(имя_указателя_на_объект_класса*) .*имя_компонентной_функции***
(*список_аргументов_вызова*)

Например, определим указатель ***pointer*** на объект ***x1*** класса ***Complex***, с помощью которого определим значения компонентных данных этого объекта и вызовем одну из его компонентных функций:

```
Complex x1;
Complex* pointer = &x1;
pointer->print(x1);     // (0, 0)
pointer->re = -1;       // x1.re = -1.0
pointer->im = 5;        // x1.im = 5.0
pointer->print(x1);     // (-1, 5)
(*pointer).re = 0;      // x1.re = 0.0
(*pointer).im = 0;      // x1.im = 0.0
(*pointer).print(x1);   // (0, 0)
```

Конструкторы, деструкторы и доступ к компонентам класса

В определении класса ***Complex*** есть очевидные недостатки. Первый из них – это отсутствие автоматической инициализации создаваемых объектов. Для каждого вновь создаваемого объекта необходимо либо вызывать компонентную функцию ***define*()**, либо явным образом с помощью уточненных имен присваивать значения его компонентным данным. Использование компонентной функции ***define*()** для инициализации объектов класса неэлегантно и подвержено ошибкам, так как можно либо забыть об этой инициализации, либо выполнить ее, например, дважды. Лучшим подходом здесь будет предоставление пользователю класса возможности объявить функцию, имеющую явное назначение – инициализация объектов.

Итак, для инициализации объектов класса в его определение можно явно включать специальную компонентную функцию, называемую ***конструктором*** (от слова ***constructor***). Конструктор создает, т.е. конструирует значения данного типа.

Формат определения конструктора в теле класса:

***имя_класса*(*список_формальных_параметров_конструктора*)**
{ *тело_конструктора* }

В соответствии с синтаксисом C++ для конструктора не определяется тип возвращаемого значения. Имя конструктора должно совпадать с именем класса. Основное назначение конструктора – инициализация объектов класса. Если класс имеет конструктор, все объекты этого класса будут проинициализированы.

Довольно часто удобно иметь несколько способов инициализации объектов класса, поэтому в классе определяют несколько конструкторов (механизм перегрузки функций-членов класса), предназначенных только для инициализации. При помощи списка аргументов вызова таким конструкторам можно передавать любые данные, необходимые для инициализации объектов класса. Далее будет сказано, что кроме инициализации конструкторы также могут выполнять и другие операции.

Приведем пример класса ***Complex*** с конструктором объектов класса:

```
// Пример 7
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)
  {
    re = c.re;
    im = c.im;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Complex y;
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  y.assign(y.add(x1, x2));
  y.print(y);                            // (9, 12)
  return 0;
}
```

Второй недостаток класса ***Complex*** – это общедоступность его компонентов, т.е. в любом месте программы можно получить доступ, например, к компонентам данных какого-либо объекта либо с помощью уточненных имен, либо с помощью указателя на этот объект класса и соответственно операции выбора члена или операций разыменования и выбора члена. Тем самым не выполняется основной принцип абстракции данных – сокрытие данных внутри объектов класса.

Все компоненты класса, определяемого с помощью ключа класса ***struct***, являются по умолчанию общедоступными (***public***). Для изменения видимости компонентов в определении класса можно использовать спецификаторы доступа.

Спецификатор доступа – это одно из трех ключевых слов: ***private*** (собственный или закрытый), ***public*** (общедоступный или открытый) и ***protected*** (защищенный). Появление любого из спецификаторов доступа в тексте определения класса означает, что до конца определения либо до другого спецификатора доступа все компоненты класса имеют указанный статус. Защищенные компоненты классов необходимы только в случае построения иерархии классов, в противном случае применение спецификатора ***protected*** эквивалентно использованию спецификатора ***private***. Далее будет сказано, что в иерархиях классов при объявлении производных классов их базовые классы, как и компоненты класса, можно объявлять открытыми (***public***), закрытыми (***private***) или защищенными (***protected***). Спецификаторы доступа при объявлении производных классов в иерархиях классов определяют, как компоненты базовых классов наследуются производным классом.

Например, представим определение класса ***Complex***, в котором наряду с его общедоступными компонентами будут объявлены и собственные компоненты:

```
struct Complex {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  // Компонентные данные - все собственные (private)
  private:
  double re;
  double im;
};
```

Как видим, теперь определение класса явно разделено на две части – открытую и закрытую. Здесь в открытом разделе класса объявлены его компонентные функции, а в закрытом – компонентные данные.

В общем случае компоненты класса могут быть объявлены во всех трех его разделах – открытом, закрытом и защищенном.

Конечно же, собственные компоненты класса ***Complex*** можно объявить и перед объявлением его общедоступных компонентов, однако для ключа класса ***struct*** такое чередование закрытого и открытого разделов класса было бы неестественным.

В качестве примера, которого на практике, как правило, следует избегать, можно представить и такое определение класса ***Complex***:

```
struct Complex {
  // Компонентные данные - все собственные (private)
  private:
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

Заметим здесь, что стремление полагаться на умолчание при использовании ключа класса ***struct*** является именно одним из тех шагов, которые ведут к естественному разделению определения класса. С другой стороны, есть стремление ограничить использование ключа класса ***struct*** только для структур как простейших форм класса и тем самым как бы забыть о возможности объявлять с его помощью классы. Однако автор не считает такую позицию оправданной.

Благодаря использованию в качестве ключа класса ключевого слова ***class*** можно изменить статус доступа к компонентам класса по умолчанию. Все компоненты такого класса являются по умолчанию собственными (***private***). Это означает, что они имеют локальную видимость и тем самым являются недоступными для внешних обращений. Это означает также, что компонентные данные такого класса доступны только его компонентным функциям (как будет сказано далее, и его друзьям).

Представим определение класса ***Complex***, в котором чередование его закрытого и открытого разделов теперь уже будет естественным:

```
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
```

```
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

Как видим, стремление полагаться на умолчание при использовании ключа класса ***class*** здесь так же ведет к естественному разделению определения класса.

Применение в качестве ключа класса ключевого слова ***union*** приводит к созданию классов с несколько необычными свойствами, которые нужны для весьма специфических приложений. Например, для экономии памяти за счет многократного использования одних и тех же участков памяти для разных целей. Все компоненты такого класса являются общедоступными, но доступ может быть изменен с помощью спецификаторов доступа. Заметим, что компоненты такого класса не могут быть статическими или членами-объектами с конструкторами и деструкторами.

Итак, пользователю предоставлен широкий выбор средств для реализации принципа сокрытия компонентов класса, однако при этом крайне важно стремление к реализации открытого интерфейса, предполагающего, что некоторые или все принадлежащие классу компонентные функции оставались бы доступными извне, что позволило бы свободно манипулировать его компонентными данными.

В следующем определении класса ***Complex*** введем компонентные функции ***get_re*()** и ***get_im*()**, позволяющие получать доступ к собственным компонентным данным объектов благодаря возвращению ссылки соответственно на вещественную (***re***) и мнимую (***im***) части того объекта, для которого они были вызваны:

```
// Пример 8
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
```

```
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  // Ссылка на вещественную часть комплексного числа
  double& get_re()
  {
    return re;
  }
  // Ссылка на мнимую часть комплексного числа
  double& get_im()
  {
    return im;
  }
  private:
  // Компонентные данные - все собственные (private)
  double re;
  double im;
};
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  x1.print(x1.add(x1, x2));              // (9, 12)
  x1.get_re() += 10;
  x1.get_im() += 7;
  x1.print(x1);                          // (9, 12)
  return 0;
}
```

Итак, подведем некоторые итоги:

- раздел класса ***public*** используется для реализации открытого интерфейса, позволяющего обеспечить доступ к компонентам класса извне этого класса;
- раздел класса ***private*** ограничивает доступность компонентов класса, т.е. изолирует их внутри класса, в этом случае доступ к компонентным данным вне класса возможен только посредством его компонентных функций и его друзей;

- раздел класса ***protected*** содержит компоненты класса, принадлежащие только данному классу и всем его производным классам в иерархии классов.

Теперь об особенностях вызова конструктора. Во-первых, конструкторы подчиняются тем же правилам разрешения перегрузки, что и остальные функции. Если пользователь класса определил несколько конструкторов, компилятор в каждом конкретном случае по списку аргументов вызова, передаваемых конструктору при создании объекта класса, сам выбирает необходимый ему конструктор. Итак, требуемый конструктор всегда автоматически вызывается при создании объекта класса, при этом, как обычно, могут использоваться стандартные значения параметров конструктора – так называемые параметры по умолчанию.

Во-вторых, среди всех определяемых пользователем конструкторов особая роль принадлежит трем конструкторам – ***конструктору по умолчанию***, ***конструктору копирования*** и ***конструктору преобразования***.

Конструктор по умолчанию – это конструктор, не требующий параметров, т.е. это может быть либо конструктор с пустым списком параметров, либо конструктор, в котором все параметры имеют значения по умолчанию. При этом в классе может быть только один конструктор по умолчанию.

Как видим, единственный конструктор класса ***Complex*** является именно конструктором по умолчанию:

```
Complex(double r = 0.0, double i = 0.0)
{
  re = r;
  im = i;
}
```

Если единственный параметр конструктора является ссылкой на тот же тип данных, что и тип конструктора, то такой конструктор называется конструктором копирования. Конструктор копирования осуществляет почленное копирование данных при выполнении операции инициализации. Например, семантика вызова параметра функции по значению требует, чтобы локальная копия типа параметра создавалась и инициализировалась от значения выражения, переданного этой функции как аргумент вызова, поэтому для этого как раз и необходим конструктор копирования, сигнатура которого ***имя_класса*(*const имя_класса&*)**.

Так, например, все представленные ранее вызовы компонентных функций для объектов класса ***Complex***, переданных в виде аргументов по значению, приводили к неявному вызову конструктора копирования, сгенерированного компилятором, называемого также еще и конструктором копирования по умолчанию.

Конструктор копирования класса ***Complex***, который, как и конструктор копирования по умолчанию, тоже осуществлял бы почленное копирование всех компонентных данных, например, можно определить так:

```
Complex(const Complex& c)
{
  re = c.re;
  im = c.im;
}
```

Конструктор копирования (сгенерированный компилятором или определенный пользователем) позволяет при создании объекта инициализировать его другим, уже существующим, объектом этого же класса, используя оператор = в самом определении объекта.

Например, определение объекта ***y*** класса ***Complex*** можно выполнить так:

```
Complex x(-1, 5);    // x.re = -1.0, x.im = 5.0
Complex y = x;       // y.re = -1.0, y.im = 5.0
```

Здесь конструктор копирования при создании объекта ***y*** инициализирует его объектом ***x***, выполняя почленное копирование.

Отметим, что перегрузка оператора = не влияет на выполнение этой операции.

Зачастую от пользователя класса требуется реализация полного контроля над процессом образования копии объекта во время выполнения инициализации, в таких случаях пользователь всегда определяет свой собственный конструктор копирования. Как будет сказано далее, отсутствие такого контроля может стать потенциальным источником проблем, связанных с нежелательными, а порой и неожиданными эффектами.

Если единственный параметр конструктора имеет тип, отличающийся от типа конструктора, то такой конструктор называется конструктором преобразования. Конструктор преобразования задает преобразование типа своего аргумента в тип конструктора (по умолчанию неявное, если только это преобразование уникально).

Например, конструктор преобразования класса ***Complex*** можно определить так:

```
Complex(double r)
{
  re = r;
  im = 0.0;
}
```

Конструктор преобразования, как и конструктор копирования, тоже позволяет инициализировать объект, используя оператор = в самом определении объекта.

Например, объект ***x*** класса ***Complex*** можно проинициализировать и так:

```
Complex x = 5;   // x.re = 5.0, x.im = 0.0
```

Здесь вначале создается временный объект, значение которого строится благодаря неявному вызову конструктора преобразования ***Complex*(*5*)**, затем этим объектом уже инициализируется объект ***x*** с помощью конструктора копирования по умолчанию. Перегрузка оператора = здесь так же не влияет на выполнение этой операции.

Очевидно, что конструктор по умолчанию класса ***Complex*** в случае вызова только первого аргумента может выступить также и в роли конструктора преобразования, однако не стоит всегда полагаться на это его свойство. Очевидно, что при этом не должен определяться сам конструктор преобразования, иначе будет иметь место либо неоднозначность вызова среди этих конструкторов, либо ошибка приведения типа.

Отметим также, что использование конструктора для преобразования типа удобно, но может иметь неприятные побочные эффекты, кроме того, в некоторых случаях неявное преобразование просто нежелательно, поэтому для его подавления можно объявить конструктор со спецификатором ***explicit*** (***явный***) – такой конструктор всегда будет вызываться только явно.

Например:

```
explicit Complex(double r)
{
  re = r;
  im = 0.0;
}
```

В этом случае инициализацию можно выполнить только так:

```
Complex x = Complex(5);   // x.re = 5.0, x.im = 0.0
```

Другим решением этой проблемы, например, может быть стремление уменьшить набор значений, допустимых для стиля инициализации с оператором = по сравнению со стилем **()**, сделав конструктор копирования закрытым компонентом класса или объявив его со спецификатором ***explicit***.

В заключение отметим, что конструктор преобразования в принципе не может осуществить:

- неявное преобразование из класса во встроенный тип (так как встроенные типы не являются классами);
- преобразование из нового класса в ранее определенный класс (не модифицируя объявление старого класса).

Эти проблемы можно решить путем определения ***оператора преобразования*** для исходного типа. Компонентная функция класса ***имя_класса*::*operator имя_типа*()** определяет неявное преобразование исходного типа ***имя_класса*** в преобразуемый тип ***имя_типа***. Операторы преобразования представляют собой операторные функции, которые будут рассматриваться в разделе, посвященном перегрузке стандартных операторов.

В-третьих, если пользователь класса сам не определил конструктор по умолчанию и если при этом не определил другие конструкторы, компилятор при необходимости генерирует конструктор по умолчанию, статус доступа которого общедоступный. Роль конструктора по умолчанию, сгенерированного компилятором, весьма важна, он неявно вызывает конструкторы по умолчанию для компонентных данных класса, являющихся членами-объектами, и конструкторы базовых классов.

Например, после определения структурного типа ***Numbers*** при определении структуры ***a*** ее элемент ***a.c***, являющийся членом-объектом класса ***Complex***, будет проинициализирован своим конструктором по умолчанию, который был вызван конструктором по умолчанию, сгенерированным компилятором, для компонентных данных класса ***Numbers***:

```
struct Numbers {
  Complex c;
  int i;
  float f;
  double d;
};
Numbers a;      // a.c.re = 0.0, a.c.im = 0.0
```

Конструкторы базовых и производных классов будут рассматриваться в разделе, посвященном наследованию классов.

Отметим здесь также еще одно весьма важное обстоятельство, связанное с тем, что, так как константы и ссылки должны быть проинициализированы, объект класса, компоненты которого являются константами или ссылками, не может быть сконструирован по умолчанию, если только пользователь явно не определил в классе конструктор по умолчанию. Далее будет сказано, какой синтаксической формы следует придерживаться при определении конструктора по умолчанию для инициализации таких компонентов класса.

Конструкторы по умолчанию можно вызывать явно как для типов, определяемых пользователем, так и для встроенных типов, т.е. встроенные типы также имеют конструкторы по умолчанию. Результатом явного вызова конструктора по умолчанию для встроенных типов является ***0***, преобразованный в соответствующий тип, например, ***int*()** является значением ***int*** по умолчанию, т.е. равным ***0***:

```
int n = int();   // n = 0
```

Использование конструкторов по умолчанию для встроенных типов весьма важно при написании шаблонов, когда пользователь заранее не знает, какой же тип будет использоваться в качестве параметра шаблона – встроенный или определяемый пользователем.

В-четвертых, если пользователь класса сам не определил конструктор копирования, компилятор так же при необходимости генерирует конструктор копирования, статус доступа которого общедоступный. Однако при дальнейшем обсуждении особенностей вызова конструктора копирования станет ясно, почему использование конструктора копирования, который сгенерирован компилятором, может стать потенциальным источником проблем. Например, одним из способов обойти эти проблемы как раз и является явное определение пользователем класса собственного конструктора копирования.

В заключение отметим, что нельзя получить адрес конструктора и что типом параметра конструктора не может быть его собственный класс.

Для явного вызова конструктора можно использовать две синтаксические формы:

***имя_класса имя_объекта*(*список_аргументов_вызова*);**

***имя_класса*(*список_аргументов_вызова*);**

Первая форма допускается только при непустом списке аргументов вызова, передаваемых конструктору, она предусматривает вызов конструктора при создании нового объекта данного класса.

Вторая форма вызова приводит к созданию временного объекта, который может использоваться в тех выражениях, где допустимо использование объекта данного класса.

Обсудим теперь возможные способы инициализации компонентных данных объекта с помощью конструкторов. Первый способ, предусматривающий передачу аргументов вызова в тело конструктора, уже был представлен. Как видим, здесь для инициализации компонентных данных класса используется оператор присваивания. Однако определенным типам данных, например, константам или ссылкам, таким способом не могут быть присвоены значения. Для такого случая предусмотрен второй способ инициализации, основанный на применении списка инициализации компонентных данных объекта в определении конструктора.

Формат определения конструктора со списком инициализации:

***имя_класса*(*список_формальных_параметров_конструктора*) :**
***список_инициализации_компонентных_данных* { *тело_конструктора* }**

Элементы в списке инициализации разделяются знаком препинания запятая, каждый элемент списка относится к конкретному компоненту данных и имеет вид:

***имя_компонентного_данного*(*выражение*)**

Например, конструктор по умолчанию класса ***Complex*** можно определить и так:

```
Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
```

Например, конструктор копирования класса ***Complex***, который копирует все его компонентные данные, можно было бы определить еще и так:

```
Complex(const Complex& c) : re(c.re), im(c.im) {}
```

Однако поскольку конструктор копирования по умолчанию имеет тот же смысл, поэтому следует полагаться на это умолчание. Заметим здесь, что если для класса был явно объявлен какой-либо конструктор, то нельзя пользоваться списком инициализации, как это было принято для структур. Например, в случае явного определения конструктора по умолчанию класса ***Complex*** такой вид инициализации для объекта ***x*** был бы ошибочным:

```
Complex x = {1, 2};   // ошибка!
```

Третий способ инициализации, как и первый, уже был представлен, он связан с неявным вызовом конструктора копирования, когда при создании объектов класса используется операция присваивания.

Последний, четвертый, способ применяется при проектировании иерархии классов, когда используется расширенная форма объявления конструкторов производных классов, позволяющая через список инициализации членов, в котором могут быть представлены явные вызовы конструкторов прямых базовых классов, передавать по цепочке наследования всем требуемым конструкторам косвенных базовых классов необходимые им аргументы вызова.

Конструкторы используются также и при выделении свободной памяти объектам, создаваемых при помощи оператора ***new*** (или ***new*[]**) и уничтожаемых впоследствии при помощи оператора ***delete*** (или ***delete*[]**). Операторы ***new*** и ***new*[]** неявно вызывают требуемые конструкторы для инициализации создаваемых объектов, что является важным преимуществом использования именно этой операции по сравнению с другими для создания объектов в свободной памяти.

Например, оператор ***new*** для инициализации создаваемых объектов в свободной памяти в зависимости от списка аргументов вызова может вызвать один из четырех возможных видов конструкторов: конструктор по умолчанию, конструктор копирования, конструктор преобразования, а также любой конструктор, отличный от первых трех.

Создание объекта в свободной памяти и его инициализация конструктором по умолчанию:

имя_класса* имя_указателя = new имя_класса;

Создание объекта в свободной памяти и его инициализация одним из четырех возможных видов конструкторов:

***имя_класса* имя_указателя = new имя_класса*(*список_аргументов_вызова*);**

Уничтожение объекта в свободной памяти:

delete имя_указателя;

На примере класса ***Complex*** рассмотрим характерные случаи создания объектов в свободной памяти и их инициализации возможными видами конструкторов:

- конструктор по умолчанию

```
Complex* p = new Complex;        // p->re = 0.0, p->im = 0.0
```

- либо конструктор по умолчанию, либо конструктор преобразования

```
Complex* p = new Complex(1);     // p->re = 1.0, p->im = 0.0
```

- либо конструктор по умолчанию, либо конструктор с двумя параметрами

```
Complex* p = new Complex(1, 5); // p->re = 1.0, p->im = 5.0
```

- конструктор копирования

```
Complex x(1, 5);                 // x.re = 1.0, x.im = 5.0
Complex* p = new Complex(x);     // p->re = 1.0, p->im = 5.0
```

Страуструп отмечает, что не существует способа явного указания аргументов конструктора (за исключением использования списка инициализации в стиле C) при объявлении массивов объектов, в том числе и объектов в свободной памяти. Так, при создании массива объектов в свободной памяти инициализация его элементов возможна только с помощью конструктора по умолчанию, поэтому пользователь класса должен явно определить такой конструктор. В дальнейшем станет известно, как, например, можно получить такой способ с использованием локального класса (класса, определенного в теле функции).

Объявление массивов объектов в свободной памяти требует использования различного вида встроенных указателей, например, для одномерного массива – это обычный указатель, для двумерного массива – это уже указатель на указатель, для трехмерного массива – это указатель на указатель на указатель и т.д. Напомним, что при объявлении массивов объектов в свободной памяти для указания размерностей вместо константных выражений можно использовать и переменные. Ограничимся здесь, как и ранее, примерами только для одномерных и двумерных массивов.

Создание одномерного массива объектов в свободной памяти и их инициализация конструктором по умолчанию:

имя_класса* имя_указателя = new имя_класса[размерность_массива];

Уничтожение одномерного массива объектов в свободной памяти:

delete[] имя_указателя;

Создание двумерного массива объектов в свободной памяти и их инициализация конструктором по умолчанию:

имя_класса** имя_указателя = new имя_класса*[размер_столбца];
for (int i = 0; i < размер_столбца; ++i)
имя_указателя[i] = new имя_класса[размер_строки];

Уничтожение двумерного массива объектов в свободной памяти:

for (int i = 0; i < размер_столбца; ++i) delete[] имя_указателя[i];
delete[] имя_указателя;

Здесь следует отметить, что для выделения свободной памяти объекту оператор ***new*** неявно вызывает операторную функцию ***operator new()***, а для ее освобождения оператор ***delete*** неявно вызывает операторную функцию ***operator delete()***.

Аналогично, для выделения свободной памяти массиву объектов оператор ***new*[]** вызывает операторную функцию ***operator new*[]()**, а для ее освобождения оператор ***delete*[]** вызывает операторную функцию ***operator delete*[]()**.

Представим прототипы этих стандартных операторных функций:

void* operator new(size_t);
void operator delete(void*);
void* operator new[](size_t);
void operator delete[](void*);

Здесь ***void**** – это родовой (обобщенный) указатель, а ***size_t*** – это интегральный тип без знака (определен в стандартной библиотеке как тип результата оператора ***sizeof***).

Все эти операторные функции можно перегружать, тем самым пользователь может сам определить, каким образом будет осуществляться выделение и освобождение свободной памяти. В дальнейшем при обсуждении наследования классов будет сказано о возможных способах управления выделением и освобождением свободной памяти. Напомним здесь лишь, что стандартная реализация операторных функций ***operator new*()** и ***operator new*[]()** не инициализирует выделяемую память.

Отметим здесь также, что стандартная реализация оператора ***new*** выделяет памяти немного больше, чем потребовалось бы для статического объекта. Это связано с необходимостью хранения размера объекта, так как при освобождении памяти операторы ***delete*** и ***delete*[]** должны иметь возможность определить размер уничтожаемого объекта. Как правило, для хранения размера объекта используется одно дополнительное слово.

Свободная память является одним из тех ресурсов компьютера, который наиболее часто захватывается конструктором класса, другим таким ресурсом, например, является файл. Очевидно, что динамическое выделение памяти для объектов класса требует наличия в этом классе некоторой компонентной функции, которая будет гарантированно вызвана для освобождения выделенной памяти при уничтожении объекта класса, аналогично конструктору, который гарантированно вызывается при создании объекта. Возможность автоматического освобождения памяти, как и любого другого ресурса, который захватывается конструктором класса, обеспечивает специальная компонентная функция класса – ***деструктор*** (от слова ***destructor***) – разрушитель объектов класса.

Формат определения деструктора в теле класса:

***~имя_класса*() { *тело_деструктора* }**

У деструктора не может быть параметров, он, как и конструктор, не имеет возвращаемого значения, вызов деструктора выполняется неявно, например, когда уничтожается объект в свободной памяти или, например, когда объект класса выходит из области видимости.

В некоторых особых случаях, например, когда объекту выделяется область памяти, которая непосредственно не управляется стандартными средствами распределения свободной памяти, возможен и явный вызов деструктора. Однако, по совету Страуструпа, следует избегать явных вызовов деструкторов, равно как и использования глобальных распределителей памяти специального назначения, там, где это только возможно.

Итак, деструктор в классе является той компонентной функцией, которая логически дополняет конструктор, на что явно указывает и сам синтаксис определения деструктора. Если в классе не определен деструктор, то такой класс можно рассматривать как тип, деструктор которого ничего не делает. В целом же эта комплиментарная пара – конструктор и деструктор – является типичным механизмом в C++ для выражения понятия объекта переменного размера. Далее при обсуждении наследования классов будет сказано о возможной виртуальной природе деструктора.

Заметим, что для встроенных типов наряду с конструктором по умолчанию разработчиками C++ также естественным образом введен и деструктор, что является последовательной реализацией цели по обеспечению одинакового поведения всех типов языка как в семантическом, так и синтаксическом отношениях.

Представим класс ***Stack***, позволяющий реализовать стек произвольной длины в виде одномерного массива символьных объектов в свободной памяти:

```
// Пример 9
// C++ Абстрактный тип данных - стек
#include <iostream>
using namespace std;
class Stack {
  // Компонентные данные - все собственные (private)
  char* pointer;
  int top;
  int stackEmpty;
  int stackFull;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Stack(int stackSize = 100):
    stackEmpty(-1),
    stackFull(stackSize - 1)
  {
    pointer = new char[stackSize];
    top = stackEmpty;
  }
  // Вталкивание данных в стек
  void push(char symbol)
  {
    pointer[++top] = symbol;
  }
  // Выталкивание данных из стека
  char pop()
  {
    return pointer[top--];
  }
  // Проверить состояние стека - "пустой"
  bool empty()
  {
```

```
    return (top == stackEmpty);
  }
  // Проверить состояние стека - "заполнен"
  bool full()
  {
    return (top == stackFull);
  }
  // Деструктор объектов класса
  ~Stack()
  {
    delete[] pointer;
  }
};
int main()
{
  Stack stack(10);
  char line[] = "Hello, Hello!";
  int i = 0;
  cout << line << endl;
  // Вталкивание символов С-строки в стек
  while (line[i])
    if (!stack.full())
      stack.push(line[i++]);
    else ++i;
  // Выталкивание символов из стека
  while (!stack.empty()) cout << stack.pop();
  cout << endl;
  return 0;
}
```

Результат работы программы:

```
Hello, Hello!
leH ,olleH
```

Теперь рассмотрим способы создания и последующего уничтожения объектов, перечисляя их по возможности в порядке важности:

- именованный автоматический объект, создаваемый каждый раз, когда встречается его объявление во время выполнения программы и уничтожаемый при каждом выходе из блока, в котором он был объявлен;
- объект в свободной памяти, создаваемый при помощи оператора ***new*** и уничтожаемый при помощи оператора ***delete***;
- нестатический член-объект, создаваемый и уничтожаемый тогда, когда создается и уничтожается содержащий его объект;
- элемент массива, создаваемый и уничтожаемый тогда, когда создается и уничтожается массив, элементом которого он являлся;

- локальный статический объект, создаваемый, когда его объявление встречается первый раз при выполнении программы и уничтожаемый один раз при ее завершении;
- глобальный объект, объект в пространстве имен или статический объект класса, создаваемые один раз при запуске программы и уничтожаемые один раз при ее завершении;
- временный объект, создаваемый для хранения промежуточного результата вычисления части выражения и уничтожаемый по завершении вычисления всего выражения;
- временный объект, создаваемый для хранения возвращаемого функцией значения и уничтожаемый, как только это значение возвращается;
- временный объект, создаваемый для инициализации ссылки на константу или именованного объекта и уничтожаемый, когда его ссылка или именованный объект выходят из области видимости;
- временный объект, создаваемый в случае явного вызова конструктора и уничтожаемый обычным образом, как и неявно генерируемые переменные;
- объект, помещенный в память, выделенную функцией пользователя, учитывающей аргументы, которые передаются ей операцией выделения памяти;
- элемент объединения ***union***, который не может иметь ни конструктора, ни деструктора.

Обсудим некоторые из перечисленных особенностей вызова конструкторов и деструкторов при создании и уничтожении объектов.

Конструкторы локальных объектов вызываются каждый раз, когда управление передается их инструкциям объявления. Деструкторы локальных объектов выполняются в порядке, противоположном выполнению их конструкторов.

Конструкторы членов-объектов вызываются до вызова конструктора объемлющего класса в том порядке, в котором эти члены-объекты были объявлены в классе, а не в том, в котором они были записаны в списке инициализации конструктора объемлющего класса, именно поэтому лучше записывать список инициализации в порядке, соответствующем объявлению членов-объектов. Когда уничтожается объект объемлющего класса, сначала вызывается его деструктор, а потом вызываются деструкторы членов-объектов в порядке, обратном вызовам их конструкторов. Если конструктор члена-объекта не нуждается в аргументах, этот член можно не указывать в списке инициализации конструктора объемлющего класса.

Инициализаторы членов класса имеют большое значение для типов, у которых инициализация отличается от присваивания, т.е. для членов-объектов класса без конструкторов по умолчанию, для константных членов и для членов, являющихся ссылками. Использование списка инициализации в определении конструктора класса для таких его членов является единственно возможным способом их инициализации.

Компоненты класса, являющиеся статическими константами интегрального типа, можно инициализировать с помощью константных выражений при их определении, однако не следует забывать об их повторном определении вне определения класса. Внутри класса можно использовать элементы перечислений в качестве символьных констант, в этом случае не требуется их повторного определения.

Конструктор локального статического объекта вызывается один раз при первом выполнении его инструкции объявления. Деструкторы локальных статических объектов вызываются в порядке, обратном их созданию, при завершении программы.

Конструкторы нелокальных объектов программы (глобальные объекты, объекты в пространстве имен или статические объекты класса) в единице трансляции вызываются до вызова функции ***main*()** в порядке их определений. После выхода из функции ***main*()** будут вызваны деструкторы для каждого такого объекта в порядке, обратном вызовам их конструкторов. Отметим здесь, что не дается никаких гарантий по поводу порядка конструирования нелокальных объектов из различных единиц компиляции – порядок конструирования и уничтожения таких объектов определяется конкретной реализацией C++.

На примере класса ***Complex*** продолжим обсуждение особенностей объявления массивов его объектов.

Итак, известно, что если объект класса может быть создан без явного задания инициализирующего значения, т.е. при помощи конструктора по умолчанию, то можно определить массив объектов этого класса в стиле C.

Например, определим массив из 5 объектов типа ***Complex***:

```
Complex a[5];  // массив из 5 объектов
```

Здесь каждый элемент массива будет проинициализирован при помощи неявного вызова конструктора по умолчанию стандартными значениями его аргументов вызова, равными ***0***. Например, результат визуализации первого элемента этого массива будет таким:

```
a[0].print(a[0]);   // (0, 0)
```

Например, определим динамический массив из 5 объектов типа ***Complex***:

```
Complex* p = new Complex[5];  // массив из 5 объектов
```

Здесь оператор ***new*[]** неявно вызывает конструктор по умолчанию для стандартных значений его аргументов, поэтому результат визуализации, например, первого элемента этого массива будет тот же самый, что и для массива ***a***:

```
p[0].print(p[0]);   // (0, 0)
```

Известно также, что при объявлении массива объектов можно воспользоваться и списком инициализации в стиле C.

Например, определим массив из 5 объектов типа ***Complex***:

```
Complex b[5] = { 1, 2, 3, 4, 5 };  // массив из 5 объектов
```

Здесь каждый элемент массива будет проинициализирован при помощи неявного вызова конструктора по умолчанию значением первого его аргумента вызова, соответствующего значениям элементов списка инициализации, и стандартным значением второго его аргумента. Например, результат визуализации первого элемента этого массива будет таким:

```
b[0].print(b[0]);   // (1, 0)
```

Заметим здесь, что при создании массива объектов инициализация его элементов возможна также еще и с помощью неявного вызова конструктора преобразования, если используются только списки инициализации в стиле C. Если необходимо проинициализировать элементы массива различными значениями, можно, например, определить такой конструктор по умолчанию, который непосредственно или

косвенно считывает и записывает нелокальные данные, однако, как правило, следует избегать подобных сложностей. При уничтожении массива объектов деструктор для каждого его элемента так же будет вызываться неявно.

Если по какой-либо причине массивы в стиле C слишком неудобны, можно воспользоваться вместо них классом, подобным ***vector***. Напомним здесь, что вектор является одним из шаблонов класса стандартной библиотеки C++. Память для объектов класса ***vector*** выделяется и освобождается при помощи простых операторов ***new*** и ***delete***.

Например, создание и уничтожение вектора из 5 элементов типа ***Complex***:

```
vector<Complex>* p = new vector<Complex>(5);    // объект
delete p;
```

А теперь в завершение обратимся к одной из важнейших форм перегружаемого конструктора – конструктору копирования. Ранее уже было сказано, что операция копирования объектов класса требует особого внимания со стороны пользователя класса. Отсутствие контроля при выполнении этой операции может привести к нежелательным и порой неожиданным эффектам.

Известно, что значение одного объекта может быть передано другому лишь в двух случаях – при присваивании и при инициализации.

В дальнейшем при изложении механизма перегрузки стандартных операторов C++ будет сказано о проблеме определения пользователем копирующего оператора присваивания. Здесь же следует отметить, что выражение присваивания для объектов класса, как и для структур, по умолчанию означает почленное копирование данных одного объекта в другой. Такое копирование обычно является неправильным при копировании объектов, имеющих ресурсы, управляемые конструктором и деструктором. Не менее катастрофичной такая ситуация наблюдается, например, при наличии в классе членов, являющихся указателями или ссылками. Кроме того, присваивание по умолчанию не может быть сгенерировано, если нестатический компонент класса является ссылкой, константой или типом, определяемым пользователем, не имеющим копирующего оператора присваивания. Очевидно, что при копирующем присваивании не должен вызываться никакой конструктор, в том числе и конструктор копирования, так как в противном случае изменилось бы содержание объекта-копии, т.е. при копировании объекта-оригинала необходимо его текущее, а не начальное состояние.

Конструктор копирования используется только для инициализации объектов, которая может иметь место в трех случаях:

- когда в инструкции объявления один объект используется для инициализации другого;
- когда объект передается в функцию в качестве аргумента;
- когда создается временный объект для хранения возвращаемого функцией значения.

Конструктор копирования, как правило, имеет общую форму:

***имя_класса*(*const имя_класса& имя_объекта*) { *тело_конструктора* }**

Здесь объект ***имя_объекта*** используется для инициализации другого объекта, свойства которого определяются перечисленными выше случаями.

Например, для объекта ***y*** класса ***Complex*** указанные случаи инициализации могли бы выглядеть следующим образом:

- **`Complex x = y; // объект y инициализирует объект x`**
- **`Complex x(y); // объект y инициализирует объект x`**
  здесь конструктору копирования передается ссылка на объект ***y***;
- **`y.print(y); // объект y передается как аргумент`**
  здесь конструктору копирования передается ссылка на объект ***y***;
- **`y.add(x, x); // создание временного объекта для add()`**
  здесь конструктору копирования кроме ссылок на объект ***x*** передается еще и ссылка на временный объект.

В том случае, когда в инструкции объявления один объект используется для инициализации другого, пользователю класса иногда требуется изменить значение объекта-оригинала, тогда конструктор копирования может иметь и такую форму:

***имя_класса*(*имя_класса*& *имя_объекта*) { *тело_конструктора* }**

Как увидим далее, такой конструктор копирования, например, необходим для “умных указателей” – объектов класса, которые ведут себя как встроенные указатели и, кроме того, выполняют некоторые действия, когда с их помощью осуществляется доступ к компонентам другого класса, на объекты которого эти указатели ссылаются.

В качестве примера корректного управления свободной памятью, выделяемой для динамических одномерных массивов, рассмотрим определение класса ***Array1D***, в котором явно определен конструктор копирования:

```
class Array1D {
  // Компонентные данные - все собственные (private)
  int* pointer;
  int arraySize;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array1D(int size)
  {
    pointer = new int[size];
    arraySize = size;
  }
  // Конструктор копирования
  Array1D(const Array1D&);
  // Деструктор объектов класса
  ~Array1D()
  {
    delete[] pointer;
  }
};
// Конструктор копирования
Array1D::Array1D(const Array1D& c)
{
```

```
    pointer = new int[c.arraySize];
    arraySize = c.arraySize;
    for (int i = 0; i < arraySize; ++i)
      pointer[i] = c.pointer[i];
  }
```

Например, для объектов ***x*** и ***y*** класса ***Array1D*** представим теперь возможные варианты вызова его конструкторов:

```
Array1D x(10);      // вызов конструктора
Array1D y = x;      // вызов конструктора копирования
```

При определении объекта ***x*** (одномерный массив из 10 элементов встроенного типа ***int***) конструктором класса ***Array1D*** выделяется свободная память, адрес которой запоминается в указателе ***pointer*** объекта ***x***. Когда объект ***x*** используется для инициализации объекта ***y***, то вызывается конструктор копирования, для массива ***y*** выделяется свободная память, адрес которой запоминается в указателе ***pointer*** объекта ***y***, и содержимое массива ***x*** копируется в массив ***y***. В этом случае указатели ***pointer*** обоих объектов указывают на разные области памяти, в противном случае, если бы использовался конструктор копирования по умолчанию, то эти указатели указывали бы на одну и ту же область памяти.

Компонентные данные и компонентные функции

Статические компоненты класса

Как уже отмечалось, каждый вновь создаваемый объект класса имеет свою собственную копию компонентных данных. Чтобы эти компоненты класса были в единственном экземпляре и не тиражировались при определении каждого нового объекта класса, они должны быть объявлены в классе как статические (***static***):

static имя_типа имя_статического_компонента_данных;

Как видим, статический компонент данных является частью класса, но не является частью объекта этого класса. Все объекты класса, в котором были объявлены статические компоненты данных, так и объекты его производных классов, теперь могут совместно использовать эти общие для них компоненты.

Статические компоненты данных класса размещаются в памяти отдельно от его нестатических компонентов, причем память статическим компонентам выделяется только после их определения вне определения класса.

Для внешнего определения статических компонентных данных можно воспользоваться следующими конструкциями:

имя_типа имя_класса*::*имя_статического_компонента_данных;

или

***имя_типа имя_класса*::*имя_статического_компонента_данных* =**
инициализатор;

или

имя_типа имя_класса*::*имя_статического_компонента_данных
(*инициализатор*);

Эти конструкции должны быть размещены в глобальной области (***global scope***) после определения класса.

Как видим, в отличие от обычных компонентных данных статические компоненты класса необходимо еще дополнительно определять вне определения класса. Заметим, что по умолчанию значением статического компонента данных встроенного типа является ***0***, преобразованный в соответствующий тип.

Напомним здесь, что конструкторы статических компонентных данных класса в единице трансляции вызываются до вызова функции ***main*()** в порядке их определений. После выхода из функции ***main*()** будут вызваны деструкторы для каждого такого объекта в порядке, обратном вызовам их конструкторов.

Итак, статическому компоненту данных память выделяется только после его определения вне класса, и лишь после этого он становится доступным. Статические компоненты данных после определения можно использовать в программе еще до определения объектов данного класса. По сути, статический компонент данных – это просто глобальная переменная, область видимости которой ограничена классом, в котором она и была объявлена. Основной смысл поддержки в C++ статических компонентов данных класса состоит в том, что теперь отпадает необходимость в использовании глобальных переменных. При этом не следует забывать о том, что использование классов наряду с глобальными переменными почти всегда нарушает принцип инкапсуляции. Заметим также, что в некоторых случаях класс используется просто как область действия, в которую глобальные имена помещаются под видом статических компонентов, чтобы не засорять глобальное пространство имен.

Для обращения к статическим компонентам данных используются следующие имена:

квалификация без указания имени объекта

имя_класса*::*имя_статического_компонента_данных

квалификация с указанием имени объекта

имя_объекта* .*имя_класса*::*имя_статического_компонента_данных

уточненное имя

имя_объекта* .*имя_статического_компонента_данных

Другой способ доступа к статическим компонентам данных предусматривает явное использование указателя на объект класса:

либо с помощью оператора выбора члена **–>**

имя_указателя_на_объект_класса*–>*имя_статического_компонента_данных

либо с помощью оператора разыменования ***** и оператора выбора члена **.**

(имя_указателя_на_объект_класса*) .*имя_статического_компонента_данных***

Так как на статические компоненты класса распространяются правила статуса доступа (объявление статического компонента можно поместить как в закрытом, так и в открытом разделах определения класса), то для обращения к ним извне, как правило, используются общедоступные статические компонентные функции.

Формат объявления статической компонентной функции:

static имя_типа имя_статической_компонентной_функции
(*список_формальных_параметров_функции*);

Формат определения статической компонентной функции:

static имя_типа имя_статической_компонентной_функции
(*список_формальных_параметров_функции*) { *тело_функции* }

Формат внешнего определения статической компонентной функции:

имя_типа имя_класса*::*имя_статической_компонентной_функции
(*список_формальных_параметров_функции*) { *тело_функции* }

Статическая компонентная функция сохраняет все основные особенности обычных (нестатических) компонентных функций, т.е. к ней можно обращаться, используя имя уже существующего объекта класса либо указатель на такой объект.

Дополнительно статическую компонентную функцию можно вызвать, используя квалификацию без указания имени объекта:

имя_класса*::*имя_статической_компонентной_функции
(*список_аргументов_вызова*)

С помощью такой квалификации статические компонентные функции можно вызывать до определения конкретных объектов класса, так как для доступа к статическим компонентам данных этим функциям вовсе не требуется, чтобы они вызывались для конкретного объекта класса.

Статическая компонентная функция непосредственно может ссылаться на статические компоненты (данные и функции) только своего класса. Так как статическую компонентную функцию можно вызвать без ссылки на объект класса, то ей не передается указатель ***this***, который всегда неявно передается нестатическим компонентным функциям класса при их вызове для указания того объекта, для которого они и вызывались. Далее будет сказано об особой роли этой переменной, которой невозможно что-нибудь присвоить и адрес которой недоступен.

Если локальная статическая переменная позволяет обычной функции “помнить о прошлом” этой переменной, то это справедливо и по отношению к статическим компонентам данных класса – они тоже “помнят о своем прошлом”. Поэтому их применение может быть весьма полезным, например, для хранения информации о количестве объектов класса, существующих в каждый конкретный момент времени работы программы, или, например, для координации доступа к разделяемым ресурсам, таким как дисковые файлы или принтер. Страуструп настоятельно рекомендует пользоваться спецификатором ***static*** только внутри функций и классов для объявления соответственно локальных и нелокальных статических объектов.

Например, в следующем определении класса ***Complex*** введем два статических компонента класса – компонент данных ***counter*** для подсчета количества созданных в программе объектов класса ***Complex*** (конструктор будет осуществлять операцию инкрементирования, а деструктор – операцию декрементирования) и компонентную функцию ***get_counterValue*()**, которая будет возвращать значение счетчика ***counter***:

```
// Пример 10
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  static int counter;     // объявление статического компонента
```

```
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
    ++counter;
  }
  // Деструктор объектов класса
  ~Complex()
  {
    --counter;
  }
  // Доступ к счетчику
  static int get_counterValue()
  {
    return counter;
  }
};
int Complex::counter;     // определение статического компонента
int main()
{
  cout << "sizeof(Complex) = " << sizeof(Complex) << endl;
  cout << "How many objects? " << Complex::counter << ':'
       << Complex::get_counterValue() << endl;
  Complex x1(-1, 5);
  cout << "sizeof x1 = " << sizeof x1 << endl;
  cout << "How many objects? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << x1.get_counterValue() << endl;
  Complex x2(10, 7);
  cout << "sizeof x2 = " << sizeof x2 << endl;
  cout << "How many objects? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << x2.get_counterValue() << endl;
  Complex* pointer = new Complex;
  cout << "How many objects? " << Complex::counter << ':'
       << Complex::get_counterValue() << ':'
       << pointer->get_counterValue() << ':'
       << (*pointer).get_counterValue() << endl;
  delete pointer;
  cout << "How many objects? " << Complex::counter << ':'
       << Complex::get_counterValue() << endl;
  return 0;
}
```

Результат работы программы:

```
sizeof(Complex) = 16
How many objects? 0:0
sizeof x1 = 16
How many objects? 1:1:1
sizeof x2 = 16
How many objects? 2:2:2
How many objects? 3:3:3:3
How many objects? 2:2
```

Указатели на компоненты класса

В языке C++ наряду с операторами выбора члена **.** и **->** есть еще два специфичных оператора выбора члена **.*** и **->***, которые предназначены для работы с указателями на нестатические компоненты класса. Прежде чем объяснить их особенности, отметим здесь, что указатель на компонент класса не является обычным указателем, унаследованным C++ от языка C. Напомним также, что все перечисленные здесь операторы выбора члена часто называют селекторами членов класса.

Обычный указатель предназначен для адресации объекта программы, т.е. участка памяти, связанного, например, с переменной или функцией. Указатель на компонент класса не может адресовать никакого участка памяти, так как память выделяется не классу, а объектам этого класса при их создании. Указатель на компонент класса можно рассматривать как позицию этого компонента в объекте класса, но, конечно же, компилятор принимает в расчет различия между компонентами класса. Можно считать, что указатель на компонент класса в некотором смысле соответствует смещению в структуре или индексу в массиве. Таким образом, указатель на компонент класса при определении не адресует никакого конкретного объекта. Указатель на компонент класса является значением, всего лишь идентифицирующим этот компонент. Мало того, даже после инициализации значение этого указателя остается неопределенным до тех пор, пока не будут выполнены операции обращения к компонентам класса с помощью операторов выбора члена **.*** или **->***.

Итак, результатом выполнения операторов выбора члена **.*** и **->*** является либо нестатический компонент данных класса, либо его нестатическая компонентная функция.

Указатели на компоненты класса по-разному определяются для компонентных данных и компонентных функций. Отметим здесь, что компоненты класса при этом обязательно должны быть общедоступными (***public***).

Формат определения указателя на нестатические компоненты данных класса:

***имя_типа*(*имя_класса*::**имя_указателя_на_компонент_данных*);**

В определение указателя можно включить его явную инициализацию, используя адрес компонента данных:

***имя_типа*(*имя_класса*::**имя_указателя_на_компонент_данных*) =**
&*имя_класса*::*имя_компонента_данных*;

Формат определения указателя на нестатические компонентные функции:

***имя_типа*(*имя_класса*::**имя_указателя_на_компонентную_функцию*)**
(*список_формальных_параметров_функции*);

В определение указателя можно включить его явную инициализацию, используя адрес компонентной функции:

***имя_типа*(*имя_класса*::**имя_указателя_на_компонентную_функцию*)**
(*список_формальных_параметров_функции*) =
&*имя_класса*::*имя_компонентной_функции;*

Отметим, что здесь, как и в случае определения указателей на обычные функции, вовсе необязательно пользоваться оператором **&** для получения адреса компонентной функции. Например:

***имя_типа*(*имя_класса*::**имя_указателя_на_компонентную_функцию*)**
(*список_формальных_параметров_функции*) =
имя_класса*::*имя_компонентной_функции;

Отметим также, что указатели на статические компоненты класса похожи на обычные (встроенные) указатели на переменную или функцию, так как статические компоненты класса не ассоциируются с конкретным объектом. Отличие лишь в том, что для их инициализации требуется полная квалификация инициализатора.

Например:

имя_типа* имя_указателя_на_статический_компонент_данных =
&*имя_класса*::*имя_статического_компонента_данных;*

или

***имя_типа*(**имя_указателя_на_статическую_компонентную_функцию*)**
(*список_формальных_параметров_функции*) =
&*имя_класса*::*имя_статической_компонентной_функции;*

или

***имя_типа*(**имя_указателя_на_статическую_компонентную_функцию*)**
(*список_формальных_параметров_функции*) =
имя_класса*::*имя_статической_компонентной_функции;

Естественно, если известно, к какому компоненту классу будут производиться обращения, то нет необходимости в указателях на эти компоненты – к ним можно обратиться и непосредственно. Например, указатели на компонентные функции, так же как и указатели на обычные функции, часто используются тогда, когда возникает необходимость сослаться на функцию, имя которой заранее неизвестно. При этом среди компонентных функций класса здесь особый интерес представляют именно виртуальные функции. Далее будет сказано, что традиционной реализацией вызова виртуальных функций является косвенный вызов функции. Так, для каждого класса с виртуальными функциями компилятор строит свою таблицу указателей на его виртуальные функции (таблица виртуальных функций). Вызов виртуальной функции выполняется по ее индексу в таблице, соответствующей требуемому классу.

Так как указатель на виртуальную компонентную функцию является в некотором смысле смещением, он не зависит от расположения объекта в памяти. Поэтому указатель на виртуальную функцию можно безопасно передавать из одного адресного пространства в другое, при условии, что в них обоих объект размещен одинаковым образом. Напомним, что указатели на невиртуальные компонентные функции, также как и указатели на обычные функции, нельзя передавать в другое адресное пространство.

Указатель на компонент класса, как и обычный указатель, можно использовать в качестве аргумента вызова функции.

Если определены указатели на нестатические компоненты класса, то доступ к компонентам конкретных объектов можно получить с помощью оператора выбора члена класса **.***:

имя_объекта .*имя_указателя_на_компонент_данных

(*имя_объекта .*имя_указателя_на_компонентную_функцию*)

(*список_аргументов_вызова_функции*)

Первым операндом кроме имени конкретного объекта здесь также может быть и ссылка на объект. Второй операнд – указатель на компонент класса.

Если определены указатели на нестатические компоненты класса и определен указатель на объект того же класса, то доступ к компонентам конкретных объектов можно получить с помощью оператора выбора члена класса **->***:

имя_указателя_на_объект->*имя_указателя_на_компонент_данных

(*имя_указателя_на_объект->*имя_указателя_на_компонентную_функцию*)

(*список_аргументов_вызова_функции*)

Первым операндом здесь должен быть указатель на объект класса, значение которого – адрес объекта класса. Второй операнд – указатель на компонент класса.

В качестве примеров работы с указателями на нестатические компоненты класса рассмотрим класс ***Complex***:

```
// Пример 11
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
};
int main()
{
  double(Complex::*dataMemberPointer) = &Complex::re;
  void(Complex::*memberFunctionPointer)() = &Complex::print;
```

```
  Complex x1(-1, 5);
  Complex* objectPointer = &x1;
  Complex x2;
  x1.print();                                          // (-1, 5)
  x2.print();                                          // (0, 0)
  x1.*dataMemberPointer = 0;
  x2.*dataMemberPointer = -1;
  dataMemberPointer = &Complex::im;
  objectPointer->*dataMemberPointer = 0;
  objectPointer = &x2;
  objectPointer->*dataMemberPointer = 5;
  (x1.*memberFunctionPointer)();                       // (0, 0)
  (objectPointer->*memberFunctionPointer)();           // (-1, 5)
  return 0;
}
```

Если по какой-либо причине объявление указателя на компонент класса в стиле C слишком неудобно, следующие два примера показывают, как можно воспользоваться ***typedef*** для задания синонима типа:

```
// Пример 12
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
typedef double(Complex::*dataPointer);
typedef void(Complex::*functionPointer)(Complex);
int main()
{
  dataPointer dataMemberPointer = &Complex::re;
  functionPointer memberFunctionPointer = &Complex::print;
  Complex x1(-1, 5);
```

```
  Complex* objectPointer = &x1;
  Complex x2;
  x1.print(x1);                                          // (-1, 5)
  x2.print(x2);                                          // (0, 0)
  x1.*dataMemberPointer = 0;
  x2.*dataMemberPointer = -1;
  dataMemberPointer = &Complex::im;
  objectPointer->*dataMemberPointer = 0;
  objectPointer = &x2;
  objectPointer->*dataMemberPointer = 5;
  (x1.*memberFunctionPointer)(x1);                       // (0, 0)
  (objectPointer->*memberFunctionPointer)(x2);           // (-1, 5)
  return 0;
}

// Пример 13
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  // Ссылка на вещественную часть комплексного числа
  double& get_re()
  {
    return re;
  }
  // Ссылка на мнимую часть комплексного числа
  double& get_im()
  {
```

```
    return im;
  }
};
typedef double&(Complex::*functionPointer)();
int main()
{
  Complex x1(-1, 5);
  Complex* objectPointer = &x1;
  Complex x2(10, 7);
  objectPointer->print(*objectPointer);
  x2.print(x2);
  objectPointer->print(objectPointer->add(*objectPointer, x2));
  functionPointer memberFunctionPointer = &Complex::get_re;
  (x1.*memberFunctionPointer)() += 10;
  memberFunctionPointer = &Complex::get_im;
  (objectPointer->*memberFunctionPointer)() += 7;
  x1.print(x1);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
(9, 12)
```

Определение компонентных функций

Компонентные функции, которые определяются в пределах определения класса, являются неявно встроенными (***inline function***). Как правило, в классе должны определяться только небольшие по размеру и часто используемые компонентные функции. Так как не каждая компонентная функция может быть встроенной, то в классе такую функцию можно только объявить, а ее определение должно быть представлено после класса. Заметим, что наряду с внутренним возможно также и внешнее определение встраиваемых компонентных функций после класса, в котором были представлены их прототипы. Тогда спецификатор ***inline*** следует явно добавить либо в прототип функции, либо в ее внешнее определение, либо сделать то и другое одновременно. Однако компилятор может и проигнорировать такой совет по поводу встраивания. Во всех случаях внешнего определения компонентных функций, чтобы указать, к какому именно классу они относятся, требуется их полная квалификация.

Формат внешнего определения встраиваемой компонентной функции:

inline имя_типа имя_класса*::*имя_компонентной_функции
(*список_формальных_параметров_функции*) { *тело_функции* }

Формат внешнего определения невстраиваемой компонентной функции:

имя_типа имя_класса*::*имя_компонентной_функции
(*список_формальных_параметров_функции*) { *тело_функции* }

Внешнее определение компонентных функций в противоположность встроенному позволяет модифицировать их, не изменяя при этом текста объявления класса. Так как встроенные компонентные функции должны быть определены в каждом исходном файле, где они вызываются, то обычно встраиваемые компонентные функции определяют в заголовочных файлах вместе со своими классами.

Например, представим несколько возможных способов определения встраиваемых компонентных функций для класса ***Complex***:

```
// Пример 14
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  //   теперь уже невстраиваемая функция
  Complex(double, double);
  // Сложение комплексных чисел
  //   возможно, что еще будет встраиваемой функцией
  Complex add(Complex, Complex);
  // Визуализация комплексного числа
  //   встроенная функция
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
// Конструктор объектов класса
Complex::Complex(double r = 0.0, double i = 0.0)
{
  re = r;
  im = i;
}
// Сложение комплексных чисел
inline Complex Complex::add(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
```

```
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  x1.print(x1.add(x1, x2));              // (9, 12)
  return 0;
}
```

В ряде случаев пользователь класса может объявить какие-либо компонентные функции константными. С одной стороны, такие функции теперь не могут изменять компонентные данные класса. Это связано с тем, что в константной компонентной функции класса ***имя_класса*** указатель ***this***, который неявно передается любой нестатической компонентной функции, имеет тип ***const имя_класса**,*** что как раз и предотвращает любое изменение состояния объектов этого класса. С другой стороны, константные компонентные функции можно вызывать как для константных, так и для неконстантных объектов класса, в то время как неконстантную компонентную функцию можно вызвать только для неконстантных объектов. Напомним, что объект называют константным, если при его определении используется квалификатор ***const***.

Формат объявления константной компонентной функции:

имя_типа имя_компонентной_функции
(*список_формальных_параметров_функции*)*const;*

Формат определения встроенной константной компонентной функции:

имя_типа имя_компонентной_функции
(*список_формальных_параметров_функции*)*const* { *тело_функции* }

Формат внешнего определения встраиваемой константной компонентной функции:

inline имя_типа имя_класса*::*имя_компонентной_функции
(*список_формальных_параметров_функции*)*const* { *тело_функции* }

Здесь суффикс ***const*** является частью типа компонентной функции.

Например, представим класс ***Complex***, в котором определим две константные компонентные функции ***add*()** и ***print*()**:

```
// Пример 15
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные данные - все общедоступные (public)
  double re;
  double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Сложение комплексных чисел
  Complex add(Complex, Complex)const;
```

```
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)
  {
    re = c.re;
    im = c.im;
  }
  // Визуализация комплексного числа
  void print(Complex c)const
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
// Сложение комплексных чисел
inline Complex Complex::add(Complex a, Complex b)const
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}
int main()
{
  const Complex x1(-1, 5);
  const Complex x2(10, 7);
  Complex y;
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  y.print(y);                            // (0, 0)
  x1.print(x1.add(x1, x2));              // (9, 12)
  y.assign(y.add(x1, x2));
  y.print(y);                            // (9, 12)
  return 0;
}
```

Заметим, что иногда компонентная функция с логической точки зрения является константной, но, тем не менее, ей требуется модифицировать некоторые компоненты данных класса. Существуют приемы, основанные, например, на использовании оператора приведения типа ***const_cast***, которые позволяют выполнить такое изменение состояния компонента данных объекта (“снятие” ***const*** путем приведения, т.е. аннулирование действия квалификатора ***const***), однако их результат не всегда гарантирован, если объект был объявлен константным. Напомним здесь, что нотация ***const_cast<имя_типа>(выражение)*** призвана заменить ***(имя_типа)выражение*** для преобразований, которым нужно получить доступ к данным с квалификатором ***const*** или ***volatile***. В ***const_cast<имя_типа>(выражение)*** тип параметра ***имя_типа*** должен совпадать с типом аргумента ***выражение*** во всем, кроме квалификаторов ***const*** или ***volatile***. Результатом этой операции будет то же самое значение, что и у аргумента ***выражение***, только его типом станет ***имя_типа***.

Другим решением проблемы “снятия константности” у компонентных данных класса является объявление их с квалификатором хранения ***mutable***, который указывает, что эти компоненты должны храниться таким способом, чтобы допускалась их модификация, даже если они являются компонентами константного объекта. Другими словами, квалификатор ***mutable*** означает: “ни при каких условиях не является константным”. Объявление имен с квалификатором ***mutable*** наиболее приемлемо, когда только часть представления класса может быть модифицирована, в то время как с логической точки зрения объект этого класса остается константным.

Чтобы проиллюстрировать использование квалификатора хранения ***mutable*** для компонентных данных класса ***Complex***, его компонентная функция ***assign*()** и его объект ***y*** теперь тоже должны стать константными:

```
// Пример 16
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные данные - все общедоступные (public)
  mutable double re;
  mutable double im;
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    re = r;
    im = i;
  }
  // Сложение комплексных чисел
  Complex add(Complex a, Complex b)const
  {
    Complex temporary;
    temporary.re = a.re + b.re;
    temporary.im = a.im + b.im;
    return temporary;
  }
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)const
  {
    re = c.re;
    im = c.im;
  }
  // Визуализация комплексного числа
  void print(Complex c)const
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
```

```
int main()
{
  const Complex x1(-1, 5);
  const Complex x2(10, 7);
  const Complex y;
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  y.print(y);                            // (0, 0)
  x1.print(x1.add(x1, x2));              // (9, 12)
  y.assign(y.add(x1, x2));
  y.print(y);                            // (9, 12)
  return 0;
}
```

Хотя конструкторы и деструкторы не являются константными компонентными функциями, они все же могут вызываться и для константных объектов класса. Объект становится константным сразу после того, как конструктор проинициализирует его, и перестает быть таковым, как только вызывается деструктор. Таким образом, объект класса с квалификатором ***const*** трактуется как константный с момента завершения работы конструктора и до вызова деструктора.

Компонентную функцию можно также объявить с квалификатором ***volatile***:

имя_типа имя_компонентной_функции
(список_формальных_параметров_функции)volatile;

Такие компонентные функции вызываются только для тех объектов класса, которые были объявлены с тем же квалификатором.

Указатель this

Когда вызывается какая-либо нестатическая компонентная функция для обработки компонентных данных конкретного объекта класса, этой функции неявно передается специальный константный указатель на этот объект – ***this***:

имя_класса* const this = &имя_объекта;

Указатель ***this*** является дополнительным (скрытым) аргументом вызова каждой нестатической компонентной функции, т.е. эти функции знают, для каких объектов класса они были вызваны, поэтому могут явно на них ссылаться. В нестатической компонентной функции класса ***имя_класса*** указатель ***this*** имеет тип ***имя_класса****, а в константной нестатической компонентной функции этого класса – соответственно тип ***const имя_класса****. Однако указатель ***this*** – это не обычная переменная, так как невозможно получить ее адрес или присвоить ей что-нибудь.

Объект класса, который адресуется указателем ***this***, становится доступным внутри всех принадлежащих классу нестатических компонентных функций, поэтому при обращении к нестатическим компонентам класса можно было бы везде явно использовать этот указатель (заметим, что в практике программирования такая ситуация представляется весьма маловероятной).

Например, приведем такое определение конструктора класса ***Complex*** и двух его компонентных функций для доступа к закрытым компонентам класса:

```
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0)
  {
    this->re = r;
    this->im = i;
  }
  // Ссылка на вещественную часть комплексного числа
  double& get_re()
  {
    return this->re;
  }
  // Ссылка на мнимую часть комплексного числа
  double& get_im()
  {
    return this->im;
  }
```

Как видим, в таком использовании указателя ***this*** нет никаких преимуществ, так как компонентные данные внутри компонентных функций класса и так доступны с помощью своих имен. В большинстве же случаев использование указателя ***this*** является неявным, в частности, каждое обращение к нестатическому компоненту внутри класса неявно использует указатель ***this*** для доступа к компоненту данных соответствующего объекта.

В некоторых случаях явное использование указателя ***this*** полезно, а иногда просто незаменимо, например, когда внутри принадлежащей классу компонентной функции необходимо явно задать адрес объекта, для которого она была вызвана. Конечно же, такой функции можно передать ссылку или указатель на конкретный объект, однако гораздо проще явно использовать указатель ***this***.

Вернемся еще раз к решению проблемы “снятия константности” у компонентных данных класса, основанное на явном преобразовании типов (так называемое приведение типов) – **(*имя_типа*)*выражение***. Например, представим класс ***Complex***, в котором его константная компонентная функция ***assign*()** осуществляет “снятие” ***const*** путем приведения типа **(*Complex**)*this***:

```
// Пример 17
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
```

```
  // Связывание комплексного числа с другим комплексным числом
  void assign(Complex c)const
  {
    ((Complex*)this)->re = c.re;
    ((Complex*)this)->im = c.im;
  }
  // Визуализация комплексного числа
  void print(Complex c)const
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
};
int main()
{
  Complex x1(-1, 5);
  x1.print(x1);                          // (-1, 5)
  const Complex x2(10, 7);
  x2.print(x2);                          // (10, 7)
  x2.assign(x1);
  x2.print(x2);                          // (-1, 5)
  return 0;
}
```

Такого же “снятия” ***const*** можно добиться, например, и с помощью оператора приведения типа ***const_cast***, если в классе ***Complex*** его константную компонентную функцию ***assign*****()** определить так:

```
  void assign(Complex c)const
  {
    (const_cast<Complex*>(this))->re = c.re;
    (const_cast<Complex*>(this))->im = c.im;
  }
```

Ярким примером широко распространенного явного использования указателя ***this*** являются операции со связным списком из стандартной библиотеки C++.

Рассмотрим, например, реализацию двухсвязного списка в виде статической структуры данных. Приведем определение класса ***List***, где его компонентной функции ***append*****()** требуется явное использование указателя ***this*** при выполнении операции добавления элемента списка:

```
// Пример 18
// C++ Абстрактный тип данных - двухсвязный список
#include <iostream>
using namespace std;
struct List {
  // Компонентные данные – все общедоступные (public)
  List* predecessor;  // предыдущий элемент списка
  int data;           // данные элемента списка
  List* successor;    // следующий элемент списка
```

```
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  List(int item = 0)
  {
    predecessor = 0;
    data = item;
    successor = 0;
  }
  // Присоединение элемента к списку
  void append(List* pointer)
  {
    pointer->predecessor = this;
    successor = pointer;
  }
  // Визуализация элемента списка
  void print(List* pointer)
  {
    cout << pointer->data << endl;
  }
};
int main()
{
  List x1(1);           // первый элемент списка
  cout << &x1 << '\t';
  x1.print(&x1);
  List x2(2);           // второй элемент списка
  cout << &x2 << '\t';
  x2.print(&x2);
  x1.append(&x2);       // связывание первых двух элементов списка
  List x3(3);           // третий элемент списка
  cout << &x3 << '\t';
  x3.print(&x3);
  x2.append(&x3);       // присоединение элемента к списку
  cout << x1.predecessor << endl;
  cout << x1.successor << endl;
  cout << x1.successor->successor << endl;
  cout << x1.successor->successor->successor << endl;
  cout << x2.predecessor << endl;
  cout << x2.predecessor->predecessor << endl;
  cout << x2.successor << endl;
  cout << x3.predecessor << endl;
  cout << x3.predecessor->predecessor << endl;
  cout << x3.predecessor->predecessor->predecessor << endl;
  cout << x3.successor << endl;
  return 0;
}
```

Результат работы программы:

```
0x8fb00ff6      1
0x8fb00fec      2
0x8fb00fe2      3
0x00000000
0x8fb00fec
0x8fb00fe2
0x00000000
0x8fb00ff6
0x00000000
0x8fb00fe2
0x8fb00fec
0x8fb00ff6
0x00000000
0x00000000
```

Как видим, чтобы присоединить элемент к двухсвязному списку, необходимо обновить объекты, на которые указывают указатели ***predecessor***, ***this*** и ***successor*** (предыдущий, текущий и следующий элементы списка). Напомним, что каждый элемент двухсвязного списка (кроме первого и последнего) должен быть связан со своим предыдущим и следующим элементом соответственно с помощью указателей ***predecessor*** и ***successor***.

Например, явное использование указателя ***this*** незаменимо при выполнении последовательности операций с одним и тем же объектом класса ***Complex***:

```
// Пример 19
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex& add(Complex a, Complex b)
  {
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
  // Визуализация комплексного числа
  void print()
  {
```

```
    cout << '(' << re << ", " << im << ')' << endl;
  }
};
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print();                              // (-1, 5)
  x2.print();                              // (10, 7)
  x1.add(x1, x2).print();                  // (9, 12)
  return 0;
}
```

Здесь выражение ***this** означает объект, для которого была вызвана компонентная функция ***add*()**, т.е. объект ***x1***. Как видим, с помощью явного использования указателя ***this*** можно легко организовать выполнение цепочки операций, например, сложение и визуализацию, если компонентная функция ***add*()** будет возвращать ссылку на объект, для которого она была вызвана.

В дальнейшем будут приведены и другие примеры явного использования указателя ***this***.

Друзья класса

Механизм управления доступом позволяет выделять общедоступные или открытые (***public***), собственные или закрытые (***private***) и защищенные (***protected***) компоненты класса. Защищенные компоненты доступны внутри класса и в его производных (порожденных) классах. Собственные компоненты локализованы в классе, потому они доступны только внутри класса и недоступны извне. Общедоступные компоненты доступны везде, с их помощью реализуется открытый интерфейс класса с любыми частями программы. Расширить этот интерфейс, т.е. предоставить доступ к компонентам как закрытого, так и защищенного разделов определения класса позволяют его друзья – ***дружественные функции*** (или ***функции-друзья***) и ***дружественные классы*** (или ***классы-друзья***).

Дружественные функции – это такие функции, которые, не являясь компонентами класса, предоставившего дружбу, имеют доступ к его собственным и защищенным компонентам, т.е. функции-друзья являются частью интерфейса класса в той же мере, в какой ею являются его компонентные функции. Для получения прав друга функции-друзья должны быть объявлены в теле класса, который предоставил им дружбу, со спецификатором ***friend***, причем объявление функции-друга можно поместить как в закрытом, так и в открытом разделах определения класса. При этом сами функции-друзья должны быть либо явно определены в охватывающей области видимости, либо иметь аргументы вызова класса, предоставившего им дружбу.

Отметим здесь, что поиск любой функции, в том числе и функции-друга, обычно осуществляется сначала в области видимости вызова, а затем – в пространствах имен каждого аргумента вызова (включая класс каждого аргумента и его базовые классы). В противном случае функцию-друга вызвать нельзя.

Дружественные классы – это такие классы, все компонентные функции которых являются друзьями другого класса, предоставившего дружбу, т.е. компоненты этого класса становятся доступными и всем компонентным функциям классов-друзей. Для получения прав друга классы-друзья, как и функции-друзья, должны быть объявлены в теле класса, предоставившего им дружбу, со спецификатором ***friend***. При этом сами классы-друзья должны быть либо предварительно объявлены в охватывающей области видимости, либо определены в области видимости, непосредственно охватывающей класс, предоставивший им дружбу, т.е. в одном пространстве имен. Обычно классы-друзья используются тогда, когда они не связаны отношением наследования с классом, предоставившим им дружбу.

Очевидно, что как функции-друзья, так и классы-друзья должны использоваться только для отражения тесно связанных концепций. Если для пользователя класса при этом существует выбор между альтернативными способами представления операций с его объектами (в качестве компонентных функций или в качестве друзей), то и для пользователя класса-друга так же существует выбор между способами реализации самого этого класса (в качестве компонента другого класса или в качестве друга).

Здесь следует также отметить, что в пользу существования дружественных функций кроме довода, связанного с получением доступа как к закрытому, так и к защищенному разделам определения класса, предоставившего дружбу, имеется еще один, связанный с перегрузкой стандартных операторов (**>>** и **<<**) для создания специальных операторных функций ввода-вывода, называемых соответственно ***инсертерами*** (от слова ***inserter***) и ***экстракторами*** (от слова ***extractor***).

Например, представим класс ***Complex***, в закрытом разделе определения которого будут объявлены только его компонентные данные, для доступа к которым он объявляет своим другом глобальную функцию ***create*()**:

```
// Пример 20
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  // Создание объектов в свободной памяти
  friend Complex* create(double, double);
};
```

```
// Создание объектов в свободной памяти
Complex* create(double r = 0.0, double i = 0.0)
{
  Complex* pointer = new Complex(r, i);
  cout << pointer << endl;
  cout << '(' << pointer->re << ", " << pointer->im << ")\n";
  pointer->print();
  return pointer;
}
int main()
{
  Complex x1(-1, 5);
  Complex* p = &x1;
  cout << p << endl;
  p->print();
  p = create();
  cout << p << endl;
  p->print();
  delete p;
  p = create(10, 7);
  cout << p << endl;
  p->print();
  delete p;
  return 0;
}
```

Результат работы программы:

```
0x8fb10fec
(-1, 5)
0x91420004
(0, 0)
(0, 0)
0x91420004
(0, 0)
0x91420004
(10, 7)
(10, 7)
0x91420004
(10, 7)
```

В случае, когда у класса нет открытого интерфейса, его функции-друзья могут, например, помочь построить этот интерфейс. Очевидно, что один из друзей должен при этом взять на себя роль “конструктора”, неявно вызывая настоящий конструктор объектов. Представим класс ***Complex***, в закрытом разделе определения которого будут объявлены не только его компонентные данные, но и все его компонентные

функции. Теперь, чтобы построить открытый интерфейс, объявим в классе ***Complex*** двух его друзей, например, глобальные функции ***create*()** и ***add*()**, каждая из которых помимо своей основной операции выполняет еще и операцию визуализации:

```
// Пример 21
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex {
  // Закрытое представление класса (private)
  double re;
  double im;
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  // Создание объектов в свободной памяти
  friend Complex* create(double, double);
  // Сложение комплексных чисел
  friend Complex* add(Complex*, Complex*);
};
// Создание объектов в свободной памяти
Complex* create(double r = 0.0, double i = 0.0)
{
  Complex* pointer = new Complex(r, i);
  pointer->print();
  return pointer;
}
// Сложение комплексных чисел
Complex* add(Complex* p, Complex* q)
{
  Complex* pointer = new Complex;
  pointer->re = p->re + q->re;
  pointer->im = p->im + q->im;
  pointer->print();
  return pointer;
}
int main()
{
  Complex* p = create(-1, 5);
  Complex* q = create(10, 7);
  Complex* r = add(p, q);
  delete p, q, r;
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

Далее будет сказано, как при помощи дружественной классу ***Complex*** операторной функции вставки можно полностью отказаться от компонентной функции ***print*()**, чтобы, например, для рассматриваемых здесь двух функций-друзей ***create*()** и ***add*()** исключить несвойственную им операцию визуализации.

В завершение этого обсуждения рассмотрим еще два примера для иллюстрации расширения открытого интерфейса класса, памятуя о стремлении к естественному поведению типа, т.е. новые операции должны быть так же естественны, как если бы они были компонентами класса. Сначала представим класс ***Complex***, в закрытом разделе определения которого будут объявлены только его компонентные данные, для доступа к которым он объявляет своим другом глобальную функцию ***add*()**:

```
// Пример 22
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  // Сложение комплексных чисел
  friend Complex add(Complex, Complex);
};
// Сложение комплексных чисел
Complex add(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}
int main()
{
```

```
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  x1.print(add(x1, x2));                 // (9, 12)
  return 0;
}
```

А теперь представим класс ***Array2D*** с внешним определением двух его компонентных функций (конструктора и деструктора) для реализации динамических двумерных массивов, при этом конструктору класса будет отведена роль, связанная лишь с выделением свободной памяти. В закрытом разделе определения класса, как и в предыдущем примере, будут объявлены только его компонентные данные, для доступа к которым он объявляет своими друзьями три глобальные функции – ***define*()** для инициализации элементов массива, ***print*()** для визуализации элементов массива и ***find_maximum*()** для поиска максимального элемента массива. Заметим при этом, что во избежание проблем, связанных с отсутствием в классе ***Array2D*** конструктора копирования, определяемого пользователем, функциям-друзьям ***define*()**, ***print*()** и ***find_maximum*()** в качестве аргумента вызова будет передаваться ссылка на массив.

Напомним, что указатель ***base*** – это указатель на указатель на ***double***, указатель ***base*** содержит начальный адрес массива указателей (***0, 1, …, columnSize–1***), а каждый указатель этого массива – начальный адрес строки, состоящей из элементов ***double*** (***0, 1, …, rowSize–1***):

```
// Пример 23
// C++ Абстрактный тип данных - динамический двумерный массив
#include <iostream>
#include <iomanip>
using namespace std;
class Array2D {
  // Компонентные данные - все собственные (private)
  double** base;
  int columnSize;
  int rowSize;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array2D(int, int);
  // Деструктор объектов класса
  ~Array2D();
  // Инициализация элементов массива
  friend void define(Array2D&);
  // Визуализация элементов массива
  friend void print(Array2D&);
  // Поиск максимального элемента массива
  friend double find_maximum(Array2D&);
};
```

```
// Конструктор объектов класса
Array2D::Array2D(int arrayColumnSize, int arrayRowSize)
{
  base = new double*[arrayColumnSize];
  for (int i = 0; i < arrayColumnSize; ++i)
    base[i] = new double[arrayRowSize];
  rowSize = arrayRowSize;
  columnSize = arrayColumnSize;
}
// Деструктор объектов класса
Array2D::~Array2D()
{
  for (int i = 0; i < columnSize; ++i)
    delete[] base[i];
  delete[] base;
}
// Инициализация элементов массива
void define(Array2D& a)
{
  cout << "A(" << a.columnSize << 'x' << a.rowSize << ')' << endl;
  for (int i = 0; i < a.columnSize; ++i)
    for (int j = 0; j < a.rowSize; ++j)
    {
      cout << "A[" << i << ',' << j << "]? ";
      cin >> a.base[i][j];
    }
}
// Визуализация элементов массива
void print(Array2D& a)
{
  for (int i = 0; i < a.columnSize; ++i)
  {
    for (int j = 0; j < a.rowSize; ++j)
      cout << setw(4) << a.base[i][j];
    cout << endl;
  }
}
// Поиск максимального элемента массива
double find_maximum(Array2D& a)
{
  double maximum = a.base[0][0];
  for (int i = 0; i < a.columnSize; ++i)
    for (int j = 0; j < a.rowSize; ++j)
      if (a.base[i][j] > maximum) maximum = a.base[i][j];
  return maximum;
}
int main()
{
```

```
  Array2D a(2, 2);
  define(a);
  print(a);
  cout << "maximum = " << find_maximum(a) << endl;
  return 0;
}
```

Результат работы программы:

```
A(2x2)
A[0,0]? 1
A[0,1]? 2
A[1,0]? 3
A[1,1]? 4
   1   2
   3   4
maximum = 4
```

Как видим, такое представление операций с объектами класса ***Complex*** и класса ***Array2D*** в виде функций-друзей выглядит здесь вполне естественной альтернативой. Главным здесь должно быть стремление к созданию эффективного интерфейса для реализации минимально возможного доступа к представлению класса.

Продолжая обсуждение, отметим и другие особенности функций-друзей:

- функция-друг при вызове не получает указателя ***this***;
- объекты класса, предоставившего дружбу, должны передаваться функции-другу только явно как аргументы вызова;
- функция-друг по отношению к классу, предоставившему ей дружбу, может быть компонентной функцией другого класса;
- функция-друг может быть дружественной по отношению к нескольким классам.

Вначале рассмотрим пример для иллюстрации расширения открытого интерфейса класса ***Complex***, предоставившего дружбу компонентной функции ***add*()** класса ***Tools*** для доступа к своим закрытым компонентным данным:

```
// Пример 24
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex;
class Tools {
  // Компонентные данные - все собственные (private)
  Complex& operand1;
  Complex& operand2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Tools(Complex& a, Complex& b) : operand1(a), operand2(b) {}
```

```
  // Сложение комплексных чисел
  Complex add();
};
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  // Сложение комплексных чисел
  friend Complex Tools::add();
};
Complex Tools::add()
{
  Complex temporary;
  temporary.re = operand1.re + operand2.re;
  temporary.im = operand1.im + operand2.im;
  return temporary;
}
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print();                              // (-1, 5)
  x2.print();                              // (10, 7)
  Tools t(x1, x2);
  Complex r = t.add();
  r.print();                               // (9, 12)
  return 0;
}
```

Как видим, чтобы компонентная функция ***add*()** класса ***Tools*** могла быть объявлена другом класса ***Complex***, требуется выполнить два условия: во-первых, класс ***Complex***, который предоставляет дружбу функции ***add*()**, должен быть определен только после определения класса ***Tools***, и во-вторых, в классе ***Tools*** его компонентную функцию ***add*()** можно лишь объявить, а ее внешнее определение должно быть только после определения класса ***Complex***.

Теперь рассмотрим пример для иллюстрации расширения открытого интерфейса классов ***Complex*** и ***Tools***, предоставивших дружбу глобальной функции ***print*()** для доступа к своим закрытым компонентным данным:

```
// Пример 25
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex;
class Tools {
  // Компонентные данные - все собственные (private)
  Complex& operand1;
  Complex& operand2;
  Complex& result;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Tools(Complex& a, Complex& b, Complex& c)
  : operand1(a), operand2(b), result(c) {}
  // Сложение комплексных чисел
  void add();
  // Визуализация комплексных чисел
  friend void print(Complex& a, Complex& b, Tools& c);
};
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  friend void Tools::add();
  // Визуализация комплексных чисел
  friend void print(Complex& a, Complex& b, Tools& c);
};
// Сложение комплексных чисел
void Tools::add()
{
  result.re = operand1.re + operand2.re;
  result.im = operand1.im + operand2.im;
}
// Визуализация комплексных чисел
void print(Complex& a, Complex& b, Tools& c)
{
  cout << '(' << a.re << ", " << a.im << ')' << endl;
  cout << '(' << b.re << ", " << b.im << ')' << endl;
  cout << '(' << c.result.re << ", " << c.result.im << ")\n";
}
int main()
{
```

```
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Complex r;
  Tools t(x1, x2, r);
  t.add();
  print(x1, x2, t);
  return 0;
}
```

Результат работы программы:

```
(-1, 5)
(10, 7)
(9, 12)
```

А теперь перейдем к дружественным классам. Начнем с примера, где закрытые компоненты класса ***Complex*** становятся доступными компонентным функциям класса ***Tools***, которого класс ***Complex*** объявляет своим другом:

```
// Пример 26
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex {
  // Закрытый раздел определения класса (private)
  double re;
  double im;
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  public:
  // Открытый раздел определения класса (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Класс-друг
  friend class Tools;
};
class Tools {
  // Компонентные данные - все собственные (private)
  Complex operand1;
  Complex operand2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Tools(Complex a, Complex b) : operand1(a), operand2(b) {}
```

```
  // Сложение комплексных чисел
  Complex add()
  {
    Complex temporary;
    temporary.re = operand1.re + operand2.re;
    temporary.im = operand1.im + operand2.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    c.print();
  }
};
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Tools t(x1, x2);
  t.print(x1);                          // (-1, 5)
  t.print(x2);                          // (10, 7)
  t.print(t.add());                       // (9, 12)
  return 0;
}
```

И завершим примером взаимной дружбы классов – класс ***Tools*** объявляется другом класса ***Complex***, а класс ***Complex***, в свою очередь, объявляется другом класса ***Tools***:

```
// Пример 27
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Tools;
class Complex {
  // Закрытый раздел определения класса (private)
  double re;
  double im;
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  public:
  // Открытый раздел определения класса (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex& add(Tools&);
```

```
  // Класс-друг
  friend class Tools;
};
class Tools {
  // Компонентные данные - все собственные (private)
  Complex operand1;
  Complex operand2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Tools(Complex a, Complex b) : operand1(a), operand2(b) {}
  // Визуализация комплексного числа
  void print(Complex c)
  {
    c.print();
  }
  // Класс-друг
  friend class Complex;
};
// Сложение комплексных чисел
Complex& Complex::add(Tools& t)
{
  re = t.operand1.re + t.operand2.re;
  im = t.operand1.im + t.operand2.im;
  return *this;
}
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Complex r;
  Tools t(x1, x2);
  t.print(x1);                          // (-1, 5)
  t.print(x2);                          // (10, 7)
  t.print(r.add(t));                    // (9, 12)
  return 0;
}
```

В дальнейшем будет сказано, как ведут себя дружественные классы, если они связаны отношением наследования.

Перегрузка стандартных операторов

Язык C++ позволяет распространить действие стандартных операторов для встроенных типов на новые типы данных, определяемые пользователем (механизм перегрузки или расширение действия стандартных операторов). Механизм перегрузки определяет новое поведение стандартных операторов, это возможно, если хотя бы один операнд для такого оператора является объектом некоторого класса.

Распространение действия стандартных операторов на типы, определяемые пользователем, позволяет реализовать более привычную и удобную форму записи для манипулирования объектами классов по сравнению с той, которая доступна с использованием только базовой функциональной формы записи.

Для распространения действия стандартного оператора на тип, определяемый пользователем, необходимо в классе определить специальную функцию, называемую ***операторной функцией*** (от слов ***operator function***).

Формат определения операторной функции:

***имя_класса operator оператор*(*список_формальных_параметров_функции*)**
{ *тело_функции* }

Определенный таким образом ***оператор*** называется перегруженным (от слов ***overloaded operator***).

Формат объявления операторной функции:

***имя_класса operator оператор*(*список_формальных_параметров_функции*);**

Имя операторной функции начинается с ключевого слова ***operator***, за которым непосредственно может следовать сам ***оператор***, например: ***operator+***. Заметим, что между лексемами ***operator*** и ***оператор***, как это принято и в других подобных случаях, могут располагаться пробельные разделители, например: ***operator +***.

Операторная функция, как и любая другая функция, может быть вызвана явно, однако такая форма вызова, как правило, не используется. Обычно операторная функция используется как оператор, что является просто сокращенной формой ее явного вызова. К тому же использование операторной функции в виде оператора сохраняет естественность поведения кода.

Например, приведем прототипы двух операторных функций для распространения действия бинарных операторов **+** и **–** на объекты класса ***Complex***:

```
Complex operator+(Complex, Complex);
Complex operator–(Complex);
```

Здесь следует отметить, что операторная функция с двумя параметрами типа класс может быть только либо функцией-другом этого класса, либо глобальной функцией, в то время как операторная функция с одним параметром типа класс может быть только нестатической компонентной функцией этого класса.

Теперь после определения для класса ***Complex*** смысла операций сложения и вычитания в тексте программы можно записывать комплексные выражения для этих операций в форме, близкой к общепринятой.

Например, приведем фрагмент кода:

```
Complex a(-1, 5);    // a.re = -1, a.im = 5
Complex b(10, 7);    // b.re = 10, b.im = 7
Complex c = a + b;   // c.re = 9, c.im = 12
a = b + c;           // a.re = 19, a.im = 19
a = a – b;           // a.re = 9, a.im = 12
```

Приведем фрагмент кода в случае явного вызова этих двух операторных функций:

```
Complex c = operator+(a, b);   // c.re = 9, c.im = 12
a = operator+(b, c);           // a.re = 19, a.im = 19
a = a.operator–(b);            // a.re = 9, a.im = 12
```

Пользователь может объявить операторные функции, определяющие смысл следующих стандартных операторов:

+	**–**	*****	**/**	**%**	**^**	**&**
|	**~**	**!**	**=**	**<**	**>**	**+=**
–=	***=**	**/=**	**%=**	**^=**	**&=**	**|=**
<<	**>>**	**>>=**	**<<=**	**==**	**!=**	**<=**
>=	**&&**	**||**	**++**	**--**	**–>***	**,**
–>	**[]**	**()**	***new***	***new*[]**	***delete***	***delete*[]**

Количество параметров операторной функции зависит как от самого оператора, так и от способа определения этой функции. Операторная функция определяет алгоритм выполнения операции, связанной с перегруженным стандартным оператором, когда он применяется к объектам класса.

Чтобы явная связь с классом была обеспечена, операторная функция должна быть:

- либо нестатическим компонентом класса;
- либо функцией-другом;
- либо глобальной функцией хотя бы с одним параметром типа класс (или ссылка на класс).

Бинарные и унарные операторы

Любой бинарный оператор **@** определяется для объектов класса ***a*** и ***b***:

- либо как нестатическая компонентная функция с одним параметром, что для выражения ***a@b*** означает вызов:
  a.operator@(b)
- либо как глобальная или дружественная функция с двумя параметрами, что для выражения ***a@b*** означает вызов:
  operator@(a, b)

Если определены обе операторные функции, то для выяснения того, какую (возможно и никакую) из них использовать, применяется механизм разрешения перегрузки, который заключается в нахождении наилучшего соответствия типов формальных параметров этих функций и их аргументов вызова.

Например, распространим действия трех бинарных операторов (**+**, **–** и *****) на объекты класса ***Complex***, определив ***operator+()*** как дружественную функцию класса, ***operator–()*** как нестатическую компонентную функцию, ***operator*()*** как глобальную функцию с параметрами типа класс:

```
// Пример 28
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
```

```
  // Ссылка на вещественную часть комплексного числа
  double& get_re()
  {
    return re;
  }
  // Ссылка на мнимую часть комплексного числа
  double& get_im()
  {
    return im;
  }
  // Сложение комплексных чисел
  friend Complex operator+(Complex, Complex);
  // Вычитание комплексных чисел
  Complex operator-(Complex c)
  {
    Complex temporary;
    temporary.re = re - c.re;
    temporary.im = im - c.im;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  private:
  // Компонентные данные - все собственные (private)
  double re;
  double im;
};
// Сложение комплексных чисел
Complex operator+(Complex a, Complex b)
{
  Complex temporary;
  temporary.re = a.re + b.re;
  temporary.im = a.im + b.im;
  return temporary;
}
// Умножение комплексных чисел
Complex operator*(Complex a, Complex b)
{
  Complex temporary;
  temporary.get_re() = a.get_re() * b.get_re() -
    a.get_im() * b.get_im();
  temporary.get_im() = a.get_re() * b.get_im() +
    b.get_re() * a.get_im();
  return temporary;
}
```

```
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  x1.print(x1 + x2);                     // (9, 12)
  x1.print(operator+(x1, x2));           // (9, 12)
  x2 = x1 + x2;
  x2.print(x2);                          // (9, 12)
  x1 = operator+(x1, x2);
  x1.print(x1);                          // (8, 17)
  x2.print(x2 + 10);                     // (19, 12)
  x1.print(-10 + x1);                    // (-2, 17)
  x1.print(x1 - x2);                     // (-1, 5)
  x1.print(x1.operator-(x2));            // (-1, 5)
  x1 = x1 - x2;
  x1.print(x1);                          // (-1, 5)
  x2.print(x2 - 10);                     // (-1, 12)
  x2 = x2.operator-(x1);
  x2.print(x2);                          // (10, 7)
  x1 = x1 * x2;
  x1.print(x1);                          // (-45, 43)
  return 0;
}
```

Любой унарный оператор ***@***, префиксный или постфиксный, определяется для объектов класса ***a***:

- либо как нестатическая компонентная функция без параметров, что для выражений ***@a*** и ***a@*** означает вызов:
  a.operator@()
- либо как глобальная или дружественная функция с одним параметром, что для выражений ***@a*** и ***a@*** означает вызов:
  operator@(a)

Если определены обе операторные функции, то для выяснения того, какую (возможно и никакую) из них использовать, применяется механизм разрешения перегрузки, который заключается в нахождении наилучшего соответствия типов формальных параметров этих функций и их аргументов вызова.

Заметим, что согласно принятому соглашению при распространении действия постфиксных операторов инкрементирования и декрементирования, которые в то же время могут быть и префиксными, операторные функции должны иметь еще один дополнительный параметр типа ***int***, значение которого компилятор определяет равным нулю, т.е. выражение ***a@*** означает либо вызов ***a.operator@(int)*** в случае определения нестатической компонентной функции без параметров, либо вызов ***operator@(a, int)*** в случае определения глобальной или дружественной функции с одним параметром. Если определены обе операторные функции, то по-прежнему для

выяснения того, какую (возможно и никакую) из них использовать, применяется механизм разрешения перегрузки.

Например, распространим действия двух унарных операторов (префиксный **++** и постфиксный **--**) на объекты класса ***Complex***, определив ***operator*++()** и ***operator*--()** как нестатические компонентные функции, полагая, что операции инкремента и декремента будут определены только для вещественной части комплексного числа:

```
// Пример 29
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0):re(r), im(i) {}
  // Инкремент вещественной части комплексного числа
  Complex& operator++()
  {
    ++re;
    return *this;
  }
  // Декремент вещественной части комплексного числа
  Complex operator--(int)
  {
    Complex temporary = *this;
    --re;
    return temporary;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  private:
  // Компонентные данные - все собственные (private)
  double re;
  double im;
};
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  x1.print(++x1);                        // (0, 5)
  x1.print(x1);                          // (0, 5)
  x2 = ++++x1;
```

```
  x2.print(x2);                          // (2, 5)
  x1.print(x1);                          // (2, 5)
  x1.print(x1--);                        // (2, 5)
  x1.print(x1);                          // (1, 5)
  x2 = x1--;
  x2.print(x2);                          // (1, 5)
  x1.print(x1);                          // (0, 5)
  return 0;
}
```

Напомним здесь, что если какая-либо функция возвращает указатель или ссылку, то добавление квалификатора ***const*** к спецификации возвращаемого значения в определении этой функции означает, что она не может использовать возвращаемое значение для последующего изменения значения переменной, на которую указывает или ссылается эта функция. Например, с помощью операторной функции

```
  const Complex& operator++()
  {
    ++re;
    return *this;
  }
```

уже невозможно будет выполнить какую-либо многократную операцию инкремента для комплексного числа, т.е. в рассматриваемом примере такая инструкция теперь будет ошибочной:

```
  x2 = ++++x1;    // ошибка!
```

Отметим теперь характерные особенности механизма перегрузки стандартных операторов:

- нельзя изменить приоритеты стандартных операторов;
- нельзя изменить синтаксис стандартных операторов, однако им можно придать новый смысл, задав соответствующие определения;
- нельзя вводить новые лексические обозначения стандартных операторов;
- не реализуются неявные замены для стандартных операторов, определяемых как эквивалентные комбинации других стандартных операторов;
- нельзя изменить смысл выражения, если в него не входит объект класса;
- операторная функция, первый параметр которой принадлежит базовому типу, не может быть компонентной функцией (расширение действия допустимо только с помощью глобальных или дружественных операторных функций);
- перегрузка оператора возможна только в области действия того класса, в котором он выполняется.

Теперь остановимся на подробном обсуждении некоторых важных для понимания деталей механизма перегрузки стандартных операторов.

Во-первых, операторные функции ***operator*=()**, ***operator*[]()** и ***operator*−>()** должны быть нестатическими компонентными функциями, так как это гарантирует, что их первый параметр будет ***lvalue*** (именующее выражение, ссылающееся на любой объект программы).

Во-вторых, из-за исторической случайности операторы **=** (присваивание), **&**(адрес) и **,**(последовательность) имеют предопределенный смысл, когда применяются к объектам класса. Этот предопределенный смысл может стать недоступным пользователям класса, если сделать операторы закрытыми.

В-третьих, так как перечисления, как и классы, являются типами, определяемыми пользователем, поэтому для них можно ввести операторы.

В-четвертых, поиск операторов, определяемых в пространствах имен, также как и поиск функций, осуществляется на основе типов их операндов.

В-пятых, следует сводить к минимуму количество функций, непосредственно манипулирующих представлением объекта, а также, по возможности, стремиться к улучшению производительности во время выполнения перегруженных операций.

В завершение рассмотрим ряд примеров, иллюстрирующих характерные приемы перегрузки стандартных операторов для достижения эффективной реализации кода.

Смешанная арифметика

В терминологии языка Fortran операторы, которые работают с операндами разного типа, реализуют так называемую ***смешанную арифметику***.

Смешанную арифметику, например, для комплексных чисел легко реализовать, если, во-первых, как было сказано, свести к минимуму количество функций, непосредственно манипулирующих представлением объекта класса. Так, для класса ***Complex*** этого можно добиться благодаря определению в нем только тех операторов, которые будут модифицировать значение первого параметра: таких как **+=**. Например, операторная функция

```
Complex& operator+=(Complex c)
{
  re += c.re;
  im += c.im;
  return *this;
}
```

не требует временной переменной для хранения результата сложения, кроме того, в операции **+=** участвуют только два объекта (ее операнды), в то время как в операции **+** участвуют три объекта (ее операнды и результат). В результате такого избавления от временных переменных улучшается производительность во время выполнения перегруженных операций **+=**.

Во-вторых, операторные функции, которые просто вычисляют новое значение на основе своих параметров, такие как **+**, теперь определяются вне класса в каком-либо пространстве имен (возможно в глобальном), при этом они пользуются операторами, имеющими доступ к представлению объекта. Например, операторная функция

```
Complex operator+(Complex a, Complex b)
{
  Complex temporary = a;
  return temporary += b;
}
```

реализует доступ к представлению объекта класса при помощи оператора **+=**.

В-третьих, теперь требуется лишь определить арифметические операторы, которые бы работали с операндами разного типа. Например, представим определение класса ***Complex***, в котором смешанная арифметика будет реализована только для одной операции сложения с операндами типа ***Complex*** и встроенного типа ***double***:

```
// Пример 30
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
struct Complex {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Сложение комплексных чисел
  Complex& operator+=(Complex c)
  {
    re += c.re;
    im += c.im;
    return *this;
  }
  // Визуализация комплексного числа
  void print(Complex c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  private:
  // Компонентные данные - все собственные (private)
  double re;
  double im;
};
// Сложение комплексных чисел
Complex operator+(Complex a, Complex b)
{
  Complex temporary = a;
  return temporary += b;
}
// Сложение комплексного числа с действительным числом
Complex operator+(Complex a, double b)
{
  Complex temporary = a;
  return temporary += b;
}
// Сложение действительного числа с комплексным числом
Complex operator+(double a, Complex b)
{
  Complex temporary = b;
  return temporary += a;
}
```

```
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  x1.print(x1);                          // (-1, 5)
  x2.print(x2);                          // (10, 7)
  x1.print(x1 + x2);                     // (9, 12)
  x1 = x1 + 2;
  x1.print(x1);                          // (1, 5)
  x2 = 2 + x2;
  x2.print(x2);                          // (12, 7)
  return 0;
}
```

Как видим, теперь можно записывать комплексные выражения в форме, еще более близкой к общепринятой.

Вывод

В C++ вывод иногда называют вставкой (от слова ***insertion***), а перегруженный оператор сдвига влево **<<** – соответственно оператором вставки (от слова ***inserter***). Операция вывода вставляет данные различных типов в поток вывода. Оператор вывода позволяет избежать многословности, которая получается при использовании функций вывода. Когда пользователь перегружает стандартный оператор **<<** для вывода, он создает операторную функцию вставки (глобальную или дружественную функцию с двумя параметрами).

У всех операторных функций вставки общая форма, например, для класса ***имя_класса*** глобальную функцию вставки можно определить так:

ostream& operator<<(ostream& имя_потока, имя_класса имя_объекта)
{ тело_функции_вставки }

Здесь первый параметр является ссылкой на объект типа ***ostream***, это означает, что параметр ***имя_потока*** должен быть потоком вывода. Второй параметр является объектом, который будет выводиться в поток. Функция вставки ***operator<<*** возвращает ссылку на поток вывода ***ostream***, для которого она вызывалась, чтобы к этой ссылке можно было применить другой ***operator<<***. Например, вывод в одной инструкции двух объектов ***x1*** и ***x2*** класса ***имя_класса***

```
cout << x1 << x2 << endl;
```

будет интерпретирован так:

```
operator<<(cout, x1)<<(cout, x2);
```

Это позволяет сохранить предполагаемый порядок вывода – слева направо.

В большинстве случаев функция вставки является дружественной классу, для которого она создавалась. Так, дружественная классу ***Complex*** функция вставки

```
friend ostream& operator<<(ostream& stream, Complex c)
{
  return stream << '(' << c.re << ", " << c.im << ')';
}
```

позволяет воспользоваться оператором **<<** точно так же, как и для встроенных типов, и тем самым полностью заменить компонентную функцию ***print*()**. Например:

```
Complex a(-1, 5);    // a.re = -1, a.im = 5
cout << a << endl;   // (-1, 5)
```

Копирующее присваивание

Ранее уже было сказано, что операция копирования объектов класса требует особого внимания со стороны пользователя класса. Отсутствие контроля при выполнении этой операции может привести к нежелательным и порой неожиданным эффектам. Напомним, что значение одного объекта может быть передано другому лишь в двух случаях – при присваивании и при инициализации. Напомним также, что выражение присваивания для объектов класса, как и для структур, по умолчанию означает почленное копирование данных одного объекта в другой.

Следует помнить также, что такое копирование обычно является неправильным при копировании объектов, имеющих ресурсы, управляемые конструктором и деструктором. Не менее катастрофичной такая ситуация наблюдается, например, при наличии в классе членов, являющихся указателями или ссылками. Кроме того, присваивание по умолчанию не может быть сгенерировано, если нестатический компонент класса является ссылкой, константой или типом, определяемым пользователем, не имеющим копирующего оператора присваивания.

Пользователь может определить любой подходящий смысл операций копирования, реализуемых с помощью копирующего оператора присваивания или копирующего конструктора, однако традиционным решением является почленное копирование хранимых компонентных данных.

Зачастую копирующий конструктор и копирующее присваивание значительно отличаются. Основная причина этого состоит в том, что копирующий конструктор инициализирует “чистую” память, в то время как копирующий оператор присваивания должен правильно работать с уже созданным объектом.

Отметим, что семантику присваивания и копирования по умолчанию зачастую называют поверхностным копированием, так как копируются члены класса, но не объекты, на которые эти члены указывают. Рекурсивное копирование указываемых объектов (так называемое глубокое копирование) следует определять явно.

Основная стратегия при реализации копирующего оператора присваивания проста:

- защита от присваивания самому себе;
- удаление старых элементов;
- инициализация и копирование новых элементов.

Как правило, все нестатические компоненты должны быть скопированы. Очевидно также, что при копирующем присваивании не должен вызываться никакой конструктор, в том числе и конструктор копирования, так как в противном случае изменилось бы содержание объекта-копии, т.е. при копировании объекта-оригинала необходимо его текущее, а не начальное состояние.

В качестве примера рассмотрим класс ***Complex***, объекты которого копируются при помощи копирующего оператора присваивания, реализующего механизм защиты от присваивания самому себе:

```
// Пример 31
// C++ Абстрактный тип данных - комплексное число
#include <iostream>
using namespace std;
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  friend ostream& operator<<(ostream& stream, const Complex& c)
  {
    return stream << '(' << c.re << ", " << c.im << ')';
  }
  // Копирующее присваивание
  Complex& operator=(const Complex&);
};
// Копирующее присваивание
Complex& Complex::operator=(const Complex& c)
{
  if (this != &c)
  {
    re = c.re;
    im = c.im;
  }
  return *this;
}
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  Complex x3(9, 12);
  cout << x1 << ' ' << x2 << ' ' << x3 << endl;
  x1 = x2;
  cout << x1 << endl;
  x2 = x3 = x1;
  cout << x1 << ' ' << x2 << ' ' << x3 << endl;
  return 0;
}
```

Результат работы программы:

```
(-1, 5) (10, 7) (9, 12)
(10, 7)
(10, 7) (10, 7) (10, 7)
```

Вызов функции

Одним из наиболее очевидных и в то же время важных приемов использования перегруженного бинарного оператора вызова функции **()** является предоставление синтаксиса стандартного вызова функций для объектов класса, которые в некотором смысле ведут себя как функции – так называемые ***объекты-функции*** или ***функторы***. Класс, в котором определен оператор вызова функции, называется ***функциональным***.

Итак, объекты-функции – это экземпляры функционального класса. Когда они используются в качестве функций, их вызов осуществляется благодаря оператору **()**.

Объекты-функции обеспечивают механизм, посредством которого пользователь может приспособить стандартные алгоритмы библиотеки C++ для работы со своими данными. Алгоритмами называют функции, выполняющие некоторые стандартные действия, например, поиск, копирование, слияние. Ряд обобщенных алгоритмов стандартной библиотеки требуют функций в качестве аргументов. Одним из простых примеров служит обобщенный алгоритм ***for_each*()**, который вызывает функцию или объект-функцию, переданную ему в качестве одного из аргументов, для каждого элемента последовательности. Отметим здесь, что представление данных в виде последовательности объектов – как содержимого некоторого контейнера – является одной из тех концепций, что составляют основу проектирования шаблонов стандартной библиотеки C++. Доступ к элементам последовательности реализуется с помощью итераторов, представляющих собой абстракцию понятия указателя.

Отметим также, что для иллюстрации ряда примеров, связанных с использованием стандартных алгоритмов библиотеки, аргументами которых могут быть итераторы и объекты-функции, автор вынужден здесь приводить фрагменты кода, характерные только для обобщенной парадигмы программирования, которой целиком будет посвящена уже следующая книга – от шаблонов функций и шаблонов классов до обобщенных алгоритмов библиотеки.

А теперь, прежде чем перейти к функциональным классам, рассмотрим пример, который составит основу перехода от традиционной функции, используемой для каждого элемента последовательности, к объекту-функции.

Например, визуализацию элементов одномерного массива, значение которых превышает некоторый заданный порог, можно осуществить с помощью алгоритма ***for_each*()**, если ему в качестве последнего его аргумента передать указатель на функцию ***greater_then*()**, а в качестве первых двух его аргументов (которые в общем случае являются итераторами ввода) передать указатели – соответственно на первый элемент массива и за его последний элемент:

```
// Пример 32
// C++ Иллюстрация стандартного алгоритма библиотеки for_each()
#include <iostream>
#include <algorithm>
using namespace std;
void greater_then(int data)
{
  if (data > 5) cout << data << ' ';
}
```

```
int main()
{
  int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  for_each(a, a + 10, greater_then);
  cout << endl;
  return 0;
}
```

Результат работы программы:

```
6 7 8 9
```

Как видим, стандартный алгоритм ***for_each*()** последовательно друг за другом берет элементы массива (в общем случае стандартного контейнера), на которые ссылается указатель (в общем случае итератор ввода), и передает их как аргумент функции ***greater_then*()**, что в целом позволяет отказаться от явной формы организации цикла.

Отметим, что стандартная библиотека предоставляет множество полезных объектов-функций – логические предикаты (функторы, возвращающие булево значение), арифметические операции и адаптеры (унарные и бинарные функторы). Объекты-функции (как стандартные, так и созданные самим пользователем) позволяют записывать код с использованием нетривиальных операций в качестве параметра. Зачастую перегрузка оператора вызова функции оказывается весьма полезной для определения типов с одной единственной операцией или типов с одной главной операцией.

Рассмотрим три функциональных класса, объекты-функции которых реализуют единственные для каждого определяемого пользователем типа данных операции, например, сравнения и сложения в целочисленной и комплексной арифметике.

Представим первый функциональный класс – ***GreaterThen***, объект-функция которого реализует операцию сравнения данных встроенного типа ***int***:

```
// Пример 33
// C++ Функциональный класс
#include <iostream>
using namespace std;
class GreaterThen {
  public:
  // Компонентные функции - все общедоступные (public)
  // Сравнение целых чисел
  int operator()(int a, int b)const
  {
    return a > b;
  }
};
int main()
{
```

```
  GreaterThen x;
  cout << x(-1, 5) << endl;            // 0
  cout << x(5, -1) << endl;            // 1
  return 0;
}
```

Как видим, от такого функционального класса даже не требуется наличия других его компонентов. От определения функционального класса ***GreaterThen*** теперь легко перейти к определению логического предиката ***GreaterThen***. Например:

```
  class GreaterThen {
    // Компонентные данные - все собственные (private)
    const int value;
    public:
    // Компонентные функции - все общедоступные (public)
    // Конструктор объектов класса
    GreaterThen(int data) : value(data) {}
    // Сравнение с целым числом
    bool operator()(int data)const
    {
      return data > value;
    }
  };
```

Результатом такого определения является функция общего вида, выполняющая целочисленное сравнение с некоторым пороговым значением, определяемым при создании экземпляра класса ***GreaterThen*** его конструктором. Например:

```
// Пример 34
// C++ Функциональный класс
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
class GreaterThen {
  // Компонентные данные - все собственные (private)
  const int value;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  GreaterThen(int data) : value(data) {}
  // Сравнение целых чисел
  bool operator()(int data)const
  {
    return data > value;
  }
};
int main()
{
```

```
  // Объявление стандартного контейнера – пустой вектор элементов
  vector<int> v;
  // Инициализация вектора - добавление элементов в контейнер
  for (int i = 1; i <= 5; ++i) v.push_back(i);
  // Визуализация вектора – копирование элементов в поток вывода
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  // Объявление итератора произвольного доступа
  vector<int>::iterator first;
  // Поиск первого элемента, удовлетворяющего предикату
  first = find_if(v.begin(), v.end(), GreaterThen(2));
  if (first != v.end()) cout << *first << endl;
  return 0;
}
```

Результат работы программы для платформы Windows:

```
1 2 3 4 5
3
```

Теперь представим третий функциональный класс – ***Complex***, объект-функция которого реализует операцию сложения данных типа ***Complex***:

```
// Пример 35
// C++ Функциональный класс
#include <iostream>
using namespace std;
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  friend ostream& operator<<(ostream& stream, Complex c)
  {
    return stream << '(' << c.re << ", " << c.im << ')';
  }
  // Прибавление комплексного числа
  Complex& operator()(Complex c)
  {
    re += c.re;
    im += c.im;
    return *this;
  }
```

```
  // Прибавление комплексного числа
  Complex& operator()(double a, double b)
  {
    re += a;
    im += b;
    return *this;
  }
};
int main()
{
  Complex x1(-1, 5);
  Complex x2(10, 7);
  cout << x1 << endl;                  // (-1, 5)
  cout << x2 << endl;                  // (10, 7)
  cout << x1(x2) << endl;              // (9, 12)
  cout << x1(-10, -7) << endl;         // (-1, 5)
  return 0;
}
```

Как видно из этих примеров, объекты функциональных классов используются так, как если бы они были функциями.

Индексация

Как в C, так и в C++ во время выполнения программы можно выйти за границы диапазона встроенного массива без генерации сообщений об ошибках времени выполнения. Класс ***vector*** из стандартной библиотеки по умолчанию так же не обеспечивает проверку диапазона вектора. Для создания защищенного одномерного массива, например, можно определить класс, который позволяет объявить этот массив и разрешить доступ к его элементам только через перегруженный оператор индексации **[]**. С помощью операторной функции ***operator*[]()** теперь можно будет перехватывать индекс, который выходит за границы диапазона массива.

Например, приведем определение класса ***Array1D*** для защищенного одномерного массива, элементы которого принадлежат встроенному типу ***int***:

```
  class Array1D {
    // Компонентные данные - все собственные (private)
    int* base1D;
    int size1D;
    public:
    // Компонентные функции - все общедоступные (public)
    // Конструктор объектов класса
    Array1D(int);
    // Деструктор объектов класса
    ~Array1D();
    // Индексация массива
    int& operator[](int);
  };
```

```
// Конструктор объектов класса
Array1D::Array1D(int size)
{
  base1D = new int[size];
  size1D = size;
}
// Деструктор объектов класса
Array1D::~Array1D()
{
  delete[] base1D;
}
// Индексация массива
int& Array1D::operator[](int index)
{
  if (index < 0 || index > size1D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base1D[index];
}
```

Теперь после определения объекта класса ***Array1D***, например:

```
Array1D a(5);
```

все операции индексации массива ***a*** будут ограничены диапазоном изменения индекса от ***0*** до ***4***.

Отметим здесь, что операции индексации для индекса, выходящего за границы допустимого диапазона, как правило, должны приводить к аварийному завершению программы.

Для создания защищенного двумерного массива необходимо воспользоваться особенностями механизма индексации многомерных массивов – определить класс, который позволяет объявить этот массив, и разрешить доступ к его элементам только через перегруженный оператор **[]**. Например, это можно сделать, определив класс ***Array2D*** таким образом, чтобы его операторная функция ***operator*[]()** перехватывала бы индекс, выходящий за границы первой размерности массива, и возвращала бы ссылку на объект вложенного класса ***Array1D***. В свою очередь, операторная функция ***operator*[]()** вложенного класса ***Array1D*** перехватывала бы индекс, выходящий за границы второй размерности массива, и возвращала бы ссылку на элемент массива.

Пользователи класса ***Array2D*** не должны знать о существовании класса ***Array1D***, объекты класса ***Array1D*** представляют собой одномерные массивы и называются proxy-объектами (от слов ***proxy objects***) или заместителями (от слова ***surrogates***). Соответственно, классы, объекты которых являются proxy-объектами, называются proxy-классами (от слов ***proxy classes***).

Например, приведем определение класса ***Array2D*** для защищенного двумерного массива, элементы которого принадлежат встроенному типу ***int***:

```
class Array2D {
  class Array1D {
    // Компонентные данные - все собственные (private)
    int* base1D;
    int size1D;
    public:
    // Компонентные функции - все общедоступные (public)
    // Конструктор по умолчанию
    Array1D();
    // Деструктор объектов класса
    ~Array1D()
    {
      delete[] base1D;
    }
    // Выделение свободной памяти для одномерного массива
    void allocate1D(int);
    // Индексация массива
    int& operator[](int);
  };
  // Компонентные данные - все собственные (private)
  Array1D* base2D;
  int size2D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array2D(int, int);
  // Деструктор объектов класса
  ~Array2D()
  {
    delete[] base2D;
  }
  // Индексация массива
  Array1D& operator[](int);
};
// Конструктор объектов класса Array2D
Array2D::Array2D(int columnSize, int rowSize)
{
  base2D = new Array1D[columnSize];
  for (int i = 0; i < columnSize; ++i)
    base2D[i].allocate1D(rowSize);
  size2D = columnSize;
}
// Индексация массива
Array2D::Array1D& Array2D::operator[](int index)
{
```

```
  if (index < 0 || index > size2D - 1)
  {
    cout << "Первый индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base2D[index];
}
// Конструктор по умолчанию класса Array1D
Array2D::Array1D::Array1D()
{
  base1D = 0;
  size1D = 0;
}
// Выделение свободной памяти для одномерного массива
void Array2D::Array1D::allocate1D(int rowSize)
{
  base1D = new int[rowSize];
  size1D = rowSize;
}
// Индексация массива
int& Array2D::Array1D::operator[](int index)
{
  if (index < 0 || index > size1D - 1)
  {
    cout << "Второй индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base1D[index];
}
```

Теперь после определения объекта класса ***Array2D***, например:

```
Array2D a(5,6);
```

операции индексации массива ***a*** будут ограничены соответствующими диапазонами изменения индексов – первого от ***0*** до ***4***, а второго от ***0*** до ***5***.

“Умные указатели”

Перегрузка оператора разыменования **->** имеет значение для целого круга интересных задач. Причина здесь в том, что косвенный доступ является ключевой концепцией, а оператор **->** предоставляет ясный, прямой и эффективный способ его реализации в программе. Например, одним из важных примеров являются итераторы стандартной библиотеки шаблонов. Итераторы поддерживают абстрактную модель данных как последовательность объектов – “нечто такое, что можно перебирать либо от начала к концу, либо от конца к началу, используя оператор получения следующего элемента”. Итератор – это абстракция понятия указателя на элемент последовательности.

Другим ярким примером являются так называемые “умные указатели” или интеллектуальные указатели (от слов ***smart pointers***) – объекты класса, которые ведут себя как встроенные указатели и, кроме того, выполняют некоторые действия, когда с их помощью осуществляется доступ к компонентам другого класса, на объекты которого эти указатели как раз и ссылаются.

Для класса, объекты которого будут интеллектуальными указателями на объекты другого класса ***имя_класса***, должна быть определена операторная функция:

имя_класса* operator->() { тело_функции }

Интеллектуальными указателями можно пользоваться для доступа к компонентам класса ***имя_класса*** как встроенными указателями на объекты этого класса.

Напомним, что для встроенных указателей на объекты класса использование оператора **->** является синонимом некоторых применений операторов ***** и **[]**. Если имеется

имя_класса* имя_указателя_на_объект_класса = &имя_объекта_класса;

то для доступа, например, к компонентам данных можно использовать конструкции:

имя_указателя_на_объект_класса->имя_компонента_данных

(**имя_указателя_на_объект_класса***) .****имя_компонента_данных***

имя_указателя_на_объект_класса*[*0*] *.имя_компонента_данных

Если требуется подобная эквивалентность, то для класса, объекты которого будут интеллектуальными указателями на объекты класса ***имя_класса***, следует определить также еще одну операцию разыменования и операцию индексации:

имя_класса& operator*() { тело_функции }

***имя_класса& operator*[]() { *тело_функции* }**

Как правило, интеллектуальные указатели создаются на основе шаблонов классов, поскольку так же, как и встроенные указатели, они должны быть максимально типизированы. Параметр шаблона определяет тип объекта, на который указывает интеллектуальный указатель.

Если вместо встроенных указателей C++ будут использоваться интеллектуальные указатели, то пользователь теперь может управлять следующими операциями:

- созданием и уничтожением указателей;
- копированием и присваиванием указателей;
- разыменованием указателей;
- преобразованием типов указателей;
- поддержкой указателей на константные объекты.

Обычно при создании интеллектуальным указателям по умолчанию присваивается значение ***0***, чтобы не допускать ошибок, возникающих из-за неинициализированных указателей. Некоторые интеллектуальные указатели отвечают за удаление объекта сразу после того, как будет уничтожен последний указывающий на этот объект интеллектуальный указатель. Это косвенно позволяет избежать утечек памяти. Для некоторых типов интеллектуальных указателей необходимо, чтобы они неявно копировали или присваивали то, на что указывают, т.е. выполняли бы глубокое копирование. Для других должен копироваться или присваиваться только сам указатель. Для третьих эти операции следует запретить. Однако независимо от

поведения разных типов интеллектуальных указателей их грамотное использование позволяет добиться требуемого функционирования программы.

То, что должно произойти, когда осуществляется обращение к объекту, на который ссылается интеллектуальный указатель, решает теперь каждый пользователь сам. Например, интеллектуальные указатели позволяют реализовать стратегию отложенной выборки – одной из разновидностей так называемых отложенных (буквально “ленивых”) вычислений. При использовании отложенных вычислений классы определяются таким образом, что вычисления не производятся до тех пор, пока не потребуются их результаты.

Рассмотрим класс ***Complex***, на объекты которого ссылаются интеллектуальные указатели – объекты класса ***SmartPointer***:

```
// Пример 36
// C++ Абстрактный тип данных – интеллектуальный указатель
#include <iostream>
using namespace std;
class Complex {
  // Компонентные данные - все собственные (private)
  double re;
  double im;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  friend ostream& operator<<(ostream& stream, const Complex& c)
  {
    return stream << '(' << c.re << ", " << c.im << ')';
  }
  // Ссылка на вещественную часть комплексного числа
  double& get_re()
  {
    return re;
  }
  // Ссылка на мнимую часть комплексного числа
  double& get_im()
  {
    return im;
  }
};
class SmartPointer {
  // Компонентные данные - все собственные (private)
  Complex* pointer;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  SmartPointer(Complex* p = 0) : pointer(p) {}
```

```
  // Конструктор копирования
  SmartPointer(SmartPointer& p)
  {
    pointer = p.pointer;
    p.pointer = 0;
  }
  // Копирующее присваивание
  SmartPointer& operator=(SmartPointer& p)
  {
    if (this != &p)
    {
      pointer = p.pointer;
      p.pointer = 0;
    }
    return *this;
  }
  // Разыменование интеллектуального указателя
  Complex* operator->()const
  {
    return pointer;
  }
  // Разыменование интеллектуального указателя
  Complex& operator*()const
  {
    return *pointer;
  }
  // Индексация интеллектуального указателя
  Complex& operator[](int index)const
  {
    return pointer[index];
  }
};
int main()
{
  Complex x(-1, 5);
  cout << x << endl;                  // (-1, 5)
  SmartPointer p(&x);
  cout << *p << endl;                 // (-1, 5)
  cout << p[0] << endl;               // (-1, 5)
  p->get_re() = 10;                   // x.re = 10
  p->get_im() = 7;                    // x.im = 7
  cout << x << endl;                  // (10, 7)
  cout << *p << endl;                 // (10, 7)
  SmartPointer q = p;                 // новый владелец объекта
  cout << *q << endl;                 // (10, 7)
  q->get_re() = -1;                   // x.re = -1
  q->get_im() = 5;                    // x.im = 5
  cout << x << endl;                  // (-1, 5)
```

```
  SmartPointer t;
  t = q;                                  // новый владелец объекта
  cout << *t << endl;                     // (-1, 5)
  return 0;
}
```

Как видим, здесь интеллектуальный указатель всегда является владельцем того объекта, на который он указывает.

Наследование классов

Ранее неоднократно отмечалось, что терминология объектно-ориентированного программирования находится под сильным влиянием программирования на языке Smalltalk. Здесь и далее будут использоваться термины, например, такие как сообщение (вызов функции) и метод (функция-член).

Каждый объект является конкретным представителем (экземпляром) класса. Объекты одного класса имеют одинаковые по типам и внутренним именам компонентные данные. Объектам одного класса для обработки своих данных доступны одинаковые компонентные функции класса и одинаковые операции, настроенные на работу с объектами класса.

Объекты разных классов и сами классы могут находиться в отношении ***наследования***, при котором формируется иерархия объектов в соответствии с предусмотренной иерархией классов. Иерархия классов позволяет определять новые классы на основе уже имеющихся. Имеющиеся классы обычно называют ***базовыми*** (порождающими), а новые классы – ***производными*** (порожденными), иногда ***классами-потомками*** или ***наследниками***.

В иерархии классов различают два вида отношений наследования – ***одиночное*** или ***простое*** и ***множественное***. Если в иерархии классов какой-либо производный класс связан отношением наследования непосредственно только с одним базовым классом, то это пример ***одиночного наследования***.

Формат определения производного класса при одиночном наследовании:

ключ_класса имя_производного_класса **:**
  спецификатор_доступа имя_базового_класса **{**
    компоненты_производного_класса
};

Если в иерархии классов какой-либо производный класс связан отношением наследования непосредственно с множеством базовых классов, то это пример ***множественного наследования***.

Формат определения производного класса при множественном наследовании:

ключ_класса имя_производного_класса **:**
  спецификатор_доступа имя_базового_класса_1,
  …
  спецификатор_доступа имя_базового_класса_n **{**
    компоненты_производного_класса
};

Класс называют ***непосредственным базовым классом*** (***непосредственной*** или ***прямой базой*** от слов ***direct base***), если он входит в список базовых при определении производного класса. В то же время для производного класса могут существовать косвенные или непрямые предшественники, которые, в свою очередь, являются базовыми для классов, входящих в список прямых базовых при его определении.

Класс называют ***косвенным базовым классом*** (***непрямой базой*** от слов ***indirect base***), если в иерархии классов он является базовым для класса, входящего в список прямых базовых классов при определении производного класса. Например, если класс ***A*** является базовым для производного класса ***B***, а класс ***B*** – базовым для производного класса ***C***, то класс ***B*** является непосредственным базовым классом для класса ***C***, а класс ***A*** – косвенным базовым классом для класса ***C***:

```
class A { компоненты_класса_A };
class B : public A { компоненты_класса_B };
class C : public B { компоненты_класса_C };
```

В иерархии классов любой производный класс может становиться базовым для других классов, и таким образом формируется направленный ациклический граф иерархии классов и объектов (***НАГ*** или ***DAG*** **–** от слов ***directed acyclic graph***), где стрелкой изображают отношение “производный от”, производные классы при этом изображаются ниже базовых. Например:

Производные классы получают наследство – данные и методы своих базовых классов, т.е. получают к ним доступ. Производные классы могут пополняться собственными данными и методами. При наследовании методы и (или) данные базового класса могут быть переопределены в производном классе. В этом случае соответствующие компоненты базового класса становятся скрытыми и тем самым недоступными из производного класса (теперь для доступа к ним необходимо использовать операцию разрешения области видимости).

Сообщения, обработку которых не могут выполнить методы производного класса, автоматически передаются в базовый класс, если для обработки сообщения необходимы данные, отсутствующие в производном классе, то их поиск автоматически производится в базовом классе.

В производном классе наследуемые компоненты базового класса получают статус доступа ***private***, если производный класс определяется с помощью ключевого слова ***class***, и статус доступа ***public***, если производный класс определяется с помощью ключевого слова ***struct***. Кроме того, при объявлении производных классов можно явно изменить статус доступа к наследуемым компонентам базовых классов при помощи спецификаторов доступа (***public***, ***private*** и ***protected***), которые указываются непосредственно перед именами базовых классов. Отметим также, что базовые и производные классы не могут быть объявлены с помощью ключевого слова ***union***.

Спецификаторы доступа при объявлении производных классов определяют, как компоненты базового класса наследуются производным классом:

- ***public*** для наследуемого базового класса
  все открытые компоненты базового класса остаются открытыми и в производном классе;
  все закрытые компоненты базового класса остаются закрытыми и в производном классе;
  все защищенные компоненты базового класса остаются защищенными и в производном классе;
- ***private*** для наследуемого базового класса
  все открытые компоненты базового класса становятся закрытыми в производном классе;
  все закрытые компоненты базового класса остаются закрытыми и в производном классе;
  все защищенные компоненты базового класса становятся закрытыми в производном классе;
- ***protected*** для наследуемого базового класса
  все открытые компоненты базового класса становятся защищенными в производном классе;
  все закрытые компоненты базового класса остаются закрытыми и в производном классе;
  все защищенные компоненты базового класса остаются защищенными и в производном классе.

Как видим, статус доступа компонентов при наследовании классов играет важную роль. Открытое наследование делает производный класс подтипом базового класса, такая форма наследования является наиболее распространенной. Закрытое и защищенное наследование используется для деталей реализации. Закрытые базовые классы полезны в основном при определении класса, который предоставляет большие гарантии, чем его базовый класс, как бы “ужесточая” интерфейс базового класса. Защищенные базовые классы полезны в иерархиях классов, когда дальнейшее проектирование производных классов является нормой.

Спецификатор доступа к базовому классу управляет не только доступом к компонентам базового класса, но также и преобразованием указателей и ссылок из типа производного класса в тип базового класса. Так, при открытом наследовании любая функция может выполнить такое преобразование, при закрытом наследовании такое преобразование могут выполнить только методы производного класса и его друзья, а при защищенном наследовании – только методы производного класса, его друзья и методы его производных классов.

Заметим здесь, что на практике защищенные данные часто приводят к проблемам сопровождения. Такие компоненты класса более подвержены риску разрушения, чем закрытые, их также непросто реструктуризировать ввиду сложности нахождения всех случаев их использования. Все эти возражения не имеют большого значения для защищенных методов классов, так как защищенность – это прекрасный способ задания операций для использования в производных классах.

Как видим, выбор способа доступа к компонентам базовых классов при наследовании определяется теми же соображениями, что и для компонентов обычного класса:

- открытые (***public***) компоненты доступны из любого места программы, т.е. они являются глобальными;
- защищенные (***protected***) компоненты доступны внутри класса, в котором они определены, и во всех его производных классах;
- закрытые (***private***) компоненты доступны только внутри того класса, в котором они определены.

Итак, при обработке сообщения объектом рассматриваемого класса используются, во-первых, общедоступные компоненты всех классов программы, во-вторых, защищенные компоненты базовых и рассматриваемого классов, в-третьих, собственные компоненты рассматриваемого класса.

Инициализация объектов, представляющих базовые и производные классы, может быть задана в их конструкторах, причем конструкторы выполняются в порядке наследования (конструктор базового класса выполняется раньше конструктора производного класса), а деструкторы – в обратном (деструктор производного класса выполняется раньше деструктора базового класса).

Ранее было сказано, что при проектировании иерархии классов для инициализации компонентных данных можно воспользоваться расширенной формой объявления конструкторов производных классов, позволяющей через список инициализации членов, в котором могут быть представлены явные вызовы конструкторов прямых базовых классов, передавать по цепочке наследования всем требуемым конструкторам косвенных базовых классов необходимые им аргументы вызова.

Здесь же отметим, что при таком способе организации списка инициализации членов допустимо использование одних и тех же аргументов вызова для базового и производного классов. Для производного класса допустимо также игнорирование всех аргументов вызова, в этом случае необходима их передача напрямую в базовый класс. Если инициализация производится только в производном классе, аргументы вызова передаются обычным образом.

Приведем пример открытого одиночного наследования компонентов базового класса ***Complex_based*** его производным классом ***Complex_derived***, который к наследуемым компонентам добавляет собственную компонентную функцию ***add*()**:

```
// Пример 37
// C++ Одиночное наследование
#include <iostream>
using namespace std;
struct Complex_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_based(double r = 0.0, double i = 0.0) : re(r), im(i)
  {
    cout << "Конструктор базового класса" << endl;
  }
```

```
  // Деструктор объектов базового класса
  ~Complex_based()
  {
    cout << "Деструктор базового класса" << endl;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};
struct Complex_derived : Complex_based {
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_derived(double r = 0.0, double i = 0.0)
  : Complex_based(r, i)
  {
    cout << "Конструктор производного класса" << endl;
  }
  // Деструктор объектов производного класса
  ~Complex_derived()
  {
    cout << "Деструктор производного класса" << endl;
  }
  // Сложение комплексных чисел
  Complex_derived& add(Complex_derived& a, Complex_derived& b)
  {
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
};
int main()
{
  Complex_derived x1(-1, 5);
  Complex_derived x2(10, 7);
  Complex_derived x3;
  x1.print();
  x2.print();
  x3.print();
  x1.add(x1, x2);
  x1.print();
  return 0;
}
```

Результат работы программы:

```
Конструктор базового класса
Конструктор производного класса
Конструктор базового класса
Конструктор производного класса
Конструктор базового класса
Конструктор производного класса
(-1, 5)
(10, 7)
(0, 0)
(9, 12)
Деструктор производного класса
Деструктор базового класса
Деструктор производного класса
Деструктор базового класса
Деструктор производного класса
Деструктор базового класса
```

Как видим, отсутствие в производном классе собственных компонентных данных приводит к тому, что все операции с его объектами осуществляются только для наследуемых компонентных данных базового класса. Здесь конструкторы базового и производного классов используют одни и те же аргументы (что бывает не так часто). В большинстве случаев конструкторы производных классов передают конструкторам базовых классов только те аргументы вызова, которые им требуются. Например, чтобы передать конструктору базового класса необходимые ему аргументы, конструктору производного класса следует передать все аргументы вызова, необходимые конструкторам обоих классов.

Ранее было сказано, что наследуемые компоненты базового класса могут быть переопределены в его производном классе. В этом случае доступ к соответствующим компонентам базового класса может быть осуществлен с использованием операции разрешения области видимости.

Приведем пример открытого одиночного наследования компонентов базового класса ***Complex_based*** его производным классом ***Complex_derived***, который переопределяет наследуемые компонентные данные и компонентную функцию ***print*()**, а также добавляет собственную компонентную функцию ***add*()**:

```
// Пример 38
// C++ Одиночное наследование
#include <iostream>
using namespace std;
struct Complex_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_based(double r = 0.0, double i = 0.0) : re(r), im(i) {}
```

```
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};
struct Complex_derived : Complex_based {
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_derived(double r = 0.0, double i = 0.0)
  : Complex_based(r), re(r), im(i) {}
  // Сложение комплексных чисел
  Complex_derived& add(Complex_derived& a, Complex_derived& b)
  {
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};
int main()
{
  Complex_derived x1(-1, 5);
  Complex_derived x2(10, 7);
  Complex_derived x3;
  x1.Complex_based::print();          // (-1, 0)
  x1.print();                         // (-1, 5)
  x2.Complex_based::print();          // (10, 0)
  x2.print();                         // (10, 7)
  x3.Complex_based::print();          // (0, 0)
  x3.print();                         // (0, 0)
  x1.add(x1, x2);
  x1.Complex_based::print();          // (-1, 0)
  x1.print();                         // (9, 12)
  return 0;
}
```

Как и следовало ожидать, теперь все операции с объектами производного класса осуществляются только для его собственных компонентных данных.

Теперь обратимся к проблеме создания защищенных массивов на основе иерархии классов с открытым одиночным наследованием. Здесь, как и ранее, следует определить классы, позволяющие объявлять защищенные массивы, и разрешить доступ к их элементам только через перегруженный оператор **[]**. Отличие лишь в том, что благодаря иерархии классов теперь можно использовать общие средства для создания как одномерных, так и многомерных массивов. Например, для создания защищенных двумерных массивов это можно сделать, определив производный класс ***Array2D*** таким образом, чтобы его операторная функция ***operator*[]()** перехватывала бы индекс, выходящий за границы первой размерности массива, и возвращала бы ссылку на объект прямого базового класса ***Array1D***. В свою очередь, операторная функция ***operator*[]()** базового класса ***Array1D*** перехватывала бы индекс, выходящий за границы второй размерности массива, и возвращала бы ссылку на элемент массива. Защищенный одномерный массив здесь создается обычным образом.

Например, представим иерархию классов ***Array1D*** и ***Array2D*** для создания защищенных одномерных и двумерных массивов, элементы которых принадлежат встроенному типу ***int***:

```
class Array1D {
  // Компонентные данные - все собственные (private)
  int* base1D;
  int size1D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор по умолчанию
  Array1D();
  // Конструктор объектов класса
  Array1D(int);
  // Деструктор объектов класса
  ~Array1D();
  // Выделение свободной памяти для одномерного массива
  void allocate1D(int);
  // Индексация массива
  int& operator[](int);
};
// Конструктор по умолчанию класса Array1D
Array1D::Array1D()
{
  base1D = 0;
  size1D = 0;
}
// Конструктор объектов класса Array1D
Array1D::Array1D(int rowSize)
{
```

```
  allocate1D(rowSize);
}
// Деструктор объектов класса Array1D
Array1D::~Array1D()
{
  delete[] base1D;
}
// Выделение свободной памяти для одномерного массива
void Array1D::allocate1D(int rowSize)
{
  base1D = new int[rowSize];
  size1D = rowSize;
}
// Индексация массива
int& Array1D::operator[](int index)
{
  if (index < 0 || index > size1D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base1D[index];
}
class Array2D : public Array1D {
  // Компонентные данные - все собственные (private)
  Array1D* base2D;
  int size2D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array2D(int, int);
  // Деструктор объектов класса
  ~Array2D();
  // Выделение свободной памяти для двумерного массива
  void allocate2D(int, int);
  // Индексация массива
  Array1D& operator[](int);
};
// Конструктор объектов класса Array2D
Array2D::Array2D(int columnSize, int rowSize)
: Array1D()
{
  allocate2D(columnSize, rowSize);
}
```

```
// Деструктор объектов класса Array2D
Array2D::~Array2D()
{
  delete[] base2D;
}
// Выделение свободной памяти для двумерного массива
void Array2D::allocate2D(int columnSize, int rowSize)
{
  base2D = new Array1D[columnSize];
  for (int i = 0; i < columnSize; ++i)
    base2D[i].allocate1D(rowSize);
  size2D = columnSize;
}
// Индексация массива
Array1D& Array2D::operator[](int index)
{
  if (index < 0 || index > size2D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base2D[index];
}
```

Теперь после определения объектов класса ***Array1D*** и класса ***Array2D***, например:

```
Array1D a(3);
Array2D b(4, 5);
```

операции индексации массива ***a*** будут ограничены диапазоном изменения индекса от ***0*** до ***2***, а массива ***b*** – диапазонами изменения индексов от ***0*** до ***3*** и от ***0*** до ***4***.

Очевидно, что очередное добавление в иерархию классов с открытым одиночным наследованием каждого нового класса позволит теперь без особых усилий создавать любой защищенный многомерный массив. Например, представим иерархию классов ***Array1D***, ***Array2D*** и ***Array3D*** для создания защищенных одномерных, двумерных и трехмерных массивов, элементы которых принадлежат встроенному типу ***int***:

```
class Array1D {
  // Компонентные данные - все собственные (private)
  int* base1D;
  int size1D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор по умолчанию
  Array1D();
  // Конструктор объектов класса
  Array1D(int);
  // Деструктор объектов класса
  ~Array1D();
```

```
  // Выделение свободной памяти для одномерного массива
  void allocate1D(int);
  // Индексация массива
  int& operator[](int);
};
// Конструктор по умолчанию класса Array1D
Array1D::Array1D()
{
  base1D = 0;
  size1D = 0;
}
// Конструктор объектов класса Array1D
Array1D::Array1D(int rowSize)
{
  allocate1D(rowSize);
}
// Деструктор объектов класса Array1D
Array1D::~Array1D()
{
  delete[] base1D;
}
// Выделение свободной памяти для одномерного массива
void Array1D::allocate1D(int rowSize)
{
  base1D = new int[rowSize];
  size1D = rowSize;
}
// Индексация массива
int& Array1D::operator[](int index)
{
  if (index < 0 || index > size1D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base1D[index];
}
class Array2D : public Array1D {
  // Компонентные данные - все собственные (private)
  Array1D* base2D;
  int size2D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор по умолчанию
  Array2D();
```

```
  // Конструктор объектов класса
  Array2D(int, int);
  // Деструктор объектов класса
  ~Array2D();
  // Выделение свободной памяти для двумерного массива
  void allocate2D(int, int);
  // Индексация массива
  Array1D& operator[](int);
};
// Конструктор по умолчанию класса Array2D
Array2D::Array2D()
{
  base2D = 0;
  size2D = 0;
}
// Конструктор объектов класса Array2D
Array2D::Array2D(int columnSize, int rowSize)
: Array1D()
{
  allocate2D(columnSize, rowSize);
}
// Деструктор объектов класса Array2D
Array2D::~Array2D()
{
  delete[] base2D;
}
// Выделение свободной памяти для двумерного массива
void Array2D::allocate2D(int columnSize, int rowSize)
{
  base2D = new Array1D[columnSize];
  for (int i = 0; i < columnSize; ++i)
    base2D[i].allocate1D(rowSize);
  size2D = columnSize;
}
// Индексация массива
Array1D& Array2D::operator[](int index)
{
  if (index < 0 || index > size2D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base2D[index];
}
```

```
class Array3D : public Array2D {
  // Компонентные данные - все собственные (private)
  Array2D* base3D;
  int size3D;
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов класса
  Array3D(int, int, int);
  // Деструктор объектов класса
  ~Array3D();
  // Выделение свободной памяти для трехмерного массива
  void allocate3D(int, int, int);
  // Индексация массива
  Array2D& operator[](int);
};
// Конструктор объектов класса Array3D
Array3D::Array3D(int numberOfArrays2D, int columnSize,
                 int rowSize) : Array2D()
{
  allocate3D(numberOfArrays2D, columnSize, rowSize);
}
// Деструктор объектов класса Array3D
Array3D::~Array3D()
{
  delete[] base3D;
}
// Выделение свободной памяти для трехмерного массива
void Array3D::allocate3D(int numberOfArrays2D,
                         int columnSize, int rowSize)
{
  base3D = new Array2D[numberOfArrays2D];
  for (int i = 0; i < numberOfArrays2D; ++i)
    base3D[i].allocate2D(columnSize, rowSize);
  size3D = numberOfArrays2D;
}
// Индексация массива
Array2D& Array3D::operator[](int index)
{
  if (index < 0 || index > size3D - 1)
  {
    cout << "Индекс за границами диапазона!" << endl;
    exit(1);
  }
  return base3D[index];
}
```

Как видим, добавление очередного производного класса в иерархию с одиночным наследованием вызывает добавление конструктора по умолчанию для его прямого базового класса.

А теперь в завершение обсуждения обратимся к непростой, но очень важной проблеме, связанной с механизмом разрешения перегрузки компонентных функций для иерархии классов с одиночным наследованием. Нередко требуется, чтобы выбор между одноименными компонентными функциями базового и производного классов осуществлялся по типам аргументов. Так как разрешение перегрузки не пересекает границ областей видимости классов, то в этом случае в производном классе одноименные компонентные функции его базового класса объявляются при помощи ***using-объявлений***, что позволяет ввести эти функции в общую область видимости производного класса, чтобы затем на основе механизма разрешения перегрузки как раз и попытаться осуществить их вызов.

Приведем пример открытого одиночного наследования компонентов базового класса ***Complex_based*** его производным классом ***Complex_derived***, который переопределяет наследуемые компонентные данные, добавляет собственные компонентные функции ***add*()** и ***print*()**, а также вводит в свою область видимости компонентную функцию базового класса ***print*()**:

```
// Пример 39
// C++ Одиночное наследование
#include <iostream>
using namespace std;
struct Complex_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_based(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print(Complex_based& c)
  {
    cout << '(' << c.re << ", " << c.im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};
struct Complex_derived : Complex_based {
  using Complex_based::print;
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_derived(double r = 0.0, double i = 0.0)
  : Complex_based(r), re(r), im(i) {}
  // Сложение комплексных чисел
  Complex_derived& add(Complex_derived& a, Complex_derived& b)
  {
```

```
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};
int main()
{
  Complex_derived x1(-1, 5);
  Complex_derived x2(10, 7);
  Complex_derived x3;
  x1.print(x1);                          // (-1, 0)
  x1.print();                            // (-1, 5)
  x2.print(x2);                          // (10, 0)
  x2.print();                            // (10, 7)
  x3.print(x3);                          // (0, 0)
  x3.print();                            // (0, 0)
  x1.add(x1, x2);
  x1.print(x1);                          // (-1, 0)
  x1.print();                            // (9, 12)
  return 0;
}
```

Как видим, такой способ явного устранения неоднозначности вызова одноименных компонентных функций базового и производного классов является более удобным по сравнению с использованием операции разрешения области видимости.

Множественное наследование и виртуальные базовые классы

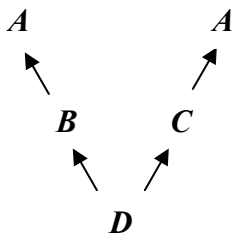
При множественном наследовании базовые классы так же, как и при одиночном наследовании базовый и его производный классы, могут иметь компоненты с одинаковым именем. Зачастую таковыми являются именно компонентные функции. Явное устранение такого рода неоднозначности при вызове этих функций в производном классе с использованием операции разрешения области видимости довольно неудобно. Одним из возможных способов решения этой проблемы является определение в производном классе компонентной функции с тем же самым именем. В этом случае компонентная функция производного класса скрывает компонентные функции (невиртуальные нестатические) базовых классов или замещает (***override***) их, если они виртуальные. Как и при одиночном наследовании, зачастую требуется, чтобы выбор между одноименными компонентными функциями различных базовых

классов осуществлялся по типам аргументов. В этом случае, как и ранее, в производном классе одноименные компонентные функции базовых классов объявляются при помощи ***using-объявлений***, что позволяет ввести эти функции в общую область видимости производного класса, чтобы затем на основе механизма разрешения перегрузки как раз и попытаться осуществить их вызов.

Заметим при этом, что ***using-объявление*** в определении производного класса должно относиться к компонентам его базовых классов, ***using-объявление*** нельзя использовать для компонента класса вне этого класса, его производных классов или их компонентных функций. Заметим также, что в определение класса нельзя поместить ***using-директиву***, и она не может использоваться для класса.

При множественном наследовании никакой класс не может больше одного раза использоваться в качестве прямого базового, однако класс может больше одного раза быть непрямым базовым классом, например:

```
class A { компоненты_класса_A };
class B : public A { компоненты_класса_B };
class C : public A { компоненты_класса_C };
class D : public B, public C { компоненты_класса_D };
```



Здесь непрямой базовый класс ***A*** дважды опосредованно наследуется классом ***D***, такое дублирование класса приводит к двукратному тиражированию объектов непрямого базового класса в производном объекте.

Кроме дублирования в производном классе ***D*** компонентных данных непрямого базового класса ***A*** здесь возможна также и неоднозначность доступа к ним, поэтому если из класса ***D*** необходим доступ к компонентным данным класса ***A***, ссылки на них должны быть явно квалифицированы.

Вначале приведем пример открытого множественного наследования компонентов непрямого базового класса ***A_based*** и двух прямых базовых классов ***B_derived*** и ***C_derived*** их производным классом ***D_derived***:

```
// Пример 40
// C++ Множественное наследование
#include <iostream>
using namespace std;
struct A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
```

```
  // Деструктор объектов базового класса
  ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // Визуализация компонента базового класса
  void print()
  {
    cout << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
struct B_derived : A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(int i = 0, long l = 0) : A_based(i), b_data(l)
  {
    cout << "Конструктор производного класса B_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~B_derived()
  {
    cout << "Деструктор производного класса B_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
    cout << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  long b_data;
};
struct C_derived : A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  C_derived(int i = 0, float f = 0.0) : A_based(i), c_data(f)
  {
    cout << "Конструктор производного класса C_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~C_derived()
  {
```

```
    cout << "Деструктор производного класса C_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
    cout << c_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  float c_data;
};
struct D_derived : B_derived, C_derived {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  D_derived(int i = 0, long l = 0, float f = 0.0, double d = 0.0)
  : B_derived(i, l), C_derived(i, f), d_data(d)
  {
    cout << "Конструктор производного класса D_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~D_derived()
  {
    cout << "Деструктор производного класса D_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << B_derived::a_data << ' ';
    B_derived::print();
    cout << b_data << endl;
    cout << C_derived::a_data << ' ';
    C_derived::print();
    cout << c_data << endl;
    cout << d_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double d_data;
};
int main()
{
  cout << "sizeof(A_based) = " << sizeof(A_based) << endl;
  cout << "sizeof(B_derived) = " << sizeof(B_derived) << endl;
  cout << "sizeof(C_derived) = " << sizeof(C_derived) << endl;
  cout << "sizeof(D_derived) = " << sizeof(D_derived) << endl;
  D_derived d(1, 2, 3, 4);
```

```
  d.print();
  d.B_derived::print();
  d.C_derived::print();
  return 0;
}
```

Результат работы программы:

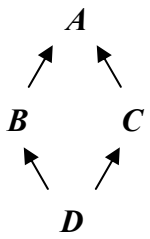
```
sizeof(A_based) = 4
sizeof(B_derived) = 8
sizeof(C_derived) = 8
sizeof(D_derived) = 24
Конструктор базового класса A_based
Конструктор производного класса B_derived
Конструктор базового класса A_based
Конструктор производного класса C_derived
Конструктор производного класса D_derived
1 1 1
2
2
1 1 1
3
3
4
1 1
2
1 1
3
Деструктор производного класса D_derived
Деструктор производного класса C_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
```

Непрямой базовый класс ***A_based*** в соответствии с размером своего компонента данных имеет размер 4 байта (32-х разрядная архитектура). Размер производного класса ***B_derived*** равен сумме размера своего компонента данных (4 байта) и размера наследуемого базового класса ***A_based*** (4 байта). Размер производного класса ***C_derived*** равен сумме размера своего компонента данных (4 байта) и размера наследуемого базового класса ***A_based*** (4 байта). Размер производного класса ***D_derived*** равен сумме размера своего компонента данных (8 байт) и размеров наследуемых классов ***B_derived*** (8 байт) и ***C_derived*** (8 байт).

Чтобы устранить многократное тиражирование компонентных данных какого-либо непрямого базового класса при множественном наследовании, этот базовый класс объявляют ***виртуальным*** с помощью спецификатора ***virtual***.

Например, непрямой базовый класс ***A*** будет виртуальным при таком определении:

```
class A { компоненты_класса_A };
class B : public virtual A { компоненты_класса_B };
class C : public virtual A { компоненты_класса_C };
class D : public B, public C { компоненты_класса_D };
```



Теперь объект производного класса ***D*** будет включать только один объект виртуального базового класса ***A***, доступ к которому равноправно разделяют его производные классы ***B*** и ***C***. Если из класса ***D*** необходим доступ к компонентным данным класса ***A***, то ссылки на них теперь уже могут быть и не квалифицированы.

Напомним здесь, что для инициализации компонентных данных непрямых базовых классов конструкторы производных классов с помощью списка инициализации должны передавать конструкторам своих прямых базовых классов только те аргументы, которые им требуются. И здесь не так важно, какой именно конструктор иерархии классов будет использоваться для создания конкретного объекта. Если какой-либо непрямой базовый класс становится виртуальным, то инициализация его компонентных данных может быть выполнена только тем конструктором иерархии классов, объект которого как раз и создается.

Приведем пример открытого множественного наследования компонентов виртуального базового класса ***A_based*** и двух прямых базовых классов ***B_derived*** и ***C_derived*** их производным классом ***D_derived***:

```
// Пример 41
// C++ Множественное виртуальное наследование
#include <iostream>
using namespace std;
struct A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
  // Деструктор объектов базового класса
  ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // Визуализация компонента базового класса
  void print()
  {
```

```
    cout << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
struct B_derived : virtual A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(int i = 0, long l = 0) : A_based(i), b_data(l)
  {
    cout << "Конструктор производного класса B_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~B_derived()
  {
    cout << "Деструктор производного класса B_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
    cout << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  long b_data;
};
struct C_derived : virtual A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  C_derived(int i = 0, float f = 0.0) : A_based(i), c_data(f)
  {
    cout << "Конструктор производного класса C_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~C_derived()
  {
    cout << "Деструктор производного класса C_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
    cout << c_data << endl;
  }
```

```
  protected:
  // Компонентные данные - все защищенные (protected)
  float c_data;
};
struct D_derived : B_derived, C_derived {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  D_derived(int i = 0, long l = 0, float f = 0.0, double d = 0.0)
  : A_based(i), B_derived(i, l), C_derived(i, f), d_data(d)
  {
    cout << "Конструктор производного класса D_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~D_derived()
  {
    cout << "Деструктор производного класса D_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    A_based::print();
    cout << a_data << ' ';
    B_derived::print();
    cout << b_data << endl;
    C_derived::print();
    cout << c_data << endl;
    cout << d_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double d_data;
};
int main()
{
  cout << "sizeof(A_based) = " << sizeof(A_based) << endl;
  cout << "sizeof(B_derived) = " << sizeof(B_derived) << endl;
  cout << "sizeof(C_derived) = " << sizeof(C_derived) << endl;
  cout << "sizeof(D_derived) = " << sizeof(D_derived) << endl;
  A_based a(1);
  a.print();
  B_derived b(2, 3);
  b.print();
  C_derived c(4, 5);
  c.print();
  D_derived d(6, 7, 8, 9);
  d.print();
  return 0;
}
```

Результат работы программы:

```
sizeof(A_based) = 4
sizeof(B_derived) = 12
sizeof(C_derived) = 12
sizeof(D_derived) = 28
Конструктор базового класса A_based
1
Конструктор базового класса A_based
Конструктор производного класса B_derived
2 2
3
Конструктор базового класса A_based
Конструктор производного класса C_derived
4 4
5
Конструктор базового класса A_based
Конструктор производного класса B_derived
Конструктор производного класса C_derived
Конструктор производного класса D_derived
6
6 6 6
7
7
6 6
8
8
9
Деструктор производного класса D_derived
Деструктор производного класса C_derived
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор производного класса C_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор базового класса A_based
```

Наличие виртуального базового класса ***A_based*** привело к увеличению размеров производных классов ***B_derived*** и ***C_derived*** на 4 байта, которые необходимы для связи в иерархии виртуальных классов (указатель на разделяемый производными классами объект виртуального класса). Размер производного класса ***D_derived*** равен сумме размера наследуемого базового класса ***A_based*** (4 байта), размера своего компонента (8 байт) и размеров наследуемых классов ***B_derived*** (4 байта + 4 байта для связи класса) и ***C_derived*** (4 байта + 4 байта для связи класса).

При множественном виртуальном наследовании конструктору производного класса разрешается явно вызывать не только конструкторы его прямых базовых классов, но также и конструкторы их предшествующих классов в иерархии.

Так как при создании объектов производных классов конструкторы наследуемых виртуальных базовых классов иерархии всегда вызываются только один раз, то здесь, например, при создании объекта производного класса ***D_derived*** будут подавлены оба неявных вызова конструктора по умолчанию виртуального базового класса ***A_based*** при вызове конструкторов прямых базовых классов ***B_derived*** и ***C_derived***. Тем самым налицо очевидная избыточность при передаче аргументов конструкторам производных классов ***B_derived*** и ***C_derived***. Это означает, что при множественном виртуальном наследовании следует различать два состояния у производного класса: либо он является так называемым ближайшим производным классом в иерархии (когда создается его объект), либо промежуточным производным классом (когда создается не его объект).

Приведем теперь пример открытого множественного наследования компонентов виртуального базового класса ***A_based*** и двух прямых базовых классов ***B_derived*** и ***C_derived*** их производным классом ***D_derived***, где с помощью защищенных конструкторов по умолчанию решается проблема избыточности при передаче аргументов конструкторам классов, когда они выступают в роли промежуточных производных классов:

```
// Пример 42
// C++ Множественное виртуальное наследование
#include <iostream>
using namespace std;
struct A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
  // Деструктор объектов базового класса
  ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // Визуализация компонента базового класса
  void print()
  {
    cout << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
```

```
struct B_derived : virtual A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов ближайшего производного класса
  B_derived(int i, long l) : A_based(i), b_data(l)
  {
    cout << "Конструктор производного класса B_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~B_derived()
  {
    cout << "Деструктор производного класса B_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
    cout << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  long b_data;
  // Компонентные функции – все защищенные (protected)
  // Конструктор объектов промежуточного производного класса
  B_derived()
  {
    cout << "Конструктор производного класса B_derived*" << endl;
  }
};
struct C_derived : virtual A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов ближайшего производного класса
  C_derived(int i = 0, float f = 0.0) : A_based(i), c_data(f)
  {
    cout << "Конструктор производного класса C_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~C_derived()
  {
    cout << "Деструктор производного класса C_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    cout << a_data << ' ';
    A_based::print();
    cout << c_data << endl;
  }
```

```
  protected:
  // Компонентные данные - все защищенные (protected)
  float c_data;
  // Компонентные функции – все защищенные (protected)
  // Конструктор объектов промежуточного производного класса
  C_derived(float f = 0.0) : c_data(f)
  {
    cout << "Конструктор производного класса C_derived*" << endl;
  }
};
struct D_derived : B_derived, C_derived {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  D_derived(int i = 0, long l = 0, float f = 0.0, double d = 0.0)
  : A_based(i), C_derived(f), d_data(d)
  {
    cout << "Конструктор производного класса D_derived" << endl;
    b_data = l;
  }
  // Деструктор объектов производного класса
  ~D_derived()
  {
    cout << "Деструктор производного класса D_derived" << endl;
  }
  // Визуализация компонентов классов
  void print()
  {
    A_based::print();
    cout << a_data << ' ';
    B_derived::print();
    cout << b_data << endl;
    C_derived::print();
    cout << c_data << endl;
    cout << d_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double d_data;
};
int main()
{
  A_based a(1);
  a.print();
  B_derived b(2, 3);
  b.print();
  C_derived c(4, 5);
  c.print();
```

```
  D_derived d(6, 7, 8, 9);
  d.print();
  return 0;
}
```

Результат работы программы:

```
Конструктор базового класса A_based
1
Конструктор базового класса A_based
Конструктор производного класса B_derived
2 2
3
Конструктор базового класса A_based
Конструктор производного класса C_derived
4 4
5
Конструктор базового класса A_based
Конструктор производного класса B_derived*
Конструктор производного класса C_derived*
Конструктор производного класса D_derived
6
6 6 6
7
7
6 6
8
8
9
Деструктор производного класса D_derived
Деструктор производного класса C_derived
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор производного класса C_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор базового класса A_based
```

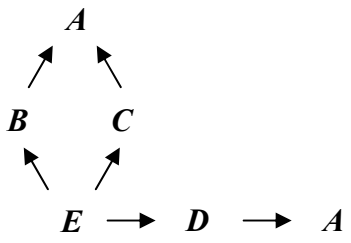
А теперь обратимся к примеру открытого множественного наследования компонентов виртуального базового класса ***Complex_A*** и двух прямых базовых классов ***Complex_B*** и ***Complex_C*** их производным классом ***Complex_D***, где выбор между одноименными компонентными функциями ***add*()** прямых базовых классов осуществляется по типам аргументов вызова благодаря ***using-объявлениям*** этих функций в производном классе ***Complex_D***:

```
// Пример 43
// C++ Множественное виртуальное наследование
#include <iostream>
using namespace std;
struct Complex_A {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  Complex_A(double r = 0.0, double i = 0.0) : re(r), im(i) {}
  // Визуализация комплексного числа
  void print()
  {
    cout << '(' << re << ", " << im << ')' << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  double re;
  double im;
};
struct Complex_B : virtual Complex_A {
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_B(double r = 0.0, double i = 0.0) : Complex_A(r, i) {}
  // Сложение комплексных чисел
  Complex_B& add(Complex_B a, Complex_B b)
  {
    re = a.re + b.re;
    im = a.im + b.im;
    return *this;
  }
};
struct Complex_C : virtual Complex_A {
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_C(double r = 0.0, double i = 0.0) : Complex_A(r, i) {}
  // Сложение комплексных чисел
  Complex_C& add(Complex_C a, double b)
  {
    re = a.re + b;
    return *this;
  }
};
struct Complex_D : Complex_B, Complex_C {
  using Complex_B::add;
  using Complex_C::add;
  // Компонентные функции – все общедоступные (public)
  // Конструктор объектов производного класса
  Complex_D(double r = 0.0, double i = 0.0)
  : Complex_A(r, i), Complex_B(r, i), Complex_C(r, i) {}
```

```
  // Сложение комплексных чисел
  Complex_D& add(double a, Complex_D b)
  {
    re = b.re + a;
    return *this;
  }
};
int main()
{
  Complex_D x1(-1, 5);
  Complex_D x2(10, 7);
  x1.print();                          // (-1, 5)
  x2.print();                          // (10, 7)
  x1.add(x1, x2);
  x1.print();                          // (9, 12)
  x1.add(x1, 2);
  x1.print();                          // (11, 12)
  x2.add(2, x2);
  x2.print();                          // (12, 7)
  return 0;
}
```

При множественном наследовании один и тот же базовый класс может быть включен в производный класс одновременно несколько раз и как виртуальный, и как невиртуальный, например, при таком определении:

```
  class A { компоненты_класса_A };
  class B : public virtual A { компоненты_класса_B };
  class C : public virtual A { компоненты_класса_C };
  class D : public A { компоненты_класса_D };
  class E : public B, public C, public D {
    компоненты_класса_E };
```



В заключение отметим, что если какой-либо класс разработан с целью сделать его базовым (даже если он пока и не используется в качестве такового), он должен иметь открытый виртуальный деструктор. Такой класс не влияет на других пользователей иерархии классов, если они добавляют в нее новые производные от него классы, имеющих деструкторы. Отметим здесь также, что виртуальные базовые классы должны иметь конструкторы по умолчанию, в противном случае с такими классами очень сложно работать. И, наконец, ромбовидное наследование, называемое иногда “ужасным бриллиантом наследования” будет хорошо управляемым, если

виртуальный базовый класс либо классы, производные непосредственно от него, являются абстрактными. Далее будет сказано, что класс становится абстрактным, если в нем объявляется хотя бы одна чисто виртуальная компонентная функция (например, деструктор).

Виртуальные функции

К механизму виртуальных функций обращаются в тех случаях, когда в базовый класс необходимо поместить компонентную функцию, которая должна по-разному выполняться в производных классах, т.е. в каждом производном классе требуется свой вариант такой функции.

Виртуальные функции используются для поддержки ***динамического полиморфизма***. В C++ полиморфизм поддерживается тремя способами: при компиляции программы полиморфизм поддерживается посредством перегрузки операторов и функций (статический полиморфизм) или посредством механизма шаблонов с использованием типа в качестве параметра при определении семейства функций или классов (параметрический полиморфизм), а во время выполнения программы полиморфизм поддерживается посредством виртуальных функций (динамический полиморфизм). Основой виртуальных функций и динамического полиморфизма являются ссылки и указатели на объекты производных классов.

По существу, виртуальная функция реализует идею “один интерфейс, множество методов”, которая лежит в основе полиморфизма. Виртуальная функция в базовом классе задает свой ***интерфейс*** замещающим ее функциям, определенным в производных классах, а переопределение этой виртуальной функции в каждом производном классе определяет ее ***реализацию***, связанную со спецификой производного класса. Таким образом, переопределение виртуальной функции создает конкретный ***метод***. Классы, в которых объявлены виртуальные функции, называются ***полиморфными***. Чтобы объявление виртуальной функции работало в качестве интерфейса к функции производного класса, типы аргументов замещающей функции не должны отличаться от типов аргументов виртуальной функции базового класса, и только некоторые ослабления допускаются к типу возвращаемого значения.

Отметим здесь, что традиционной реализацией вызова виртуальных функций является косвенный вызов функции. Для каждого класса с виртуальными функциями компилятор строит свою таблицу указателей на его виртуальные функции (***virtual function table***). Вызов виртуальной функции выполняется по ее индексу в таблице. Следует признать, что накладные расходы такого вызова могут оказаться немалыми.

Рассмотрим поведение кода при наследовании невиртуальных компонентных функций с одинаковыми именами, типами возвращаемых значений и сигнатурами параметров. Если в базовом классе определена некоторая компонентная функция, то такая же функция (с тем же именем, того же типа и с той же сигнатурой параметров) может быть введена и в производном классе. Полные квалифицированные имена позволяют в теле какого-либо класса однозначно получить доступ к таким функциям. Если обращения к ним выполнены с помощью ссылок или указателей на объекты соответствующих классов, то выбор функции зависит только от типа ссылки или указателя, но не от их значения.

Итак, выбор требуемой невиртуальной компонентной функции определяется уже при кодировании и не изменяется после компиляции – такой режим называется ***ранним*** или ***статическим связыванием*** (от слов ***early binding*** или ***static binding***).

Проиллюстрируем сказанное следующим примером для невиртуальной нестатической компонентной функции ***print()***, которая определяется в базовом классе ***A_based*** и переопределяется в производном от него классе ***B_derived***:

```
// Пример 44
// C++ Раннее (статическое) связывание
#include <iostream>
using namespace std;
struct A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i) {}
  // Визуализация компонента данных
  void print()
  {
    cout << "A_based::data_member = " << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
struct B_derived : A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(int i = 0) : b_data(i) {}
  // Визуализация компонента данных
  void print()
  {
    cout << "B_derived::data_member = " << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int b_data;
};
int main()
{
  A_based a(1);
  B_derived b(2);
  A_based* pointerA_based = &a;
  B_derived* pointerB_derived = &b;
  a.print();
  pointerA_based->print();
  b.print();
  pointerB_derived->print();
```

```
  pointerA_based = &b;
  pointerA_based->print();
  A_based& referenceA_based = b;
  referenceA_based.print();
  return 0;
}
```

Результат работы программы:

```
A_based::data_member = 1
A_based::data_member = 1
B_derived::data_member = 2
B_derived::data_member = 2
A_based::data_member = 0
A_based::data_member = 0
```

Как видим, вызовы компонентной функции ***print*()**, выполненные с помощью указателей на объекты классов ***A_based*** и ***B_derived***, а также ссылки на объект класса ***B_derived*** зависят только от типа указателя и ссылки, но не от их значения.

Наряду с режимом раннего или статического связывания существует и режим ***позднего*** (***отложенного***) или ***динамического связывания*** (от слов ***late binding*** или ***dynamic binding***), который предоставляется механизмом виртуальных функций. Любая нестатическая компонентная функция базового класса может стать виртуальной, если в ее объявлении используется спецификатор ***virtual***. В этом случае интерпретация вызова виртуальной функции через ссылку или указатель на базовый класс будет зависеть от значения ссылки или указателя, т.е. от типа объекта, для которого как раз и выполняется вызов. Этот процесс является реализацией принципа динамического полиморфизма. Говорят, что функция производного класса с тем же именем, того же типа и с той же сигнатурой параметров, что и виртуальная функция базового класса, замещает виртуальную функцию базового класса.

Например, в базовом классе ***A_based*** объявим виртуальной нестатическую компонентную функцию ***print*()**:

```
// Пример 45
// C++ Позднее (динамическое) связывание
#include <iostream>
using namespace std;
struct A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i) {}
  // Визуализация компонента данных
  virtual void print()
  {
    cout << "A_based::data_member = " << a_data << endl;
  }
```

```
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
struct B_derived : A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(int i = 0) : b_data(i) {}
  // Визуализация компонента данных
  void print()
  {
    cout << "B_derived::data_member = " << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int b_data;
};
int main()
{
  A_based a(1);
  B_derived b(2);
  A_based* pointerA_based = &a;
  B_derived* pointerB_derived = &b;
  a.print();
  pointerA_based->print();
  b.print();
  pointerB_derived->print();
  pointerA_based = &b;
  pointerA_based->print();
  A_based& referenceA_based = b;
  referenceA_based.print();
  return 0;
}
```

Результат работы программы:

```
A_based::data_member = 1
A_based::data_member = 1
B_derived::data_member = 2
B_derived::data_member = 2
B_derived::data_member = 2
B_derived::data_member = 2
```

Как видим, вызовы компонентной функции ***print*()**, выполненные с помощью указателей на объекты классов ***A_based*** и ***B_derived***, а также ссылки на объект класса ***B_derived*** зависят теперь от значения указателя и ссылки, т.е. от типа объекта, для которого как раз и выполнялся вызов.

Отметим характерные особенности механизма виртуальных функций:

- виртуальная функция является нестатической компонентной функцией класса, она объявляется в базовом классе и если этот класс наследуется, то эта функция переопределяется в производном классе;
- виртуальная функция должна быть определена для класса, в котором она впервые объявлена (если она не объявлена как чистая виртуальная функция, в этом случае класс называют абстрактным);
- виртуальная функция может быть объявлена дружественной в другом классе;
- виртуальную функцию (если она не объявлена как чистая виртуальная функция) можно вызывать, даже если у ее класса нет производных классов;
- переопределение виртуальной функции в производном классе создает в этом классе новую виртуальную функцию, причем использование спецификатора ***virtual*** в этом случае не обязательно;
- виртуальная функция может вызываться так же, как и обычная компонентная функция; самым распространенным является вызов виртуальных функций через указатель базового класса на объекты производных от него классов; компилятор определяет какую версию виртуальной функции вызвать на основе типа объекта, на который указывает этот указатель;
- механизм виртуального вызова может быть подавлен в случае использования полной квалификации; если виртуальная функция была объявлена встраиваемой, то только такой способ вызова может привести к ее встраиванию, что, например, важно, когда одна виртуальная функция вызывает другую с тем же объектом;
- если виртуальная функция не переопределена в производном классе, то при ее вызове через указатель базового класса, указывающего на объект производного класса, используется ее версия из базового класса;
- определение в производном классе функции с тем же именем и с той же сигнатурой параметров, но с другим типом возвращаемого значения, чем у виртуальной функции базового класса, приводит к ошибке времени компиляции;
- определение в производном классе функции с тем же именем и типом возвращаемого значения, что и виртуальная функция базового класса, но с другой сигнатурой параметров, приводит к потере ее виртуальной природы, т.е. эта функция производного класса не будет уже виртуальной, а будет его новой компонентной функцией, а именно – перегружаемой;
- деструкторы могут быть виртуальными функциями, а конструкторы не могут.

Здесь следует отметить, что тип замещающей функции должен быть такой же, как тип виртуальной функции, которую она замещает, за исключением того, что допускаются некоторые ослабления по отношению к типу возвращаемого значения. Например, если исходный тип возвращаемого значения был ***имя_базового_класса****, то тип возвращаемого значения замещающей функции может быть ослаблен до ***имя_производного_класса**** при условии, что класс ***имя_базового_класса*** является открытым базовым классом для класса ***имя_производного_класса***. Аналогично, вместо ***имя_базового_класса&*** тип возвращаемого значения может быть ослаблен до ***имя_производного_класса&***.

Проиллюстрируем одно из правил “ослабления типа” на примере так называемых “виртуальных конструкторов” – виртуальных компонентных функций класса, неявно вызывающих его конструкторы и возвращающих созданные объекты. Зачастую это необходимо для создания объектов, точный тип которых не известен. Представим иерархию классов с открытым одиночным наследованием, где “виртуальные конструкторы” ***newA_based*()** и ***cloneA_based*()** определяются в базовом классе ***A_based*** и переопределяются в производном от него классе ***B_derived***:

```
// Пример 46
// C++ "Виртуальные конструкторы"
#include <iostream>
using namespace std;
struct A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
  // Деструктор объектов базового класса
  virtual ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // "Виртуальный конструктор" базового класса
  virtual A_based* newA_based()
  {
    return new A_based;
  }
  // "Виртуальный конструктор копирования" базового класса
  virtual A_based* cloneA_based()
  {
    return new A_based(*this);
  }
  // Визуализация компонента данных
  virtual void print()
  {
    cout << a_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
struct B_derived : A_based {
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  B_derived(float f = 0.0) : b_data(f)
  {
```

```
    cout << "Конструктор производного класса B_derived" << endl;
  }
  // Деструктор объектов производного класса
  ~B_derived()
  {
    cout << "Деструктор производного класса B_derived" << endl;
  }
  // "Виртуальный конструктор" производного класса
  B_derived* newA_based()
  {
    return new B_derived;
  }
  // "Виртуальный конструктор копирования" производного класса
  B_derived* cloneA_based()
  {
    return new B_derived(*this);
  }
  // Визуализация компонента данных
  void print()
  {
    cout << b_data << endl;
  }
  protected:
  // Компонентные данные - все защищенные (protected)
  float b_data;
};
int main()
{
  A_based a(1);
  A_based* p = &a;
  p->print();
  A_based* q = p->newA_based();
  q->print();
  A_based* r = p->cloneA_based();
  r->print();
  delete q;
  B_derived b(2);
  B_derived* s = &b;
  s->print();
  B_derived* t = s->newA_based();
  t->print();
  B_derived* u = s->cloneA_based();
  u->print();
  delete t;
  return 0;
}
```

Результат работы программы:

```
Конструктор базового класса A_based
1
Конструктор базового класса A_based
0
1
Деструктор базового класса A_based
Конструктор базового класса A_based
Конструктор производного класса B_derived
2
Конструктор базового класса A_based
Конструктор производного класса B_derived
0
2
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор базового класса A_based
```

Как видим, значения, возвращаемые "виртуальными конструкторами" ***B_derived*::*newA_based*()** и ***B_derived*::*cloneA_based*()**, имеют тип ***B_derived****, а не ***A_based****. Это как раз и позволяет создавать копии объектов класса ***B_derived*** без потери информации о типе.

Отметим здесь, что класс с "виртуальными конструкторами" иногда называют ***фабрикой***, и, как правило, такие классы являются абстрактными классами, где "виртуальные конструкторы" представляют собой чисто виртуальные функции. Каждая такая функция фабрики после ее переопределения в производном классе выполняет заданную операцию конструирования объекта с требуемым интерфейсом и типом реализации. Обычно конструирование объектов с заданными свойствами осуществляется не без помощи указателя на такую фабрику.

В заключение приведем пример реализации идеи "один интерфейс, множество методов", который в дальнейшем станет основой для обобщения, связанного с переходом к ***абстрактным базовым классам***.

Зададимся целью вычисления площади какой-либо геометрической фигуры по двум ее размерностям, например, прямоугольника или треугольника. Создадим базовый класс ***Area***, компонентные данные которого определяют две размерности фигуры. В базовом классе определим две компонентные функции: нестатическую функцию ***set*()** для задания двух размерностей фигуры и виртуальную функцию ***get_area*()**, которая при переопределении в производном классе возвращает значение площади фигуры, вид которой определяется производным классом. В этом случае определение ***get_area*()** в базовом классе задает интерфейс, а конкретные реализации, т.е. методы, определяются производными классами ***Rectangle*** и ***Triangle***, каждый из которых наследует базовый класс ***Area***. Например, проиллюстрируем реализацию этой идеи с помощью иерархии классов с открытым одиночным наследованием:

```
// Пример 47
// C++ Реализация идеи "один интерфейс, множество методов"
#include <iostream>
using namespace std;
class Area {
  protected:
  // Компонентные данные - все защищенные (protected)
  double dimension1;
  double dimension2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Задание размерностей фигуры
  void set(double figureD1 = 1.0, double figureD2 = 1.0)
  {
    dimension1 = figureD1;
    dimension2 = figureD2;
  }
  // Интерфейс - площадь фигуры
  virtual double get_area()
  {
    return 0.0;
  }
};
class Rectangle : public Area {
  public:
  // Компонентные функции - все общедоступные (public)
  // Метод - площадь прямоугольника
  double get_area()
  {
    return dimension1 * dimension2;
  }
};
class Triangle : public Area {
  public:
  // Компонентные функции - все общедоступные (public)
  // Метод - площадь треугольника
  double get_area()
  {
    return dimension1 * dimension2 / 2;
  }
};
int main()
{
  Rectangle r;
  Triangle t;
  r.set(1.2, 3.4);
  t.set(1.2, 3.4);
  Area* p = &r;
```

```
  cout << "Площадь прямоугольника = " << p->get_area() << endl;
  p = &t;
  cout << "Площадь треугольника = " << p->get_area() << endl;
  return 0;
}
```

Результат работы программы:

```
Площадь прямоугольника = 4.08
Площадь треугольника = 2.04
```

Как видим, виртуальная функция, объявленная в базовом классе, может и не выполнять никаких значимых действий, конкретные действия выполняются лишь благодаря механизму переопределения виртуальной функции базового класса в каждом его производном классе. Это вполне обычная ситуация, базовый класс просто содержит базовый набор компонентов, для которых производный класс задает все недостающее. При этом, как видим, можно даже обойтись и без конструкторов, определяемых самим пользователем, полагаясь лишь на те, которые генерируются компилятором по умолчанию.

Итак, если в виртуальной функции базового класса отсутствует значимое действие, то хотя бы в одном из его производных классов эта функция обязательно должна быть переопределена. В C++ реализация этого принципа обеспечивается механизмом ***чисто виртуальных функций*** (от слов ***pure virtual function***), называемых также иногда исключительно виртуальными функциями.

Абстрактные классы

Абстрактным классом (от слов ***abstract class***) называется класс, в котором есть хотя бы одна чисто виртуальная функция. Нередко чисто виртуальным оказывается деструктор класса. Чисто виртуальные функции не определяются в абстрактном классе, они представлены в нем лишь своими прототипами.

Для объявления чисто виртуальной функции используется общая форма:

virtual имя_типа имя_функции
(список_формальных_параметров_функции) = 0;

Использование инициализатора функции, равного нулю, сообщает компилятору, что не существует тела функции. Так как в этом случае определение класса будет неполным, это означает, что невозможно создать ни одного объекта абстрактного класса, т.е. абстрактные классы могут быть только наследуемыми. Если функция задается как чисто виртуальная в каком-либо классе иерархии, это предполагает, что она должна переопределяться хотя бы в одном из его производных классов. Если этого нет, то при компиляции возникнет ошибка.

Таким образом, создание чисто виртуальных функций гарантирует, что производные классы обеспечат их переопределение. Чисто виртуальная функция недоступна для вызовов, она служит лишь основой для замещающих ее функций в производных классах. В терминологии ООП такое “отложенное” решение о реализации функции называется ***отсроченным методом***.

Чисто виртуальная функция, если ее не переопределить в производном классе, так и останется чисто виртуальной, поэтому такой класс тоже является абстрактным. Это позволяет теперь строить реализацию программы поэтапно в стиле модульного программирования с использованием классов в качестве строительных блоков.

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. Критерий общности, например, может отражать сходство наборов операций или сходство реализаций. Эти общие понятия невозможно использовать непосредственно, но на их основе можно построить производные классы, пригодные для описания конкретных объектов.

Важным примером использования абстрактных классов является предоставление интерфейса с полным отсутствием каких-либо деталей реализации. Детали оставляют специализированным производным классам, от которых непосредственно зависит реализация общих понятий. Абстрактный класс – идеальное средство для объявления указателей, которые затем будут использоваться в производных от него классах. Допускаются также и ссылки на абстрактный класс.

Отметим характерные особенности механизма абстрактных классов:

- абстрактный класс нельзя использовать для задания типа параметра функции или в качестве типа возвращаемого функцией значения;
- формальным параметром функции может быть указатель на абстрактный класс, в этом случае значение указателя на производный объект как фактический параметр при вызове функции заменит указатель на абстрактный базовый класс;
- абстрактный класс нельзя использовать при явном приведении типов;
- абстрактный класс обычно не нуждается в конструкторе, тем не менее, абстрактный класс может иметь явно определенный конструктор; из конструктора возможен вызов компонентных функций класса, но любые прямые или опосредованные обращения к чисто виртуальным функциям приведут к ошибкам времени выполнения программы.

Продолжая развивать реализацию идеи “один интерфейс, множество методов”, приведем теперь еще два примера открытого одиночного наследования компонентов абстрактного базового класса ***Area*** его производными классами ***Rectangle*** и ***Triangle***:

```
// Пример 48
// C++ Абстрактный класс
#include <iostream>
using namespace std;
class Area {
  protected:
  // Компонентные данные - все защищенные (protected)
  double dimension1;
  double dimension2;
  public:
  // Компонентные функции - все общедоступные (public)
  // Задание размерностей фигуры
  void set(double figureD1 = 1.0, double figureD2 = 1.0)
  {
```

```
    dimension1 = figureD1;
    dimension2 = figureD2;
  }
  // Интерфейс - площадь фигуры
  virtual double get_area() = 0;
};
class Rectangle : public Area {
  public:
  // Компонентные функции - все общедоступные (public)
  // Метод - площадь прямоугольника
  double get_area()
  {
    return dimension1 * dimension2;
  }
};
class Triangle : public Area {
  public:
  // Компонентные функции - все общедоступные (public)
  // Метод - площадь треугольника
  double get_area()
  {
    return dimension1 * dimension2 / 2;
  }
};
int main()
{
  Rectangle r;
  Triangle t;
  r.set(1.2, 3.4);
  t.set(1.2, 3.4);
  Area* p = &r;
  cout << "Площадь прямоугольника = " << p->get_area() << endl;
  p = &t;
  cout << "Площадь треугольника = " << p->get_area() << endl;
  return 0;
}
```

Результат работы программы:

```
Площадь прямоугольника = 4.08
Площадь треугольника = 2.04
```

Очевидно, что именно такой способ перехода к иерархии с абстрактным классом является всего лишь одним из первых шагов на пути представления общих понятий. Для критерия общности, основанного здесь на сходстве набора операций, можно предложить еще и другой шаг, связанный, например, с реализацией операции инициализации компонентных данных с помощью конструкторов:

```
// Пример 49
// C++ Абстрактный класс
#include <iostream>
using namespace std;
class Area {
  public:
  // Компонентные функции - все общедоступные (public)
  // Интерфейс - площадь фигуры
  virtual double get_area() = 0;
  protected:
  // Компонентные данные - все защищенные (protected)
  double dimension1;
  double dimension2;
  // Компонентные функции - все защищенные (protected)
  // Конструктор базового абстрактного класса
  Area(double figureD1, double figureD2)
  : dimension1(figureD1), dimension2(figureD2) {}
};
class Rectangle : public Area {
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  Rectangle(double figureD1 = 1.0, double figureD2 = 1.0)
  : Area(figureD1, figureD2) {}
  // Метод - площадь прямоугольника
  double get_area()
  {
    return dimension1 * dimension2;
  }
};
class Triangle : public Area {
  public:
  // Компонентные функции - все общедоступные (public)
  // Конструктор объектов производного класса
  Triangle(double figureD1 = 1.0, double figureD2 = 1.0)
  : Area(figureD1, figureD2) {}
  // Метод - площадь треугольника
  double get_area()
  {
    return dimension1 * dimension2 / 2;
  }
};
int main()
{
  Rectangle r(1.2, 3.4);
  Triangle t(1.2, 3.4);
  Area* p = &r;
  cout << "Площадь прямоугольника = " << p->get_area() << endl;
```

```
  p = &t;
  cout << "Площадь треугольника = " << p->get_area() << endl;
  return 0;
}
```

Результат работы программы:

```
Площадь прямоугольника = 4.08
Площадь треугольника = 2.04
```

Иерархии классов и абстрактные классы

Абстрактный класс является интерфейсом, иерархия классов – средством последовательного построения классов. Интерфейс и строительные блоки – вот главные роли абстрактных классов и иерархии классов.

Классическая иерархия (или традиционная иерархия классов, существующая в языках Simula, Smalltalk и ранних программах на C++) – это иерархия, в которой отдельные классы предоставляют пользователям полезные функции и одновременно являются строительными блоками для реализации более мощных или специализированных классов. Такие иерархии классов идеальны для поддержки программирования методом последовательных усовершенствований. Они предоставляют максимум поддержки для реализации новых классов, в то время как новые классы в значительной степени используют существующую иерархию.

Классические иерархии имеют тенденцию смешивать вопросы реализации с интерфейсами для пользователей. В этих случаях на помощь приходят абстрактные классы. Иерархии абстрактных классов предоставляют ясный и мощный способ выражения концепций, не загроможденных вопросами реализации, и при этом не приводят к значительным накладным расходам, так как вызов чисто виртуальной функции не дороже вызова любой другой виртуальной функции.

Проектирование на основе абстрактных классов почти настолько же просто, как эквивалентный метод с использованием общего базового класса. Однако у первого подхода есть неоспоримое преимущество – пользователи иерархии абстрактных классов меньше рискуют попасть в зависимость от способа реализации общих понятий, чем пользователи классической иерархии. Пользователи абстрактного класса не могут случайно воспользоваться механизмами реализации, потому что им доступны только те средства, которые явно указаны в иерархии этого абстрактного класса, и ничто не наследуется неявно из зависящего от реализации базового класса.

Применение динамического полиморфизма

Осознавая важность механизма виртуальных функций, обсудим теперь преимущества и недостатки использования раннего и позднего связывания.

Функции раннего связывания – это обычные функции, перегруженные функции, невиртуальные компонентные функции и функции-друзья. Главное преимущество раннего связывания (и довод в пользу его широкого использования) – обеспечение высокого быстродействия программ. Главный недостаток – потеря гибкости.

Функции позднего связывания – это виртуальные нестатические компонентные функции класса. Главное преимущество позднего связывания – гибкость во время работы программы. Главный недостаток – снижение быстродействия программ за счет более медленного вызова виртуальных функций.

Динамический полиморфизм важен потому, что может сильно упростить сложные системы. Один, хорошо спроектированный интерфейс, используется для доступа к некоторому числу различных, но связанных действий, и, таким образом, устраняется искусственная сложность. Полиморфизм позволяет сделать очевидной логическую близость схожих действий, поэтому программа становится легче для понимания и сопровождения, т.е. если связанные действия реализуются через общий интерфейс, то это действительно облегчает понимание.

Вложенные и локальные классы

Класс может быть определен внутри другого класса. Такой класс называется ***вложенным*** (***nested class***) классом. Вложение классов – это не более чем соглашение о записи, поскольку вложенный класс не является скрытым в области видимости лексически объемлющего класса. Имя вложенного класса является локальным в его объемлющем классе.

Отметим характерные особенности вложенных классов:

- объявления во вложенном классе могут использовать из объемлющего класса лишь имена типов, статические (***static***) компоненты и элементы перечислений;
- компонентные функции вложенного класса не имеют специального доступа к компонентам объемлющего класса – они подчиняются обычным правилам доступа, так же и компонентные функции объемлющего класса подчиняются обычным правилам доступа к компонентам вложенного класса;
- компонентные функции и статические компоненты данных вложенного класса могут быть определены в глобальной области видимости.

Класс может быть определен внутри блока, например, внутри определения функции. Такой класс называется ***локальным*** (***local class***) классом. Имя локального класса является локальным в окружающем контексте. Локальный класс имеет тот же доступ к именам вне функции, что и сама функция.

Отметим характерные особенности локальных классов:

- локализация класса предполагает недоступность его компонентов вне области определения класса (вне тела функции или блока, в котором он определен);
- локальный класс не может иметь статических данных, так как компоненты локального класса не могут быть определены вне контекста класса;
- в локальном классе разрешено использовать из окружающего контекста только имена типов, статические (***static***) переменные, внешние (***extern***) переменные, внешние функции и элементы перечислений;
- в локальном классе запрещено использование переменных автоматической памяти (***auto***);
- компонентные функции локального класса могут быть только встроенными;
- вложенный класс локального класса сам является локальным.

Ранее отмечалось, что при объявлении массива объектов класса явное указание аргументов вызова для его конструктора возможно не только в случае использования списка инициализации в стиле C, но также и в случае использования локального класса. Например, при объявлении одномерного массива объектов локального производного класса ***B_derived*** из иерархии с открытым одиночным наследованием инициализация его элементов будет выполнена благодаря явному вызову конструктора базового класса ***A_based*** для указанных аргументов:

```
// Пример 50
// C++ Локальный класс
#include <iostream>
using namespace std;
class A_based {
  protected:
  // Компонентные функции - все защищенные (protected)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i)
  {
    cout << "Конструктор базового класса A_based" << endl;
  }
  // Деструктор объектов базового класса
  ~A_based()
  {
    cout << "Деструктор базового класса A_based" << endl;
  }
  // Визуализация компонента данных
  friend ostream& operator<<(ostream& stream, A_based& item)
  {
    return stream << item.a_data;
  }
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
const int size = 2;
void create()
{
  static int value;
  class B_derived : public A_based {
    public:
    // Компонентные функции - все общедоступные (public)
    // Конструктор объектов производного класса
    B_derived() : A_based(value++)
    {
      cout << "Конструктор производного класса B_derived" << endl;
    }
    // Деструктор объектов производного класса
    ~B_derived()
    {
```

```
      cout << "Деструктор производного класса B_derived" << endl;
    }
  };
  B_derived array[size];
  for (int i = 0; i < size; ++i) cout << array[i] << endl;
}
int main()
{
  create();
  return 0;
}
```

Результат работы программы:

```
Конструктор базового класса A_based
Конструктор производного класса B_derived
Конструктор базового класса A_based
Конструктор производного класса B_derived
0
1
Деструктор производного класса B_derived
Деструктор базового класса A_based
Деструктор производного класса B_derived
Деструктор базового класса A_based
```

Подобным образом можно инициализировать, например, также и динамические одномерные массивы объектов локального производного класса ***B_derived***:

```
// Пример 51
// C++ Локальный класс
#include <iostream>
using namespace std;
class A_based {
  protected:
  // Компонентные функции - все защищенные (protected)
  // Конструктор объектов базового класса
  A_based(int i = 0) : a_data(i) {}
  // Деструктор объектов базового класса
  virtual ~A_based() {}
  // Визуализация компонента данных
  friend ostream& operator<<(ostream& stream, A_based& item)
  {
    return stream << item.a_data;
  }
  // Компонентные данные - все защищенные (protected)
  int a_data;
};
```

```
void create(int size)
{
  static int value;
  class B_derived : public A_based {
    public:
    // Компонентные функции - все общедоступные (public)
    // Конструктор объектов производного класса
    B_derived() : A_based(value++) {}
    // Деструктор объектов производного класса
    ~B_derived() {}
  };
  B_derived* base = new B_derived[size];
  for (int i = 0; i < size; ++i) cout << base[i] << endl;
  delete[] base;
}
int main()
{
  create(2);
  return 0;
}
```

Результат работы программы:

```
0
1
```

Напомним здесь, что компилятор определяет размер удаляемого объекта на основании вызываемого деструктора. Виртуальный деструктор класса обеспечивает правильность размера, передаваемого оператору ***delete***. Известно, что правильный размер будет передаваться при соблюдении любого из трех условий:

- деструктор является виртуальным;
- указатель ссылается на настоящий тип объекта в свободной памяти;
- тип, на который ссылается указатель, имеет тот же размер, что и настоящий тип.

Как видим, здесь нет необходимости в использовании виртуального деструктора, так как для указателя ***base*** соблюдаются оба последних условия.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Шилдт Г. Теория и практика C++: Пер. с англ. – СПб.: BHV-Санкт-Петербург, 1996. – 416 с.
2. Паппас К., Мюррей У. Руководство программиста по C/C++. В 2 кн. Кн. I. – М.: “СК Пресс”, 1997. – 520 с.
3. Паппас К., Мюррей У. Руководство программиста по C/C++. В 2 кн. Кн. II. – М.: “СК Пресс”, 1997. – 452 с.
4. Сэвитч У. C++ в примерах: Пер. с англ. – М.: ЭКОМ, 1997. – 736 с.
5. Шилдт Г. Самоучитель C++: Пер. с англ. – СПб.: BHV-Санкт-Петербург, 1997. – 512 с.
6. Дейтел Х., Дейтел П. Как программировать на C++: Пер. с англ. – М.: ЗАО “Издательство БИНОМ”, 1998. – 1024 с.
7. Страуструп Б. Язык программирования C++, 3-е изд.: Пер. с англ. – СПб.; М.: “Невский Диалект” – “Изд-во БИНОМ”, 1999. – 991 с.
8. Топп У., Форд У. Структуры данных в C++: Пер. с англ. – М.: ЗАО “Издательство БИНОМ”, 1999. – 816 с.
9. Страуструп Б. Дизайн и эволюция C++: Пер. с англ. – М.: ДМК Пресс, 2000. – 448 с.

СОДЕРЖАНИЕ