

УЧЕБНОЕ / ПОСОБИЕ

Т. А. Павловская, Ю. А. Щупак

C++

Объектно-ориентированное
программирование

Практикум



 ПИТЕР®

300



С Е Р И Я

У Ч Е Б Н О Е / П О С О Б И Е

 ПИТЕР®

УЧЕБНОЕ / ПОСОБИЕ

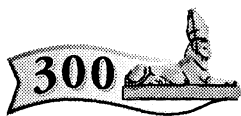
Т. А. Павловская, Ю. А. Щупак

C++

**Объектно – ориентированное
программирование**

Практикум

Допущено Министерством образования Российской Федерации в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника»



300.piter.com

Издательская программа

**300 лучших учебников для высшей школы
в честь 300-летия Санкт-Петербурга**

осуществляется при поддержке Министерства образования РФ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж · Ростов-на-Дону
Новосибирск · Екатеринбург · Самара · Киев · Харьков · Минск

2006

ББК 32.973-018.1

УДК 681.3.06

П12

Рецензенты:

Смирнова Н. Н., зав. кафедрой вычислительной техники Балтийского государственного технического университета «Военмех» им. Д.Ф. Устинова,
кандидат технических наук, профессор

Трофимов В. В., зав. кафедрой информатики Санкт-Петербургского государственного университета экономики и финансов,
доктор технических наук, профессор

Павловская Т. А., Щупак Ю. А.

П12 С++. Объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2006. — 265 с.: ил.

ISBN 5-94723-842-X

Практикум предназначен для студентов, изучающих язык С++ на семинарах или самостоятельно. Классы, шаблоны, наследование, исключения, стандартная библиотека, UML, концепции программной инженерии (software engineering) и паттерны проектирования рассматриваются на примерах, сопровождаемых необходимыми теоретическими сведениями. Обсуждаются алгоритмы, приемы отладки и вопросы качества. По каждой теме приведено по 20 вариантов заданий.

Допущено Министерством образования Российской Федерации в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению «Информатика и вычислительная техника».

ББК 32.973-018.1

УДК 681.3.06

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 5-94723-842-X

© ЗАО Издательский дом «Питер», 2006

Краткое содержание

Предисловие	9
Семинар 1. Классы	12
Семинар 2. Наследование	58
Семинар 3. Шаблоны классов. Обработка исключительных ситуаций	105
Семинар 4. Стандартные потоки	143
Семинар 5. Файловые и строковые потоки. Строки класса string	181
Семинар 6. Стандартная библиотека шаблонов	196
Приложение. Паттерны проектирования	241
Литература	260
Алфавитный указатель	261

Содержание

Предисловие	9
От издательства	11
Семинар 1. Классы	12
Появление ООП — реакция на кризис программного обеспечения	12
Критерии качества декомпозиции проекта	13
Что принесло с собой ООП	14
От структуры — к классу	16
Задача 1.1. Поиск в массиве структур	16
Отладка программы	25
Инициализаторы конструктора	27
Конструктор копирования	28
Перегрузка операций	29
Перегрузка операций инкремента	30
Перегрузка операции присваивания	31
Статические элементы класса	32
Задача 1.2. Реализация класса треугольников	33
Этап 1	34
Тестирование и отладка первой версии программы	41
Этап 2	42
Этап 3	44
Этап 4	48
Задания	53
Семинар 2. Наследование	58
Наследование классов	58
Замещение функций базового класса	59
Конструкторы и деструкторы в производном классе	60
Устранение неоднозначности при множественном наследовании	61
Доступ к объектам иерархии	62
Виртуальные методы	63
Абстрактные классы. Чисто виртуальные методы	64

Отношения между классами. Диаграммы классов на языке UML	65
Ассоциация	66
Наследование	67
Агрегация	67
Зависимость	68
Проектирование программы с учетом будущих изменений	69
Задача 2.1. Функциональный калькулятор	72
Задача 2.2. Продвинутой функциональный калькулятор	81
Задача 2.3. Работа с объектами символьных и шестнадцатеричных строк	88
Задания	102

Семинар 3. Шаблоны классов. Обработка исключительных ситуаций 105

Шаблоны классов	105
Определение шаблона класса	106
Использование шаблона класса	107
Организация исходного кода	107
Параметры шаблонов	108
Специализация	110
Использование классов функциональных объектов для настройки шаблонных классов	110
Разработка шаблонного класса для представления разреженных массивов	112
Задача 3.1. Шаблонный класс для разреженных массивов	113
Обработка исключительных ситуаций	119
Определение исключений	120
Перехват исключений	121
Неперехваченные исключения	122
Классы исключений. Иерархии исключений	123
Спецификации исключений	124
Исключения в конструкторах	125
Исключения в деструкторах	129
Задача 3.2. Шаблонный класс векторов (динамических массивов)	129
Задания	141

Семинар 4. Стандартные потоки 143

Потоковые классы	143
Классы стандартных потоков	144
Заголовочные файлы библиотеки ввода/вывода C++	144
Объекты и методы стандартных потоков ввода/вывода	144
Обработка ошибок потоков	147
Перегрузка операций извлечения и вставки для типов, определенных программистом	148
Задача 4.1. Разработка потоковых классов, поддерживающих ввод/вывод кириллицы	149
Задача 4.2. Первичный ввод и поиск информации в базе данных	166
Задания	173

Семинар 5. Файловые и строковые потоки.	
Строки класса string	181
Файловые потоки	181
Строковые потоки	185
Строки класса string	186
Задача 5.1. Подсчет количества вхождений слова в текст	189
Задача 5.2. Вывод вопросительных предложений	191
Задания	193
Семинар 6. Стандартная библиотека шаблонов	196
Основные концепции STL	196
Контейнеры	197
Итераторы	197
Общие свойства контейнеров	200
Алгоритмы	201
Использование последовательных контейнеров	202
Задача 6.1. Сортировка вектора	204
Шаблонная функция print() для вывода содержимого контейнера	205
Адаптеры контейнеров	206
Использование алгоритмов	208
Использование ассоциативных контейнеров	215
Множества	215
Словари	217
Задача 6.2. Формирование частотного словаря	218
Задача 6.3. Морской бой	220
Задания	233
Приложение. Паттерны проектирования	241
Порождающие паттерны	243
Структурные паттерны	245
Паттерны поведения	247
Паттерн Стратегия (Strategy)	252
Паттерн Компоновщик (Composite)	255
Литература	260
Алфавитный указатель	261

Предисловие

Перед вами практикум по изучению объектно-ориентированного программирования (ООП). Наша цель — научить читателя самостоятельно создавать грамотные и по возможности профессиональные программы на C++.

В подавляющем большинстве учебников по C++ излагаются конструкции языка с иллюстрациями на примерах. Любой программист, работавший над реальными проектами, понимает, что знания синтаксиса и правил выполнения операторов далеко не достаточно для того, чтобы писать программы приемлемого качества. Это особенно справедливо для такого многогранного языка, как C++.

Если вы не знакомы с основами ООП и с базовыми концепциями программной инженерии (software engineering), то написанная вами программа, если и заработает, скорее всего, будет неудобной для сопровождения и модификации, а повторное использование программного кода окажется почти невозможным. Именно поэтому вопросам программной инженерии в нашей книге уделяется особое внимание.

Вот краткий перечень этих вопросов.

- ❑ Из каких компонентов — модулей, функций, классов — должна состоять программа?
- ❑ Как распределяются обязанности между этими компонентами?
- ❑ Как компоненты программы взаимодействуют друг с другом?
- ❑ Каким критериям должен удовлетворять программный проект, чтобы его было легко сопровождать и модифицировать?
- ❑ Как применять шаблоны (паттерны) проектирования для достижения указанных целей?

Все эти проблемы рассматриваются на конкретных задачах, причем особая роль отведена первым двум семинарам, посвященным изучению базовых концепций ООП. Здесь особенно подробно разбирается процесс проектирования программы, активно используются средства отладки и тестирования. Иногда мы специально вносим в исходный текст программы ошибку или проявляем «забывчивость», чтобы продемонстрировать возникающие последствия и, кроме того, привить

читателю вкус к аналитической работе детектива, идущего по следу коварного «преступника» — программной ошибки.

Для эффективного восприятия новых технологических идей сегодня не обойтись без знания основ унифицированного языка моделирования UML, уже ставшего стандартным средством представления проектных решений. Поэтому в практикуме показано применение диаграмм классов языка UML для отображения взаимоотношений между классами.

Кроме чисто учебных задач в практикуме рассматриваются и более сложные, изучение которых, как мы надеемся, ускорит вхождение читателя в мир профессионального программирования. Например, одной из реальных проблем при разработке программного обеспечения является необходимость адаптации библиотеки стороннего производителя к потребностям заказчика. В четвертом семинаре рассматривается адаптация стандартной библиотеки Microsoft для обеспечения ввода/вывода кириллицы на платформе Windows.

Эта книга является второй частью практикума, являющегося дополнением к учебнику Т. А. Павловской «С/С++. Программирование на языке высокого уровня» («Питер», 2001, 2003)¹. В дальнейшем для краткости мы будем называть его «учебник». По содержанию книга соответствует материалам второй и третьей частей учебника, но выходит довольно далеко за его рамки.

В начале каждого семинара приведены ссылки на разделы учебника, содержащие соответствующий материал. Тем не менее при разборе текстов программ поясняются все использованные в них возможности языка, поэтому практикум можно и нужно рассматривать как самостоятельное издание. Тексты заданий на лабораторные работы частично соответствуют учебнику, а частично переработаны для более углубленного изучения материала.

Все ключевые слова, стандартные типы, константы, функции, макросы и классы, описанные в книге, можно найти по предметному указателю, что позволяет использовать ее в качестве справочника.

Средства С++, рассматриваемые в данной книге, соответствуют стандарту ANSI ISO/IEC 14882 (1998). Из распространенных компиляторов ему в достаточной степени соответствуют, например, Microsoft Visual C++ 6.0 (ОС Windows) и gcc (GNU C Compiler — ОС Linux, Cygwin — ОС Windows). Приведенные в книге примеры программ протестированы с помощью первого из указанных средств.

Практикум поддержан проектом «Разработка концепции и научно-методического обеспечения регионального центра целевой подготовки разработчиков программного обеспечения и компьютерных технологий» программы Министерства образования Российской Федерации «Государственная поддержка региональной научно-технической политики высшей школы и развития ее научного потенциала» на 2003 год. В основу книги положены семинары, проводимые авторами в Санкт-Петербургском государственном университете информационных технологий, механики и оптики (СПбГУИТМО).

¹ Первая книга практикума — «С/С++. Структурное программирование» — была выпущена издательством «Питер» в 2002 году.

Если в первой книге практикума 80% текста было написано Т. А. Павловской, а оставшиеся 20% — Ю. А. Щупаком, то в этой книге авторы поменялись ролями. Тем не менее мы несем полную равную ответственность за все недочеты, которые может найти наш взыскательный читатель. Замечания, пожелания и дополнения к практикуму будут с благодарностью приняты по адресу shupak@mail.ru.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на web-сайте издательства <http://www.piter.com>.

Семинар 1 Классы

Теоретический материал: с. 178–199.

Появление ООП — реакция на кризис программного обеспечения

Объектно-ориентированное программирование (ООП) — это технология, возникшая как реакция на очередную фазу кризиса программного обеспечения, когда методы структурного программирования уже не позволяли справляться с растущей сложностью промышленного программного продукта. Следствия — срыв сроков проектов, перерасход бюджета, урезанная функциональность и множество ошибок.

Существенная черта промышленной программы — ее *сложность*: один разработчик не в состоянии охватить все аспекты системы, поэтому в ее создании участвует целый коллектив. Следовательно, к первичной сложности самой задачи, вытекающей из предметной области, добавляется управление процессом разработки с учётом необходимости координации действий в команде разработчиков.

Так как сложные системы разрабатываются в расчете на длительную эксплуатацию, то появляются еще две проблемы: *сопровождение* системы (устранение обнаруженных ошибок) и ее *модификация*, поскольку у заказчика постоянно появляются новые требования и пожелания. Иногда затраты на сопровождение и модификацию сопоставимы с затратами на собственно разработку системы.

Способ управления сложными системами был известен еще в древности — *divide et impera* (разделяй и властвуй). То есть выход — в *декомпозиции* системы на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. Здесь вы, наверное, вспомните о методе нисходящего проектирования, которым мы активно пользовались в первой книге практикума. Но если в рамках структурного подхода декомпозиция понимается как разбиение алгоритма, когда каждый из модулей системы выполняет один из этапов общего процесса, то ООП предлагает совершенно другой подход.

Суть его в том, что в качестве *критерия декомпозиции* принимается принадлежность ее элементов к различным *абстракциям проблемной области*. Откуда же берутся эти абстракции? Исключительно из головы программиста, который, анализируя предметную область, вычленяет из нее отдельные объекты. Для каждого из этих объектов определяются свойства, существенные для решения задачи. Затем каждому реальному объекту предметной области ставится в соответствие программный объект.

Почему *объектно-ориентированная декомпозиция* оказалась более эффективным средством борьбы со сложностью процессов проектирования и сопровождения программных систем, чем *функциональная*¹ декомпозиция? Тому есть много причин. Чтобы в них разобраться, рассмотрим критерии качества проекта, связанные с его декомпозицией.

Критерии качества декомпозиции проекта

Со *сложностью приложения* трудно что-либо сделать — она определяется целью создания программы. А вот *сложность реализации* можно попытаться контролировать. Первый вопрос, возникающий при декомпозиции: на какие компоненты (модули, функции, классы) нужно разбить программу? Очевидно, что с ростом числа компонентов сложность программы растет, поскольку необходима кооперация, координация и коммуникация между компонентами. Особенно негативны последствия неоправданного разбиения на компоненты, когда оказываются разделенными действия, по сути тесно связанные между собой.

Вторая проблема связана с организацией взаимодействия между компонентами. Взаимодействие упрощается и его легче взять под контроль, если каждый компонент рассматривается как некий «черный ящик», внутреннее устройство которого неизвестно, но известны выполняемые им функции, а также «входы» и «выходы» этого ящика. Вход компонента позволяет ввести в него значение некоторой *входной переменной*, а выход — получить значение некоторой *выходной переменной*. В программировании совокупность входов и выходов черного ящика определяет *интерфейс* компонента. Интерфейс реализуется как набор некоторых функций (или запросов к компоненту), вызывая которые *клиент* либо получает какую-то информацию, либо меняет состояние компонента.

Модное нынче словечко «клиент» означает просто-напросто компонент, которому понадобились услуги другого компонента, исполняющего в этом случае роль *сервера*. Взаимоотношение *клиент/сервер* на самом деле очень старо и использовалось уже в рамках структурного программирования, когда функция-клиент пользовалась услугами функции-сервера путем ее вызова.

¹ Синонимами являются термины *структурная, процедурная и алгоритмически-ориентированная декомпозиция*.

Подытожим сказанное о проблемах разбиения программы на компоненты и организации их взаимодействия. Для оценки качества программного проекта нужно учитывать, кроме всех прочих, следующие два показателя:

- ❑ *Сцепление (cohesion)* внутри компонента — показатель, характеризующий степень взаимосвязи отдельных его частей. Простой пример: если внутри компонента решаются две подзадачи, которые легко можно разделить, то компонент обладает слабым (плохим) сцеплением.
- ❑ *Связанность (coupling)* между компонентами — показатель, описывающий интерфейс между компонентом-клиентом и компонентом-сервером. Общее число входов и выходов сервера есть мера связанности. Чем меньше связанность между двумя компонентами, тем проще понять и отслеживать в будущем их взаимодействие. А так как в больших проектах эти компоненты часто разрабатываются разными людьми, то очень важно уменьшать связанность между компонентами.

Заметим, что описанные показатели, конечно, имеют относительный характер, и пользоваться ими следует благоразумно. Например, фанатичное следование первому показателю (сильное сцепление) может привести к дроблению проекта на очень большое количество мелких функций, и сопровождающий программист вряд ли помянет вас добрым словом.

Почему в случае функциональной декомпозиции трудно достичь слабой связанности между компонентами? Дело в том, что интерфейс между компонентами в таком проекте реализуется либо через глобальные переменные, либо через механизм формальных/фактических параметров. В сложной программной системе, реализованной в рамках структурной парадигмы, практически невозможно обойтись без связи через глобальные структуры данных, а это означает, что фактически любая функция в случае ошибки может испортить эти данные. Подобные ошибки очень трудно локализовать в процессе отладки. Добавьте к этому головную боль для сопровождающего программиста, который должен помнить десятки (если не сотни) обращений к общим данным из разных частей проекта. Соответственно, модификация существующего проекта в связи с новыми требованиями заказчика также потребует очень большой работы, так как возникнет необходимость проверить влияние внесенных изменений практически на все компоненты проекта.

Другой проблемой в проектах с функциональной декомпозицией было «проклятие» общего глобального пространства имен. Члены команды, работающей над проектом, должны были тратить немалые усилия по согласованию применяемых имен для своих функций, чтобы они были уникальными в рамках всего проекта.

Что принесло с собой ООП

Первым бросающимся в глаза отличием ООП от структурного программирования является использование классов. *Класс* — это тип, определяемый программистом, в котором объединяются структуры данных и функции их обработки.

Конкретные переменные типа данных «класс» называются *экземплярами класса*, или *объектами*. Программы, разрабатываемые на основе концепций ООП, реализуют алгоритмы, описывающие взаимодействие между объектами.

Класс содержит константы и переменные, называемые *полями*, а также выполняемые над ними операции и функции. Функции класса называются *методами*¹. Предполагается, что доступ к полям класса возможен только через вызов соответствующих методов. Поля и методы являются *элементами*, или *членами* класса.

Эффективным механизмом ослабления связанности между компонентами в случае объектно-ориентированной декомпозиции является так называемая инкапсуляция.

Инкапсуляция — это ограничение доступа к данным и их объединение с методами, обрабатывающими эти данные. Доступ к отдельным частям класса регулируется с помощью специальных ключевых слов: *public* (открытая часть), *private* (закрытая часть) и *protected* (защищенная часть)².

Методы, расположенные в открытой части, формируют *интерфейс* класса и могут свободно вызываться клиентом через соответствующий объект класса. Доступ к закрытой секции класса возможен только из его собственных методов, а к защищенной — из его собственных методов, а также из методов классов-потомков³.

Инкапсуляция повышает надежность программ, предотвращая непреднамеренный ошибочный доступ к полям объекта. Кроме этого, программу легче модифицировать, поскольку при сохранении интерфейса класса можно менять его реализацию, и это не затронет внешний программный код (код клиента).

С понятием инкапсуляции тесно связано понятие *сокрытия информации*. С другой стороны, понятие сокрытия информации соприкасается с понятием *разделения ответственности* между клиентом и сервером. Клиент не обязан знать, как реализованы те или иные методы в сервере. Для него достаточно знать, что делает данный метод и как к нему обратиться. При хорошем проектировании имена методов обычно отражают суть выполняемой ими работы, поэтому чтение кода клиента для сопровождающего программиста превращается просто в удовольствие, если не сказать — в наслаждение.

Заметим, что класс одаривает своего программиста-разработчика надежным «укрытием», обеспечивая локальную (в пределах класса) область видимости имен. Теперь можно сократить штат бригады программистов: специалист, отвечающий за согласование имен функций и имен глобальных структур данных между членами бригады, стал не нужен. В *разных* классах методы, реализующие схожие подзадачи, могут преспокойно иметь *одинаковые* имена. То же относится и к полям разных классов.

С ООП связаны еще два инструмента, грамотное использование которых повышает качество проектов: наследование классов и полиморфизм.

¹ Широко используется и другое название — *функции-члены*, возникшее при подстрочном переводе англоязычной литературы.

² Последний вид доступа имеет значение при наследовании классов и будет рассмотрен на втором семинаре.

³ Подробнее об этом — на втором семинаре.

Наследование — механизм получения нового класса из существующего. Производный класс создается путем дополнения или изменения существующего класса. Благодаря этому реализуется концепция повторного использования кода. С помощью наследования может быть создана иерархия родственных типов, которые совместно используют код и интерфейсы.

Полиморфизм дает возможность создавать множественные определения для операций и функций. Какое именно определение будет использоваться, зависит от контекста программы. Вы уже знакомы с одной из разновидностей полиморфизма в языке C++ — перегрузкой функций. Программирование с классами предоставляет еще две возможности: перегрузку операций и использование так называемых виртуальных методов. Перегрузка операций позволяет применять для собственных классов те же операции, которые используются для встроенных типов C++. Виртуальные методы обеспечивают возможность выбрать на этапе выполнения нужный метод среди одноименных методов базового и производного классов.

Кроме наследования, классы могут находиться также в отношении *агрегации*¹: например, когда в составе одного класса имеются объекты другого класса. Совместное использование наследования, композиции и полиморфизма лежит в основе элегантных проектных решений, обеспечивающих наибольшую простоту модификации программы (эта тема будет рассмотрена на втором семинаре).

Ну и, наконец, отметим, что в реальном проекте, разработанном на базе объектно-ориентированной декомпозиции, находится место и для алгоритмически-ориентированной декомпозиции (например, при реализации сложных методов).

От структуры — к классу

Прообразом класса в C++ является структура в C. В то же время в C++ структура обрела новые свойства и теперь является частным видом класса, все элементы которого по умолчанию являются открытыми. Со структурой `struct` в C++ можно делать все, что можно делать с классом. Тем не менее в C++ структуры обычно используют лишь для удобства работы с небольшими наборами данных без какого-либо собственного поведения.

Новые понятия легче изучать, отталкиваясь от уже освоенного материала. Давайте возьмем задачу 6.1 из первой книги практикума и посмотрим, что можно с ней сделать, применяя средства ООП.

Задача 1.1. Поиск в массиве структур

В текстовом файле хранится база отдела кадров предприятия. На предприятии 100 сотрудников. Каждая строка файла содержит запись об одном сотруднике. Формат записи: фамилия и инициалы (30 позиций, фамилия должна начинаться с первой позиции), год рождения (5 позиций), оклад (10 позиций). Написать программу, которая по заданной фамилии выводит на экран сведения о сотруднике, подсчитывая средний оклад всех запрошенных сотрудников.

¹ Более подробно об этом — также на втором семинаре.

Напомним, что в программе, предложенной для решения задачи 6.1 (П1)¹, для хранения сведений об одном сотруднике была использована следующая структура `Man`:

```
struct Man {
    char name[l_name + 1];
    int  birth_year;
    float pay;
};
```

Начнем с того, что преобразуем эту структуру в класс, так как мы предполагаем, что наш новый тип будет обладать более сложным поведением, чем просто чтение и запись его полей:

```
class Man {
    char name[l_name + 1];
    int  birth_year;
    float pay;
};
```

Замечательно. Это у нас здорово получилось! Все поля класса по умолчанию — закрытые (`private`). Так что если клиентская функция `main()` объявит объект `Man` `man`, а потом попытается обратиться к какому-либо его полю, например: `man.pay = value`, то компилятор быстро пресечет это безобразие, отказавшись компилировать программу. Поэтому в состав класса надо добавить методы доступа к его полям. Эти методы должны быть общедоступными, или открытыми (`public`).

Однако предварительно взглянем внимательнее в определения полей. В решении задачи 6.1 (П1) поле `name` объявлено как статический массив длиной `l_name + 1`. Это не очень гибкое решение. Мы хотели бы, чтобы наш класс `Man` можно было использовать в будущем в разных приложениях. Например, если предприятие находится в России, то значение `l_name = 30`, по-видимому, всех устроит, если же приложение создается для некой восточной страны, может потребоваться, скажем, значение `l_name = 200`. Решение состоит в использовании динамического массива символов с требуемой длиной. Поэтому заменим поле `char name[l_name + 1]` на поле `char* pName`. Сразу возникает вопрос: кто и где будет выделять память под этот массив? Вспомним один из принципов ООП: все объекты должны быть самодостаточными, то есть полностью себя обслуживать.

Таким образом, в состав класса необходимо включить метод, который обеспечит бы выделение памяти под указанный динамический массив при создании объекта (переменной типа `Man`). Метод, который автоматически вызывается при создании экземпляра класса, называется *конструктором*. Компилятор безошибочно находит этот метод среди прочих методов класса, поскольку его имя всегда совпадает с именем класса.

Парным конструктору является другой метод, называемый *деструктором*, который автоматически вызывается перед уничтожением объекта. Имя деструктора

¹ Запись в скобках «П1» означает, что имеется в виду задача 6.1 из первой книги практикума.

отличается от имени конструктора только наличием предваряющего символа ~ (тильда).

Ясно, что если в конструкторе была выделена динамическая память, то в деструкторе нужно побеспокоиться об ее освобождении. Напомним, что объект, созданный как локальная переменная в некотором блоке { }, уничтожается, когда при выполнении достигнут конец блока. Если же объект создан с помощью операции new, например:

```
Man* pMan = new Man;
```

то для его уничтожения применяется операция delete, например: delete pMan.

Итак, наш класс принимает следующий вид:

```
class Man {
public:
    Man(int lName = 30) { pName = new char[lName + 1]; } // конструктор
    ~Man() { delete [] pName; } // деструктор
private:
    char* pName;
    int birth_year;
    float pay;
};
```

Обратим ваше внимание на одну *синтаксическую деталь* — объявление класса должно обязательно завершаться точкой с запятой (;). Если вы забудете это сделать, то получите от компилятора длинный список маловразумительных сообщений о чем угодно, но только не об истинной ошибке. Что поделаешь, таков уж наш компилятор...

Рассмотрим теперь одну важную *семантическую деталь*: в конструкторе класса параметр lName имеет значение по умолчанию (30). Если все параметры конструктора имеют значения по умолчанию или если конструктор вообще не имеет параметров, он называется *конструктором по умолчанию*. Зачем понадобилось специальное название для такой разновидности конструктора? Разве это не просто удобство для клиента — передать некоторые значения по умолчанию одному из методов класса? Нет! Конструктор — это особый метод, а конструктор по умолчанию имеет несколько специальных областей применения.

Во-первых, такой конструктор используется, если компилятор встречает определение массива объектов, например: Man man[25]. Здесь объявлен массив из 25 объектов типа Man, и каждый объект этого массива создан путем вызова конструктора по умолчанию! Поэтому если вы забудете снабдить класс конструктором по умолчанию, то вы *не сможете объявлять массивы* объектов этого класса¹.

Второе применение конструкторов по умолчанию относится к механизму наследования классов. Об этом мы будем говорить на втором семинаре.

Вернемся к приведенному выше описанию класса. В нем методы класса *определены* как *встроенные* (inline) функции. При другом способе методы только *объ-*

¹ Исключение представляют классы, в которых нет ни одного конструктора, так как в таких ситуациях конструктор по умолчанию создается компилятором.

являются внутри класса, а их реализация записывается вне определения класса, как показано ниже:

```
// Man.h (интерфейс класса)
class Man {
public:
    Man(int lName = 30);           // конструктор
    ~Man();                       // деструктор
private:
    char* pName;
    int  birth_year;
    float pay;
};

// Man.cpp (реализация класса)
#include "Man.h"
Man::Man(int lName) { pName = new char[lName + 1]; }
Man::~Man() { delete [] pName; }
```

При внешнем определении метода перед его именем указывается имя класса, за которым следует операция доступа к области видимости ::. Выбор способа определения метода зависит в основном от его размера: короткие методы можно определить как встроенные, что может привести к более эффективному коду. Впрочем, компилятор все равно сам решит, может он сделать метод встроенным или нет¹.

Продолжим процесс проектирования интерфейса нашего класса. Какие методы нужно добавить в класс? С какими сигнатурами?? На этом этапе очень полезно задаться следующим вопросом: *какие обязанности* должны быть возложены на класс Man?

Первую обязанность мы уже реализовали: объект класса хранит сведения о сотруднике. Чтобы воспользоваться этими сведениями, клиент должен иметь возможность получить эти сведения, изменить их и вывести на экран. Кроме этого, для поиска сотрудника желательно иметь возможность сравнивать его имя с заданным.

Начнем с методов, обеспечивающих доступ к полям класса. Для считывания значений полей добавим методы GetName(), GetBirthYear(), GetPay(). Очевидно, что аргументы здесь не нужны, а возвращаемое значение совпадает с типом поля.

Для записи значений полей добавим методы SetName(), SetBirthYear(), SetPay(). Чтобы определиться с сигнатурой этих методов, надо представить себе, как они будут вызываться клиентом. Напомним, что в задаче 6.1 (П1) фрагмент ввода

¹ В реальных (не учебных) проектах конструкторы и деструкторы делать встроенными не рекомендуется, поскольку кроме кода, который мы пишем самостоятельно, в них может содержаться и большой объем инструкций, генерируемых автоматически, особенно при использовании виртуальных методов в сложных иерархиях классов.

² *Сигнатура, прототип, заголовок функции* — термины-синонимы (см. учебник, раздел «Функции»).

исходных данных был реализован следующим образом (здесь мы используем идентификатор `man` вместо идентификатора `dbase`):

```
int i = 0;
while (fin.getline(buf, l_buf)) {
    // . . .
    strncpy(man[i].name, buf, l_name);
    man[i].name[l_name] = '\0';
    man[i].birth_year = atoi(&buf[l_name]);
    man[i].pay = atof(&buf[l_name + l_year]);
    i++;
}
```

Как видно из этого фрагмента, в теле цикла `while` на i -й итерации выполняются следующие действия:

- ❑ очередная строка читается из файла `fin` в символьный массив `buf`;
- ❑ из массива `buf` в поле `name` структуры `man[i]` копируются первые `l_name` символов;
- ❑ из буфера `buf` извлекается очередная порция символов, отличных от пробела, преобразуется к типу `int` и записывается в поле `birth_year` структуры `man[i]`;
- ❑ извлекается следующая порция символов, отличных от пробела, преобразуется к типу `float` и записывается в поле `pay` структуры `man[i]`.

Если чисто механически скопировать такое разделение обязанностей между клиентом (`main`) и сервером (объект класса `Man`), то мы получим в теле цикла код вида:

```
man[i].SetName(buf);
man[i].SetBirthYear(atoi(&buf[l_name]));
man[i].SetPay(atof(&buf[l_name + l_year]));
i++;
```

Вряд ли такое решение можно признать удачным. Глаз сопровождающего программиста наверняка «споткнется» на аргументах вида `atoi(&buf[l_name])`. Ведь подобные выражения фактически являются сущностями типа «как делать», а не сущностями типа «что делать»! Иными словами, крайне нежелательно нагружать клиента избыточной информацией. Второстепенные детали, или детали реализации, должны быть упрятаны (инкапсулированы) внутрь класса. Поэтому в данной задаче более удачной является сигнатура метода `void SetBirthYear(const char* fromBuf)`. Аналогичные размышления можно повторить и для метода `SetPay()`.

СОВЕТ

Распределяя обязанности между клиентом и сервером, инкапсулируйте подробности реализации в серверном компоненте.

Разобравшись с методами доступа, перейдем теперь к основной части алгоритма функции `main()`, решающей подзадачу поиска сотрудника в базе по заданной фамилии и вывода сведений о нем.

Напомним, что в задаче 6.1 (П1) фрагмент поиска сотрудника с заданным именем (name) был реализован с помощью следующего цикла:

```
for (i = 0; i < n_record; ++i) {
    if (strstr(man[i].name, name)) // 1
        if (man[i].name[strlen(name)] == ' ') { // 2
            strcpy(name, man[i].name); // 3
            cout << name << man[i].birth_year << ' ' << man[i].pay << endl; // 4
        }
    // . . .
}
```

Опять задумаемся над вопросом: какая информация в этом решении *избыточна для клиента*? Здесь решаются две подзадачи: сравнение имени в очередной прочитанной записи с заданным именем name (операторы 1 и 2) и вывод информации в поток cout (операторы 3 и 4).

Поручим решение первой подзадачи методу CompareName(const char* name), инкапсулировав в него подробности решения, а решение второй подзадачи — методу Print().

Проектирование интерфейса класса Man завершено. Давайте посмотрим на текст программы, в которой реализован и использован описанный выше класс Man.

ПРИМЕЧАНИЕ

Мы будем придерживаться следующего стиля программирования:

Код каждого класса размещается в двух файлах: интерфейс — в заголовочном (.h), реализация — в исходном (.cpp), поэтому все программы будут реализовываться как многофайловые проекты, в которых главный клиент main() также занимает отдельный файл.

Имена всех классов, а также имена методов всегда начинаются с прописной буквы, имена полей (переменных) — со строчной буквы.

```
////////////////////////////////////
// Проект Task1_1
////////////////////////////////////
// Man.h // for Visual C++ 6.0
#include "CyrIOS.h"

const int l_name = 30;
const int l_year = 5;
const int l_pay = 10;
const int l_buf = l_name + l_year + l_pay;

class Man {
public:
    Man(int lName = 30);
    ~Man();
    bool CompareName(const char*) const;
    int GetBirthYear() const { return birth_year; }
```

```

float GetPay() const { return pay; }
char* GetName() const { return pName; }
void Print() const;
void SetBirthYear(const char*);
void SetName(const char*);
void SetPay(const char*);
private:
char* pName;
int birth_year;
float pay;
};
/////////////////////////////////////////////////////////////////
// Man.cpp
#include <iostream>
#include <cstring>
#include "Man.h"
using namespace std;

Man::Man(int lName) {
    cout << "Constructor is working" << endl;
    pName = new char[lName + 1];
}
Man::~Man() {
    cout << "Destructor is working" << endl;
    delete [] pName;
}
void Man::SetName(const char* fromBuf) {
    strncpy(pName, fromBuf, l_name);
    pName[l_name] = 0;
}
void Man::SetBirthYear(const char* fromBuf) {
    birth_year = atoi(fromBuf + l_name);
}
void Man::SetPay(const char* fromBuf) {
    pay = atof(fromBuf + l_name + l_year);
}
bool Man::CompareName(const char* name) const {
    if ((strstr(pName, name)) && (pName[strlen(name)] == ' '))
        return true;
    else
        return false;
}
void Man::Print() const {
    cout << pName << birth_year << ' ' << pay << endl;
}
/////////////////////////////////////////////////////////////////
// Main.cpp
#include <fstream>
#include "Man.h"
const char filename[] = "dbase.txt";

```

```

int main() {
    const int maxn_record = 10;
    Man man[maxn_record];
    char buf [l_buf + 1];
    char name[l_name + 1];

    ifstream fin(filename);
    if (!fin) {
        cout << "Нет файла " << filename << endl; return 1;
    }
    int i = 0;
    while (fin.getline(buf, l_buf)) {
        if (i >= maxn_record) {
            cout << "Слишком длинный файл"; return 1; }
        man[i].SetName(buf);
        man[i].SetBirthYear(buf);
        man[i].SetPay(buf);
        i++;
    }
    int n_record = i, n_man = 0;
    float mean_pay = 0;

    while (true) {
        cout << "Введите фамилию или слово end: ";
        cin >> name;
        if (0 == strcmp(name, "end")) break;
        bool not_found = true;
        for (i = 0; i < n_record; ++i) {
            if (man[i].CompareName(name)) {
                man[i].Print();
                n_man++; mean_pay += man[i].GetPay();
                not_found = false;
                break;
            }
        }
        if (not_found) cout << "Такого сотрудника нет" << endl;
    }
    if (n_man) cout << " Средний оклад: " << mean_pay / n_man << endl;
    return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

Обратите внимание на следующие моменты.

Ввод/вывод кириллицы. В первой части практикума мы неоднократно обсуждали этот вопрос. Если вы работаете в интегрированной среде, поддерживающей кодировку символов в стандарте ASCII (например, любой среде на платформе MS-DOS), то проблем с вводом/выводом кириллицы у вас не будет. Проблемы возникают при работе в интегрированной среде на платформе Windows (например, Visual C++ 6.0), поскольку кодировка по стандарту ANSI, принятая в Windows, отличается во второй (национальной) части кодов от кодировки ASCII.

В первой книге практикума было показано возможное решение проблемы. Для вывода информации в поток `cout` использовалась предложенная нами функция `Rus(строковая_константа)`, которая вызывала серверную функцию `CharToOem` (из Win32 API). При вводе текстовой информации из потока `cin` приходилось преобразовывать этот текст, вызывая функцию `OemToChar`. Такое прямолинейное решение довольно неудобно для клиента, но в рамках структурной парадигмы было трудно предложить что-то иное.

В этой книге мы представляем более элегантное, как нам кажется, решение: на семинаре 4 разработаны два класса — `CyrIStream` и `CyrOStream`, — объекты которых `Cin` и `Cout` подменяют стандартные объекты `cin` и `cout` путем макроподстановок и обеспечивают необходимые преобразования при потоковом вводе/выводе кириллицы. Чтобы воспользоваться указанными классами, необходимо подключить к проекту два файла: `CyrIOS.h` и `CyrIOS.cpp`, тексты которых находятся на с. 159–163¹, и кроме этого, добавить в начало модулей, содержащих потоковый ввод/вывод, директиву `#include «CyrIOS.h»`.

Заголовочные файлы. Стандартная библиотека C++ имеет несколько реализаций. В первоначальной версии библиотеки использовались заголовочные файлы с расширением `.h`, например `<iostream.h>`. Если вы работаете с компилятором, который поддерживает версию библиотеки, вошедшую в стандарт языка ISO/IEC 14882 (1998), то заголовочные файлы нужно указывать без расширения, например `<iostream>`. Кроме того, обычно используется директива `using namespace std;`, так как все имена в стандартной версии библиотеки принадлежат пространству `std`. В этой книге все примеры программ даются в расчете на версию библиотеки, соответствующую указанному стандарту, поскольку в старых версиях отсутствуют некоторые очень удобные средства, например не реализован класс `string`².

Константные методы. Обратите внимание, что заголовки тех методов класса, которые *не должны изменять поля* класса, снабжены модификатором `const` после списка параметров. Если вы по ошибке попытаетесь в теле метода что-либо присвоить полю класса, компилятор не позволит вам это сделать. Другое достоинство ключевого слова `const` — оно четко показывает сопровождающему программисту намерения разработчика программы. Например, если обнаружено некорректное поведение приложения и выяснено, что «кто-то» портит одно из полей объекта класса, то сопровождающий программист сразу может исключить из списка подозреваемых методы класса, объявленные как `const`. Поэтому использование `const` в объявлениях методов, не изменяющих объект, считается хорошим стилем программирования.

Отладочная печать в конструкторе и деструкторе. Вывод сообщений типа «Constructor is working», «Destructor is working» очень помогает на начальном этапе освоения классов. Да и не только на начальном — мы сможем убедиться в этом, когда столкнемся с проблемой локализации неочевидных ошибок в программе.

¹ А также на сайте издательства <http://www.piter.com>.

² В интегрированной среде Microsoft Visual Studio.NET реализована только версия библиотеки, соответствующая стандарту ISO/IEC 14882 (1998).

СОВЕТ

Вставляйте отладочную печать типа «Здесь был я!» в тела конструкторов и деструкторов, чтобы увидеть, как работают эти невидимки. Использование этого приема особенно полезно при поиске трудно диагностируемых ошибок.

Отладка программы

С технологией создания многофайловых проектов вы уже знакомы по первой книге практикума¹. Создайте проект с именем Task1_1 и добавьте к нему приведенные выше файлы. Не забудьте поместить в текущий каталог проекта файл с базой данных dbase.txt, заполнив его соответствующей информацией, например²:

Иванов И.П.	1957	4200
Петров А.Б.	1947	3800
Сидоров С.С.	1938	3000
Ивановский Р.Е.	1963	6200

Скомпилируйте и запустите программу на выполнение. Результат смотрится особенно хорошо под тихо льющуюся «Ave Maria» Франца Шуберта в аранжировке Джеймса Ласта. Вы должны увидеть следующий вывод в консольном окне программы:

```
Constructor is working
Constructor is working
Constructor is working
Constructor is working
Constructor is working
Constructor is working
Constructor is working
Constructor is working
Constructor is working
Constructor is working
Введите фамилию или слово end:
```

Теперь вы можете оценить, насколько полезна отладочная печать типа «Здесь был я!» в теле конструктора. Мы четко видим, как создаются все 10 элементов массива map.

Продолжим тестирование программы. Введите с клавиатуры текст: «Сидоров». После нажатия Enter вы должны увидеть на экране новый текст:

```
Сидоров С.С.      1938   3000
Введите фамилию или слово end:
```

Введите с клавиатуры текст: «Петров». После нажатия Enter появятся две строки:

```
Петров А.Б.      1947   3800
Введите фамилию или слово end:
```

¹ В приложении 1 первой книги практикума изложена технология для Visual C++ 6.0, в приложении 2 — для Borland C++ 3.1.

² При работе в среде Visual C++ 6.0 данный файл должен быть создан и заполнен текстом в одном из «виндовских» текстовых редакторов, например в Блокноте (NotePad).

Введите с клавиатуры текст: «end». После нажатия Enter появится вывод:

```
Средний оклад: 3400
Destructor is working
Destructor is working
Destructor is working
Destructor is working
Destructor is working
Destructor is working
Destructor is working
Destructor is working
Destructor is working
Destructor is working
Press any key to continue
```

Все верно! После ввода end программа покидает цикл while (true) и печатает величину среднего оклада для вызванных ранее записей из базы данных. Средний оклад рассчитан правильно. Затем программа завершается (return 0), а при выходе из области видимости все объекты вызывают свои деструкторы, и мы это тоже четко наблюдаем.

ПРИМЕЧАНИЕ

В реальных программах доступ к полям класса с помощью методов преследует еще одну важную цель — проверку заносимых значений. В качестве самостоятельного упражнения дополните методы SetBirthYear и SetPay проверками успешности преобразования из строки в число и осмысленности получившегося результата (например, на неотрицательность).

Итак, мы завершили перестройку структурной программы в программу, реализующую концепции ООП. Рекомендуем вам сравнить решение задачи 6.1 из первой книги практикума с новым решением. Вот хотя бы один характерный фрагмент — ввод исходных данных:

❑ в старой программе:

```
strncpy(man[i].pName, buf, l_name);
man[i].pName[l_name] = '\0';
man[i].birth_year = atoi(&buf[l_name]);
man[i].pay = atof(&buf[l_name + l_year]);
```

❑ в новой программе:

```
man[i].SetName(buf);
man[i].SetBirthYear(buf);
man[i].SetPay(buf);
```

Видите, как упростится теперь жизнь сопровождающего программиста? В новой программе все понятно без лишних комментариев. В старой — нужно разбираться на уровне клиента с реализацией более низкого уровня.

ВНИМАНИЕ

Если ваши последующие программы будут компилироваться в интегрированной среде на платформе Windows и в них возникнет потребность в потоковом вводе/выводе кириллицы, не забудьте подключить к проекту два файла: `CyrIOS.h` и `CyrIOS.cpp`, тексты которых находятся на с. 160–164 и на сайте <http://www.piter.com>, и добавить директиву `#include «CyrIOS.h»` в начало модулей, содержащих потоковый ввод/вывод.

Перейдем теперь к рассмотрению других аспектов, необходимых для создания хорошо спроектированного класса.

Инициализаторы конструктора

Существует альтернативный способ инициализации отдельных частей объекта в конструкторе — с помощью *списка инициализаторов*, расположенного после двоеточия между заголовком и телом конструктора. Каждый инициализатор имеет вид `имя_поля (выражение)` и обеспечивает инициализацию указанного поля объекта значением указанного выражения. Если инициализаторов несколько, они разделяются запятыми. Если полем объекта является объект другого класса, для его инициализации будет вызван соответствующий конструктор.

Например, для класса `Point`, определяющего точку на плоскости в декартовой системе координат:

```
class Point {
    double x, y;
public:
    Point(double _x = 0, double _y = 0) { x = _x; y = _y; }
    // . . .
};
```

конструктор можно записать в альтернативной форме:

```
Point(double _x = 0, double _y = 0) : x(_x), y(_y) {}
```

Обратите внимание, что тело конструктора пустое, а действия по присваиванию начальных значений полям перенесены в инициализаторы конструктора. В этом примере может быть выбран любой вариант конструктора. Однако есть три ситуации, в которых инициализация полей объекта возможна только с использованием инициализаторов конструктора:

- для полей, являющихся объектами класса, в котором есть один или более конструкторов, но отсутствует конструктор по умолчанию;
- для констант;
- для ссылок.

Обратите также внимание на то, что все параметры конструктора имеют значения по умолчанию. Благодаря этому данный конструктор может использоваться в двух формах: как конструктор с параметрами и как конструктор по умолчанию.

Вообще говоря, инициализация элементов класса с помощью списка инициализаторов является более эффективной по быстродействию, чем присваивание начальных значений в теле конструктора. Например, если объявлен объект `Point p(1.5, 2.0)`, а в классе использован конструктор без списка инициализаторов, то при создании объекта `p` его поля `x` и `y` будут сначала инициализированы значением `0` (стандартное требование C++), а затем в теле конструктора они получат значения `1.5` и `2.0`. Если же в классе `Point` использован конструктор с инициализаторами, то при создании объекта `p` его полям `x` и `y` сразу будут присвоены значения `1.5` и `2.0`.

Конструктор копирования

Конструктор так же, как и остальные методы класса, можно перегружать. Одной из важнейших форм перегружаемого конструктора является *конструктор копирования* (*copy constructor*). Конструктор копирования вызывается в трех ситуациях:

- при инициализации нового объекта другим объектом в операторе объявления;
- при передаче объекта в функцию в качестве аргумента по значению;
- при возврате объекта из функции по значению.

Если при объявлении класса конструктор копирования не задан, компилятор C++ создает *конструктор копирования по умолчанию*, который просто дублирует объект, осуществляя побайтное копирование всех его полей. Однако при этом возможно появление проблем, связанных с копированием указателей. Например, если объект, передаваемый в функцию в качестве аргумента по значению, содержит некоторый указатель `pMem` на выделенную область памяти, то в копии объекта этот указатель будет ссылаться на ту же самую область памяти. При возврате из функции для копии объекта будет вызван деструктор, освобождающий память, на которую указывает `pMem`. При выходе из области видимости исходного объекта его деструктор попытается еще раз освободить память, на которую указывает `pMem`. Результат обычно весьма плачевный.

Для того чтобы исключить нежелательные побочные эффекты в каждой из трех указанных ситуаций, необходимо создать конструктор копирования, в котором реализуется необходимый контроль за созданием копии объекта.

Конструктор копирования имеет следующую общую форму:

```
имя_класса(const имя_класса & obj) {
    // тело конструктора
}
```

Здесь `obj` — ссылка на объект, для которого должна создаваться копия. Например, пусть имеется класс `Соо`, а `y` — объект типа `Соо`. Тогда следующие операторы вызовут конструктор копирования класса `Соо`:

```
Соо x = y;           // y явно инициализирует x
Func1(y);           // y передается в качестве аргумента по значению
y = Func2();        // y получает возвращаемый объект
```

В двух первых случаях конструктору копирования передается ссылка на *u*. В последнем случае конструктору копирования передается ссылка на объект, возвращаемый функцией `Func2()`.

Пример реализации конструктора копирования будет показан при решении задачи 1.2.

Перегрузка операций

Любая операция¹, определенная в C++, может быть перегружена для созданного вами класса. Это делается с помощью функций специального вида, называемых *функциями-операциями* (операторными функциями). Общий вид такой функции:

```
возвращаемый_тип operator # (список параметров) { тело функции }
```

где вместо знака # ставится знак перегружаемой операции.

Функция-операция может быть реализована либо как функция класса, либо как внешняя (обычно дружественная) функция. В первом случае количество параметров у функции-операции на единицу меньше, так как первым операндом при этом считается сам объект, вызвавший данную операцию.

Например, покажем два варианта перегрузки операции сложения для класса `Point`:

Первый вариант — в форме метода класса:

```
class Point {
    double x, y;
public:
    // . . .
    Point operator +(Point&);
};
Point Point::operator +(Point& p) {
    return Point(x + p.x, y + p.y);
}
```

Второй вариант — в форме внешней глобальной функции, причем функция, как правило, объявляется дружественной классу, чтобы иметь доступ к его закрытым элементам:

```
class Point {
    double x, y;
public:
    // . . .
    friend Point operator +(Point&, Point&);
};
Point operator +(Point& p1, Point& p2) {
    return Point(p1.x + p2.x, p1.y + p2.y);
}
```

¹ За исключением «::», «?:», «.», «.*», «#», «##».

Независимо от формы реализации операции «+» мы можем теперь написать:

```
Point p1(0, 2), p2(-1, 5);  
Point p3 = p1 + p2;
```

Будет неплохо, если вы будете понимать, что, встретив выражение `p1 + p2`, компилятор в случае первой формы перегрузки вызовет метод `p1.operator +(p2)`, а в случае второй формы перегрузки — глобальную функцию `operator +(p1, p2)`. Результатом выполнения данных операторов будет точка `p3` с координатами `x = -1, y = 7`. Заметим, что для инициализации объекта `p3` будет вызван конструктор копирования по умолчанию, но он нас устраивает, поскольку в классе нет полей-указателей.

Если операция может перегружаться как внешней функцией, так и функцией класса, какую из двух форм следует выбирать? Ответ: используйте перегрузку в форме метода класса, если нет каких-либо причин, препятствующих этому. Например, если первый аргумент (левый операнд) относится к одному из базовых типов (к примеру, `int`), то перегрузка операции возможна только в форме внешней функции.

Перегрузка операций инкремента

Операция инкремента имеет две формы: *префиксную* и *постфиксную*. Для первой формы сначала изменяется состояние объекта в соответствии с данной операцией, а затем он (объект) используется в том или ином выражении. Для второй формы объект используется в том состоянии, которое у него было до начала операции, а потом уже его состояние изменяется.

Чтобы компилятор смог различить эти две формы операции инкремента, для них используются разные сигнатуры, например:

```
Point& operator ++():           // префиксный инкремент  
Point operator ++(int):        // постфиксный инкремент
```

Покажем реализацию данных операций на примере класса `Point`:

```
Point& Point::operator ++() {  
    x++;    y++;  
    return *this;  
}  
Point Point::operator ++(int) {  
    Point old = *this;  
    x++;    y++;  
    return old;  
}
```

Обратите внимание, что в префиксной операции осуществляется возврат результата по ссылке. Это предотвращает вызов конструктора копирования для создания возвращаемого значения и последующего вызова деструктора. В постфиксной операции инкремента возврат по ссылке не подходит, поскольку необходимо вернуть первоначальное состояние объекта, сохраненное в локальной перемен-

ной `old`. Таким образом, префиксный инкремент является более эффективной операцией, чем постфиксный инкремент.

Заметим, что ранее мы не обращали внимания на запись инкремента в заголовке цикла `for`. Так, в первой книге практикума во всех примерах использовалась *постфиксная* форма инкремента: `for (i = 0; i < n; i++)`. Дело в том, что пока параметр `i` является переменной встроенного типа, форма инкремента безразлична: программа будет работать одинаково. Ситуация меняется, если параметр `i` есть объект некоторого класса — в этом случае префиксная форма инкремента оказывается более эффективной. Когда мы будем использовать контейнерные классы стандартной библиотеки, параметр `i` в заголовке цикла очень часто будет представлять объект-итератор.

СОВЕТ

Всегда используйте *префиксный инкремент* для параметра цикла `for` — это дает более эффективный программный код.

ПРИМЕЧАНИЕ

Все сказанное о данной операции относится также и к *операции декремента*.

Перегрузка операции присваивания

О перегрузке этой операции следует поговорить особо по нескольким причинам. Во-первых, если вы не определите эту операцию в некотором классе, то компилятор создаст операцию присваивания по умолчанию, которая выполняет поэлементное копирование объекта. В этом случае возможно появление тех же проблем, которые возникают при использовании конструктора копирования по умолчанию (см. выше). Поэтому запомните золотое правило: если в классе требуется определить конструктор копирования, то его верной спутницей должна быть перегруженная операция присваивания, и наоборот.

Во-вторых, операция присваивания может быть определена только в форме метода класса.

В-третьих, операция присваивания не наследуется (в отличие от всех остальных операций).

Например, для класса `Man` из задачи 1.1 перегрузку операции присваивания можно определить следующим образом:

```
// Man.h (интерфейс класса)
class Man {
public:
    // . . .
    Man& operator =(const Man&); // операция присваивания
private:
    char* pName;
    // . . .
};
```



```
// Man.cpp (реализация класса)
// . . .
Man& Man::operator =(const Man& man) {
    if (this == &man) return *this; // проверка на самоприсваивание
    delete [] pName;                // уничтожить предыдущее значение
    pName = new char[strlen(man.pName) + 1];
    strcpy(pName, man.pName);
    birth_year = man.birth_year;
    pay = man.pay;
    return *this;
}
```

Обратите внимание на несколько простых, но важных моментов при реализации операции присваивания:

- ❑ убедитесь, что не выполняется присваивание вида $x = x$. Если левая и правая части ссылаются на один и тот же объект, то делать ничего не надо. Если не перехватить этот особый случай, то следующий шаг уничтожит значение, на которое указывает `pName`, еще до того, как оно будет скопировано;
- ❑ удалите предыдущие значения полей в динамически выделенной памяти;
- ❑ выделите память под новые значения полей;
- ❑ скопируйте в нее новые значения всех полей;
- ❑ возвратите значение объекта, на которое указывает `this` (то есть `*this`).

Статические элементы класса

До сих пор одноименные поля разных объектов одного и того же класса были уникальными. Но что делать, если необходимо создать переменную, значение которой будет общим для всех объектов конкретного класса? Если воспользоваться глобальной переменной, то это нарушит принцип инкапсуляции данных. Модификатор `static` как раз и позволяет объявить поле в классе, которое будет общим для всех экземпляров класса. Кроме *объявления* статического поля в классе, необходимо также дать его *определение* в глобальной области видимости программы, например:

```
class Coo {
    static int count; // объявление в классе
    // остальной код
};
int Coo::count = 1; // определение и инициализация
// int Coo::count; // по умолчанию инициализируется нулем
```

Аналогично статическим полям могут быть объявлены и статические методы класса (с модификатором `static`). Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, потому что им не передается скрытый указатель `this`. Статические методы не могут быть константными (`const`) и виртуальными (`virtual`). Обращение к стати-

ческим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта. Все рассмотренные выше аспекты найдут применение при решении второй задачи этого семинара.

Задача 1.2. Реализация класса треугольников

Для некоторого множества заданных координатами своих вершин треугольников найти треугольник максимальной площади (если максимальную площадь имеют несколько треугольников, то найти первый из них). Предусмотреть возможность перемещения треугольников и проверки включения одного треугольника в другой.

Для реализации этой задачи составить описание класса треугольников на плоскости. Предусмотреть возможность объявления в клиентской программе (`main`) экземпляра треугольника с заданными координатами вершин. Предусмотреть наличие в классе методов, обеспечивающих: 1) перемещение треугольников на плоскости; 2) определение отношения $>$ для пары заданных треугольников (мера сравнения — площадь треугольников); 3) определение отношения включения типа: «Треугольник 1 входит в (не входит в) Треугольник 2».

Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Отметим, что сейчас мы еще не готовы провести полномасштабную объектно-ориентированную декомпозицию программы. Для этого нам не хватает знаний, которые будут получены на втором семинаре. Поэтому применим гибридный подход: разработку главного клиента `main()` проведем по технологии функциональной декомпозиции, а функции-серверы, вызываемые из `main()`, будут использовать объекты.

Начнем с выявления основных понятий/классов для нашей задачи. Первый очевидный класс `Triangle` необходим для представления треугольников. Из нескольких способов определения треугольника на плоскости выберем самый простой — через три точки, задающие его вершины. Сделанный выбор опирается на новое понятие, отсутствующее в условии задачи, — понятие *точки*. Точку на плоскости можно представить различными способами; остановимся на наиболее популярном — с помощью пары вещественных чисел, задающих координаты точки по осям x и y .

Таким образом, с понятием точки связывается как минимум пара атрибутов. В принципе, этого уже достаточно, чтобы подумать о создании класса `Point`. Если же представить, что можно делать с объектом типа точки — например, перемещать ее на плоскости или определять ее вхождение в заданную фигуру, — то становится ясным, что такой класс `Point` будет полезен.

Итак, объектно-ориентированная декомпозиция дала нам два класса: `Triangle` и `Point`. Как эти два класса должны быть связаны друг с другом? На втором

семинаре мы будем более подробно рассматривать взаимоотношения между классами. Пока же отметим, что наиболее часто для двух классов имеет место одно из двух:

- отношение наследования (отношение *is a*);
- отношение агрегации или включения (отношение *has a*).

Если класс B является «частным случаем» класса A, то говорят, что *B is a A* (например, класс треугольников есть частный вид класса многоугольников: *Triangle is a Polygon*).

Если класс A содержит в себе объект класса B, то говорят, что *A has a B* (например, класс треугольников может содержать в себе объекты класса точек: *Triangle has a Point*).

Теперь уточним один вопрос: *в каком порядке* мы будем перечислять точки, задавая треугольник? Порядок перечисления вершин важен для нас потому, что в дальнейшем, решая подзадачу определения отношения включения одного треугольника в другой, мы будем рассматривать стороны треугольника как *векторы*. Условимся, что вершины треугольника перечисляются в направлении *по часовой стрелке*.

Займемся теперь основным клиентом — `main()`. Здесь мы применяем функциональную декомпозицию, или технологию нисходящего проектирования. В соответствии с данной технологией основной алгоритм представляется как последовательность нескольких подзадач. Каждой подзадаче соответствует вызываемая серверная функция. На начальном этапе проектирования тела этих функций могут быть заполнены «заглушками» — отладочной печатью. Если при этом в какой-то серверной функции окажется слабое сцепление, то она в свою очередь разбивается на несколько подзадач.

То же самое происходит и с классами, используемыми в программе: по мере реализации подзадач они пополняются необходимыми для этого методами. Такая технология облегчает отладку и поиск ошибок, сокращая общее время разработки программы.

В соответствии с описанной технологией мы представим решение задачи 1.2 как последовательность нескольких этапов. Иногда как бы по забывчивости мы будем допускать некоторые «ляпы», поскольку в процессе поиска ошибок можно глубже понять нюансы программирования с классами.

На первом этапе мы напишем код для начального представления классов `Point` и `Triangle`, достаточный для того, чтобы создать несколько объектов типа `Triangle` и реализовать первый пункт меню — вывод всех объектов на экран.

Этап 1

```

////////////////////////////////////
// Проект Task1_2
////////////////////////////////////
// Point.h
#ifdef POINT_H
#define POINT_H

```

```

class Point {
public:
    // Конструктор
    Point(double _x = 0, double _y = 0) : x(_x), y(_y) {}
    // Другие методы
    void Show() const;
public:
    double x, y;
};
#endif /* POINT_H */
/////////////////////////////////////////////////////////////////
// Point.cpp
#include <iostream>
#include "Point.h"
using namespace std;

void Point::Show() const {
    cout << " (" << x << ", " << y << ")";
}
/////////////////////////////////////////////////////////////////
// Triangle.h
#ifndef TRIANGLE_H
#define TRIANGLE_H

#include "Point.h"

class Triangle {
public:
    Triangle(Point, Point, Point, const char*); // конструктор
    Triangle(const char*); // конструктор пустого (нулевого) треугольника
    ~Triangle(); // деструктор
    Point Get_v1() const { return v1; } // Получить значение v1
    Point Get_v2() const { return v2; } // Получить значение v2
    Point Get_v3() const { return v3; } // Получить значение v3
    char* GetName() const { return name; } // Получить имя объекта
    void Show() const; // Показать объект
    void ShowSideAndArea() const; // Показать стороны и площадь объекта
public:
    static int count; // количество созданных объектов
private:
    char* objID; // идентификатор объекта
    char* name; // наименование треугольника
    Point v1, v2, v3; // вершины
    double a; // сторона, соединяющая v1 и v2
    double b; // сторона, соединяющая v2 и v3
    double c; // сторона, соединяющая v1 и v3
};
#endif /* TRIANGLE_H */

```

```

////////////////////////////////////
//Triangle.cpp
// Реализация класса Triangle
#include <math.h>
#include <iostream>
#include <iomanip>
#include <cstring>
//#include "CyrIOS.h" // for Visual C++ 6.0
#include "Triangle.h"
using namespace std;

// Конструктор
Triangle::Triangle(Point _v1, Point _v2, Point _v3, const char* ident)
    : v1(_v1), v2(_v2), v3(_v3) {
    char buf[16];

    objID = new char[strlen(ident) + 1];
    strcpy(objID, ident);

    count++;
    sprintf(buf, "Треугольник %d", count);
    name = new char[strlen(buf) + 1];
    strcpy(name, buf);
    a = sqrt((v1.x - v2.x) * (v1.x - v2.x) + (v1.y - v2.y) * (v1.y - v2.y));
    b = sqrt((v2.x - v3.x) * (v2.x - v3.x) + (v2.y - v3.y) * (v2.y - v3.y));
    c = sqrt((v1.x - v3.x) * (v1.x - v3.x) + (v1.y - v3.y) * (v1.y - v3.y));
    cout << "Constructor_1 for: " << objID
        << " (" << name << ")" << endl; // отладочный вывод
}

// Конструктор пустого (нулевого) треугольника
Triangle::Triangle(const char* ident) {
    char buf[16];
    objID = new char[strlen(ident) + 1];
    strcpy(objID, ident);

    count++;
    sprintf(buf, "Треугольник %d", count);
    name = new char[strlen(buf) + 1];
    strcpy(name, buf);
    a = b = c = 0;
    cout << "Constructor_2 for: " << objID
        << " (" << name << ")" << endl; // отладочный вывод
}

// Деструктор
Triangle::~Triangle() {
    cout << "Destructor for: " << objID << endl;
}

```

```

    delete [] objID;
    delete [] name;
}

// Показать объект
void Triangle::Show() const {
    cout << name << " ";
    v1.Show();  v2.Show();  v3.Show();
    cout << endl;
}

// Показать стороны и площадь объекта
void Triangle::ShowSideAndArea() const {
    double p = (a + b + c) / 2;
    double s = sqrt(p * (p - a) * (p - b) * (p - c));
    cout << "-----" << endl;
    cout << name << " ";
    cout.precision(4);
    cout << " a= " << setw(5) << a;
    cout << ", b= " << setw(5) << b;
    cout << ", c= " << setw(5) << c;
    cout << ";\ts= " << s << endl;
}
/////////////////////////////////////////////////////////////////
// Main.cpp
#include <iostream>
#include "Triangle.h"
// #include "CyrIOS.h"           // for Visual C++ 6.0
using namespace std;

int  Menu();
int  GetNumber(int, int);
void ExitBack();
void Show(Triangle* [], int);
void Move(Triangle* [], int);
void FindMax(Triangle* [], int);
void IsIncluded(Triangle* [], int);

// Инициализация глобальных переменных
int Triangle::count = 0;

// ----- главная функция
int main() {

// Определения точек
    Point p1(0, 0);   Point p2(0.5, 1);
    Point p3(1, 0);   Point p4(0, 4.5);
    Point p5(2, 1);   Point p6(2, 0);
    Point p7(2, 2);   Point p8(3, 0);

```

```

// Определения треугольников
Triangle triaA(p1, p2, p3, "triaA");
Triangle triaB(p1, p4, p8, "triaB");
Triangle triaC(p1, p5, p6, "triaC");
Triangle triaD(p1, p7, p8, "triaD");

// Определение массива указателей на треугольники
Triangle* pTria[] = { &triaA, &triaB, &triaC, &triaD };
int n = sizeof (pTria) / sizeof (pTria[0]);

// Главный цикл
bool done = false;
while (!done) {
    switch (Menu()) {
        case 1: Show(pTria, n);          break;
        case 2: Move(pTria, n);         break;
        case 3: FindMax(pTria, n);      break;
        case 4: IsIncluded(pTria, n);   break;
        case 5: cout << "Конец работы." << endl;
               done = true;          break;
    }
}
return 0;
}
// ----- вывод меню
int Menu() {

    cout << "\n===== Г л а в н о е   м е н ю =====" <<
endl;
    cout << "1 - вывести все объекты\t 3 - найти максимальный" << endl;
    cout << "2 - переместить\t\t 4 - определить отношение включения" << endl;
    cout << "\t\t 5 - выход" << endl;

    return GetNumber(1, 5);
}
// ----- ввод целого числа в заданном диапазоне
int GetNumber(int min, int max) {
    int number = min - 1;
    while (true) {
        cin >> number;
        if ((number >= min) && (number <= max) && (cin.peek() == '\n'))
            break;
        else {
            cout << "Повторите ввод (ожидается число от " << min
                << " до " << max << "):" << endl;
            cin.clear();
            while (cin.get() != '\n') {};
        }
    }
}

```

```

    }
    return number;
}
// ----- возврат в функцию с основным меню
void ExitBack() {
    cout << "Нажмите Enter." << endl;
    cin.get(); cin.get();
}
// ----- вывод всех треугольников
void Show(Triangle* p_tria[], int k) {
    cout << "===== Перечень треугольников =====" << endl;
    for (int i = 0; i < k; ++i) p_tria[i]->Show();
    for (i = 0; i < k; ++i) p_tria[i]->ShowSideAndArea();
    ExitBack();
}
// ----- перемещение
void Move(Triangle* p_tria[], int k) {
    cout << "===== Перемещение =====" << endl;
    // здесь будет код функции...
    ExitBack();
}
// ----- поиск максимального треугольника
void FindMax(Triangle* p_tria[], int k) {
    cout << "=== Поиск максимального треугольника ===" << endl;
    // здесь будет код функции...
    ExitBack();
}
// ----- определение отношения включения
void IsIncluded(Triangle* p_tria[], int k) {
    cout << "===== Отношение включения =====" << endl;
    // здесь будет код функции...
    ExitBack();
}
//----- конец проекта Task1_2 -----
////////////////////////////////////

```

Рекомендуем вам обратить внимание на следующие моменты в проекте Task1_2.

□ Класс Point (файлы Point.h, Point.cpp).

- Реализация класса Point пока что содержит единственный метод Show(), назначение которого очевидно: показать объект типа Point на экране. Здесь следует заметить, что при решении реальных задач в какой-либо графической оболочке метод Show() действительно нарисовал бы нашу точку, да еще в цвете. Но мы-то изучаем «чистый» C++, так что придется удовольствоваться текстовым выводом на экран основных атрибутов точки — ее координат.

□ Класс Triangle (файлы Triangle.h, Triangle.cpp).

- Назначение большинства полей и методов очевидно из их имен и комментариев.

- Поле `static int count` играет роль глобального¹ счетчика создаваемых объектов; мы сочли удобным в конструкторах генерировать имена треугольников автоматически: «Треугольник 1», «Треугольник 2» и т. д., используя текущее значение `count` (возможны и другие способы именования треугольников).
 - Поле `char* objID` избыточно для решения нашей задачи — оно введено исключительно для целей отладки и обучения; вскоре вы увидите, что благодаря отладочным операторам печати в конструкторах и деструкторе удобно наблюдать за созданием и уничтожением объектов.
 - Метод `ShowSideAndArea()` введен также только для целей отладки, — удивившись, что стороны треугольника и его площадь вычисляются правильно (с помощью калькулятора), в дальнейшем этот метод можно удалить.
 - Конструктор пустого (нулевого) треугольника предусмотрен для создания временных объектов, которые могут модифицироваться с помощью присваивания.
 - Метод `Show()` — см. комментарий выше по поводу метода `Show()` в классе `Point`. К сожалению, здесь нам тоже не удастся нарисовать треугольник на экране; вместо этого печатаются координаты его вершин.
- Основной модуль (файл `Main.cpp`).
- Инициализация глобальных переменных: обратите внимание на оператор `int Triangle::count = 0;` — если вы забудете это написать, компилятор очень сильно обидится.
 - Функция `main()`:
 - определения восьми точек `p1, ..., p8` выбраны произвольно, но так, чтобы из них можно было составить треугольники;
 - определения четырех треугольников сделаны тоже произвольно, впоследствии на них будут демонстрироваться основные методы класса; однако не забывайте, что вершины в каждом треугольнике должны перечисляться *по часовой стрелке*;
 - далее определяются массив указателей `Triangle* pTria[]` с адресами объявленных выше треугольников и его размер `n`; в таком виде удобно передавать адрес `pTria` и величину `n` в вызываемые серверные функции;
 - главный цикл функции довольно прозрачен и дополнительных пояснений не требует.
 - Функция `Menu()` — после вывода на экран списка пунктов меню вызывается функция `GetNumber()`, возвращающая номер пункта, введенный пользователем с клавиатуры. Для чего написана такая сложная функция — `GetNumber()`? Ведь можно было просто написать: `cin >> number;`? Но тогда мы не обеспечили бы защиту программы от непреднамеренных ошибок при вводе. Вообще-то

¹ Свойство глобальности обеспечивается благодаря модификатору `static`.

вопрос надежного ввода чисел с клавиатуры подробно разбирается на семинаре 4 при решении задачи 4.2. Там же вы можете найти описание работы аналогичной функции `GetInt()`¹.

- Функция `Show()` просто выводит на экран перечень всех треугольников. В завершение вызывается функция `ExitBack()`, которая обеспечивает заключительный диалог с пользователем после обработки очередного пункта меню.
- Остальные функции по обработке оставшихся пунктов меню выполнены в виде заглушек, выводящих только наименование соответствующего пункта.

Тестирование и отладка первой версии программы

После компиляции и запуска программы вы должны увидеть на экране следующий текст:

```
Constructor_1 for: triaA (Треугольник 1)
Constructor_1 for: triaB (Треугольник 2)
Constructor_1 for: triaC (Треугольник 3)
Constructor_1 for: triaD (Треугольник 4)
```

```
===== Г л а в н о е   м е н ю =====
1 - вывести все объекты   3 - найти максимальный
2 - переместить          4 - определить отношение включения
5 - выход
```

Введите с клавиатуры цифру 1². Программа выведет:

```
1
===== Перечень треугольников =====
Треугольник 1: (0, 0) (0.5, 1) (1, 0)
Треугольник 2: (0, 0) (0, 4.5) (3, 0)
Треугольник 3: (0, 0) (2, 1) (2, 0)
Треугольник 4: (0, 0) (2, 2) (3, 0)

-----
Треугольник 1: a= 1.118, b= 1.118, c=   1;   s= 0.5
-----
Треугольник 2: a=   4.5, b= 5.408, c=   3;   s= 6.75
-----
Треугольник 3: a= 2.236, b=   1, c=   2;   s= 1
-----
Треугольник 4: a= 2.828, b= 2.236, c=   3;   s= 3
Нажмите Enter.
```

¹ В задаче 9.2 из первой книги практикума было дано другое решение проблемы ввода номера пункта меню (с защитой от ошибок), но оно рассчитано на использование функций библиотеки `stdio`, унаследованных из языка C.

² После ввода числовой информации всегда подразумевается нажатие клавиши `Enter`.

Выбор первого пункта меню проверен. Нажмите Enter. Программа выведет:

```
===== Главное меню =====
```

Теперь проверим выбор второго пункта меню. Введите с клавиатуры цифру 2. На экране должно появиться:

```
2
===== Перемещение =====
```

Нажмите Enter.

Выбор второго пункта проверен. Нажмите Enter. Программа выведет:

```
===== Главное меню =====
```

Теперь проверим ввод ошибочного символа. Введите с клавиатуры любой буквенный символ, например w, и нажмите Enter. Программа должна выругаться:

Повторите ввод (ожидается число от 1 до 5):

Проверяем завершение работы. Введите цифру 5. Программа выведет:

```
5
Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: triaB
Destructor for: triaA
```

Тестирование закончено. Обратите внимание на то, что деструкторы объектов вызываются в порядке, обратном вызову конструкторов.

Продолжим разработку программы. На втором этапе мы добавим в классы Point и Triangle методы, обеспечивающие перемещение треугольников, а в основной модуль — реализацию функции Move(). Кроме этого, в классе Triangle мы удалим метод ShowSideAndArea(), поскольку он был введен только для целей отладки и свою роль уже выполнил.

Этап 2

Внесите следующие изменения в тексты модулей проекта.

1. Модуль Point.h: добавьте сразу после объявления метода Show() объявление операции-функции «+=», которая позволит нам впоследствии реализовать метод перемещения Move() в классе Triangle:

```
void operator +=(Point&);
```

2. Модуль Point.cpp. Добавьте код реализации данной функции:

```
void Point::operator +=(Point& p) {
    x += p.x;    y += p.y;
}
```

3. Модуль Triangle.h.

- Удалите объявление метода ShowSideAndArea().
- Добавьте объявление метода:


```
void Move(Point);
```

4. Модуль Triangle.cpp.

- Удалите метод ShowSideAndArea().
- Добавьте код метода Move():


```
// Переместить объект на величину (dp.x, dp.y)
void Triangle::Move(Point dp) {
    v1 += dp;    v2 += dp;    v3 += dp;
}
```

5. Модуль Main.cpp.

- В список прототипов функций в начале файла добавьте сигнатуру:


```
double GetDouble();
```
- Добавьте в файл текст новой функции GetDouble() либо сразу после функции Show(), либо в конец файла. Эта функция предназначена для ввода вещественного числа и вызывается из функции Move(). В ней предусмотрена защита от ввода недопустимых (например, буквенных) символов аналогично тому, как это решено в функции GetNumber():

```
// ----- ввод вещественного числа
double GetDouble() {
    double value;
    while (true) {
        cin >> value;
        if (cin.peek() == '\n') break;
        else {
            cout << "Повторите ввод (ожидается вещественное число):" << endl;
            cin.clear();
            while (cin.get() != '\n') {};
        }
    }
    return value;
}
```

- Замените заглушку функции Move() следующим кодом:

```
// ----- перемещение
void Move(Triangle* p_tria[], int k) {
    cout << "===== Перемещение =====" << endl;
    cout << "Введите номер треугольника (от 1 до " << k << "): ";
    int i = GetNumber(1, k) - 1;
    p_tria[i]->Show();

    Point dp;
```

```

cout << "Введите смещение по x: ";
dp.x = GetDouble();
cout << "Введите смещение по y: ";
dp.y = GetDouble();

p_tria[i]->Move(dp);
cout << "Новое положение треугольника:" << endl;
p_tria[i]->Show();
ExitBack();
}

```

Выполнив компиляцию проекта, проведите его тестирование аналогично тестированию на первом этапе. После выбора второго пункта меню и ввода данных, задающих номер треугольника, величину сдвига по x и величину сдвига по y , вы должны увидеть на экране примерно следующее:¹

```

2
===== Перемещение =====
Введите номер треугольника (от 1 до 4): 1
Треугольник 1: (0, 0) (0.5, 1) (1, 0)
Введите смещение по x: 2.5
Введите смещение по y: -7
Новое положение треугольника:
Треугольник 1: (2.5, -7) (3, -6) (3.5, -7)
Нажмите Enter.

```

Продолжим разработку программы.

Этап 3

На этом этапе мы добавим в класс `Triangle` метод, обеспечивающий сравнение треугольников по их площади, а в основной модуль — реализацию функции `FindMax()`. Внесите следующие изменения в тексты модулей проекта:

1. Модуль `Triangle.h`: добавьте объявление функции-операции:

```
bool operator >(const Triangle&) const;
```

2. Модуль `Triangle.cpp`: добавьте код реализации функции-операции:

```

// Сравнить объект (по площади) с объектом tria
bool Triangle::operator >(const Triangle& tria) const {
    double p = (a + b + c) / 2;
    double s = sqrt(p * (p - a) * (p - b) * (p - c));
    double p1 = (tria.a + tria.b + tria.c) / 2;
    double s1 = sqrt(p1 * (p1 - tria.a) * (p1 - tria.b) * (p1 - tria.c));
    if (s > s1) return true;
    else return false;
}

```

¹ Жирным шрифтом выделено то, что вводилось с клавиатуры.

3. Модуль Main.cpp: замените заглушку функции FindMax() следующим кодом:

```
// ————— поиск максимального треугольника
void FindMax(Triangle* p_tria[], int k) {
    cout << "=== Поиск максимального треугольника ===" << endl;
    // Создаем объект triaMax, который по завершении поиска будет
    // идентичен максимальному объекту.
    // Инициализируем его значением 1-го объекта из массива объектов.
    Triangle triaMax("triaMax");
    triaMax = *p_tria[0];
    // Поиск
    for (int i = 1; i < 4; ++i)
        if (*p_tria[i] > triaMax)
            triaMax = *p_tria[i];
    cout << "Максимальный треугольник: " << triaMax.GetName() << endl;
    ExitBack();
}
```

Откомпилируйте программу и запустите. Выберите третий пункт меню. На экране должно появиться:

```
3
=== Поиск максимального треугольника ===
Constructor_2 for: triaMax (Треугольник 5)
Максимальный треугольник: Треугольник 2
Нажмите Enter.
```

Как видите, максимальный треугольник найден правильно. Повинуясь указанию, нажмите Enter. Появится текст:

```
Destructor for: triaB
===== Г л а в н о е   м е н ю =====
1 - вывести все объекты   3 - найти максимальный
2 - переместить          4 - определить отношение включения
5 - выход
```

Чтобы завершить работу, введите цифру 5 и нажмите Enter. Что за ... — получили??? Ха-ха-ха!!! Программа вылетела! Если это происходит в Visual Studio, то вас порадует нагло выскочившая на передний план диалоговая панель с жирным белым крестом на красном кружочке и с сообщением об ошибке:

```
Debug Assertion Failed!
Program: C:\. . . MAIN.EXE
File: dbgdel.cpp
Line 47
```

А далее добрый совет, смысл которого в переводе на русский язык следующий: «Если вас интересует, по каким причинам могут выскакивать такие диалоговые панельки, обратитесь к разделу документации по Visual C++, посвященному макросу assert». Попробуйте обратиться. Скорее всего, вы будете разочарованы...

Давайте лучше посмотрим на нашу отладочную печать. Перед тем как так некрасиво умереть, программа успела вывести на экран следующее:

```
5
Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: ННННННННННээээЭЭЭЭЭЭЭЭЭЭЭЭ
```

Пришло время для аналитической работы нашим серым клеточкам. Вспомните, как выглядела отладочная печать при завершении нормально работающей первой версии программы? Деструкторы вызывались в порядке, обратном вызову конструкторов. Значит, какая-то беда случилась после вызова деструктора для объекта `triaB`! Почему?

Всмотримся внимательней в предыдущий вывод нашей программы. После того как функция `FindMax()` выполнила основную работу и вывела на экран сообщение

```
Максимальный треугольник: Треугольник 2
```

программа пригласила пользователя нажать клавишу `Enter`. Это приглашение выводится функцией `ExitBack()`. А вот после нажатия клавиши `Enter` на экране появился текст:

```
Destructor for: triaB
```

после которого опять было выведено главное меню.

Значит, деструктор для объекта `triaB` был вызван в момент возврата из функции `FindMax()`. Но почему? Ведь объект `triaB` создается в основной функции `main()`, а значит, там же должен и уничтожаться! Что-то нехорошее происходит, однако, в теле функции `FindMax()`. Хотя внешне вроде все прилично... Стоп! Ведь внутри функции объявлен объект `triaMax`, и мы даже видели работу его конструктора:

```
Constructor_2 for: triaMax (Треугольник 5)
```

А где же вызов деструктора, который по идее должен произойти в момент возврата из функции `FindMax()`? Кажется, мы нашли причину ненормативного поведения нашей программы. Объект `triaMax` после своего объявления неоднократно модифицируется с помощью операции присваивания. Последняя такая модификация происходит в цикле, причем объекту `triaMax` присваивается значение объекта `triaB`. А теперь давайте вспомним, что если мы не перегрузили операцию присваивания для некоторого класса, то компилятор сделает это за нас, но в такой «операции присваивания по умолчанию» будут поэлементно копироваться все поля объекта. При наличии же полей типа указателей возможны проблемы, что мы и получили.

В поля `objID` и `name` объекта `triaMax` были скопированы значения одноименных полей объекта `triaB`. В момент выхода из функции `FindMax()` деструктор объекта освободил память, на которую указывали эти поля. А при выходе из основной

функции `main()` деструктор объекта `tria` в попытался еще раз освободить эту же память. Это делать нельзя, потому что этого делать нельзя никогда.

Займемся починкой нашей программы. Нужно добавить в класс `Triangle` перегрузку операции присваивания, а заодно и конструктор копирования.

Внесите следующие изменения в тексты модулей проекта:

1. Модуль `Triangle.h`.

- Добавьте объявление конструктора копирования:

```
Triangle(const Triangle&); // конструктор копирования
```

- Добавьте объявление операции присваивания:

```
Triangle& operator =(const Triangle&);
```

2. Модуль `Triangle.cpp`.

- Добавьте реализацию конструктора копирования:

```
// Конструктор копирования
Triangle::Triangle(const Triangle& tria) : v1(tria.v1), v2(tria.v2),
v3(tria.v3) {
    cout << "Copy constructor for: " << tria.objID << endl; // отладочный вывод

    objID = new char[strlen(tria.objID) + strlen("(копия)") + 1];
    strcpy(objID, tria.objID);
    strcat(objID, "(копия)");

    name = new char[strlen(tria.name) + 1];
    strcpy(name, tria.name);
    a = tria.a;
    b = tria.b;
    c = tria.c;
}
```

- Добавьте реализацию операции присваивания:

```
// Присвоить значение объекта tria
Triangle& Triangle::operator =(const Triangle& tria) {
    cout << "Assign operator: " << objID << " = " << tria.objID << endl;
    // отладочный вывод

    if (&tria == this) return *this;
    delete [] name;
    name = new char[strlen(tria.name) + 1];
    strcpy(name, tria.name);
    a = tria.a; b = tria.b; c = tria.c;
    return *this;
}
```

И в конструкторе копирования, и в операторе присваивания перед копированием содержимого полей, на которые указывают поля типа `char*`, для них выделя-

ется новая память. Обратите внимание, что в конструкторе копирования после переписи поля objID мы добавляем в конец этой строки текст «(копия)». А в операции присваивания это поле, идентифицирующее объект, вообще не затрагивается и остается в своем первоначальном значении. Все это полезно для целей отладки. Откомпилируйте и запустите программу. Выберите третий пункт меню. На экране должен появиться текст:

```
3
=== Поиск максимального треугольника ===
Constructor_2 for: triaMax (Треугольник 5)
Assign operator: triaMax = triaA
Assign operator: triaMax = triaB
Максимальный треугольник: Треугольник 2
Нажмите Enter.
```

Обратите внимание на отладочный вывод операции присваивания. Продолжим тестирование. Нажмите Enter. Программа выведет:

```
Destructor for: triaMax
===== Г л а в н о е   м е н ю =====
1 - вывести все объекты   3 - найти максимальный
2 - переместить           4 - определить отношение включения
5 - выход
```

Обратите внимание на то, что был вызван деструктор для объекта triaMax, а не triaB. Продолжим тестирование. Введите цифру 5. Программа выведет:

```
5
Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: triaB
Destructor for: triaA
```

Все. Программа работает, как часы.

Но, однако, мы еще не все сделали. Нам осталось решить самую сложную подзадачу — определение *отношения включения* одного треугольника в другой.

Этап 4

Из многочисленных подходов к решению этой подзадачи наш выбор остановился на алгоритме, в основе которого лежит определение относительного положения точки и вектора на плоскости¹. Вектор — это направленный отрезок прямой линии, начинающийся в точке `beg_p` и заканчивающийся в точке `end_p`. При графическом изображении конец вектора украшают стрелкой. Теперь призовите

¹ Известен другой алгоритм решения этой задачи, основанный на использовании формулы Герона. Наш выбор обоснован тем, что это решение гармонично вписывается в технологию «нисходящего проектирования».

ваше пространственное воображение или вооружитесь карандашом и бумагой, чтобы проверить следующее утверждение. Вектор (beg_p , end_p) делит плоскость на пять непересекающихся областей:

- 1) все точки *слева* от воображаемой бесконечной прямой¹, на которой лежит наш вектор (область LEFT),
- 2) все точки *справа* от воображаемой бесконечной прямой, на которой лежит наш вектор (область RIGHT),
- 3) все точки на воображаемом продолжении прямой *назад* от точки beg_p в бесконечность (область BEHIND),
- 4) все точки на воображаемом продолжении прямой *вперед* от точки end_p в бесконечность (область AHEAD),
- 5) все точки, *принадлежащие* самому вектору (область BETWEEN).

Для выяснения относительного положения точки, заданной некоторым объектом класса Point, добавим в класс Point перечисляемый тип:

```
enum ORIENT { LEFT, RIGHT, AHEAD, BEHIND, BETWEEN };
```

а также метод Classify(beg_p , end_p), возвращающий значение типа ORIENT для данной точки относительно вектора (beg_p , end_p).

Обладая этим мощным методом, совсем нетрудно определить, находится ли точка внутри некоторого треугольника. Мы договорились перед началом решения задачи, что треугольники будут задаваться перечислением их вершин в порядке изображения их на плоскости *по часовой стрелке*. То есть каждая пара вершин образует вектор, и эти векторы следуют один за другим по часовой стрелке. При этом условии некоторая точка находится внутри треугольника тогда и только тогда, когда ее ориентация относительно каждой вектора-стороны треугольника имеет одно из двух значений: либо RIGHT, либо BETWEEN. Эту подзадачу будет решать метод InTriangle() в классе Point.

Изложим по порядку, какие изменения нужно внести в тексты модулей.

1. Модуль Point.h.

- Добавьте перед объявлением класса Point объявление нового типа ORIENT, а также упреждающее объявление типа Triangle:

```
enum ORIENT { LEFT, RIGHT, AHEAD, BEHIND, BETWEEN };
class Triangle;
```

Последнее необходимо, чтобы имя типа Triangle было известно компилятору в данной единице трансляции, так как оно используется в сигнатуре метода InTriangle().

- Добавьте внутри класса Point объявления функций:

```
Point operator +(Point&);
Point operator -(Point&);
```

¹ Точнее говоря, от воображаемого *бесконечного вектора*, поскольку направление здесь очень важно.

```

double Length() const;           // определяет длину вектора точки
                                // в полярной системе координат
ORIENT Classify(Point&, Point&) const; // определяет положение точки
                                // относительно вектора,
                                // заданного двумя точками
bool InTriangle(Triangle&) const; // определяет,
                                // находится ли точка внутри
                                // треугольника

```

Функция-операция «-» и метод Length() будут использованы при реализации метода Classify(), а функция-операция «+» добавлена для симметрии. Метод Classify(), в свою очередь, вызывается из метода InTriangle().

1. Модуль Point.cpp.

- Добавьте после директивы #include <iostream> директиву #include <math.h>. Она необходима для использования функции sqrt(x) из математической библиотеки C++ в алгоритме метода Length().
- Добавьте после директивы #include «Point.h» директиву #include «Triangle.h».
- Последняя необходима в связи с использованием имени класса Triangle в данной единице трансляции.
- Добавьте реализацию функций-операций:

```

Point Point::operator +(Point& p) {
    return Point(x + p.x, y + p.y);
}
Point Point::operator -(Point& p) {
    return Point(x - p.x, y - p.y);
}

```

- Добавьте реализацию метода Length():

```

double Point::Length() const {
    return sqrt(x*x + y*y);
}

```

- Добавьте реализацию метода Classify():

```

ORIENT Point::Classify(Point& beg_p, Point& end_p) const {
    Point p0 = *this;
    Point a = end_p - beg_p;
    Point b = p0 - beg_p;
    double sa = a.x * b.y - b.x * a.y;

    if (sa > 0.0) return LEFT;
    if (sa < 0.0) return RIGHT;

    if ((a.x * b.x < 0.0) || (a.y * b.y < 0.0)) return BEHIND;
    if (a.Length() < b.Length()) return AHEAD;

    return BETWEEN;
}

```

Алгоритм заимствован из [6], поэтому мы не будем здесь подробно его разбирать. Обратите внимание, что аргументы передаются в функцию по ссылке — это позволяет избежать вызова конструктора копирования.

- Добавьте реализацию метода `InTriangle()`:

```
bool Point::InTriangle(Triangle& tria) const {
    ORIENT or1 = Classify(tria.Get_v1(), tria.Get_v2());
    ORIENT or2 = Classify(tria.Get_v2(), tria.Get_v3());
    ORIENT or3 = Classify(tria.Get_v3(), tria.Get_v1());

    if ((or1 == RIGHT || or1 == BETWEEN)
        && (or2 == RIGHT || or2 == BETWEEN)
        && (or3 == RIGHT || or3 == BETWEEN)) return true;
    else return false;
}
```

2. Модуль `Triangle.h`: добавьте в классе `Triangle` объявление дружественной функции (все равно, в каком месте):

```
friend bool TriaInTria(Triangle, Triangle); // Определить,
// входит ли один треугольник во второй
```

3. Модуль `Triangle.cpp`: добавьте в конец файла реализацию внешней дружественной функции:

```
// Определить, входит ли треугольник trial в треугольник tria2
bool TriaInTria(Triangle trial, Triangle tria2) {
    Point v1 = trial.Get_v1();
    Point v2 = trial.Get_v2();
    Point v3 = trial.Get_v3();
    return (v1.InTriangle(tria2) &&
            v2.InTriangle(tria2) &&
            v3.InTriangle(tria2));
    return true;
}
```

Результат, возвращаемый функцией, основан на проверке вхождения каждой вершины первого треугольника (`trial`) во второй треугольник (`tria2`).

4. Модуль `Main.cpp`: замените заглушку функции `IsIncluded()` следующим кодом:

```
// ----- определение отношения включения
void IsIncluded(Triangle* p_tria[], int k) {
    cout << "=====  
cout << "Введите номер 1-го треугольника (от 1 до " << k << "): ";
    int i1 = GetNumber(1, k) - 1;

    cout << "Введите номер 2-го треугольника (от 1 до " << k << "): ";
    int i2 = GetNumber(1, k) - 1;

    if (TriaInTria(*p_tria[i1], *p_tria[i2]))
        cout << p_tria[i1]->GetName() << " - входит в - "
            << p_tria[i2]->GetName() << endl;
```

```

else
    cout << p_tria[i1]->GetName() << " - не входит в - "
    << p_tria[i2]->GetName() << endl;
ExitBack();
}

```

Модификация проекта завершена. Откомпилируйте и запустите программу. Выберите четвертый пункт меню. Следуя указаниям программы, введите номера сравниваемых треугольников, например 1 и 2. Вы должны получить следующий результат:

```

4
===== Отношение включения =====
Введите номер 1-го треугольника (от 1 до 4): 1
Введите номер 2-го треугольника (от 1 до 4): 2
Copy constructor for: triaB
Copy constructor for: triaA
Destructor for: triaA(копия)
Destructor for: triaB(копия)
Треугольник 1 - входит в - Треугольник 2
Нажмите Enter.

```

Обратите внимание на отладочную печать: конструкторы копирования вызываются при передаче аргументов в функцию `TrianTria()`, а перед возвратом из этой функции вызываются соответствующие деструкторы.

Проведите следующий эксперимент: удалите с помощью скобок комментария конструктор копирования в файлах `Triangle.h` и `Triangle.cpp`, откомпилируйте проект и повторите тестирование четвертого пункта меню. Полюбовавшись результатом, верните проект в нормальное состояние.

Протестируйте остальные пункты меню, вызывая их в произвольном порядке.

Когда вы сочтете, что тестирование завершено, уберите с помощью символов комментария `//` всю отладочную печать из конструкторов, деструктора и операции присваивания. Повторите тестирование без отладочной печати.

Решение задачи 1.2 завершено.

Давайте повторим наиболее важные моменты этого семинара.

1. Использование классов лежит в основе объектно-ориентированной декомпозиции программных систем, которая является более эффективным средством борьбы со сложностью систем, чем функциональная декомпозиция.
2. В результате декомпозиции система разделяется на компоненты (модули, функции, классы). Чтобы обеспечить высокое качество проекта, его способность к модификациям, удобство сопровождения, необходимо учитывать сцепление внутри компонента (оно должно быть сильным) и связанность между компонентами (она должна быть слабой), а также правильно распределять обязанности между компонентом-клиентом и компонентом-сервером.
3. Класс — это определяемый пользователем тип, лежащий в основе ООП. Класс содержит ряд полей (переменных), а также методов (функций), имеющих доступ к этим полям.

4. Доступ к отдельным частям класса регулируется с помощью ключевых слов: `public` (открытая часть), `private` (закрытая часть) и `protected` (защищенная часть). Последний вид доступа имеет значение только при наследовании классов. Методы, расположенные в открытой части, формируют *интерфейс* класса и могут вызываться через соответствующий объект.
5. Если в классе имеются поля типа указателей и осуществляется динамическое выделение памяти, то необходимо позаботиться о создании конструктора копирования и перегрузке операции присваивания.
6. Для создания полного, минимального и интуитивно понятного интерфейса класса широко применяется перегрузка методов и операций.

Задания

Вариант 1

Описать класс, реализующий стек. Написать программу, использующую этот класс для моделирования T-образного сортировочного узла на железной дороге. Программа должна разделять на два направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность формирования состава из файла и с клавиатуры.

Вариант 2

Описать класс, реализующий бинарное дерево, обладающее возможностью добавления новых элементов, удаления существующих, поиска элемента по ключу, а также последовательного доступа ко всем элементам.

Написать программу, использующую этот класс для представления англо-русского словаря. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса. Предусмотреть возможность формирования словаря из файла и с клавиатуры.

Вариант 3

Построить систему классов для описания плоских геометрических фигур: круга, квадрата, прямоугольника. Предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и вращения на заданный угол.

Написать программу, демонстрирующую работу с этими классами. Программа должна содержать меню, позволяющее осуществить проверку всех методов классов.

Вариант 4

Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность отдельного изменения составных частей адреса, создания и уничтожения объектов этого класса.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 5

Составить описание класса для представления комплексных чисел. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 6

Составить описание класса для объектов-векторов, задаваемых координатами концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 7

Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости, изменение размеров, построение наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 8

Составить описание класса для определения одномерных массивов целых чисел (векторов). Предусмотреть возможность обращения к отдельному элементу массива с контролем выхода за пределы массива, возможность задания произвольных границ индексов при создании объекта, возможность выполнения операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов, умножения и деления всех элементов массива на скаляр, вывода на экран элемента массива по заданному индексу, вывода на экран всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 9

Составить описание класса для определения одномерных массивов строк фиксированной длины. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы массива, выполнения опера-

ций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывод на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 10

Составить описание класса многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Предусмотреть методы для вычисления значения многочлена для заданного аргумента, операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена, вывод на экран описания многочлена.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 11

Составить описание класса одномерных массивов строк, каждая строка задается длиной и указателем на выделенную для нее память. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы массивов, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывод на экран элемента массива и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 12

Составить описание класса, обеспечивающего представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывода на экран подматрицы любого размера и всей матрицы.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 13

Написать класс для эффективной работы со строками, позволяющий форматировать и сравнивать строки, хранить в строках числовые значения и извлекать их. Для этого необходимо реализовать:

- перегруженные операции присваивания и конкатенации;
- операции сравнения и приведения типов;
- преобразование в число любого типа;
- форматный вывод строки.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 14

Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки книг по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 15

Описать класс «записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 16

Описать класс «студенческая группа». Предусмотреть возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 17

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- сложение, вычитание, умножение, деление (+, -, *, /) (умножение и деление, как на другую матрицу, так и на число);
- комбинированные операции присваивания (+=, -=, *=, /=);
- операции сравнения на равенство/неравенство;
- операции вычисления обратной и транспонированной матрицы, операцию возведения в степень;
- методы вычисления детерминанта и нормы;
- методы, реализующие проверку типа матрицы (квадратная, диагональная, нулевая, единичная, симметрическая, верхняя треугольная, нижняя треугольная);
- операции ввода/вывода в стандартные потоки.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 18

Описать класс «множество», позволяющий выполнять основные операции — добавление и удаление элемента, пересечение, объединение и разность множеств.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 19

Описать класс, реализующий стек. Написать программу, использующую этот класс для отыскания прохода по лабиринту.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Каждый квадрат определяется его координатами в матрице. После отыскания прохода программа печатает найденный путь в виде координат квадратов.

Вариант 20

Описать класс «предметный указатель». Каждый компонент указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Предусмотреть возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Семинар 2 Наследование

Теоретический материал: с. 200–210.

На самом деле на этом семинаре мы рассмотрим более широкий круг вопросов.

- Наследование классов и полиморфизм.
- Отношения между классами. Диаграммы классов на языке UML.
- Технология проектирования программ с учетом будущих изменений. Введение в использование шаблонов (паттернов) проектирования.

Наследование классов

Классы в объектно-ориентированных программах используются для моделирования концепций реального и программного мира. Концепции (или сущности) предметной области находятся в различных взаимоотношениях. Одно из таких взаимоотношений — *отношение наследования* (именуемое также отношением *родитель/потомок* или отношением *обобщение/специализация*).

Например, если бы в условии задачи 1.2 содержалось требование обработки не только треугольников, но и четырехугольников, то анализ нового класса Tetragon выявил бы наличие общих черт с классом Triangle. Причиной такой общности является то, что и треугольники, и четырехугольники являются частным (специальным, конкретным) случаем более общего понятия «многоугольник». Поэтому было бы логичным создать класс Polygon, содержащий элементы, общие для классов Triangle и Tetragon, а последние два класса объявить «наследниками» базового (родительского) класса Polygon. Язык C++ позволяет легко это сделать:

```
class Polygon {  
    // . . .  
};  
class Triangle : public Polygon {  
public:  
    Show();  
};
```

```
class Tetragon : public Polygon {
public:
    Show();
};
```

В этом примере производные классы Triangle и Tetragon наследуют все элементы базового класса Polygon, но каждый из них имеет свой собственный метод Show(). Иногда отношение наследования называют отношением «*is a*», что можно перевести как «представляет собой». В данном примере Triangle *is a* Polygon, в такой же мере и Tetragon *is a* Polygon.

Общий синтаксис создания производного класса при *простом наследовании*:

```
class имя : ключ_доступа имя_базового_класса {
    // тело класса
};
```

В случае *множественного наследования* после двоеточия перечисляются через запятую все базовые классы со своими ключами (модификаторами) доступа.

Производный класс, в свою очередь, сам может служить базовым классом. Такой набор связанных классов традиционно называется *иерархией классов*. Иерархия чаще всего является деревом, но может иметь и более общую структуру графа.

В примерах предыдущего семинара доступ к элементам класса регулировался с помощью двух модификаторов (спецификаторов): `private` — закрытая часть класса, `public` — открытая часть класса. Для базовых классов возможно использование еще одного модификатора — `protected`, который определяет так называемую *защищенную* часть класса. Смысл «защиты» заключается в том, что элементы этой части класса являются доступными для любого производного класса, но в то же время они недоступны извне классов данной иерархии.

Кроме этого, доступность в производном классе регулируется *ключом доступа к базовому классу*, указываемому в объявлении производного класса. Этот ключ определяет *вид наследования*: открытое (`public`), защищенное (`protected`) или закрытое (`private`).

Открытое наследование сохраняет статус доступа всех элементов базового класса, *защищенное* — понижает статус доступа `public` элементов базового класса до `protected`, и наконец, *закрытое* — понижает статусы доступа `public` и `protected` элементов базового класса до `private`. Заметим, что в C++ отношение между классами «*is a*» имеет место только при открытом наследовании.

Замещение функций базового класса

Иногда в производном классе требуется несколько иная реализация метода, унаследованного из базового класса. Язык позволяет это сделать. *Замещение (переопределение)* метода производится путем объявления в производном классе метода с таким же именем. Если понадобится все-таки вызвать из потомка метод предка, используется операция доступа к области видимости `::`, например:

```
#include <iostream>
using namespace std;
```

```
class Base {
public:
    void Display() { cout << "Hello, world!" << endl; }
};

class Derived : public Base {
public:
    void Display() {
        Base::Display(); // Вызов метода базового класса
        cout << "How are you?" << endl;
    }
};

class SubDerived : public Derived {
public:
    void Display() {
        Derived::Display(); // Вызов метода базового класса
        cout << "Bye!" << endl;
    }
};

int main() {
    SubDerived sd;
    sd.Display();
    return 0;
}
```

Эта программа, не лишенная оригинальности, выведет на экран:

```
Hello, world!
How are you?
Bye!
```

Конструкторы и деструкторы в производном классе

Конструкторы и деструкторы из базового класса не наследуются, поэтому при создании производного класса встает вопрос, нужны ли эти методы и как они должны быть реализованы.

Решая вопрос о *конструкторах* производного класса, руководствуйтесь следующими правилами:

- ❑ Если в базовом классе вообще нет конструктора или есть конструктор по умолчанию, то производному классу конструктор нужен только в том случае, когда требуется инициализировать поля, введенные в этом классе.
- ❑ Если вы не определили ни одного конструктора, компилятор самостоятельно создаст конструктор по умолчанию, из которого будет вызван конструктор по умолчанию базового класса.

- Если в базовом классе есть конструктор с аргументами, то в производном классе, как правило, требуется задать конструктор со списком аргументов, включающим значения для передачи конструктору базового класса; конструктор базового класса надо вызвать в списке инициализации.

Необходимость в *деструкторе* для производного класса определяется тем, нужно ли освобождать какие-либо ресурсы, выделенные в конструкторе. Если такой необходимости нет, то можно доверить компилятору создать деструктор по умолчанию. В нем обеспечивается вызов деструктора базового класса.

На этапе выполнения программы при создании объекта производного класса сначала вызываются конструкторы базовых классов, начиная с самого верхнего уровня, затем конструкторы объектов-элементов класса, и в последнюю очередь — конструктор класса. При уничтожении объекта (например, когда покидается область его видимости) деструкторы вызываются в порядке, обратном вызову конструкторов.

Устранение неоднозначности при множественном наследовании

Предположим, что от базового класса А, имеющего некоторый элемент *x*, наследуются два класса В и С, а класс D является производным от этих двух классов (множественное наследование). Если попытаться обратиться к элементу *x* из методов класса D, то компилятор воспримет выражение `D::x` как неоднозначное¹ и прекратит работу. Для решения этой проблемы в C++ предусмотрен механизм, благодаря которому в класс D будет включена только одна копия класса А. Достигается это добавлением спецификатора `virtual` перед модификаторами доступа к А в объявлениях классов В и С:

```
#include <iostream>
using namespace std;

class A {
public:
    A(int _x = 0) { x = _x; }
protected:
    int x;
};

class B : virtual public A {
public:
    void AddB(int y) { x += y; }
};

class C : virtual public A {
public:
    void AddC(int y) { x += y; }
};
```

¹ Поскольку неясно, какой элемент класса А имеется в виду: унаследованный через класс В или унаследованный через класс С.

```

class D : public B, public C {
public:
    void ShowX() { cout << "x = " << x << endl; }
};

int main() {
    D d;
    d.ShowX();
    d.AddB(10); d.ShowX();
    d.AddB(5); d.ShowX();
    return 0;
}

```

Доступ к объектам иерархии

Эффективнее всего работать с объектами одной иерархии через указатель на базовый класс. Дело в том, что при открытом (`public`) наследовании можно присваивать такому указателю адрес объекта как базового класса, так и любого производного класса. Такая возможность представляется вполне логичной: в качестве представителя более общего класса (например, класса `Polygon`) используется объект специализированного класса (например, класса `Triangle` или `Tetragon`).

Однако, если не принять специальных мер (о которых говорится в следующем разделе), то при работе с таким указателем — независимо от того, какой адрес ему присвоен — оказываются доступными *только элементы базового класса*. Причиной является то, что в общем случае во время компиляции компилятор не может знать, с каким объектом на самом деле связан этот указатель. Например, рассмотрим следующий фрагмент:

```

class Base {
public:
    // . . .
    void Modify();
};

class Derived : public Base {
public:
    void Modify();
};

int main() {
    int mode;
    Base* pB;
    cin >> mode;
    if (mode == 1) pB = new Base;
    else pB = new Derived;
    pB->Modify(); // на что указывает pB?
    return 0;
}

```

Так как компилятор не может предсказать, какой выбор будет произведен на этапе выполнения, то он выбирает метод по типу указателя pB, то есть Base::Modify(). Такая стратегия называется *ранним*, или *статическим, связыванием*. Поэтому, если в производном классе замещен некоторый метод базового класса (например, Derived::Modify()), то он оказывается недоступным. Продемонстрируем это на примере:

```
#include <iostream>
using namespace std;

class Base {
public:
    Base(int _x = 10) { x = _x; }
    void ShowX() { cout << "x = " << x << " "; }
    void ModifyX() { x *= 2; }
protected:
    int x;
};

class Derived : public Base {
public:
    void ModifyX() { x /= 2; }
};

int main() {
    Base b;    Derived d;
    b.ShowX(); d.ShowX();

    Base* pB;
    pB = &b;   pB->ModifyX();   pB->ShowX();
    pB = &d;   pB->ModifyX();   pB->ShowX();
    return 0;
}
```

Эта программа выведет на экран:

```
x = 10; x = 10; x = 20; x = 20;
```

то есть второй вызов метода ModifyX() происходит также из базового класса.

Виртуальные методы

Проблема доступа к методам, переопределенным в производных классах, через указатель на базовый класс решается в C++ посредством использования *виртуальных методов*. Чтобы сделать некоторый метод виртуальным, надо в базовом классе предварить его заголовок спецификатором virtual. После этого он будет восприниматься как виртуальный во всех производных классах иерархии.

По отношению ко всем виртуальным методам компилятор применяет стратегию *позднего*, или *динамического, связывания*. Это означает, что на этапе компиляции

он не определяет, какой из методов должен быть вызван, а передает ответственность программе, которая принимает решение на этапе выполнения, когда уже точно известно, каков тип объекта, на который указывает наш указатель. Все сказанное относится также к вызову методов *по ссылке* на базовый класс.

Для реализации динамического связывания компилятор создает *таблицу виртуальных методов (vtbl)*, а к каждому объекту добавляет скрытое поле ссылки (*vptr*) на таблицу *vtbl*¹. Это несколько снижает эффективность программы, поэтому рекомендуется делать виртуальными только те методы, которые переопределяются в производных классах².

Чтобы увидеть динамическое связывание в действии, модифицируйте приведенную выше программу добавлением спецификатора `virtual` перед заголовком метода `ModifyX()` в базовом классе, так что теперь определение метода примет вид:

```
virtual void ModifyX() { x *= 2; }
```

Модифицированная программа должна вывести:

```
x = 10; x = 10; x = 20; x = 5;
```

то есть второй вызов метода `ModifyX()` происходит теперь из производного класса.

Виртуальный механизм работает только при использовании указателей и ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется *полиморфным*.

Если базовый класс содержит хотя бы один виртуальный метод, то рекомендуется всегда снабжать этот класс *виртуальным деструктором*, даже если он ничего не делает³. Наличие такого виртуального деструктора предотвратит некорректное удаление объектов производного класса, адресуемых через указатель на базовый класс, так как в противном случае деструктор производного класса вызван не будет⁴.

Абстрактные классы. Чисто виртуальные методы

Иногда, разрабатывая иерархию классов, можно получить базовый класс, для которого создание объекта как бы теряет смысл. Например, иерархия классов геометрических фигур может произрастать из базового класса `Shape`, а в качестве производных будут выступать классы `Polygon`, `Ellipse` и т. д. Одним из методов базового класса будет, видимо, функция `Show()` — показать объект. Но как можно показывать объект, не имеющий конкретной формы?

Классы, для которых нет смысла создавать объекты, объявляют как *абстрактные*. Абстрактный класс — это класс, содержащий хотя бы один чисто виртуаль-

¹ Более подробно об этом см. учебник.

² Или являются потенциальными кандидатами на такое переопределение.

³ То есть имеет пустое тело.

⁴ В небольшой учебной программе уничтожение объекта производного класса без вызова его деструктора может пройти без последствий. В нетривиальных программах это может привести к очень трудно диагностируемым ошибкам.

ный метод. *Чисто виртуальный метод* — это метод, объявленный в классе, но не имеющий конкретной реализации. Синтаксис объявления чисто виртуального метода дополняется конструкцией = 0, например:

```
virtual void Show() = 0;
```

Компилятор не допустит создания объекта для абстрактного класса, если вы по забывчивости попытаетесь это сделать. Если в производном классе хотя бы одна чисто виртуальная функция базового класса останется без конкретной реализации, то такой производный класс также будет абстрактным классом, для которого запрещено создавать объекты.

ПРИМЕЧАНИЕ

Дополнительные проблемы, связанные с замещением методов в производном классе, рассматриваются на семинаре 4 при решении задачи 4.1.

Отношения между классами. Диаграммы классов на языке UML

Полиморфизм и наследование делают язык C++ чрезвычайно мощным инструментом для реализации объектно-ориентированной технологии проектирования. Но для эффективного восприятия новых технологических идей сегодня не обойтись без знания хотя бы основ ставшего уже стандартом *унифицированного языка моделирования UML*¹. Язык UML является визуальным средством представления моделей программ.

Под *моделями программ* понимается их графическое представление в виде различных диаграмм, отражающих связи между объектами в программном коде. Одной из основных диаграмм языка UML является *диаграмма классов*². Она описывает классы и отражает отношения, существующие между ними. Диаграмма классов чрезвычайно удобна для сопоставления различных вариантов проектных решений. Кроме того, эти диаграммы используют для описания так называемых *шаблонов проектирования*³ (*design patterns*), которые в сочетании с объектно-ориентированной парадигмой лежат в основе современного подхода к разработке программного обеспечения.

Класс изображается на диаграмме классов UML в виде прямоугольника, состоящего из трех частей. Имя класса указывается в верхней части. Список *атрибутов* (полей⁴), возможно, с указанием типов и значений, приводится в средней

¹ UML — Unified Modeling Language.

² Далее (при решении задачи 2.3) мы покажем использование еще одной разновидности диаграмм UML: *диаграммы видов деятельности*.

³ Слово «шаблон» в отношении термина «pattern» не вполне удачно, поскольку вызывает путаницу с термином «template». Поэтому на сегодняшний день во многих книгах издательства «Питер» термин «паттерн» используется без перевода. — *Примеч. ред.*

⁴ Термин *атрибут класса* в UML соответствует термину *поле класса* в C++.

части. В нижней части записывается список *операций* (методов¹), возможно, с указанием списка типов аргументов и типа возвращаемого значения. Имя *абстрактного класса*, так же как и имена *абстрактных операций* (чисто виртуальных методов), выделяется курсивом. Перед именем поля или метода может указываться спецификатор доступа с помощью одного из трех символов: + для public, - для private, # для protected. Для статических элементов класса после спецификатора доступа записывается символ \$.

Во второй и третьей частях могут указываться не все элементы класса, а только те, которые представляют интерес на данном уровне абстракции. Если обе эти части пусты, они могут быть опущены. Например, варианты изображения класса Triangle на различных этапах анализа решения задачи 1.2, которую мы рассматривали на семинаре 1, показаны на рис. 2.1.

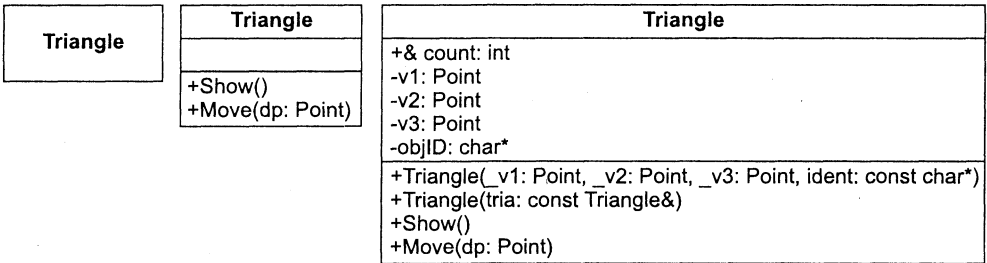


Рис. 2.1. Варианты изображения класса Triangle на диаграмме классов UML

Большинство объектно-ориентированных языков поддерживают следующие виды отношений между классами:

- ассоциация;
- наследование;
- агрегация;
- зависимость.

Рассмотрим, что означают и как изображаются эти виды отношений на диаграмме классов.

Ассоциация

Если два класса концептуально взаимодействуют друг с другом, то такое взаимодействие называется *ассоциацией*. Например, желая смоделировать торговую точку, мы обнаруживаем две абстракции: товары (класс Product) и продажи (класс Sale). Объект класса Sale — это некоторая сделка, в которой продано от 1 до n объектов класса Product. На рис. 2.2 ассоциация между этими двумя классами показана соединяющей их линией. Над линией рядом с обозначением класса может быть указана так называемая *кратность* (*multiplicity*²), указывающая, сколько объек-

¹ Термин *операция класса* в UML соответствует термину *метод класса* в C++.

² Используется также термин-синоним *cardinality*.

тов данного класса может быть связано с одним объектом другого класса. Для представления понятия «произвольное количество» в UML используется символ звездочки *.



Рис. 2.2. Отношения ассоциации

Ассоциация представляет наиболее абстрактную семантическую связь между двумя классами, которая выявляется на ранней стадии анализа. В дальнейшем она, как правило, конкретизируется и принимает вид одного из рассматриваемых далее отношений.

Наследование

Отношение *обобщения* (наследования, «*is a*») между классами показывает, что *подкласс* (производный класс) разделяет атрибуты и операции, определенные в одном или нескольких *суперклассах* (базовых классах). На диаграмме классов отношение наследования показывают линией со стрелкой в виде незакрашенного треугольника, которая указывает на базовый класс. Допускается объединять несколько стрелок в одну, как это показано на рис. 2.3, с тем чтобы разгрузить диаграмму.

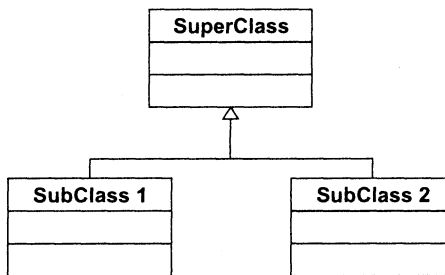


Рис. 2.3. Отношения наследования

Агрегация

Отношение *агрегации* между классами имеет место, когда один класс содержит в качестве составной части объекты другого класса. Иными словами, это отношение *целое/часть*, или отношение «*has a*», между двумя классами. На диаграмме такая связь обозначается линией со стрелкой в виде незакрашенного ромба, которая указывает на целое. На рис. 2.4 показан пример так называемой *нестрогой* агрегации (или просто агрегации). Действительно, конкретный объект класса СпортЗал может содержать не все компоненты (спортивные снаряды), присутствующие на схеме.

Как распознать агрегацию в программном коде? Для реализации нестрогой агрегации часть включается в целое *по ссылке*: на языке C++ это обычно указатель на соответствующий класс. Таким образом, если этот указатель равен нулю, то

компонент отсутствует. В зависимости от решаемой задачи такой компонент может появляться и исчезать динамически в течение жизни объекта целое.



Рис. 2.4. Отношения агрегации

Строгая агрегация имеет специальное название — *композиция*. Она означает, что компонент не может исчезнуть, пока объект целое существует. Пример такого отношения мы уже встречали на семинаре 1 при решении задачи 1.2: напомним, что класс `Triangle` содержал в себе три объекта класса `Point`. На диаграмме отношение композиции обозначается линией со стрелкой в виде закрашенного ромба (рис. 2.5).

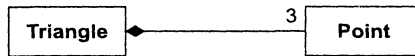


Рис. 2.5. Отношение композиции

Проще всего композицию реализовать включением объектов-компонентов *по значению*, как это и было сделано в приведенном примере. В то же время возможна реализация и включением по ссылке, если обеспечить следующее требование: время жизни компонентов должно либо совпадать с временем жизни объекта целое, либо перекрывать его.

Зависимость

Отношение *зависимости* (отношение *использования*) между двумя классами показывает, что один класс (`Client`) пользуется некоторыми услугами другого класса (`Supplier`), например:

- ❑ метод класса `Client` использует значения некоторых полей класса `Supplier`;
- ❑ метод класса `Client` вызывает некоторый метод класса `Supplier`;
- ❑ метод класса `Client` имеет сигнатуру, в которой упоминается класс `Supplier`.

На диаграмме такая связь обозначается пунктирной линией со стрелкой, указывающей на класс `Supplier`. Пример отношения зависимости, взятый опять из задачи 1.2, показан на рис. 2.6, где роль класса `Client` играет класс `Triangle`, а в качестве класса `Supplier` выступает класс `Point`.

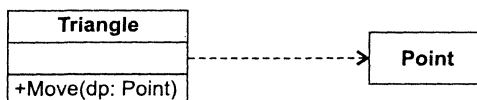


Рис. 2.6. Отношение зависимости (использования)

Проектирование программы с учетом будущих изменений

Одним из показателей качества программной системы является ее способность к модификациям в связи с появлением новых требований заказчика или пользователя. На первом семинаре мы уже говорили о критериях качества проекта и отмечали, что большое значение имеет правильная декомпозиция системы на компоненты (модули, классы, функции).

В случае удачной декомпозиции отдельные компоненты характеризуются сильным внутренним сцеплением и слабой внешней связанностью. Если система разработана с использованием такой стратегии, то, как правило, ее легче сопровождать и модифицировать. Однако тактические решения локальных проблем также могут влиять на то, сколь сложным окажется добавление новой функциональности к уже разработанному программному продукту. В данном разделе рассматриваются некоторые базовые концепции поиска таких решений.

Одной из причин негибкости процедурно-ориентированных систем является частое использование в них управляющих конструкций на базе оператора `switch`. Рассмотрим, например, типичную ситуацию: некоторая функция `SomeFunction()` в зависимости от значения параметра `mode` вызывает одну из обрабатывающих процедур:

```
typedef enum { mode1, mode2, mode3 } ModeType;
void SomeFunction(ModeType mode) {
    switch (mode) {
        case mode1: DoSomething1();    break;
        case mode2: DoSomething2();    break;
        case mode3: DoSomething3();    break;
    }
}
```

При этом перечень значений переменных типа `ModeType` согласован с заказчиком и утвержден техническим заданием. Вся система, таким образом, разрабатывается с использованием перечисляемого типа `ModeType`. Однако по истечении некоторого промежутка времени заказчик обнаруживает, что необходимо предусмотреть еще одно значение `mode4` параметра `mode`. Хорошо, если он еще платежеспособен и сможет заказать модификацию системы. Но для разработчиков системы наступят черные дни, когда нужно будет искать по всем модулям переключатели `switch` и вносить нужные добавления. Программисты хорошо знают, что после самого невинного изменения в коде любой функции проекта необходимо полное повторное тестирование всех модулей, использующих эту функцию, так как могут появиться неожиданные побочные эффекты.

Что же нового для решения таких проблем дает нам объектно-ориентированная технология? Оказывается, очень много, если грамотно воспользоваться полиморфизмом, а также композицией классов. Заметим, что любую задачу можно решить, используя только наследование классов, но, как правило, такие решения

получаются не гибкими для дальнейшей модификации. Самые оптимальные и красивые решения характеризуются сочетанием наследования и композиции классов. Кстати, программисты уже давно пытаются обобщить опыт своих предшествующих коллег, выделив самые удачные решения схожих проблем. Так появилось направление, называемое *шаблонами проектирования* (*design patterns*¹). Одну из пионерских работ в этой области опубликовали Э. Гамма (E. Gamma), Р. Хелм (R. Helm), Р. Джонсон (R. Johnson) и Дж. Влссидес (J. Vlissides) [7].

С формальной точки зрения шаблон проектирования представляет собой диаграмму классов, дающую решение некоторой часто встречающейся проблемы. Кроме этого, шаблон имеет некоторое имя, «пояснительную записку», содержащую сведения о том, в каких ситуациях он может оказаться полезным, и пример реализации на каком-либо языке программирования. Впоследствии указанная работа стала широко цитироваться и обсуждаться, а ее авторы получили шуточную кличку «банда четырех». Это очень упрощает ссылки: достаточно написать «GoF95» — аббревиатуру от «Gang of four, 1995²». В своей книге банда четырех описала 23 шаблона проектирования. Наиболее интересные из этих шаблонов приведены в приложении, а кое-какие «бандитские» идеи мы используем уже сейчас.

ПРИМЕЧАНИЕ

В рассматриваемых далее примерах мы воспользуемся очень удобными классами из стандартной библиотеки C++: `string` и `vector`. Хотя изучению этих классов посвящены семинары 5 и 6, тем не менее в ограниченном объеме их можно применять уже сейчас: строки типа `string` — взамен традиционных C-строк, объекты типа `vector` — взамен традиционных одномерных массивов. Благодаря автоматическому управлению выделением и освобождением памяти в этих классах мы можем писать более компактный программный код, не отвлекаясь на детали реализации. Соответствующий минимум сведений по работе с этими классами будет приведен ниже.

Вернемся к вопросу поиска более удачного решения проблемы типа «переключатель». Одна из рекомендаций, приведенных в [7], звучит так: «Найдите то, что должно (или может) *измениться* в вашем дизайне, и *инкапсулируйте сущности*, подверженные изменениям!». В нашем случае переменной сущностью являются вызываемые процедуры. Можем ли мы их инкапсулировать? Да — если реализуем эти процедуры как виртуальные методы полиморфных объектов!

Идея заключается в следующем: объект типа «переключатель» (класс `Switch`) должен иметь дело с некоторой абстрактной «процедурой вообще», а это можно реализовать, создав абстрактный класс `AbstractEntity` с чисто виртуальным методом `DoSomething()`. Конкретные процедуры в виде одноименных замещенных методов будут содержаться в производных от `AbstractEntity` классах `Entity1`, `Entity2` и т. д. Чтобы делать «осознанный» выбор, класс `Switch` должен быть осведомлен об адресах объектов `Entity1`, `Entity2` и т. д., — это можно обеспечить, поместив список

¹ В отечественной литературе можно встретить и другой перевод этого термина — *паттерны проектирования*.

² Книга была опубликована в оригинале в 1995 г.

указанных адресов в одно из его полей типа `std::vector<AbstractEntity*>`. Описываемая идея поясняется на диаграмме классов (рис. 2.7).

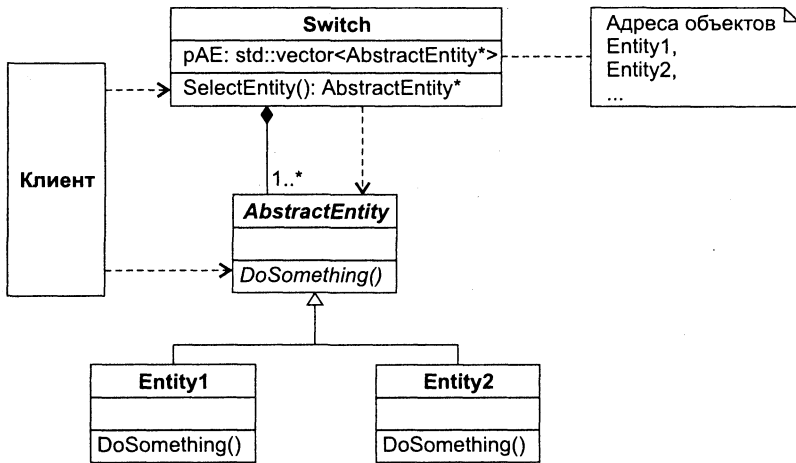


Рис. 2.7. Шаблон Switch (переключатель)

Обратите внимание, что между классами `Switch` и `AbstractEntity` имеется отношение *композиции* (стрелка с закрашенным ромбиком), так как класс `Switch` содержит поле типа `std::vector<AbstractEntity*>`. В то же время класс `Switch` *использует* класс `AbstractEntity` (пунктирная стрелка), поскольку один из его методов возвращает значение типа `AbstractEntity*`.

Как же все это работает? Клиент обращается с запросом `SelectEntity` к объекту `Switch` (то есть вызывает одноименный метод). Неважно, как реализован метод `SelectEntity()`, существенно то, что он возвращает значение типа `AbstractEntity*`, содержащее адрес объекта одного из подклассов класса `AbstractEntity`. После этого через полученный указатель клиент вызывает конкретную процедуру `DoSomething()` одного из объектов — `Entity1`, `Entity2`...

Чем примечательно предлагаемое решение? Необычайной легкостью модификации. Если возникла необходимость дополнить список переключаемых сущностей еще одним k -м экземпляром, то достаточно добавить производный класс `Entity k` , объявить объект этого класса и добавить адрес нового объекта в список `vector<AbstractEntity*>`.

А теперь мы рискнем заявить, что предложенное на рис. 2.7 решение есть не что иное, как еще один *шаблон проектирования*, который мы назвали `Switch`. Пример реализации этого шаблона будет дан в решении задачи 2.1. Мы отдаем себе отчет, что скорее всего аналогичное решение уже многократно применялось программистами на практике, но ведь решение *становится* шаблоном только после его *описания* на требуемом уровне абстракции! Во всяком случае, среди паттернов, описанных бандой четырех, наиболее близким к нашему является шаблон `Strategy`, который определяет семейство *алгоритмов*, инкапсулирует каждый из них и делает их взаимозаменяемыми.

ПРИМЕЧАНИЕ

При первом взгляде на диаграмму классов шаблона Strategy (он описан в приложении) может даже показаться, что он решает ту же задачу, что и шаблон Switch (класс Context вместо класса Switch, класс Strategy вместо класса AbstractEntity). Но, во-первых, есть отличия в мотивации: шаблон Switch предназначен для выбора произвольной сущности, которая не обязательно является алгоритмом, во-вторых, сама проблема выбора «выведена за рамки» шаблона Strategy, в то время как в шаблоне Switch ее решение поручено методу SelectEntity(). Эти различия становятся особенно наглядными при сравнении примеров реализации данных шаблонов.

Ну и, наконец, проблему «переключателя» можно обобщить для случая *многоуровневого* переключения. Например, решая задачу 2.2, в которой имеется двухуровневое переключение, мы выведем соответствующий шаблон DoubleSwitch.

Задача 2.1. Функциональный калькулятор

Разработать программу, имитирующую работу функционального калькулятора, который позволяет выбрать с помощью меню какую-либо из известных ему функций, затем предлагает ввести значение аргумента u , возможно, коэффициентов и после ввода выдает соответствующее значение функции.

В первой версии калькулятора «база знаний» содержит две функции:

- экспоненту $y = e^x$;
- линейную функцию $y = ax + b$.

Решение задачи начнем с выявления понятий/классов и их существенных взаимосвязей.

Интерфейс пользователя нашего калькулятора, как следует из его описания, должен обеспечить некоторое меню для выбора вида функции. У нас уже есть опыт создания меню для консольных приложений, в которых отсутствует оконная графика, — например, такое меню реализовано в задаче 1.2 (семинар 1). Функция menu() в той программе выводит список выбираемых пунктов с номерами и после ввода пользователем номера пункта возвращает это значение главной функции. Далее следует традиционный переключатель:

```
switch (Menu()) {
    case 1: Show(...); break;
    case 2: Move(...); break;
    ...
    case 5: //Конец работы
}
```

Но вряд ли стоит идти по пути повторного использования этого кода. Ведь мы уже знаем, что подобная реализация переключателя чревата проблемами в случае модификации программы. Мы также вооружены новым шаблоном проектирования Switch, который дает красивое, легко модифицируемое решение. Попробуем

применить этот шаблон на практике. Несложно увидеть, что роль класса Switch здесь будет играть класс Menu.

Что является изменяемой сущностью в нашей задаче? — Вид функции, для которой нужно выполнить вычисления (в более общем случае — вид некоторого объекта, для которого нужно выполнить некоторую операцию). Следовательно, требуется инкапсулировать эту изменяемую сущность так, чтобы класс Menu имел дело с некоторой абстрактной «функцией вообще» (с некоторым «объектом вообще»). Отсюда вывод: необходим абстрактный класс Function, обеспечивающий единый унифицированный интерфейс для всех его производных классов, в данном случае — для классов Exp и Line. В результате мы приходим к диаграмме классов, показанной на рис. 2.8.

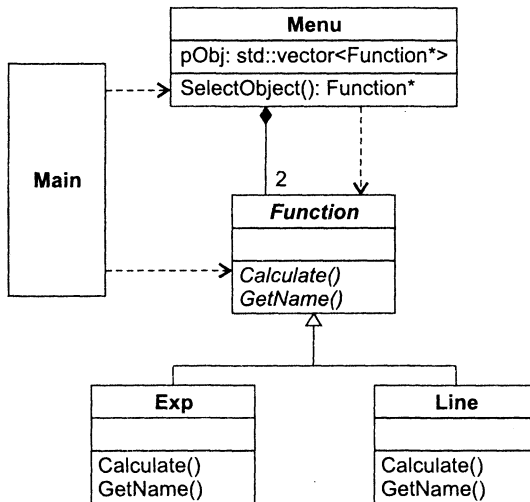


Рис. 2.8. Диаграмма классов программы «Функциональный калькулятор»

На диаграмме, естественно, показаны не все элементы классов, а только наиболее существенные для данного этапа анализа. Кроме метода Calculate(), исполняющего роль метода DoSomething() в шаблоне Switch, здесь появился еще метод GetName(), который извлекает значение поля name, содержащего наименование функции. Этим методом будет пользоваться объект Menu для вывода перечня выбираемых функций.

Прежде чем привести реализацию программы, скажем несколько слов о работе с классами std::string и std::vector.

Класс string позволяет создать строку s типа string, с которой очень удобно работать: ее можно инициализировать значением C-строки: std::string s("C-строка"); или присвоить ей значение другой строки посредством операции присваивания =. Память, выделенная для объекта s, автоматически освобождается, как только программа покидает область его видимости. Для использования класса необходимо подключить заголовочный файл <string>.

Контейнер `vector` является шаблонным классом и позволяет создавать динамические массивы¹. Для его использования необходимо подключить заголовочный файл `<vector>`. В классе есть конструктор по умолчанию, создающий вектор нулевой длины, а также конструктор с инициализацией создаваемого вектора обычным одномерным C-массивом. К имеющемуся вектору можно добавить в его конец новый элемент с помощью метода `push_back()`. Доступ к любому элементу вектора осуществляется по индексу, как и в обычном массиве². Количество элементов в векторе в любой момент можно определить методом `size()`. Поскольку `vector` — шаблонный класс, то он позволяет создавать конкретные экземпляры классов для любого типа элементов, задаваемого в качестве аргумента в угловых скобках после имени класса³. Поясним использование этих средств на следующем примере:

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int a[5] = { 5, 4, 3, 2, 1 };
    double b[] = { 1.1, 2.2, 3.3, 4.4 };

    vector<char> v1;
    v1.push_back('A'); v1.push_back('B'); v1.push_back('C');
    for (int i = 0; i < v1.size(); ++i)
        cout << v1[i] << ' ';
    cout << endl;

    vector<int> v2(a, a + 5);
    for (i = 0; i < v2.size(); ++i)
        cout << v2[i] << ' ';
    cout << endl;

    vector<double> v3(b, b + sizeof(b) / sizeof(double));
    for (i = 0; i < v3.size(); ++i)
        cout << v3[i] << ' ';
    cout << endl;

    return 0;
}
```

Эта программа должна вывести на экран:

```
A B C
5 4 3 2 1
2.2 3.3 4.4
```

¹ Термины *вектор* и *динамический массив* мы будем использовать как синонимы (иногда опуская слово *динамический*, когда это очевидно).

² Возможен доступ и через итератор, но это тема семинара 6.

³ Более подробно о шаблонных классах см. семинар 3.

В программе показано использование вектора v_1 , предназначенного для хранения элементов типа `char`, вектора v_2 — для хранения элементов типа `int`, и вектора v_3 — для элементов типа `double`.

Обратите внимание на два способа инициализации вектора одномерным массивом. Первый (для вектора v_2) используется, когда длина массива задана константой. Второй (для вектора v_3) оказывается единственным возможным, когда длина массива определяется на стадии компиляции.

Теперь приведем возможную реализацию программы «Функциональный калькулятор»:

```

////////////////////////////////////
// Проект Task2_1
////////////////////////////////////
// Function.h
#ifndef FUNCTION_H
#define FUNCTION_H

#include <string>

class Function {
public:
    virtual ~Function() {}
    virtual const std::string& GetName() const = 0;
    virtual void Calculate() = 0;
protected:
    double x;    // аргумент
};

#endif /* FUNCTION_H */
////////////////////////////////////
// Exp.h
#include "Function.h"

// Класс для представления функции  $y = e^x$ 
class Exp : public Function {
public:
    Exp() : name("e ^ x") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name; // мат. обозначение функции
};
extern Exp f_exp;
////////////////////////////////////
// Exp.cpp
#include <iostream>
#include <math.h>

```

```

#include "Exp.h"
using namespace std;
void Exp::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter x = ";    cin >> x;
    cin.get();
    cout << "y = " << exp(x) << endl;
    cin.get();
}
// Глобальный объект
Exp f_exp;
/////////////////////////////////////////////////////////////////
// Line.h
#include "Function.h"
// Класс для представления функции  $y = a * x + b$ 
class Line : public Function {
public:
    Line() : name("a * x + b") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name;    // мат. обозначение функции
    double a;
    double b;
};
extern Line f_line;
/////////////////////////////////////////////////////////////////
// Line.cpp
#include <iostream>
#include "Line.h"
using namespace std;

void Line::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter a = ";    cin >> a;
    cout << "Enter b = ";    cin >> b;
    cout << "Enter x = ";    cin >> x;
    cin.get();
    cout << "y = " << (a * x + b) << endl;
    cin.get();
}
// Глобальный объект
Line f_line;
/////////////////////////////////////////////////////////////////
// Menu.h
#include <vector>
#include "Function.h"

```

```

class Menu {
public:
    Menu(std::vector<Function*>);
    Function* SelectObject() const;
private:
    int SelectItem(int) const;
    std::vector<Function*> pObj;
};
////////////////////////////////////
// Menu.cpp
#include <iostream>
#include "Menu.h"
using namespace std;

Menu::Menu(vector<Function*> _pObj) : pObj(_pObj) {
    pObj.push_back(0); // для выбора пункта 'Exit'
}

Function* Menu::SelectObject() const {
    int nItem = pObj.size();
    cout << "=====\n";
    cout << "Select one of the following function:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i+1 << ". "; // номер пункта меню на единицу
                           // больше, чем индекс массива pObj
        if (pObj[i]) cout << pObj[i]->GetName() << endl;
        else cout << "Exit" << endl;
    }
    int item = SelectItem(nItem);
    return pObj[item - 1];
}

int Menu::SelectItem(int nItem) const {
    cout << "-----\n";
    int item;
    while (true) {
        cin >> item;
        if ((item > 0) && (item <= nItem)
            && (cin.peek() == '\n')) {
            cin.get(); break;
        }
        else {
            cout << "Error (must be number from 1 to "
                << nItem << "):" << endl;
            cin.clear();
            while (cin.get() != '\n') {};
        }
    }
    return item;
}

```

```

////////////////////////////////////
// Main.cpp
#include <iostream>
#include "Function.h"
#include "Exp.h"
#include "Line.h"
#include "Menu.h"
using namespace std;

Function* pObjjs[] = { &f_exp, &f_line };
vector<Function*> funcList(pObjjs, pObjjs +
                          sizeof(pObjjs) / sizeof(Function*));

int main() {

    Menu menu(funcList);

    while (Function* pObjj = menu.SelectObject())
        pObjj->Calculate();

    cout << "Bye!\n";
    return 0;
}
//————— конец проекта Task2_1 —————
////////////////////////////////////

```

Несколько пояснений по тексту программы.

- ❑ В абстрактном классе `Function` конструктор отсутствует, так как объекты этого класса создаваться не будут. Класс содержит *виртуальный* деструктор (см. рекомендацию, приведенную в разделе «Виртуальные методы»). В связи с тем, что все методы класса — чисто виртуальные, в проекте отсутствует модуль реализации класса (файл `Function.cpp`). Заметим, что существует рекомендация размещать в абстрактном классе *только методы* (то есть делать класс чисто интерфейсным), а все поля объявлять в производных классах. При таком подходе гарантируется, что любые изменения и дополнения не затронут абстрактный класс. В данном случае мы все-таки поместили поле `x` для значения аргумента, поскольку рассматриваем только функции с одним аргументом и при этом полагаем, что аргумент всегда имеет тип `double`. А вот значения для коэффициентов мы перенесли в поля производных классов, так как эта сущность более специализированная.
- ❑ Обратите внимание на сигнатуру метода `virtual const std::string& GetName() const` в классе `Function`. Метод объявлен константным (модификатор `const` в конце сигнатуры), так как он не должен изменять состояние полей класса. Кроме того, для повышения эффективности (чтобы исключить вызов конструктора копирования) мы хотим вернуть значение по ссылке: `std::string&`. Однако, если мы опишем сигнатуру `virtual std::string& GetName() const`, ее не пропустит компилятор, так как возврат по ссылке `string&` противоречит модифика-

тору `const`. Выход из этого противоречия — возврат по константной ссылке: `const std::string&`.

- ❑ Конкретные методы `Calculate()` в классах `Exp` и `Line` обеспечивают ввод исходных данных (аргумент `i`, по необходимости, значения коэффициентов) и вычисляют значение соответствующей функции.
- ❑ Отметим, что в модуле `Exp.cpp` объявлен глобальный объект `Exp f_exp`, а в модуле `Line.cpp` — глобальный объект `Line f_line`.
- ❑ Конструктор класса `Menu` обеспечивает инициализацию поля `pObj` вектором объектов типа `Function*`, передаваемым через его аргумент. После копирования аргумента в поле `pObj` (в инициализаторе конструктора) к созданному массиву добавляется нулевой указатель — это делает метод `pObj.push_back(0)`. Нулевой указатель используется для реализации последнего стандартного пункта меню под названием «Exit» (см. метод `Menu::SelectObject()`).
- ❑ В методе `Menu::SelectObject()` количество пунктов меню определяется вызовом `pObj.size()`. После вывода оператором `for` списка пунктов меню вызывается метод `SelectItem()`, обеспечивающий защищенный от ошибок ввод пользователем номера выбираемого пункта. Заметим, что метод `SelectItem()` объявлен в секции `private` класса `Menu`, так как он используется только внутри класса.
- ❑ В главном модуле `main.cpp` объявлен глобальный массив `pObjs[]`, содержащий список адресов объектов `&f_exp`, `&f_line`. Этим массивом инициализируется вектор `funclist`.
- ❑ Функция `main()` получилась на удивление лаконичной: в ней объявлен объект `menu` класса `Menu`, которому передается вектор `funclist`, а затем идет цикл `while`, работа которого очевидна. В том смысле, что метод `menu.SelectObject()` возвращает адрес (указатель) конкретного объекта — `f_exp` или `f_line`, — через который вызывается метод `Calculate()` соответствующего класса. При возврате нулевого адреса цикл завершается, и программа прощается с пользователем.
- ❑ Пояснения к программе останутся неполными, если мы не расскажем, по какому принципу размещаются *директивы* `using` в тексте модулей. Напомним, что благодаря директиве `using namespace std` становятся видимыми (известными компилятору) все имена из пространства имен `std` стандартной библиотеки C++. Например, в нашей программе такими именами являются `cin`, `cout`, `string`, `vector`. Есть и другие способы сделать видимым некоторое конкретное имя: с помощью *объявления* `using` (например, `using std::string`)¹ или с помощью квалификатора `std::` (например, `std::string`). Проще всего использовать директиву `using`. Но при этом возникают две опасности: возможность появления конфликта имен из пространства `std` с вашими именами и, что более неприятно, возможность неоднозначной трактовки вашего кода компилятором при беспорядочном употреблении директив `using`. В этом вопросе мы солидарны с Гербом Саттером [8] и приводим ниже его рекомендации.

¹ Такое объявление действует до конца блока, в котором оно сделано.

СОВЕТ

Правило № 1. *Не используйте* директивы `using`, равно как и объявления `using`, в заголовочных файлах. Именно в этом случае указанные выше опасности становятся весьма вероятными. Поэтому в заголовочных файлах используйте только квалификатор `std::` перед конкретным именем.

Правило № 2. *В файлах реализации* директивы и объявления `using` не должны располагаться *перед* директивами `include`. В противном случае имеется вероятность того, что использование `using` изменит семантику заголовочного файла из-за введения неожиданных имен.

Откомпилируйте и проверьте работу программы. Еще раз сравните код функций `main()` в задачах 1.2 и 2.1, чтобы увидеть элегантность нового решения, основанного на шаблоне проектирования `Switch`. А теперь представим, что от заказчика поступило предложение сделать наш калькулятор более «мощным», добавив в его «базу знаний» новую функцию — параболу. Доработка сведется к добавлению в проект класса `Parabola`, то есть следующих двух файлов:

```

////////////////////////////////////
// Parabola.h
#include "Function.h"
// Класс для представления функции  $y = a * x^2 + b * x + c$ 
class Parabola : public Function {
public:
    Parabola() : name("a * x^2 + b * x + c") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name; // мат. обозначение функции
    double a, b, c;
};
extern Parabola f_parabola;
////////////////////////////////////
// Parabola.cpp
#include <iostream>
#include "Parabola.h"
using namespace std;

void Parabola::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter a = "; cin >> a;
    cout << "Enter b = "; cin >> b;
    cout << "Enter c = "; cin >> c;
    cout << "Enter x = "; cin >> x;
    cin.get();
    cout << "y = " << (a * x * x + b * x + c) << endl;
    cin.get();
}
// Глобальный объект
Parabola f_parabola;
////////////////////////////////////

```

Кроме этого, в основном модуле `main.cpp` необходимо добавить директиву `#include «Parabola.h»`; а в списке инициализации массива `pObjs` — адрес нового объекта `&f_parabola`. Все!!! Новая версия калькулятора готова.

Задача 2.2. Продвинутый функциональный калькулятор

Наш заказчик никак не уgomонится. Теперь ему пришла идея расширить набор операций, которые способен выполнять функциональный калькулятор. «Пусть, — говорит заказчик, — калькулятор по желанию пользователя либо вычисляет значение заданной функции для некоторого аргумента, либо осуществляет табуляцию функции в заданном интервале с заданным шагом».

Идея неплохая. Это позволит резко повысить спрос на рынке на наш калькулятор, поскольку калькуляторы других фирм не умеют это делать. За работу!

Нам нужно решить две проблемы. Во-первых, видимо, придется добавить в базовый класс `Function` новый чисто виртуальный метод `Tabulation`, а также заместить его во всех подклассах. То есть придется вносить изменения в код всех классов иерархии `Function`, а это плохо. Во-вторых, в продвинутом функциональном калькуляторе необходимо реализовать двухуровневое меню: на первом уровне выбирается вид функции, на втором уровне — вид операции. В предыдущей задаче мы справились с проблемой *легко модифицируемого кода* одноуровневого меню с помощью шаблона проектирования `Switch`. Неужели на втором уровне придется писать по старинке гирлянду `case'ов` в блоке оператора `switch`? Как мы знаем, это тоже плохо для дальнейшей модификации программы.

Попробуем найти более удачное решение. А что если попытаться развить идею одноуровневого переключателя (шаблон `Switch`) для случая двух уровней переключения? Например, переключатель такого типа мог бы выбирать на первом уровне некоторую конкретную сущность `EntityA_I` из иерархии абстрактного базового класса `AbstractEntityA`, а затем — на втором уровне — некоторую конкретную сущность `EntityB_J` из иерархии абстрактного базового класса `AbstractEntityB`, после чего клиенту предоставляется возможность реализовать требуемую ассоциацию между выбранными сущностями¹. Шаблон проектирования `DoubleSwitch`, реализующий эту идею, показан на рис. 2.9.

Обратите внимание, что метод `SelectEntityA()` класса `Switch` возвращает указатель `AbstractEntityA*`, который передается далее в качестве аргумента методу `SelectEntityB()`, осуществляющему выбор на втором уровне уже *для конкретной сущности первого уровня*. Абстрактный класс второго уровня `AbstractEntityB` позволяет клиенту абстрагироваться от вида выполняемой операции: он имеет дело с «операцией вообще» `Operate()`.

После выбора на втором уровне благодаря полиморфизму клиент может вызвать метод `Operate()`, относящийся к конкретной сущности второго уровня. Передача

¹ Красиво сказано, не правда ли?

ссылки `AbstractEntityA*` методу `Operate()` разрешает последнему иметь доступ к атомарным операциям `AtomOperate1()`, `AtomOperate2()` и т. п., определенным в конкретном объекте иерархии `AbstractEntityA`. Совокупность этих примитивов должна быть достаточной для реализации любого метода `Operate()` в иерархии `AbstractEntityB`. Как правило, это операции, обеспечивающие доступ ко всем защищенным или закрытым полям объектов иерархии `AbstractEntityA` или выполняющие конкретную обработку информации, специфичную для данного класса.

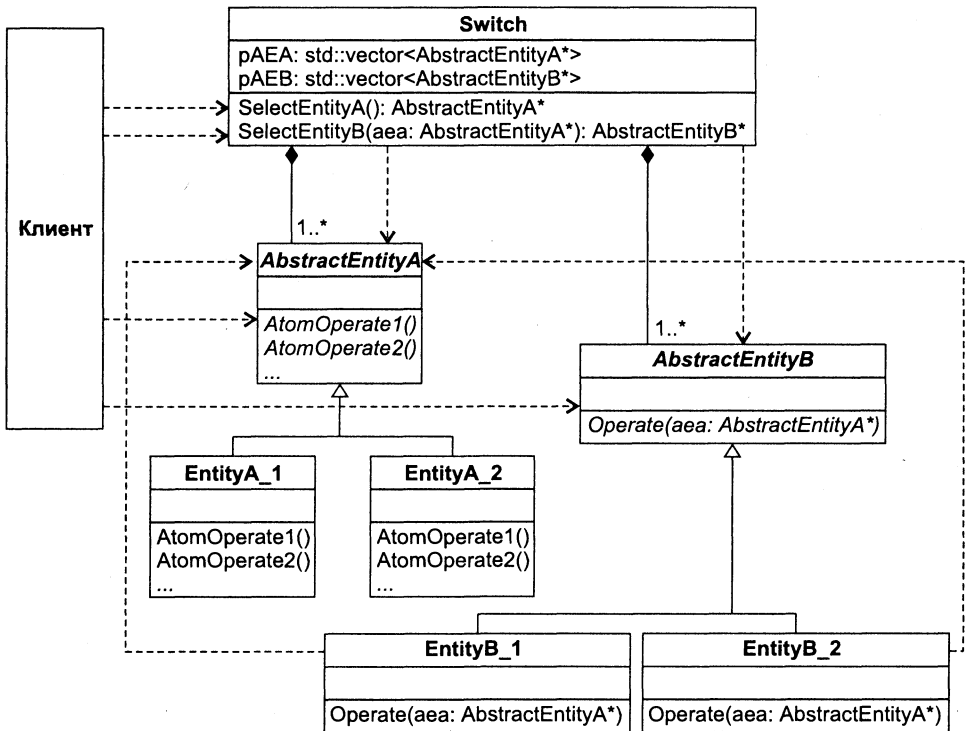


Рис. 2.9. Шаблон DoubleSwitch

Теперь попытаемся применить этот шаблон к нашей задаче. Ясно, что в качестве класса `Switch` здесь будет выступать класс `Menu`, как и в предыдущей задаче, а в качестве класса `AbstractEntityA` — класс `Action`. Роль класса `AbstractEntityB` мы поручим новому абстрактному классу `Action`, обеспечивающему унифицированный интерфейс для конкретных классов `Calculation`, `Tabulation`, `AnyAction`. Последний класс-пустышку мы добавим просто для иллюстрации того, что список выполняемых операций может легко расширяться.

Роль метода `AbstractEntityB::Operate(AbstractEntityA*)` будет возложена на метод `Action::Operate(Function*)`. Ну и, наконец, метод `Calculate` переключает из иерархии базового класса `Action` в конкретный класс `Calculation`, принадлежащий иерархии базового класса `Action`. Взамен мы должны пополнить иерархию `Action` такими атомарными операциями, как `SetArg` — установить значение аргумента, `SetCoeff` — установить значения коэффициентов, `GetVal` — получить значение функции.


```

// Класс для представления функции  $y = a * x + b$ 
class Line : public Function {
public:
    Line() : name("a * x + b") {}
    const std::string& GetName() const { return name; }
    void SetCoeff();
    double GetVal() const { return (a * x + b); }
private:
    std::string name; // мат. обозначение функции
    double a;
    double b;
};

extern Line f_line;
/////////////////////////////////////////////////////////////////
// Line.cpp
#include <iostream>
#include "Line.h"
using namespace std;

void Line::SetCoeff() {
    cout << "Enter a = ": cin >> a;
    cout << "Enter b = ": cin >> b;
}

// Глобальный объект
Line f_line;
/////////////////////////////////////////////////////////////////
// Action.h
#ifndef ACTION_H
#define ACTION_H

#include "Function.h"

class Action {
public:
    virtual ~Action() {}
    virtual void Operate(Function*) = 0;
    virtual const std::string& GetName() const = 0;
};

#endif /* ACTION_H */
/////////////////////////////////////////////////////////////////
// Calculation.h
#include "Action.h"

class Calculation : public Action {
public:
    Calculation() : name("Calculation") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
};

```

```

private:
    std::string name:    // обозначение операции
};
extern Calculation calculation;
////////////////////////////////////
// Calculation.cpp
#include <iostream>
#include "Calculation.h"
using namespace std;

void Calculation::Operate(Function* pFunc) {
    cout << "Calculation for function y = ";
    cout << pFunc->GetName() << endl;
    pFunc->SetCoeff();
    double x;
    cout << "Enter x = ";    cin >> x;
    cin.get();
    pFunc->SetArg(x);
    cout << "y = " << pFunc->GetVal() << endl;
    cin.get();
}

// Глобальный объект
Calculation calculation;
////////////////////////////////////
// Tabulation.h
#include "Action.h"

class Tabulation : public Action {
public:
    Tabulation() : name("Tabulation") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name:    // обозначение операции
};
extern Tabulation tabulation;
////////////////////////////////////
// Tabulation.cpp
#include <iostream>
#include <iomanip>
#include "Tabulation.h"
using namespace std;

void Tabulation::Operate(Function* pFunc) {
    cout << "Tabulation for function y = ";
    cout << pFunc->GetName() << endl;
    pFunc->SetCoeff();
    double x_beg, x_end, x_step;

```

```

cout << "Enter x_beg = ";   cin >> x_beg;
cout << "Enter x_end = ";   cin >> x_end;
cout << "Enter x_step = ";  cin >> x_step;
cin.get();
cout << "_____ " << endl;
cout << "      x          y" << endl;
cout << "_____ " << endl;
double x = x_beg;
while (x <= x_end) {
    pFunc->SetArg(x);
    cout << setw(6) << x << setw(14) << pFunc->GetVal() << endl;
    x += x_step;
}
cin.get();
}

```

```

// Глобальный объект
Tabulation tabulation;
////////////////////////////////////
// AnyAction.h
#include "Action.h"

class AnyAction : public Action {
public:
    AnyAction() : name("Any action") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name; // обозначение операции
};
extern AnyAction any_action;
////////////////////////////////////
// AnyAction.cpp
#include <iostream>
#include "AnyAction.h"
using namespace std;

void AnyAction::Operate(Function*) {
    cout << "Your advertising might be here!" << endl;
    cin.get();
}

```

```

// Глобальный объект
AnyAction any_action;
////////////////////////////////////
// Menu.h
#include <vector>
#include "Function.h"
#include "Action.h"

```

```

class Menu {
public:
    Menu(std::vector<Function*>, std::vector<Action*>);
    Function* SelectObject() const;
    Action* SelectAction(Function*) const;
private:
    int SelectItem(int) const;
    std::vector<Function*> pObj;
    std::vector<Action*> pAct;
}; ////////////////////////////////////////////////////////////////////
// Menu.cpp
#include <iostream>
#include "Menu.h"
using namespace std;

Menu::Menu(vector<Function*> _pObj, vector<Action*> _pAct)
    : pObj(_pObj), pAct(_pAct) {
    pObj.push_back(0); // для выбора пункта 'Exit'
}

Function* Menu::SelectObject() const {
    // Возьмите код аналогичной функции из проекта Task2_1
}

Action* Menu::SelectAction(Function* pObj) const {
    int nItem = pAct.size();
    cout << "=====\n";
    cout << "Select one of the following Actions:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". ";
        cout << pAct[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return pAct[item - 1];
}

int Menu::SelectItem(int nItem) const {
    // Возьмите код аналогичной функции из проекта Task2_1
}
//////////////////////////////////////////////////////////////////
// Main.cpp
#include <iostream>
#include "Function.h"
#include "Exp.h"
#include "Line.h"
#include "Action.h"
#include "Calculation.h"
#include "Tabulation.h"
#include "AnyAction.h"

```



```

#include "Menu.h"
using namespace std;

Function* pObjs[] = { &f_exp, &f_line };
vector<Function*> funcList(pObjs, pObjs + sizeof(pObjs)/sizeof(Function*));

Action* pActs[] = { &calculation, &tabulation, &any_action };
vector<Action*> operList(pActs, pActs + sizeof(pActs) / sizeof(Action*));

int main() {
    Menu menu(funcList, operList);

    while (Function* pObj = menu.SelectObject()) {
        Action* pAct = menu.SelectAction(pObj);
        pAct->Operate(pObj);
    }

    cout << "Bye!\n";
    return 0;
}
//————— конец проекта Task2_2 —————
////////////////////////////////////

```

Обратите внимание на следующее:

- метод `Exp::SetCoeff()` имеет пустое тело, то есть ничего не делает, поскольку для вычисления экспоненты коэффициенты не требуются;
- в связи с тем, что все методы класса `Exp` — встроенные, файл реализации `Exp.cpp` содержит только объявление глобального объекта `f_exp`.

Если вы хорошо поработали с текстом предыдущей задачи, то последняя программа не должна вызвать у вас каких-либо сложностей с восприятием ее кода. Отметим, что предложенное решение позволяет очень легко модифицировать наш калькулятор, решая как проблему добавления новых функций, так и проблему добавления новых операций.

Задача 2.3. Работа с объектами символьных и шестнадцатеричных строк

Написать программу, демонстрирующую работу с объектами двух типов: символьная строка (`SymbString`) и шестнадцатеричная строка (`HexString`), для чего создать систему соответствующих классов.

Каждый объект должен иметь как минимум два атрибута: идентификатор и значение, представленные в виде строк символов. В поле значения объекты `SymbString`

могут хранить произвольный набор символов, в то время как объекты `HexString` — только изображение шестнадцатеричного числа.

Клиенту (то есть функции `main`) должны быть доступны следующие операции: создать объект, удалить объект, показать значение объекта (строку символов), показать изображение эквивалентного десятичного числа (только для объектов `HexString`), показать изображение эквивалентного двоичного числа (только для объектов `HexString`). Предусмотреть меню, позволяющее продемонстрировать перечисленные операции.

При решении данной задачи мы воспользуемся еще одним из видов диаграмм UML, а именно *диаграммой видов деятельности* (*Activity Diagram*), чтобы расширить наш кругозор в сфере современных методов проектирования.

Диаграмма видов деятельности UML очень похожа на старые блок-схемы алгоритмов. В ней точками принятия решений и переходами описывается последовательность шагов, именуемых в UML *видами деятельности* (*Activity*). Каждый вид деятельности, ассоциируемый обычно с некоторой процедурой, изображается на диаграмме прямоугольником с округленными углами, а переходы между ними — стрелками. Точки принятия решений можно изображать одним из двух способов. В первом просто показываются все возможные переходы после завершения некоторого вида деятельности. Во втором способе изображается переход к маленькому ромбику, похожему на блок ветвления в блок-схемах, а затем все возможные пути выходят из этого ромбика.

Анализируя условие задачи, мы пытаемся, во-первых, выявить объекты/классы предметной области и, во-вторых, сконструировать основной алгоритм¹, работающий с этими объектами. Для рассматриваемой задачи нам представляется естественным алгоритм, изображенный на рис. 2.10 в виде диаграммы видов деятельности. Заметим, что начальная точка алгоритма на такой диаграмме обозначается в виде закрашенного кружка, а конечная точка — в виде «глазка». При необходимости на диаграмме можно размещать комментарии в виде прямоугольника с текстом, который напоминает листок бумаги с отогнутым углом.

Графическое представление алгоритма, как обычно, позволяет более четко выявить те проблемы, которые нужно решить. Так, меню в нашей программе должно обеспечить выбор на трех уровнях:

- 1) выбор вида работы (добавить объект к коллекции, удалить объект из коллекции, работать с коллекцией объектов, выйти из программы);
- 2) выбор объекта из коллекции объектов;
- 3) выбор операции из коллекции операций.

Имея опыт реализации двухуровневого меню в предыдущей задаче, мы должны принять решение о способе реализации трехуровневого меню. Первая мысль, которая приходит в голову: а не создать ли нам шаблон проектирования `ThreeLevelSwitch`, развивающий идеи шаблона `DoubleSwitch`? Однако с каждым повышением размерности такого переключателя программа будет становиться все сложнее для

¹ Реализуемый в клиенте `main()`.

понимания. Поэтому важно вовремя остановиться, проявляя заботу о тех программах, кому, возможно, придется сопровождать нашу программу в будущем¹.

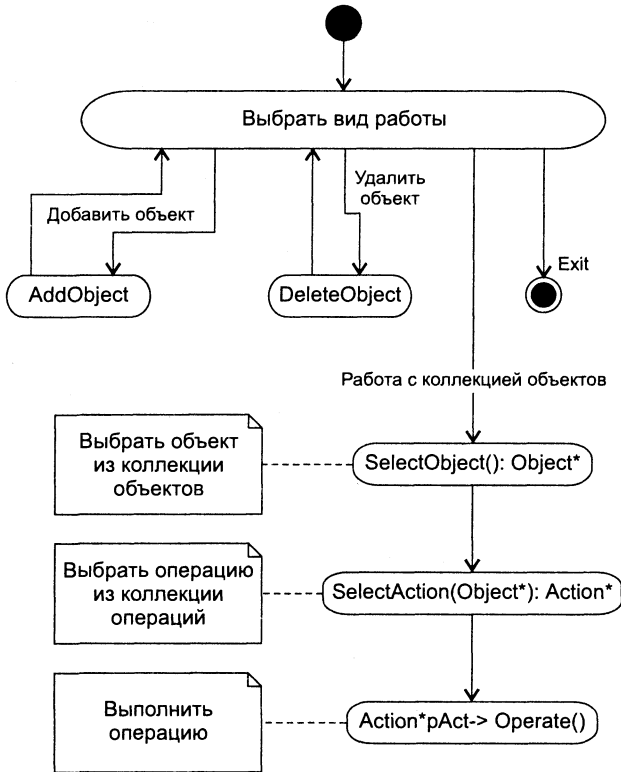


Рис. 2.10. Диаграмма видов деятельности для задачи 2.3

Принимаем следующий постулат: перечень действий (работ), выбираемых на первом уровне, тщательно продуман заказчиком, и вероятность его изменения крайне мала. В связи с этим первый уровень меню мы реализуем по старинке — в блоке оператора `switch`, а вот для второго и третьего уровней применим уже испытанный шаблон проектирования `DoubleSwitch`, который позволит в будущем легко добавлять как типы обрабатываемых объектов, так и виды применяемых операций.

Теперь можно поговорить и о классах. Два из них: `SymbString` и `HexString` — продиктованы условием задачи. Именно объекты этих классов будут пополнять коллекцию объектов, с которыми в дальнейшем можно осуществлять те или иные операции (в нашем случае — показать значение объекта, показать изображение эквивалентного десятичного числа, показать изображение эквивалентного двоичного числа). Для реализации шаблона проектирования `DoubleSwitch` нам по-

¹ Вообще, в любом деле важно вовремя остановиться, помня о том, что любую идею можно довести до абсурда.

требуется абстрактный базовый класс `AString`, наследниками которого будут классы `SymbString` и `HexString`, а также абстрактный базовый класс `Action`, имеющий в качестве дочерних классы `ShowStr`, `ShowDec`, `ShowBin`.

Для реализации меню создадим, как и в предыдущей задаче, класс `Menu`, содержащий аналогичные методы `SelectObject()` и `SelectAction()`, но к ним в компанию добавим метод `SelectJob()`. Поскольку список операций, применяемых к объектам, заранее известен (эти операции инкапсулированы в классах `ShowStr`, `ShowDec` и `ShowBin`), то этот список, как и раньше, мы разместим в поле `std::vector<Action*> pAct`. А вот со списком объектов ситуация более сложная, чем в задаче 2.2, так как эти объекты должны создаваться и уничтожаться динамически — в процессе работы программы. Пока отложим этот вопрос, мы вскоре к нему вернемся.

Какие сущности в решаемой задаче не охвачены перечисленными классами? Перечитаем еще раз условие задачи. В нем упоминаются операции: *создать объект* и *удалить объект*. Конечно, можно поручить эти операции функции `main()`, разместив заодно в ней и коллекцию обрабатываемых объектов, но это было бы в высшей степени нехорошо! Почему? — Потому, что такое проектное решение понизило бы сцепление внутри главного клиента `main()`¹.

СОВЕТ

Прилагайте максимум усилий к тому, чтобы каждая часть кода — каждый модуль, класс, функция — отвечали за выполнение одной четко определенной задачи.

Последуем этому совету и создадим класс `Factory`, отвечающий за создание и удаление объектов из иерархии `AString`. Класс будет содержать поле `std::vector<AString*> pObj` для хранения коллекции объектов и два метода: `AddObject()` и `DeleteObject()`.

Пора переходить к кодированию. Вот одно из возможных решений:

```

////////////////////////////////////
// Проект Task2_3
////////////////////////////////////
// AString.h
#ifndef ASTRING_H
#define ASTRING_H

#include <string>

class AString {
public:
    virtual ~AString() {}
    virtual const std::string& GetName() const = 0;
    virtual const std::string& GetVal() const = 0;

```

¹ Напомним, что *сцепление* внутри компонента — это показатель, характеризующий степень взаимосвязи отдельных его частей. Критерии качества проекта рассматривались на первом семинаре.

```

    virtual int GetSize() const = 0;
};

#ifdef ASTRING_H
/////////////////////////////////////////////////////////////////
// SymbString.h
#include <string>
#include "AString.h"

class SymbString : public AString {
public:
    SymbString(std::string _name) : name(_name) {}
    SymbString(std::string _name, std::string _val) :
        name(_name), val(_val) {}

    const std::string& GetName() const { return name; }
    const std::string& GetVal() const { return val; }
    int GetSize() const { return val.size(); }
private:
    std::string name;
    std::string val;
};
/////////////////////////////////////////////////////////////////
// HexString.h
#include <string>
#include "AString.h"

const std::string alph = "0123456789ABCDEF";
bool IsHexStrVal(std::string);

class HexString : public AString {
public:
    HexString(std::string _name) : name(_name) {}
    HexString(std::string, std::string);
    const std::string& GetName() const { return name; }
    const std::string& GetVal() const { return val; }
    int GetSize() const { return val.size(); }
private:
    std::string name;
    std::string val;
};
/////////////////////////////////////////////////////////////////
// HexString.cpp
#include <iostream>
#include "HexString.h"
using namespace std;

bool IsHexStrVal(string _str) {
    for (int i = 0; i < _str.size(); ++i)

```



```

class ShowStr : public Action {
public:
    ShowStr() : name("Show string value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string name;    // обозначение операции
};

extern ShowStr show_str;
////////////////////////////////////
// ShowStr.cpp
#include <iostream>
#include "ShowStr.h"
using namespace std;

void ShowStr::Operate(AString* pObj) {
    cout << pObj->GetName() << ": ";
    cout << pObj->GetVal() << endl;
    cin.get();
}

// Глобальный объект
ShowStr show_str;
////////////////////////////////////
// ShowDec.h
#include "Action.h"

class ShowDec : public Action {
public:
    ShowDec() : name("Show decimal value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string name;    // обозначение операции
};

extern ShowDec show_dec;
////////////////////////////////////
// ShowDec.cpp
#include <iostream>
#include "ShowDec.h"
#include "HexString.h"
using namespace std;

void ShowDec::Operate(AString* pObj) {
    cout << pObj->GetName() << ": ";
    long decVal = GetDecimal(pObj);
    if (decVal != -1)
        cout << GetDecimal(pObj);
    cout << endl;
    cin.get();
}

```

```

// Глобальный объект
ShowDec show_dec;
////////////////////////////////////
// ShowBin.h
#include "Action.h"

class ShowBin : public Action {
public:
    ShowBin() : name("Show binary value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string GetBinary(AString*) const;
    std::string name; // обозначение операции
};

extern ShowBin show_bin;
////////////////////////////////////
// ShowBin.cpp
#include <iostream>
#include "ShowBin.h"
#include "ShowDec.h"
#include "AString.h"
using namespace std;

void ShowBin::Operate(AString* pObj) {
    cout << pObj->GetName() << ": ";
    cout << GetBinary(pObj) << endl;
    cin.get();
}

string ShowBin::GetBinary(AString* pObj) const {
    int nBinDigit = 4 * pObj->GetSize();
    char* binStr = new char[nBinDigit + 1];
    for (int k = 0; k < nBinDigit; ++k) binStr[k] = '0';
    binStr[nBinDigit] = 0;
    long decVal = GetDecimal(pObj);
    if (-1 == decVal)
        return string("");
    int i = nBinDigit - 1;
    while (decVal > 0) {
        binStr[i--] = 48 + (decVal % 2);
        decVal >>= 1;
    }
    string temp(binStr);
    delete [] binStr;
    return temp;
}
// Глобальный объект
ShowBin show_bin;

```



```

////////////////////////////////////
// Factory.h
#ifndef FACTORY_H
#define FACTORY_H

#include <vector>
#include "AString.h"

class Factory {
    friend class Menu;
public:
    Factory() {}
    void AddObject();
    void DeleteObject();
private:
    std::vector<AString*> pObj;
};

#endif //FACTORY_H
////////////////////////////////////
// Factory.cpp
#include <iostream>
#include "Factory.h"
#include "Menu.h"
#include "SymbString.h"
#include "HexString.h"
using namespace std;

#define MAX_LEN_STR 100

void Factory::AddObject() {
    cout << "_____\\n";
    cout << "Select object type:\\n";
    cout << "1. Symbolic string" << endl;
    cout << "2. Hexadecimal string" << endl;
    int item = Menu::SelectItem(2);

    string name;
    cout << "Enter object name: ";
    cin >> name;
    cin.get();
    cout << "Enter object value: ";
    char buf[MAX_LEN_STR];
    cin.getline(buf, MAX_LEN_STR);
    string value = buf;

    AString* pNewObj;
    switch (item) {
    case 1:

```

```

        pNewObj = new SymbString(name, value);
        break;
    case 2:
        if (!IsHexStrVal(value)) {
            cout << "Error!" << endl;    return;
        }
        pNewObj = new HexString(name, value);
        break;
    }
    pObj.push_back(pNewObj);
    cout << "Object added." << endl;
}

void Factory::DeleteObject() {
    int nItem = pObj.size();
    if (!nItem) {
        cout << "There are no objects." << endl;
        cin.get();
        return;
    }
    cout << ".....\n";
    cout << "Delete one of the following Object:\n";
    for (int i = 0; i < nItem; ++i)
        cout << i + 1 << ". " << pObj[i]->GetName()
            << endl;
    int item = Menu::SelectItem(nItem);
    string objName = pObj[item - 1]->GetName();
    pObj.erase(pObj.begin() + item - 1);
    cout << "Object " << objName << " deleted." << endl;
    cin.get();
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Menu.h
#include <vector>
#include "AString.h"
#include "Action.h"
#include "Factory.h"

typedef enum { AddObj, DelObj, WorkWithObj, Exit } JobMode;

class Menu {
public:
    Menu(std::vector<Action*>);
    JobMode    SelectJob() const;
    AString*   SelectObject(const Factory&) const;
    Action*    SelectAction(const AString*) const;
    static int SelectItem(int);
private:
    std::vector<Action*> pAct;
};

```

```

////////////////////////////////////
// Menu.cpp
#include <iostream>
#include "AString.h"
#include "SymbString.h"
#include "HexString.h"
#include "Menu.h"
using namespace std;

Menu::Menu(vector<Action*> _pAct) : pAct(_pAct) {}

JobMode Menu::SelectJob() const {
    cout << "=====\n";
    cout << "Select one of the following job modes:\n";
    cout << "1. Add object" << endl;
    cout << "2. Delete object" << endl;
    cout << "3. Work with object" << endl;
    cout << "4. Exit" << endl;
    int item = SelectItem(4);
    return (JobMode)(item - 1);
}

AString* Menu::SelectObject(const Factory& fctry) const {
    int nItem = fctry.pObj.size();
    if (!nItem) {
        cout << "There are no objects." << endl;
        cin.get();
        return 0;
    }
    cout << ".....\n";
    cout << "Select one of the following Object:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". ";
        cout << fctry.pObj[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return fctry.pObj[item - 1];
}

Action* Menu::SelectAction(const AString* pObj) const {
    if (!pObj) return 0;
    int nItem = pAct.size();
    cout << ".....\n";
    cout << "Select one of the following Actions:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". ";
        cout << pAct[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return pAct[item - 1];
}

```

```

int Menu::SelectItem(int nItem) {
    // Возьмите код аналогичной функции из проекта Task2_1
}
////////////////////////////////////
// Main.cpp
#include <iostream>
#include "AString.h"
#include "SymbString.h"
#include "HexString.h"
#include "Action.h"
#include "ShowStr.h"
#include "ShowDec.h"
#include "ShowBin.h"
#include "Factory.h"
#include "Menu.h"
using namespace std;

Action* pActs[] = { &show_str, &show_dec, &show_bin };
vector<Action*> actionList(pActs,
    pActs + sizeof(pActs)/sizeof(Action*));

int main() {
    Factory factory;
    Menu menu(actionList);
    JobMode jobMode;

    while ((jobMode = menu.SelectJob()) != Exit ) {
        switch (jobMode) {
            case AddObj: factory.AddObject();
                break;
            case DelObj: factory.DeleteObject();
                break;
            case WorkWithObj:
                AString* pObj = menu.SelectObject(factory);
                Action* pAct = menu.SelectAction(pObj);
                if (pAct)
                    pAct->Operate(pObj);
                break;
        }
        cin.get();
    }

    cout << "Bye!\n";
    return 0;
}
//————— конец проекта Task2_3 —————
////////////////////////////////////

```

Обратим внимание на наиболее интересные моменты реализации.

- ❑ В модуле `HexString.cpp` размещена глобальная функция `IsHexStrVal(string _str)`, проверяющая, соответствует ли предъявленная строка `_str` изображению шестнадцатеричного числа. Выяснение этого вопроса осуществляется с помощью метода `find_first_of()` класса `string`, который возвращает индекс вхождения символа, представленного аргументом, в строку `alph`. Последняя объявлена в файле `HexString.h` и содержит набор допустимых символов для изображения шестнадцатеричного числа. Если символ не найден, то метод `find_first_of()` возвращает значение `-1`. Функция `IsHexStrVal()` используется в конструкторе класса `HexString`, предотвращая присваивание полю `val` некорректного значения, а также в методе `AddObject()` класса `Factory`, блокируя ошибочный ввод информации пользователем.
- ❑ В базовом классе `Action` реализован метод `GetDecimal(AString* pObj)`, возвращающий значение десятичного числа для передаваемого через указатель `pObj` изображения шестнадцатеричного числа. Почему в базовом классе? — Потому, что это значение необходимо получать как в методе `ShowDec::Operate()`, так и в методе `ShowBin::GetBinary()` — вычисление двоичного изображения числа реализовано через предварительное получение его десятичного значения.
- ❑ В методе `Action::GetDecimal()` помимо всего прочего применена технология использования информации о типе на этапе выполнения, или сокращенно *RTTI* — *Run-Time Type Information*. Дело в том, что заранее (на этапе компиляции) не известно, объекты каких типов будут передаваться в качестве аргумента данного метода. В принципе, возможно появление объекта любого производного класса из иерархии `AString`. В то же время из условия задачи известно, что получение десятичного и двоичного представлений корректно только для шестнадцатеричных строк. Поэтому возникает проблема, для решения которой мы используем операцию динамического приведения типов `dynamic_cast`, которая позволяет осуществить понижающее преобразование из типа базового класса к типу производного класса. Операция возвращает адрес объекта производного класса, указанного в угловых скобках: `<HexString*>`, если такое преобразование возможно (то есть если `pObj` действительно является адресом объекта класса `HexString`), либо `0`, если преобразование невозможно.
- ❑ В методе `ShowBin::GetBinary()` сначала вычисляется десятичное значение `decVal` для шестнадцатеричной строки, а затем в цикле `while` формируется символьный массив `binStr`, содержащий символы '0' и '1' для двоичного изображения числа. Очередной символ вычисляется извлечением младшего бита из `decVal` операцией `decVal % 2` и последующим его преобразованием к коду ASCII (ANSI), для чего к значению бита (0 или 1) прибавляется 48. После извлечения двоичное представление `decVal1` сдвигается на один разряд вправо.

¹ Мы очень надеемся, что вы не забыли, что в памяти компьютера все числа представлены в двоичном виде. (Просьба к продвинутым программистам эту сноску не читать.)

- В классе `Menu` метод `SelectItem()` объявлен как статический. Зачем? Напомним, что статические методы класса могут использоваться как глобальные. В данном случае подзадачу ввода целого числа для выбора пункта меню нужно решать не только в классе `Menu`, но и в классе `Factory`. Чтобы не дублировать код, мы сделали метод `SelectItem()` статическим и теперь можем его вызывать из другого класса, разумеется, предваряя операцией доступа к области видимости `Menu::`.
- В методе `Factory::AddObject()` добавление нового объекта к коллекции `vector<AString*>` `pObj` производится вызовом метода `push_back()` класса `vector`.
- В методе `Factory::DeleteObject()` удаление объекта из коллекции осуществляется с помощью метода `erase()` класса `vector`. В качестве аргумента метод принимает адрес объекта. Так как значение `item` на единицу больше, чем индекс элемента в контейнере `vector`, то адрес удаляемого объекта вычисляется выражением `pObj.begin() + item - 1`, где `begin()` — метод, возвращающий адрес начального элемента в контейнере.

Давайте повторим наиболее важные моменты этого семинара.

1. Наследование и полиморфизм — важнейшие механизмы ООП. Наследование позволяет объединять классы в семейства связанных типов, что дает им возможность совместно использовать общие поля и методы.
2. В языке C++ родительский класс называется *базовым*, а дочерний класс — *производным*. Отношения между родительскими классами и их потомками называются *иерархией наследования*.
3. Полиморфное использование объектов из одной иерархии базируется на двух механизмах: а) возможности использования указателя или ссылки на базовый класс для работы с объектами любого из производных классов, б) технологии *динамического связывания* при выборе виртуального метода через указатель на базовый класс. Фактический выбор операции, которая будет вызвана, происходит только во время выполнения программы в зависимости от типа выбранного объекта.
4. Наряду с отношением наследования, классы могут находиться и в других отношениях: *ассоциации*, *агрегации*, *композиции* и *зависимости (использования)*. Удобным средством отображения взаимоотношений классов является *диаграмма классов* на языке UML.
5. Современное программирование базируется не только на идеях ООП, но и на применении шаблонов проектирования, аккумулирующих в себе наиболее удачные решения типичных проблем, возникавших неоднократно при разработке программ.
6. Одной из целей применения шаблонов проектирования является разработка программного кода с учетом его будущих изменений. В задачах 2.1 и 2.2 предложены два новых шаблона: `Switch` и `DoubleSwitch`, целью применения которых является реализация одноуровневого и двухуровневого меню, позволяющая вносить последующие изменения в меню наиболее легким и приятным способом.

Задания

Общая часть заданий для вариантов 1–20

Написать программу, демонстрирующую работу с объектами двух типов: T1 и T2, для чего создать систему соответствующих классов. Каждый объект должен иметь идентификатор (в виде произвольной строки символов) и одно или несколько полей для хранения состояния объекта (один класс является потомком другого).

Клиенту (функции main) должны быть доступны следующие основные операции (методы): создать объект, удалить объект, показать значение объекта и прочие дополнительные операции (зависят от варианта). Операции по созданию и удалению объектов инкапсулировать в классе Factory. Предусмотреть меню, позволяющее продемонстрировать заданные операции.

При необходимости в разрабатываемые классы добавляются дополнительные методы (например, конструктор копирования, операция присваивания и т. п.) для обеспечения надлежащего функционирования этих классов.

Варианты 1–10

В табл. 2.1 и 2.2 перечислены возможные типы объектов и возможные дополнительные операции над ними.

Таблица 2.1. Перечень типов объектов

Класс	Объект
SymbString	Символьная строка (произвольная строка символов)
BinString	Двоичная строка (изображение двоичного числа) ¹
OctString	Восьмеричная строка (изображение восьмеричного числа)
DecString	Десятичная строка (изображение десятичного числа)
HexString	Шестнадцатеричная строка (изображение шестнадцатеричного числа)

Таблица 2.2. Перечень дополнительных операций (методов)

Операция (метод)	Описание
ShowBin()	Показать изображение двоичного значения объекта
ShowOct()	Показать изображение восьмеричного значения объекта
ShowDec()	Показать изображение десятичного значения объекта
ShowHex()	Показать изображение шестнадцатеричного значения объекта
operator +(T& s1, T& s2) ²	Для объектов SymbString — конкатенация строк s1 и s2; для объектов прочих классов — сложение соответствующих численных значений с последующим преобразованием к типу T

¹ Здесь и далее в таблице рассматриваются только целые положительные числа.

² Здесь T — любой из типов T1 или T2.

Операция (метод)	Описание
operator -(T& s1, T& s2)	Для объектов SymbString — если s2 содержится как подстрока в s1, то результатом является строка, полученная из s1 удалением подстроки s2; в противном случае возвращается значение s1; для объектов прочих классов — вычитание соответствующих численных значений с последующим преобразованием к типу T

Примечание: Первые четыре операции могут применяться к объектам любых классов, за исключением класса SymbString.

Таблица 2.3 содержит спецификации вариантов.

Таблица 2.3. Спецификации вариантов 1–10

Вариант	T1	T2	Операции (методы)
1	SymbString	BinString	ShowOct(), ShowDec(), ShowHex()
2	SymbString	BinString	operator +(T&, T&)
3	SymbString	BinString	operator -(T&, T&)
4	SymbString	OctString	operator +(T&, T&)
5	SymbString	OctString	operator -(T&, T&)
6	SymbString	DecString	ShowBin(), ShowOct(), ShowHex()
7	SymbString	DecString	operator +(T&, T&)
8	SymbString	DecString	operator -(T&, T&)
9	SymbString	HexString	operator +(T&, T&)
10	SymbString	HexString	operator -(T&, T&)

Варианты 11–20

В табл. 2.4 и 2.5 перечислены возможные типы объектов и возможные дополнительные операции над ними.

Таблица 2.4. Перечень типов объектов

Класс	Объект
Triangle	Треугольник
Quadrate	Квадрат
Rectangle	Прямоугольник
Tetragon	Четырехугольник
Pentagon	Пятиугольник

Таблица 2.5. Перечень дополнительных операций (методов)

Операция (метод)	Описание
Move()	Переместить объект на плоскости
Compare(T& ob1, T& ob2)	Сравнить объекты ob1 и ob2 по площади
IsIntersect(T& ob1, T& ob2)	Определить факт пересечения объектов ob1 и ob2 (есть пересечение или нет)
IsInclude(T& ob1, T& ob2)	Определить факт включения объекта ob2 в объект ob1

Таблица 2.6 содержит спецификации вариантов.

Таблица 2.6. Спецификации вариантов 11–20

Вариант	T1	T2	Операции (методы)
11	Triangle	Quadrate	Move(), Compare(T&, T&)
12	Quadrate	Pentagon	Move(), IsIntersect(T&, T&)
13	Triangle	Rectangle	Move(), Compare(T&, T&)
14	Triangle	Rectangle	Move(), IsIntersect(T&, T&)
15	Rectangle	Pentagon	Move(), IsInclude(T&, T&)
16	Triangle	Tetragon	Move(), Compare(T&, T&)
17	Triangle	Tetragon	Move(), IsIntersect(T&, T&)
18	Triangle	Tetragon	Move(), IsInclude(T&, T&)
19	Triangle	Pentagon	Move(), Compare(T&, T&)
20	Triangle	Pentagon	Move(), IsIntersect(T&, T&)

Семинар 3 Шаблоны классов. Обработка исключительных ситуаций

Теоретический материал: с. 211–230.

Шаблоны классов

Шаблоны классов, так же как и шаблоны функций, поддерживают в C++ парадигму *обобщенного программирования*, то есть программирования с использованием типов в качестве параметров. Механизм шаблонов в C++ допускает применение абстрактного типа в качестве параметра при определении класса или функции. После того как шаблон класса определен, он может использоваться для определения конкретных классов. Процесс генерации компилятором определения конкретного класса по шаблону класса и аргументам шаблона называется *инстанцированием шаблона* (template instantiation).

Рассмотрим, например, точку на плоскости. Для ее представления в задаче 1.2 мы разработали класс Point, в котором положение точки задавалось двумя координатами x и y — полями типа double. Представим теперь, что в другом приложении требуется задавать точки для целочисленной системы координат, то есть использовать поля типа int. Если не ограничивать нашу фантазию, то можно вообразить себе системы, в которых координаты точки имеют тип short или unsigned char. Так что же — определять каждый раз новый класс Point для каждой из этих задач? Бьерна Страуструпа очень раздражала такая перспектива, и он добавил в C++ поддержку того, чем мы будем заниматься на этом семинаре.

Определение шаблона класса

Определение шаблонного (обобщенного, родового) класса имеет вид:

```
template <параметры_шаблона> class имя_класса { /* ... */ };
```

Например, определение шаблонного класса Point будет выглядеть следующим образом:

```
template <class T> class Point {
public:
    Point(T _x = 0, T _y = 0) : x(_x), y(_y) {}
    void Show() const {
        cout << " (" << x << ", " << y << ")" << endl;
    };
private:
    T x, y;
};
```

Вы заметили, чем отличается это определение от определения обычного класса? Префикс `template <class T>` указывает, что объявлен шаблон класса, в котором `T` — некоторый абстрактный тип. То есть ключевое слово `class` в этом контексте задает вовсе не класс, а означает лишь то, что `T` — это параметр шаблона. Вместо `T` может использоваться любое имя. После объявления `T` используется внутри шаблона точно так же, как имена других типов. Отметим, что язык позволяет вместо ключевого слова `class` перед параметром шаблона использовать другое ключевое слово — `typename`, то есть написать:

```
template < typename T> class Point { /* ... */ };
```

В литературе встречаются оба стиля объявления, но первый, пожалуй, более распространен¹.

Определение встроенных методов внутри шаблона класса практически не отличается от записи в обычном классе. Но если определение метода выносится за пределы класса, то синтаксис его заголовка усложняется. Покажем это на примере метода `Show()`:

```
// Версия с внешним определением метода Show()
template <class T> class Point {
public:
    Point(T _x = 0, T _y = 0) : x(_x), y(_y) {}
    void Show() const;
private:
    T x, y;
};
template <class T> void Point<T>::Show() const {
    cout << " (" << x << ", " << y << ")" << endl;
}
```

¹ Вам, наверное, интересна аргументация Б. Страуструпа по поводу его выбора: «Лично я неважно печатаю, и мне вечно не хватает места на экране, поэтому я предпочитаю более короткое: `template <class T> ...`»

Обратите внимание на появление того же префикса `template <class T>`, который предварял объявление шаблона класса, а также на более сложную запись операции квалификации области видимости для имени `Show()`: если раньше мы писали `Point::`, то теперь пишем `Point<T>::`, добавляя к имени класса список параметров шаблона, заключенный в угловые скобки (в данном случае один параметр `T`)¹.

Использование шаблона класса

При включении шаблона класса в программу никакие классы на самом деле не генерируются до тех пор, пока не будет создан экземпляр шаблонного класса, в котором вместо абстрактного типа `T` указывается некоторый конкретный тип. Такая подстановка приводит к *актуализации*, или *инстанцированию*, шаблона. Как и для обычного класса, экземпляр создается либо объявлением объекта, например:

```
Point<int> anyPoint(13, -5);
```

либо объявлением указателя на актуализированный шаблонный тип с присваиванием ему адреса, возвращаемого операцией `new`, например:

```
Point<double>* pOtherPoint = new Point<double>(9.99, 3.33);
```

Встретив подобные объявления, компилятор генерирует код соответствующего класса.

Организация исходного кода

В многофайловом проекте определение шаблона класса обычно выносится в отдельный файл. В то же время для инстанцирования компилятором конкретного экземпляра шаблона класса необходимо, чтобы определение шаблона находилось в *одной единице трансляции* с данным экземпляром.

В связи с этим принято размещать *все определение* шаблонного класса в некотором заголовочном файле, например `Point.h`, и подключать его к нужным файлам с помощью директивы `#include`. Для предотвращения повторного включения этого файла, которое может иметь место в многофайловом проекте, обязательно используйте «стражи включения», реализуемые посредством директивы `#ifndef` (о стражах включения см. первую часть практикума, с. 153).

Продемонстрируем эту технику на примере нашего класса `Point`:

```
////////////////////////////////////  
// Point.h  
#ifndef POINT_H  
#define POINT_H  
  
template <class T> class Point {
```

¹ На первых порах это сложно запомнить (что является причиной постоянных ошибок компиляции), но, написав пару-другую десятков шаблонных классов, вы привыкнете к этому вычурному синтаксису.

```

public:
    Point(T _x = 0, T _y = 0) : x(_x), y(_y) {}
    void Show() const:
private:
    T x, y;
};

template <class T> void Point<T>::Show() const {
    cout << " (" << x << ", " << y << ")" << endl;
}
#endif /* POINT_H */
////////////////////////////////////
// Main.cpp
#include <iostream>
#include "Point.h"
using namespace std;

int main() {
    Point<double> p1;           // 1
    Point<double> p2(7.32, -2.6); // 2
    p1.Show(); p2.Show();
    Point<int> p3(13, 15);     // 3
    Point<short> p4(17, 21);   // 4
    p3.Show(); p4.Show();
    return 0;
}

```

Обратите внимание на использование шаблонного класса `Point` клиентом `main()`: в строках 1 и 2 шаблон инстанцируется в конкретный класс `Point` с подстановкой вместо `T` типа `double`, в строке 3 — в класс `Point` с подстановкой типа `int`, в строке 4 — с подстановкой типа `short`.

Заметим, что наиболее широкое применение шаблоны классов нашли при создании контейнерных классов¹ стандартной библиотеки шаблонов (STL), которые предназначены для работы с такими стандартными структурами, как вектор, список, очередь, множество и т. д. Кстати, на втором семинаре мы уже воспользовались одним из этих классов, а именно классом `vector`. Так что сейчас вы можете бросить беглый ретроспективный взгляд на пройденный материал, испытывая чувство глубокого удовлетворения от более тонкого понимания новой материи — шаблонных классов.

Параметры шаблонов

Параметрами шаблонов могут быть абстрактные типы или переменные встроенных типов, таких как `int`². Первый вид параметров мы уже рассмотрели. При ин-

¹ *Контейнеры* — это объекты, предназначенные для хранения других объектов.

² Стандарт C++ допускает также использование параметров-шаблонов, но это реализовано далеко не во всех компиляторах, поэтому лучше ими не пользоваться.

станционировании на их место подставляются аргументы либо встроенных типов, либо типов, определенных программистом.

Второй вид используется, когда для шаблона предусматривается его настройка некоторой константой. Например, можно создать шаблон для массивов, содержащих n элементов типа T :

```
template <class T, int n> class Array { /*...*/ };
```

Тогда, объявив объект

```
Array<Point, 20> ap;
```

мы создадим массив из 20 элементов типа `Point`.

Приведем менее тривиальный пример использования параметров второго вида¹:

```
void f1() { cout << "I am f1()." << endl; }
void f2() { cout << "I am f2()." << endl; }

template<void (*pf)()> struct A { void Show() { pf(); } };

int main() {
    A<&f1> aa;
    aa.Show(); // вывод: I am f1().
    A<&f2> ab;
    ab.Show(); // вывод: I am f2().
    return 0;
}
```

Здесь параметр шаблона имеет тип указателя на функцию. При инстанцировании класса в качестве аргумента подставляется адрес соответствующей функции².

Естественно, у шаблона может быть несколько параметров. Например, ассоциативный контейнер `map` из библиотеки STL имеет следующее определение:

```
template <class Key, class T, class Compare = less<Key> >
class map { /*...*/ };
```

Из последнего примера видно, что параметры шаблона так же, как и параметры обычных функций, могут иметь значения по умолчанию. Обратите внимание на наличие пробела между двумя последними символами `>`. Если вы забудете поставить этот пробел, то компилятор воспримет последовательность `>>` как опера-

¹ В таких маленьких примерах мы будем опускать директивы:

```
#include <iostream>
using namespace std;
```

Но вы, пожалуйста, не забывайте добавлять их в начале текста программы.

² Адрес функции также является константой, создаваемой компилятором.

цию сдвига вправо и выдаст сообщение об ошибке, текст которого чаще всего слабо помогает локализовать ее место¹.

Специализация

Иногда возникает необходимость определить специализированную версию шаблона для некоторого конкретного типа одного из его параметров. Например, невозможно дать обобщенный алгоритм, проверяющий отношение < (меньше) для двух аргументов типа T, который одновременно подходил бы и для числовых типов, и для традиционных C-строк, завершающихся нулевым байтом. В таких случаях применяется специализация шаблона, имеющая следующую общую форму:

```
template <> class имя_класса <имя_специализированного_типа> { /* ... */ };
```

Например:

```
// общий шаблон
template <class T> class Sample {
    bool Less(T) const; /*...*/ };
// специализация для char*
template <> class Sample<char*> {
    bool Less(char*) const; /*...*/ };
```

Если шаблонный класс имеет несколько параметров, то возможна *частичная специализация*. Например:

```
// общий шаблон
template <class T1, class T2> class Pair { /*...*/ };
// специализация, где для T2 установлен тип int
template <class T1> class Pair <T1, int> { /*...*/ };
```

В любом случае общий шаблон должен быть объявлен прежде любой специализации.

Иногда есть смысл специализировать не весь класс, а какой-либо из его методов.

Например:

```
// обобщенный метод
template <class T> bool Sample<T>::Less(T ob) const { /*...*/ };
// специализированный метод
void Sample<char*>::Less(char* ob) const { /*...*/ };
```

¹ Например, компилятор Visual C++ 6.0 сообщает: «*fatal error ... : end of file found before the left angle-bracket '<' at '...\main.cpp(...)' was matched*».

Использование классов функциональных объектов для настройки шаблонных классов

Напомним, что *функциональным объектом* называется объект, для которого определена операция вызова функции `operator()`. Соответственно, класс, экземпляром которого является этот объект, называется *классом функционального объекта*, или просто *функциональным классом*. Приведем простой пример:

```
#include <iostream>
using namespace std;
struct LessThan {
    bool operator() (const int x, const int y) {
        return x < y;
    }
};

int main() {
    LessThan lt;           // 1
    int a = 5, b = 9;

    if (lt(a, b))         // 2
        cout << a << " less than " << b << endl;

    if (LessThan()(a, b)) // 3
        cout << a << " less than " << b << endl;

    return 0;
}
```

Здесь `LessThan` — функциональный класс¹ с единственной операцией `operator()`. В строке, помеченной номером 1, объявлен объект `lt` этого класса. Да, конечно, это функциональный объект. Далее он используется в операторе `if` (оператор 2) для вызова функции `operator()`, передавая ей аргументы `a` и `b`.

Во втором операторе `if` (оператор 3) демонстрируется другой способ использования функционального объекта без его предварительного объявления. Запись `LessThan()` означает создание анонимного экземпляра класса `LessThan`, то есть функционального объекта. Запись `LessThan()(a, b)` означает вызов функции `operator()` с передачей ей аргументов `a` и `b`. Естественно, что результат выполнения второго оператора `if` такой же, как и первого.

Вы, конечно, удивитесь: какой смысл использовать здесь класс `LessThan` вместо того, чтобы просто написать `if (a < b)`? И будете абсолютно правы. Но ситуация резко меняется, когда один из параметров шаблонного класса используется для настройки класса на некоторую стратегию.

¹ Еще раз напомним, что `struct` определяет частный вид класса, в котором по умолчанию все элементы открытые.

Рассмотрим пример. Сегодня утром вас вызвал шеф и дал задание создать шаблонный класс `PairSelect`, предназначенный для выбора одного значения из пары значений, хранящихся в этом классе. Выбор должен осуществляться в соответствии с критерием (стратегией), который передается как параметр шаблона. Этот параметр должен называться `class Compare`. Для начала нужно реализовать две стратегии: `LessThan` (первое значение меньше второго) и `GreaterThan` (первое значение больше второго).

Анализируя постановку задачи, вы приходите к заключению, что реализация стратегий в виде глобальных функций `LessThan()` и `GreaterThan()` быстро заведет в тупик, так как компилятор не позволит передавать адреса функций на место параметра `class Compare`. Остается единственное решение — использовать функциональные классы:

```
template <class T> struct LessThan {
    bool operator() (const T& x, const T& y) {
        return x < y;
    }
};

template <class T> struct GreaterThan {
    bool operator() (const T& x, const T& y) {
        return x > y;
    }
};

template <class T, class Compare>
class PairSelect {
public:
    PairSelect(const T& x, const T& y) : a(x), b(y) {}
    void OutSelect() const {
        cout << (Compare()(a, b) ? a : b) << endl;
    }
private:
    T a, b;
};

int main() {
    PairSelect<int, LessThan<int> > ps1(13, 9);
    ps1.OutSelect();           // вывод: 9

    PairSelect<double, GreaterThan<double> > ps2(13.8, 9.2);
    ps2.OutSelect();         // вывод: 13.8
    return 0;
}
```

Обратите внимание на использование функционального объекта `Compare()(a, b)` в методе `OutSelect()`, а также на настройку шаблонного класса `PairSelect` при его инстанцировании. При первом инстанцировании, объявляя объект `ps1`, мы пере-

даем вторым аргументом класса PairSelect функциональный класс LessThan<int>. При втором инстанцировании, объявляя ps2, мы передаем GreaterThan<double>.

Разработка шаблонного класса для представления разреженных массивов

Для закрепления изложенного материала рассмотрим пример разработки шаблонного класса, предназначенного для представления разреженных массивов. *Разреженный массив* — это такой массив, в котором не все элементы фактически используются, присутствуют на своих местах или нужны.

Структура данных этого типа появилась в процессе решения тех научных и инженерных задач, в которых нужны многомерные массивы очень большого объема, и в то же время на каждом этапе обработки информации оказывается, что фактически используется только незначительная часть элементов массива. Например, к таким задачам относятся приложения, связанные с анализом матриц. Другой пример — электронные таблицы, использующие матрицу для хранения формул, значений и строк, ассоциируемых с каждой из ячеек. Благодаря использованию разреженных массивов память для хранения каждого из элементов выделяется из пула свободной памяти только по мере необходимости.

Для простоты изложения мы будем рассматривать далее только одномерные разреженные массивы, так как все обсуждаемые идеи несложно применить и для реализации многомерных массивов. В среде приличных людей, так или иначе связанных с разреженными массивами, в ходу два термина: *логический массив* и *физический массив*. Логический массив является воображаемым, а физический массив — это массив, который фактически существует в системе. Например, если вы определите разреженный массив:

```
SparseArr sa(1000000);
```

то логический массив будет состоять из 1 000 000 элементов даже в том случае, если этот массив в системе физически не существует. Таким образом, если фактически используются только 100 элементов этого массива, то только они и будут занимать физическую память компьютера.

Обычно каждый элемент физического разреженного массива содержит как минимум два поля: логический индекс элемента и его значение. Для хранения физического массива, как правило, используют одну из динамических структур данных.

Задача 3.1. Шаблонный класс для разреженных массивов

Разработать шаблонный класс для представления разреженных одномерных массивов. Размер логического массива передавать через аргумент конструктора.

Класс должен обеспечивать хранение данных любого типа T, для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция

присваивания. Класс должен содержать операцию индексирования, возвращающую ссылку на найденный элемент в массиве. Если элемент с заданным индексом не найден, то операция должна создать новый элемент с этим индексом и поместить его в массив.

При необходимости добавить в класс другие методы. В клиенте `main()` продемонстрировать использование этого класса.

Заметим, что согласно условию задачи нужно разработать очень примитивный класс с минимальной функциональностью. В реальных приложениях такой класс будет, конечно, содержать и другие методы, например удаление из физического массива элемента с заданным индексом.

Выше было сказано, что каждый элемент физического массива должен содержать два поля: логический индекс элемента и его значение. Поэтому давайте начнем с разработки серверного класса для представления одного элемента физического массива. Этот класс также должен быть шаблонным, иначе как же он будет использоваться клиентом — шаблонным же классом разреженного массива? Результатом проработки этого вопроса может быть, например, следующий класс:

```
template <class DataT> class SA_item {
public:
    SA_item(long i, DataT d) : index(i), info(d) {}
    long index;
    DataT info;
};
```

Второй вопрос, который нужно решить до начала кодирования — какую структуру данных мы выберем для хранения физического массива и какими средствами ее реализуем? Чаще всего используются линейные списки, бинарные деревья или структуры данных с хешированием индексов.

Линейный список имеет наихудшие показатели по времени поиска информации с заданным ключом (индексом)¹, но в то же время он наиболее прост для программирования. Более того, чтобы не отвлекаться на детали реализации и написать компактный код, мы воспользуемся контейнерным классом `list` из STL. Вообще-то контейнерным классам STL будет посвящен семинар 6, но у нас уже есть опыт использования класса `vector` на семинаре 2; так же и здесь, приведя только минимально необходимые сведения о классе `list`, мы сможем воспользоваться многими его преимуществами.

Контейнерный класс `list` является шаблонным классом и реализован в STL в виде двусвязного списка, каждый узел которого содержит ссылки на последующий и предыдущий элементы. Для использования класса необходимо подключить заголовочный файл `<list>`.

В классе есть конструктор по умолчанию, создающий список нулевой длины. К имеющемуся списку можно добавить в его конец новый элемент с помощью

¹ Это может иметь существенное значение, если количество элементов в физическом массиве достаточно велико.

метода `push_back()`. Доступ к любому элементу списка осуществляется через *итератор* — переменную типа `list<T>::iterator`. Поскольку с понятием итератора вы еще не знакомы¹, скажем о нем несколько слов.

Проще всего рассматривать итератор как указатель на элемент списка. Он используется для просмотра списка в прямом или обратном направлении. В первом случае к итератору применяется операция инкремента, во втором — декремента.

В классе `list` есть два метода, которые позволяют организовать цикл просмотра всех элементов списка: `begin()` возвращает указатель на первый элемент, `end()` возвращает указатель на элемент, следующий *за* последним. Текущее значение итератора в цикле сравнивается со значением, полученным от метода `end()` с помощью операции `!=`, так как при произвольном размещении в памяти соседних элементов списка операция `<` для адресов элементов теряет смысл.

Поясним использование класса `list` на следующем примере:

```
#include <iostream>
#include <list>
using namespace std;

int main() {
    list<char> v1;
    v1.push_back('A');
    v1.push_back('B');
    v1.push_back('C');

    list<char>::iterator i = v1.begin();
    list<char>::iterator n = v1.end();
    for (i; i != n; ++i)
        cout << *i << ' '; // содержимое ячейки памяти,
                           // на которую указывает i
    cout << endl;
    return 0;
}
```

Мы можем позволить себе не комментировать этот пример, так как уровень ваших знаний на текущий момент, безусловно, достаточен, чтобы понять все с полувзгляда.

Теперь все готово, чтобы привести возможное решение задачи:

```
////////////////////////////////////
// Проект Task3_1
////////////////////////////////////
// SparseArr.h
#ifndef SPARSE_ARR_H
#define SPARSE_ARR_H

#include <list>
```

¹ Контейнерные классы и итераторы рассматриваются на семинаре 6.

```

template <class DataT> class SA_item {
public:
    SA_item() : index(-1), info(DataT()) {}
    SA_item(long i, DataT d) : index(i), info(d) {}
    long index;
    DataT info;
};

```

```

template <class T> class SparseArr {
public:
    SparseArr(long len) : length(len) {}
    T& operator [] (long ind);
    void Show(const char*);
private:
    std::list<SA_item<T> > arr;
    long length;
};

```

```

template <class T>
void SparseArr<T>::Show(const char* title) {
    cout << "==== " << title << "====\n";
    list<SA_item<T> >::iterator i = arr.begin();
    list<SA_item<T> >::iterator n = arr.end();
    for (i; i != n; ++i)
        cout << i->index << "\t" << i->info << endl;
}

```

```

template <class T>
T& SparseArr<T>::operator [] (long ind) {

    if ((ind < 0) || (ind > length - 1)) {
        cerr << "Error of index: " << ind << endl;
        SA_item<T> temp;
        return temp.info;
    }

    list<SA_item<T> >::iterator i = arr.begin();
    list<SA_item<T> >::iterator n = arr.end();

    for (i; i != n; ++i)
        if (ind == i->index)
            return i->info; // элемент найден

    // Элемент не найден, создаем новый элемент
    arr.push_back(SA_item<T>(ind, T()));
    i = arr.end();
    return (--i)->info;
}

```

```

#endif /* SPARSE_ARR_H */
/////////////////////////////////////////////////////////////////
// Main.cpp
#include <iostream>
#include <string>
#include "SparseArr.h"
using namespace std;

int main() {
    SparseArr<double> sa1(2000000);           // 1
    sa1[127649] = 1.1;                       // 2
    sa1[38225] = 1.2;                        // 3
    sa1[2004056] = 1.3;                     // 4
    sa1[1999999] = 1.4;                     // 5
    sa1.Show("sa1");                        // 6

    cout << "sa1[38225] = " << sa1[38225] << endl; // 7
    sa1[38225] = sa1[93];                    // 8
    cout << "After the modification of sa1:\n"; // 9
    sa1.Show("sa1");                         // 10

    SparseArr<string> sa2(1000);             // 11
    sa2[333] = "Nick";                       // 12
    sa2[222] = "Peter";                      // 13
    sa2[444] = "Ann";                        // 14
    sa2.Show("sa2");                         // 15
    sa2[222] = sa2[555];                     // 16
    sa2.Show("sa2");                         // 17
    return 0;
}
//————— конец проекта Task3_1 —————
/////////////////////////////////////////////////////////////////

```

Обратите внимание на следующие моменты.

- ❑ Поле `arr` в шаблонном классе `SparseArr`, объявленное как объект шаблонного класса `list`, актуализированного шаблонным же параметром `SA_item<T>`, предназначено для хранения физического массива. Заметим, что аргумент `T` здесь совпадает с параметром шаблонного класса `SparseArr`.
- ❑ В операции индексирования `operator[]()` прежде всего проверяется, не выходит ли значение индекса `ind` за допустимые границы.
- ❑ Если выходит, то формируется сообщение об ошибке, выводимое в поток `cerr`¹, после чего нужно либо прервать выполнение программы, либо вернуть некоторое приемлемое значение. В данном случае выбран второй вариант: создается временный объект `temp` с вызовом конструктора по умолчанию `SA_item()` и возвращается ссылка на его поле `info`.

¹ `cerr` — объект класса `ostream`, представляющий *стандартное устройство для вывода сообщений об ошибках*. Аналогично объекту `cout`, он направляет данные на терминал, но вывод объекта `cerr` не буферизируется.

- А теперь обратим наши взоры на конструктор по умолчанию в классе `SA_item` — в первоначальном варианте класса, рассмотренном выше, этого конструктора не было. Здесь же он добавлен именно для использования в нестандартных ситуациях: конструктор создает клон объекта физического массива со значением индекса, равным -1 . Этим гарантируется, что созданный элемент нигде и никогда не будет использован. Таким образом, после вывода сообщения об ошибке программа продолжит свое выполнение.
- Вряд ли стоит считать такое решение идеальным — ведь память будет замусориваться неиспользуемыми объектами. С другой стороны, терминальное прерывание работы программы вызовом `exit()` тоже не назовешь эстетичным решением. Короче говоря, перед нами — классическая ситуация, требующая генерации и последующей обработки исключения, но эту тему мы рассмотрим чуть ниже. А пока вернемся к анализу работы операции индексирования.
- Если с индексом все в порядке, то далее в цикле `for` ищется элемент физического массива с заданным индексом. В случае успеха возвращается ссылка на поле `info` найденного элемента. В случае неуспеха создается и добавляется к списку `arr` новый элемент физического массива:

```
arr.push_back(SA_item<T>(ind. T()));
```

- При этом полю `info` присваивается значение, сформированное конструктором по умолчанию `T()`. Затем вызывается метод `arr.end()`, возвращающий указатель (итератор) на элемент, следующий за *последним* элементом списка. Чтобы получить доступ к последнему элементу, применяется операция *префиксного* декремента $-i$. Полученное значение итератора позволяет вернуть из функции ссылку на поле `info` нового элемента.
- В класс добавлен метод `Show()`, который просто выводит в поток `cout` перечень элементов (индексы и значения) физического массива.
- В клиенте `main()` показано два варианта инстанцирования класса `SparseArr`. В первом случае создается логический массив `sal` из 2 000 000 элементов типа `double`. Поначалу он не содержит физических элементов. Выполнение оператора присваивания

```
sal[127649] = 1.1;
```

начнется с вызова операции `sal.operator[](127649)`. Так как элемента с указанным индексом в массиве нет, то будет создан и добавлен в конец массива элемент, имеющий индекс 127649 и значение 0.0 (значение для типа `double` по умолчанию). Операция индексирования вернет ссылку на поле `info` со значением 0.0, поэтому следующая операция присваивания изменит это значение на 1.1. Выполнение операторов 3 и 5 будет идти аналогично, а вот при выполнении оператора 4 будет обнаружена ошибка индексирования.

В операторе 7 проверяется обращение к существующему элементу массива для его вывода в поток `cout`. Интересным является оператор 8 — в результате его выполнения элемент `sal[38225]` получит значение 0. Если вы внимательно

читали этот пункт с самого начала, то такой результат не вызовет у вас вопросов. В итоге вывод на экран первой части программы (работа с массивом `sal`) будет следующим:

```
Error of index: 2004056
===== sal =====
127649  1.1
38225   1.2
1999999 1.4
sal[38225] = 1.2
After the modification of sal:
===== sal =====
127649  1.1
38225   0
1999999 1.4
93      0
```

- Во втором случае создается логический массив `sa2` из 1000 элементов типа `string`. Действия по проверке его использования аналогичны предыдущим. В результате вывод на экран второй части программы будет следующим:

```
===== sa2 =====
333 Nick
222 Peter
444 Ann
===== sa2 =====
333 Nick
222
444 Ann
555
```

Обработка исключительных ситуаций

При решении предыдущей задачи мы столкнулись с проблемой — что делать методу класса (операции индексирования), если он обнаруживает некоторую ошибку, вызванную некорректным обращением клиента к методу (недопустимое значение индекса)? Инструментарий, которым мы пользовались до сих пор, позволял предпринять одно из следующих действий:

- прервать выполнение программы;
- вернуть значение, означающее «ошибка»;
- вывести сообщение об ошибке в поток `cerr` и вернуть вызывающей программе некоторое приемлемое значение, которое позволит ей продолжать работу.

Первый вариант решения не понравится пользователю программы, так как в самый неподходящий момент она будет «ломаться» без всякого уведомления о причине своей гибели. Второй вариант возможен лишь тогда, когда возвращаемое значе-

ние функции предназначено именно для кодирования статуса ее завершения. Так бывает далеко не всегда: например, в операции индексирования `operator[]()` класса `SparseArr` возвращаемое значение есть ссылка на поле `info` объекта. Ну и, наконец, третий вариант решения тоже часто связан с трудно решаемыми вопросами: что есть «приемлемое значение» и почему программа продолжает свою работу независимо от обнаруженной ошибки?

С подобными проблемами часто сталкиваются разработчики библиотек классов широкого применения (библиотек типа STL). Автор библиотечного класса может обнаружить ошибки времени выполнения, но, в общем случае, не знает, как должен реагировать клиент на эти ошибки.

Для решения подобных проблем в C++ были введены средства генерации и обработки *исключений* (exception). Заметим, что такими средствами пользуются не только при разработке библиотек. Например, в процессе выполнения конструктора класса может возникнуть какая-то нештатная ситуация (скажем, нехватка памяти). Поскольку конструктор не имеет возвращаемого значения, то единственным способом для него уведомить об этом клиента также является генерация исключения.

Определение исключений

Для того чтобы работать с исключениями, необходимо:

- Выделить контролируемый блок — блок `try`.
- Предусмотреть генерацию одного или нескольких исключений операторами `throw` внутри блока `try` или внутри функций, вызываемых из этого блока.
- Разместить сразу за блоком `try` один или несколько обработчиков исключений `catch`.

Контролируемый блок — это составной оператор, перед которым записано ключевое слово `try`:

```
try {
    // фрагмент кода
}
```

Оператор `throw`, предназначенный для генерации исключения, имеет вид `throw выражение`. Тип выражения, стоящего после `throw`, определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, и происходит поиск соответствующего обработчика и передача ему управления.

Синтаксис обработчиков напоминает определение функции с одним параметром и именем `catch`:

```
catch (/* ... */) {
    // действия по обработке исключения
}
```

Объявление параметра обработчика возможно в одной из трех форм:

```
catch (Type) { // обработка исключения типа Type }
catch (Type info) {
    // обработка исключения типа Type
    // с использованием значения info
}
catch (...) { // обработка исключений всех типов }
```

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками исключений. Туда же, минуя код всех обработчиков, передается управление, если исключение в try-блоке не было сгенерировано.

Если обработчик не в состоянии полностью обработать ошибку, он может сгенерировать исключение повторно с помощью оператора throw без параметров. В этом случае предполагается наличие внешних объемлющих блоков, в которых может находиться другой обработчик для этого типа исключения.

Перехват исключений

Когда с помощью throw генерируется исключение, функции исполнительной библиотеки C++ выполняют следующие действия:

- ❑ создают копию параметра throw в виде статического объекта, который сохраняется до тех пор, пока исключение не будет обработано;
- ❑ в поисках подходящего обработчика раскручивают стек;
- ❑ передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

А что такое подходящий разработчик? Рассмотрим такой пример:

```
try {
    throw E();
}
catch (H) {
    // когда мы сюда попадем?
}
```

Обработчик будет вызван, если:

- ❑ H и E — одного типа;
- ❑ H является открытым базовым классом для E;
- ❑ H и E — указатели либо ссылки и для них справедливо одно из предыдущих утверждений.

Следующая маленькая программка демонстрирует перехват исключений:

```
class A {
public:
    A() { cout << "Constructor of A\n"; }
    ~A() { cout << "Destructor of A\n"; }
};
```

```

class Error {};
class ErrorOfA : public Error {};

void foo() {
    A a;
    throw 1;
    cout << "This message is never printed" << endl;
}

int main() {
    try {
        foo();
        throw ErrorOfA();
    }
    catch(int)      { cerr << "Catch of int\n"; }
    catch(ErrorOfA) { cerr << "Catch of ErrorOfA\n"; }
    catch(Error)    { cerr << "Catch of Error\n"; }
    return 0;
}

```

Она выдаст следующий результат:

```

Constructor of A
Destructor of A
Catch of int

```

Обратите внимание на то, что первым генерируется исключение `throw 1` внутри функции `foo()`. Исполнительная система C++ создает копию объекта «целая константа 1» и начинает поиск подходящего обработчика. Поскольку внутри `foo()` ничего подходящего нет, поиск переносится вовне — в клиент `main()`. Но (очень важная деталь!) перед тем, как покинуть тело функции `foo()`¹, система вызывает деструкторы всех ее локальных объектов!

В функции `main()` нужный обработчик обнаруживается и дает о себе знать сообщением «Catch of int». Таким образом, генерация второго исключения `throw ErrorOfA()` здесь невозможна.

Поменяйте местами строки в блоке `try` и запустите программу снова. Теперь исключение `throw ErrorOfA()` будет сгенерировано и вы получите результат:

```
Catch of ErrorOfA
```

Интересно, что класс `ErrorOfA` совершенно пустой. Но этого достаточно, чтобы использовать его экземпляр (в данном случае анонимный) для генерации исключения. А теперь прокомментируйте строку с обработчиком `catch(ErrorOfA)`. Программа выдаст:

```
Catch of Error
```

¹ Это и есть раскрутка стека.

Неперехваченные исключения

Если исключение сгенерировано, но не перехвачено, вызывается стандартная функция `std::terminate()`. Функция `terminate()` будет также вызвана, если механизм обработки исключения обнаружит, что стек разрушен, или если деструктор, вызванный во время раскрутки стека, пытается завершить свою работу при помощи исключения. По умолчанию `terminate()` вызывает функцию `abort()`.

Чтобы увидеть, что происходит при вызове этих функций, скомпилируйте и выполните следующую программу:

```
int main() {
    throw 5;    // произвольное значение
    return 0;
}
```

На нашем компьютере (Windows 2000 Professional и Visual Studio 6.0) на экран вываливается окно с сообщением об аварийном завершении программы (рис. 3.1).

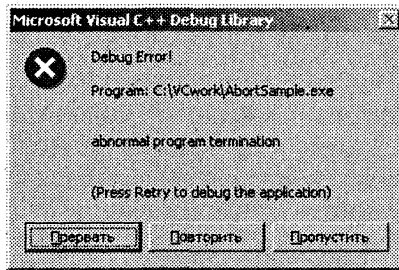


Рис. 3.1. Сообщение об аварийном завершении программы (проект AbortSample)

Вы можете заменить (если захотите) вызов функции `abort()` вызовом своего обработчика. Это делается с помощью функции `set_terminate()`. Например:

```
void SoftAbort() {
    cerr << "Program is terminated." << endl;
    exit(1);
}

int main() {
    set_terminate(SoftAbort);
    throw 5;
    return 0;
}
```

Проверьте, что завершение программы теперь действительно окажется более мягким.

Классы исключений. Иерархии исключений

Отметим, что хотя язык позволяет генерировать исключения любого встроенного типа (как, например, типа `int` в рассмотренных выше примерах), все же в реальных программных системах такие исключения используются редко. Гораздо удобнее создавать специальные классы исключений (такие, как `ErrorOfA`) и использовать в операторе `throw` либо объекты этих классов, либо анонимные экземпляры¹.

При необходимости в этих классах можно передавать через параметры конструктора и сохранять для последующей обработки любую информацию о состоянии программы в момент генерации исключения. Другое преимущество этого подхода — возможность создания иерархии исключений. Например, исключения для математической библиотеки можно организовать следующим образом:

```
// Базовый класс обработки ошибок
class MathError { /* ... */ };
// Класс ошибки переполнения
class Overflow : public MathError { /* ... */ };
// Класс ошибки "деление на ноль"
class ZeroDivide : public MathError { /* ... */ };
// ...
```

Если к тому же не полениться и добавить в базовый класс некий виртуальный метод `ErrProcess()`, заместив его в производных классах версиями `ErrProcess()`, ориентированными на обработку конкретного типа ошибки, то это позволит задавать после блока `try` единственный обработчик `catch`, принимающий объект базового класса:

```
try {
    // генерация любых исключений MathError, Overflow, ZeroDivide, ...
}
catch (MathError& me) {
    me.ErrProcess() // Обработка любого исключения
}
```

На этапе выполнения такой обработчик `catch` будет обрабатывать как исключения типа `MathError`, так и исключения любого производного типа, причем благодаря полиморфизму каждый раз будет вызываться версия метода `ErrProcess()`, соответствующая именно данному типу исключения.

Организация исключений в виде иерархий имеет неоценимое значение, если преследуется цель разработки легко модифицируемого программного обеспечения. Ведь при структуре кода, которую мы рассмотрели, добавление в систему нового вида исключения, входящего в существующую иерархию, вообще не потребует изменять написанные ранее фрагменты кода, предназначенные для перехвата и обработки исключений.

¹ Через вызов конструктора класса.

Спецификации исключений

В заголовке функции можно задать список типов исключений, которые она может прямо или косвенно порождать. Этот список приводится в конце заголовка и предваряется ключевым словом `throw`. Например:

```
void Func(int a) throw (Foo1, Foo2);
```

Такое объявление означает, что `Func` может сгенерировать только исключения `Foo1`, `Foo2` и исключения, являющиеся производными от этих типов, но не другие. Заголовок является интерфейсом функции, поэтому такое объявление дает пользователям функции определенные гарантии. Это очень важно при использовании библиотечных функций, так как определения функций в этом случае не всегда доступны.

Что произойдет, если функция вдруг нарушит взятые на себя обязательства и в ее недрах будет возбуждено исключение, не соответствующее списку спецификации исключений? Тогда система вызовет обработчик `std::unexpected()`, который может попытаться сгенерировать свое исключение¹, и если оно не будет противоречить спецификации, то продолжится поиск подходящего обработчика, в противном случае вызывается `std::terminate()`.

Если вас не устраивает поведение `unexpected()` по умолчанию, вы можете установить собственную функцию, которую он будет вызывать. Для этого нужно воспользоваться функцией `set_unexpected()`.

Если спецификация исключений задана в виде `throw()`, это означает, что функция вообще не генерирует исключений.

Отсутствие спецификации исключений в заголовке функции, то есть запись заголовка в привычном нам виде, означает, что функция может сгенерировать любое исключение.

Если заголовок функции содержит спецификацию исключений, то каждое объявление этой функции (а также определение) должно иметь спецификацию исключений с точно таким же набором типов исключений. Виртуальная функция может быть замещена в производном классе функцией с не менее ограничительной спецификацией исключений, чем ее собственная.

Спецификация исключений *не является частью типа функции*, и `typedef` не может ее содержать. Например:

```
typedef int (*PF)() throw(A); // ошибка
```

Исключения в конструкторах

Исключения предоставляют единственную возможность передать информацию об обломе (срыве, крахе, катастрофе), случившемся в процессе создания нового объекта. Рассмотрим пример небольшой программы, написанной очень некор-

¹ Зависит от реализации.

ректно (мягко выражаясь), но именно это позволяет смоделировать ситуации, которые могут иметь место и в корректной, но большой программе:

```
class Vect {
public:
    Vect(char);
    ~Vect() { delete [] p; }
    int& operator [] (int i) { return p[i]; }
    void Print();
private:
    int* p;
    char size;
};

Vect::Vect(char n) : size(n) {
    p = new int[size];
    if (!p) {
        cerr << "Error of Vect constructor" << endl;
        return;
    }
    for (int i = 0; i < size; ++i) p[i] = int();
}

void Vect::Print() {
    for (int i = 0; i < size; ++i)
        cout << p[i] << " ";
    cout << endl;
}

int main() {
    Vect a(3);
    a[0] = 0; a[1] = 1; a[2] = 2;
    a.Print();

    Vect a1(200);
    a1[10] = 5;
    a1.Print();
    return 0;
}
```

Сначала несколько слов о возможной предыстории появления этого кода на свет. В программе реализован непритязательный класс Vect, предназначенный для создания и использования одномерных массивов типа int произвольного размера (размер передается через параметр конструктора класса).

Руководитель проекта, для которого предназначен этот класс, заявил, что размер массива никогда не превысит число 256. Проект (система) разрабатывается для «железа» с жесткими ограничениями на размер оперативной памяти. Програм-

мисты — члены бригады — ведут борьбу за каждый байт! Человек, которому поручили указанную подзадачу, написал код, приведенный выше. Для хранения в классе `Vect` информации о размере массива он использовал поле `char size`, полагаясь на то, что для типа `char` выделяется один байт, а диапазон возможных значений для восьмиразрядного двоичного числа составляет `0...255`. В функции `main()` созданный класс тестируется.

Когда дело дошло до тестирования, выяснилось, что программа выводит на экран:

```
0 1 2
Error of Vect constructor
```

после чего серьезно «ломается» (среда выполнения сообщает о попытке обратиться по несуществующему адресу)¹. Выполнение по шагам показывает, что крах происходит при попытке выполнить оператор `a1[10] = 5`.

Займемся экспресс-анализом причин ошибки. Вывод программы показывает, что работа с объектом `a` прошла нормально. Значит, сообщение «Error of Vect constructor» относится к работе конструктора объекта `a1`. Вывод сообщения происходит тогда, когда операция `new` не смогла выделить запрашиваемую память и вернула нулевое значение указателя `p`. Но разве значение `200`, переданное в качестве аргумента конструктора, такое уж невыполнимое требование²?

Обычно для начинающих программистов непросто выяснить причину. Баг кажется таинственным и непонятным... Наш герой, видимо, тоже был начинающим программистом. Более того, скорее всего, он невнимательно изучил учебник [1], где на страницах 24–25 объясняется, что тип `char` — это сокращенное обозначение типа `signed char`, в котором старший бит используется для представления знака числа. Поэтому диапазон представимых чисел для типа `char` составляет `-128...+127`.

Так, стало теплее. Как же будет интерпретироваться в этом случае десятичное число `200`? Преобразовав его в шестнадцатеричное, получим число `C8`. Преобразовав шестнадцатеричное число `C8` в двоичный эквивалент, получаем `11001000`. Вы видите? Нет, вы видите, что старший разряд равен единице?.. Это означает, что компьютер воспримет его как `-1001000`.

Обратный перевод в десятичную систему будет несколько сложнее. Напомним, что отрицательные целые числа хранятся в памяти компьютера в *дополнительном коде*. Дополнительный код двоичного числа получается инверсией всех разрядов с последующим прибавлением единицы. Значит, для обратного перевода нужно сначала вычесть единицу, а потом проинвертировать все разряды. В итоге получим число `-56`. Зачем такие пространные объяснения? Да просто мы не уверены, что каждый читатель, увидев в окне отладчика при выполнении програм-

¹ Вы можете это проверить, откомпилировав и выполнив программу на своем компьютере.

² Хотя, в конечном итоге, проект будет реализован на аппаратуре с ограниченной памятью, тестирование ведется на обычном современном компьютере с сотнями мегабайт оперативной памяти.

мы по шагам в качестве значения поля `size` странное число `-56`, сразу сообразит, откуда оно взялось.

Да, вот оно как бывает. Объяснение к программе длиннее ее самой... И так, мы нашли ошибку в реализации класса. Получив отрицательное число в качестве размера запрашиваемой памяти, операция `new`, войдя в состояние глубокого стресса, возвращает значение `0`, что означает отказ в выделении памяти. Далее следует вывод в `cout` предупреждающего сообщения о сбое в работе конструктора. Но что толку-то — ведь программа продолжает работать! И как результат — фатальная ошибка при попытке присвоить значение несуществующему (не созданному) объекту.

Исправить ошибку несложно — нужно все `char` заменить на `unsigned char`¹. Но не будем спешить. Дело в том, что мы имеем прекрасную модель ненормативного поведения операции `new`². И для дальнейших экспериментов она нам еще пригодится. Ведь в общем случае, даже если мы напишем безукоризненно корректный код класса, никто не может дать нам гарантию, что клиент не попытается непреднамеренно использовать наш класс некорректным образом.

Приведенный пример показал, что если в конструкторе класса имеется вызов функций (операций), которые могут не справиться с возложенной на них задачей, необходимо информировать об этом клиента (функцию, в теле которой осуществляется попытка создать объект класса). Единственный приличный способ сделать это — использовать механизм исключений³.

Для приведенного выше примера необходимо изменить код конструктора класса на следующий:

```
Vect::Vect(char n) : size(n) {
    p = new int[size];
    if (!p)
        throw "ErrVectConstr";

    for (int i = 0; i < size; ++i) p[i] = int();
}
```

В клиенте `main()` использование объектов класса можно организовать следующим образом:

```
int main() {
    try {
        Vect a(3);
        a[0] = 0; a[1] = 1; a[2] = 2;
        a.Print();
    }
```

¹ Рекомендуем вам проверить это.

² Попробуйте в качестве упражнения найти другой способ заставить операцию `new` вернуть нулевое значение.

³ Ну, не глобальные же переменные использовать!

```

    Vect a1(200);
    a1[10] = 5;
    a1.Print();
}
catch (char* msg) {
    cerr << "Error: " << msg << endl;
}
return 0;
}

```

Отметим, что здесь используется генерация исключения типа `const char*` только ради краткости изложения. Наше прежнее замечание о целесообразности использования классов исключений остается в силе. Рекомендуем вам проверить работу новой версии программы.

Исключения в деструкторах

Рассматривая эту тему, важно помнить, что если деструктор, вызванный во время раскрутки стека, попытается завершить свою работу при помощи исключения, то система вызовет функцию `terminate()`. На этапе отладки программы это допустимо, но в готовом продукте появление сообщений, подобных показанному на рис. 3.1, должно быть сведено к минимуму. В идеале — исключено совсем.

Отсюда наиважнейшее требование к деструктору: *ни одно из исключений, которое могло бы появиться в процессе работы деструктора, не должно покинуть его пределы!* Чтобы выполнить это требование, придерживайтесь следующих двух правил:

1. Никогда не генерируйте исключения в теле деструктора с помощью `throw`.
2. Если «погребальные» действия по отношению с деструктурируемому объекту достаточно сложны и связаны с вызовом других функций, относительно которых у вас нет гарантий отсутствия генерации исключений, то рекомендуется инкапсулировать все эти действия в некотором методе, например `Destroy()`, и вызывать данный метод с использованием блока `try/catch`:

```

T::Destroy() {
    // код, который может генерировать исключения
}
T::~T() {
    try { Destroy(); }
    catch(...) { /* ... */ }
}

```

Мы завершили рассмотрение вопросов, связанных с обработкой исключений. Рекомендуем вам в качестве упражнения доработать программу в проекте `Task3_1`, предусмотрев в реализации операции индексирования генерацию исключения для ошибочного значения индекса.

Теперь рассмотрим решение задачи, в которой требуется использовать и шаблонные классы, и обработку исключений.

Задача 3.2. Шаблонный класс векторов (динамических массивов)

Разработать шаблонный класс *Vect* для представления динамических одномерных массивов (векторов). Класс должен обеспечивать хранение данных любого типа *T*, для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция присваивания. Класс должен содержать:

- конструктор по умолчанию, создающий вектор нулевого размера;
- конструктор, создающий вектор заданного размера;
- операцию индексирования, возвращающую ссылку на соответствующий элемент вектора;
- метод, добавляющий элемент в произвольную позицию вектора;
- метод, добавляющий элемент в конец вектора;
- метод, удаляющий элемент из конца вектора.

При необходимости добавить в класс другие методы. Предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций.

В клиенте *main()* продемонстрировать использование этого класса.

Одним из принципиальных (мы бы даже сказали — стратегических) вопросов при разработке контейнерного класса является вопрос реализации хранения элементов контейнера, или, другими словами, выбор подходящей структуры данных (массив, список, бинарное дерево и т. п.).

В данном случае мы остановим наш выбор на динамическом массиве, размещаемом в памяти посредством операции *new*, поскольку это наиболее простое решение. После размещения адрес первого элемента вектора будет запоминаться в поле *T* first*, а адрес элемента, следующего за последним, — в поле *T* last*. С учетом того, что используемый в классе метод *size()* возвращает количество элементов в векторе, размещение контейнера в памяти поясняется на рис. 3.2.

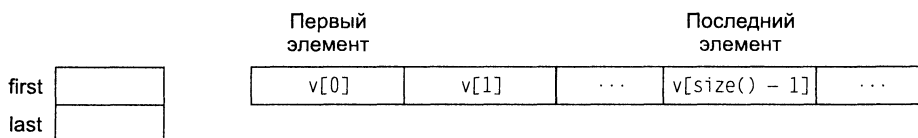


Рис. 3.2. Размещение вектора *v* в памяти

При решении этой задачи мы постараемся сделать наш класс *Vect* как можно более похожим — с точки зрения его интерфейса и, частично, реализации — на контейнерный класс `std::vector` из библиотеки STL (конечно, с учетом тех ограни-

чений, которые вытекают из постановки задачи). Поэтому в состав интерфейса будут включены методы:

- ❑ `void insert(T* _P, const T& _x)` — вставка элемента в позицию `_P`;
- ❑ `void push_back(const T& _x)` — вставка элемента в конец вектора;
- ❑ `void pop_back()` — удаление элемента из конца вектора;
- ❑ `T* begin()` — получение указателя на первый элемент;
- ❑ `T* end()` — получение указателя на элемент, следующий за последним;
- ❑ `size_t size()` — получение размера вектора¹.

Эти методы имитируют интерфейс класса `std::vector`². Целью такого подражания является психологическая подготовка вас, любезные наши читатели, к более легкому восприятию контейнерных классов STL (семинар 6).

К двум конструкторам, требующимся по заданию, мы добавим еще конструктор копирования, чтобы обеспечить возможность передачи объектов класса в качестве аргументов для любой внешней функции.

По некоторым параметрам наш класс `Vect`, однако, будет превосходить класс `std::vector`. Для целей отладки и обучения мы хотим визуализировать работу таких методов класса, как конструктор копирования и деструктор, путем вывода на терминал соответствующих сообщений. Чтобы эти сообщения были более информативны, мы снабдим каждый объект класса так называемым отладочным именем (поле `markName`), которое позволит распознавать источник сообщения.

Трудно переоценить, какую пользу приносят эти сообщения начинающим программистам, помогая прочувствовать скрытые механизмы функционирования класса. Для более опытных читателей эти средства могут оказаться полезными при поиске неочевидных ошибок в программе.

Задание требует также предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций. Наиболее очевидными из таких ситуаций являются: *ошибка индексирования*, когда клиент пытается получить доступ к несуществующему элементу вектора, и *ошибка удаления несуществующего элемента* из конца вектора — когда вектор пуст. В принципе, возможна еще одна нештатная ситуация, связанная с нехваткой памяти, когда операция `new` возвращает нулевой указатель. Но мы подробно рассмотрели, как решать эту проблему, в разделе «Исключения в конструкторах», и сейчас, ради краткости изложения, будем полагать, что обладаем неограниченной памятью.

Для обработки исключительных ситуаций создадим иерархию классов во главе с базовым классом `VectError`. Производный класс `VectRangeErr` будет предназначен для обработки ошибок индексирования, а производный класс `VectPopErr` — для обработки ошибки удаления несуществующего элемента.

Ниже приводится код предлагаемого решения задачи, после которого даются некоторые пояснения.

¹ Тип `std::size_t` является синонимом типа `unsigned int`.

² Уточним, что вместо типа `iterator`, имеющегося в `std::vector`, здесь используется указатель `T*`, но концептуально это одно и то же.

```

////////////////////////////////////
// Проект Task3_2
////////////////////////////////////
// VectError.h
#ifndef _VECT_ERROR_
#define _VECT_ERROR_
#include <iostream>

#define DEBUG

class VectError {
public:
    VectError() {}
    virtual void ErrMsg() const {
        std::cerr << "Error with Vect object.\n";
    }
    void Continue() const {
#ifdef DEBUG
        std::cerr << "Debug: program is being continued.\n";
#else
        throw;
#endif
    }
};

class VectRangeErr : public VectError {
public:
    VectRangeErr(int _min, int _max, int _actual) :
min(_min), max(_max), actual(_actual) {}
    void ErrMsg() const {
        std::cerr << "Error of index: ";
        std::cerr << "possible range: " << min << " - " <<
            max << ", ";
        std::cerr << "actual index: " << actual << std::endl;
        Continue();
    }
private:
    int min, max;
    int actual;
};

class VectPopErr : public VectError {
public:
    void ErrMsg() const {
        std::cerr << "Error of pop\n";
        Continue();
    }
};

```

```

#endif /* _VECT_ERROR_ */
////////////////////////////////////
// Vect.h
#ifndef _VECT_
#define _VECT_
#include <iostream>
#include <string>
#include "VectError.h"

// Template class Vect
template<class T> class Vect {
public:
    explicit Vect()
        : first(0), last(0) {}
    explicit Vect(size_t _n, const T& _v = T()) {
        Allocate(_n);
        for (size_t i = 0; i < _n; ++i)
            *(first + i) = _v;
    }
    Vect(const Vect&); // конструктор копирования
    Vect& operator =(const Vect&); // операция присваивания
    ~Vect() {
#ifdef DEBUG
        cout << "Destructor of " << markName << endl;
#endif
        Destroy();
        first = 0, last = 0;
    }

    // установить отладочное имя
    void mark(std::string& name) { markName = name; }
    // получить отладочное имя
    std::string mark() const { return markName; }

    size_t size() const; // получить размера вектора
    T* begin() const { return first; } // получить указатель на 1-й элемент
    T* end() const { return last; } // получить указатель на элемент,
    // следующий за последним элементом
    T& operator[](size_t i); // операция индексирования
    void insert(T* _P, const T& _x); // вставка элемента в позицию _P
    void push_back(const T& _x); // вставка элемента в конец вектора
    void pop_back(); // удаление элемента из конца вектора
    void show() const; // вывод в cout содержимого вектора

protected:
    void Allocate(size_t _n) {
        first = new T[_n * sizeof(T)];
        last = first + _n;
    }
};

```

```

    }
    void Destroy() {
        for (T* p = first; p != last; ++p)    p->~T();
        delete [] first;
    }

    T* first; // указатель на 1-й элемент
    T* last;  // указатель на элемент, следующий за последним
    std::string markName;
};

// Конструктор копирования
template<class T>
Vect<T>::Vect(const Vect& other) {
    size_t n = other.size();
    Allocate(n);
    for (size_t i = 0; i < n; ++i)
        *(first + i) = *(other.first + i);
    markName = string("Copy of ") + other.markName;
#ifdef DEBUG
    cout << "Copy constructor: " << markName << endl;
#endif
}

// Операция присваивания
template<class T>
Vect<T>& Vect<T>::operator =(const Vect& other) {
    if (this == &other) return *this;
    Destroy();
    size_t n = other.size();
    Allocate(n);
    for (size_t i = 0; i < n; ++i)
        *(first + i) = *(other.first + i);
    return *this;
}

// Получение размера вектора
template<class T>
size_t Vect<T>::size() const {
    if (first > last)
        throw VectError();
    return (0 == first ? 0 : last - first);
}

// Операция доступа по индексу
template<class T>
T& Vect<T>::operator [](size_t i) {
    if (i < 0 || i > (size() - 1))

```

```

        throw VectRangeErr(0, last - first - 1, i);

    return (*(first + i));
}

// Вставка элемента со значением _x в позицию _P
template<class T>
void Vect<T>::insert(T* _P, const T& _x) {
    size_t n = size() + 1; // новый размер
    T* new_first = new T [n * sizeof(T)];
    T* new_last = new_first + n;

    size_t offset = _P - first;
    for (size_t i = 0; i < offset; ++i)
        *(new_first + i) = *(first + i);

    *(new_first + offset) = _x;

    for (i = offset; i < n; ++i)
        *(new_first + i + 1) = *(first + i);
    Destroy();
    first = new_first;
    last = new_last;
}

// Вставка элемента в конец вектора
template<class T>
void Vect<T>::push_back(const T& _x) {
    if (!size()) {
        Allocate(1);
        *first = _x;
    }
    else insert(end(), _x);
}

// Удаление элемента из конца вектора
template<class T>
void Vect<T>::pop_back() {
    if (last == first)
        throw VectPopErr();
    T* p = end() - 1;
    p->~T();
    last--;
}

// Вывод в cout содержимого вектора
template<class T>
void Vect<T>::show() const {
    cout << "\n==== Contents of " << markName << "====" << endl;
}

```



```

    size_t n = size();
    for (size_t i = 0; i < n; ++i)
        cout << *(first + i) << " ";
    cout << endl;
}

#endif /* _VECT_ */
////////////////////////////////////
// Main.cpp
#include "Vect.h"
#include <iostream>
#include <string>
using namespace std;

template<class T> void SomeFunction(Vect<T> v) {
    std::cout << "Reversive output for " << v.mark() << ":" << endl;
    size_t n = v.size();
    for (int i = n - 1; i >= 0; --i)
        std::cout << v[i] << " ";
    std::cout << endl;
}

int main() {
    try {
        string initStr[5] = {"first", "second", "third", "fourth", "fifth"};

        Vect<int> v1(10); v1.mark(string("v1"));
        size_t n = v1.size();
        for (int i = 0; i < n; ++i)
            v1[i] = i+1;
        v1.show();
        SomeFunction(v1);

        try {
            Vect<string> v2(5);
            v2.mark(string("v2"));
            size_t n = v2.size();
            for (int i = 0; i < n; ++i)
                v2[i] = initStr[i];

            v2.show();
            v2.insert(v2.begin() + 3, "After_third");
            v2.show();
            cout << v2[6] << endl;
            v2.push_back("Add_1");
            v2.push_back("Add_2");
            v2.push_back("Add_3");
            v2.show();

            v2.pop_back();

```

```

        v2.pop_back();
        v2.show();
    }
    catch (VectError& vre) { vre.ErrMsg(); }

    try {
        Vect<int> v3;
        v3.mark(string("v3"));

        v3.push_back(41);
        v3.push_back(42);
        v3.push_back(43);
        v3.show();

        Vect<int> v4;
        v4.mark(string("v4"));
        v4 = v3;
        v4.show();

        v3.pop_back();    v3.pop_back();
        v3.pop_back();    v3.pop_back();
        v3.show();
    }
    catch (VectError& vre) { vre.ErrMsg(); }
}
catch (...) { cerr << "Epilogue: error of Main().\n": }

return 0;
}
//————— конец проекта Task3_2 —————
////////////////////////////////////

```

Наши комментарии будут следовать в порядке размещения текста модулей.

Модуль VectError.h содержит иерархию классов исключений. В базовом классе VectError определены два метода.

Виртуальный метод ErrMsg(), обеспечивающий вывод по умолчанию сообщения об ошибке (в производных классах этот метод замещается конкретными методами).

Метод Continue(), управляющий стратегией продолжения работы программы после обнаружения ошибки. Стратегия зависит от конфигурации программы. Конфигурация программы управляется наличием или отсутствием единственной директивы в начале файла: #define DEBUG.

❑ Если лексема DEBUG определена, то программа компилируется в *отладочной конфигурации*, если не определена — то в *выпускной конфигурации*. Для метода Continue() это означает, что:

- в отладочной конфигурации выводится сообщение «Debug: program is being continued», после чего работа программы продолжается;
 - в выпускной конфигурации повторно возбуждается (оператор `throw`) то исключение, которое было первопричиной цепочки событий, завершившихся вызовом данного метода.
- В производных классах `VectRangeErr` и `VectPopErr` переопределяется метод `ErrMsg()` с учетом вывода информации о конкретной ошибке. После вывода вызывается метод `Continue()` базового класса. Обратите внимание, что класс `VectRangeErr` содержит поля `min`, `max` и `actual`, в которых запоминается и затем выводится с помощью метода `ErrMsg()` информация, существенная для диагноза ошибки.

Модуль `Vect.h` содержит определение шаблонного класса `Vect`. Отметим наиболее интересные моменты.

Для управления выделением и освобождением ресурсов класс снабжен методами `Allocate()` и `Destroy()`¹. Напомним, что операция `new T [n]`, где `n` — количество элементов, которое мы хотим поместить в массив, не только выделяет память, но и инициализирует элементы посредством выполнения `T()` — конструктора по умолчанию для типа `T`. Поэтому в методе `Destroy()` мы сначала в цикле вызываем деструкторы `~T()` для всех элементов массива, а уже потом освобождаем память операцией `delete`.

Обратите внимание на то, что в заголовке оператора `for` условие продолжения циклических вычислений записано в виде `p != last`, а не `p < last`. Здесь мы следуем традиции STL, порожденной тем обстоятельством, что для произвольной структуры данных (например, для списка) соседние элементы не обязаны занимать подряд идущие ячейки памяти, а могут быть разбросаны в памяти как угодно. Поэтому адрес любого элемента в контейнере может оказаться больше адреса `last`. Условие же `p != last` позволяет корректно организовать цикл для любой произвольной структуры данных.

Деструктор и конструктор копирования содержат вывод сообщений типа «Здесь был я!», дополненных печатью содержимого `markName`. Весь этот вывод осуществляется *только в отладочной конфигурации* программы.

Операция присваивания (верная спутница конструктора копирования) реализована стандартным способом.

В операции доступа по индексу `operator[]()` служба внутренней охраны (оператор `if`) проверяет, не несет ли индекс с собой что-либо взрывоопасное (значение, лежащее вне пределов допустимого диапазона). Если индекс оказывается неблагонадежным, то служба бьет тревогу, вызывая исключение типа `VectRangeErr`. Вызов производится через анонимный экземпляр класса с передачей конструктору трех аргументов: минимальной границы, максимальной границы и текущего значения для индекса.

¹ Методы размещены в защищенной части класса, так как относятся к его реализации.

В методе получения размера вектора `size()` предусмотрена проверка на лояльность взаимоотношений между `first` и `last`. Вообще-то непонятно, с каких это дел `first` может оказаться больше, чем `last`? Да, в нормально функционирующем классе это невозможно. Но в случае каких-либо ошибок или сбоев такое тоже может случиться¹. Поэтому для повышения надежности функционирования класса и облегчения диагноза ошибок указанная проверка весьма полезна, тем более что реализуется она всего одним оператором. Оператор, обнаружив беду, генерирует исключение типа `VectError`.

Наиболее сложным в реализации оказался метод `insert()` — вставка элемента со значением `_x` в произвольную позицию `_p`. Алгоритм вставки работает следующим образом. Создается новый вектор размером на единицу больше существующего вектора²; адреса размещения нового вектора сохраняются в локальных переменных `new_first`, `new_last`. Затем в новый вектор копируется первая часть существующего вектора, начиная с первого элемента и заканчивая элементом, предшествующим элементу с адресом `_p`.

Поскольку доступ к элементам осуществляется через смещение относительно указателя `first`, перед копированием требуется вычислить `offset` — смещение, соответствующее адресу `_p`³. Затем элементу с адресом `new_first + offset` присваивается значение `_x`. После этого оставшаяся часть существующего вектора копируется в новый вектор, начиная с позиции `new_first + offset + 1`. Вот и все. Осталось освободить ресурсы, занятые существующим вектором (`Destroy()`), и назначить полям `first` и `last` новые значения.

Метод `push_back()` — вставка элемента со значением `_x` в конец вектора — получился очень простым благодаря использованию серверной функции `insert()`. Если вектор пуст, то выделяются ресурсы для хранения одного элемента, и этому элементу присваивается значение `_x`. В противном случае вызывается `insert()` для вставки элемента в позицию, определенную с помощью `end()`.

В методе `pop_back()` — удаление элемента из конца вектора — проверяется, не пуст ли вектор. В пустом векторе `first = last = 0`. Если это так, то удалять нечего, и, значит, клиент ошибся, затребовав такую операцию, а следовательно, надо ему об этом сообщить — генерируется исключение типа `VectPopErr`. В противном случае определяется указатель на последний элемент вектора, вызывается для него деструктор `~T()`, и уменьшается значение `last` на единицу⁴.

¹ Для любознательных: попробуйте закомментировать оператор `if` в теле метода `pop_back()`, убрав таким образом генерацию исключения `VectPopErr`, — в результате первая же ошибка удаления несуществующего элемента вызовет состояние `first > last`.

² Возможны более сложные алгоритмы выделения памяти, основанные на блочном принципе: память выделяется блоками, и пока текущий вектор не выходит за пределы блока, можно не выделять дополнительную память. В нашем решении использован наипростейший, но, конечно, не самый эффективный алгоритм.

³ Напомним, что разность двух указателей — это разность их значений, деленная на размер типа в байтах. Например, в применении к массивам разность указателей на второй и седьмой элементы равна 5.

⁴ Здесь опять используется специфическая *арифметика указателей*.

Модуль `Main.cpp` содержит определение глобальной шаблонной функции `SomeFunction()` и определение функции `main()`.

Функция `SomeFunction()` обеспечивает реверсивный вывод содержимого вектора в поток `cout`. Стратегическое же ее назначение — проверить передачу объектов конкретного класса `Vect` в качестве аргументов функции.

Функция `main()` предназначена для тестирования класса `Vect`. Советуем внимательно изучить ее многослойную архитектуру: внешний блок `try` содержит в себе два внутренних блока `try`. И это не случайно!

Первый внутренний блок `try` предназначен для тестирования операций с объектом `v2`, среди которых есть операция, вызывающая ошибку индексирования. Второй блок `try` работает с объектом `v3`, причем одна из операций вызывает ошибку удаления несуществующего элемента.

Обратите внимание на то, что обработчики `catch` в обоих случаях имеют своим параметром объект базового класса `VectError`, но благодаря полиморфизму они будут перехватывать и исключения производных классов `VectRangeErr` и `VectPopErr`, так что в результате будет вызываться конкретный метод `ErrMsg()` для данного типа ошибки! Вспомним, что после вывода сообщения об ошибке метод `ErrMsg()` вызывает метод `Continue()`, а работа последнего зависит от конфигурации программы.

В отладочной конфигурации `Continue()` сообщает о продолжении работы программы и возвращает управление клиенту. В выпускной конфигурации `Continue()` возбуждает исключение повторно. Так вот, для того чтобы его поймать, и служит внешний блок `try` со своим обработчиком `catch(...)` в основной программе.

Откомпилируйте и выполните программу, чтобы увидеть, как она работает. Исходный текст настроен на работу в отладочной конфигурации. Всмотритесь внимательно в отладочные сообщения конструктора копирования и деструктора. А теперь закомментируйте директиву `#define DEBUG` и выполните программу после компиляции заново. Почувствуйте разницу!

Решение задачи завершено.

Давайте повторим наиболее важные моменты этого семинара.

1. Шаблоны классов поддерживают парадигму обобщенного программирования — программирования с использованием типов в качестве параметров.
2. Определение конкретного класса по шаблону класса и аргументам шаблона называется инстанцированием (актуализацией) шаблона.
3. Параметрами шаблонов могут быть абстрактные типы или переменные встроженных типов.
4. Для настройки шаблонного класса на некоторую стратегию возможно использование классов функциональных объектов.
5. Обработка исключительных ситуаций позволяет разделить проблему обработки ошибок на две фазы: генерацию исключения в случае нарушения каких-то заданных условий и последующую обработку сгенерированного исключения.

6. Фазы генерации и обработки обычно разнесены в программе по разным компонентам (модулям, функциям). Такая организация дает существенные преимущества, так как в месте возникновения ошибки (серверная функция) часто нет возможности корректно ее обработать, а клиентская функция такими возможностями обычно располагает.
7. Для конструктора класса использование исключений является единственным способом сообщить клиенту об ошибках или сбоях, случившихся в процессе конструирования объекта.
8. Для работы с исключениями необходимо: а) описать контролируемый блок — составной оператор с ключевым словом `try`; б) предусмотреть генерацию одного или нескольких исключений операторами `throw` внутри блока `try` или внутри функций, вызываемых из этого блока; в) разместить сразу за блоком `try` один или несколько обработчиков исключений `catch`.
9. В качестве типов исключений, генерируемых оператором `throw`, целесообразно использовать определенные программистом классы исключений. Более того, эти классы исключений исключительно полезно выстраивать в некоторую иерархию. Это позволяет создавать программы, для которых будущая модификация в связи с обработкой новых типов ошибок оказывается наименее трудоемкой.
10. Никогда не генерируйте исключения в теле деструктора. Если это могут сделать другие вызываемые функции, то примите все меры, чтобы исключение не покинуло данный деструктор.

Задания

Общие указания для всех вариантов

Во всех вариантах требуется создать шаблон некоторого целевого класса *A*, возможно, реализованный с применением некоторого серверного класса *B*. Это означает, что объект класса *B* используется как элемент класса *A*. В качестве серверного класса может быть указан либо класс, созданный программистом¹, либо класс из стандартной библиотеки — например, `std::vector`.

Варианты целевых или серверных классов, создаваемых программистом, приведены в табл. 3.1. Информацию о работе с динамическими структурами данных (линейный список, стек, односторонняя очередь, бинарное дерево) см. в учебнике (с. 114–127), а также в первой книге практикума [2] (семинар 9).

Таблица 3.1. Варианты целевых или серверных классов

Имя класса	Назначение
<code>Vect</code>	одномерный динамический массив
<code>List</code>	двунаправленный список

¹ В рамках этого же задания.

Имя класса	Назначение
Stack	стек
BinaryTree	бинарное дерево
Queue	односторонняя очередь
Deque	двусторонняя очередь (допускает вставку и удаление из обоих концов очереди)
Set	множество (повторяющиеся элементы в множество не записываются; элементы в множестве хранятся отсортированными)
SparseArray	разреженный массив

Если вместо серверного класса указан динамический массив, то это означает, что для хранения элементов контейнерного класса используется массив, размещаемый с помощью операции `new`.

Во всех вариантах необходимо предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций.

Во всех вариантах показать в клиенте `main()` использование созданного класса, включая ситуации, приводящие к генерации исключений. Показать инстанцирование шаблона для типов `int`, `double`, `std::string`.

Варианты заданий приведены в табл. 3.2.

Таблица 3.2. Варианты заданий

Вариант	Целевой шаблонный класс	Реализация с применением
1	Vect	<code>std::list</code>
2	List	—
3	Stack	динамический массив
4	Stack	Vect
5	Stack	List
6	Stack	<code>std::vector</code>
7	Stack	<code>std::list</code>
8	BinaryTree	—
9	Queue	Vect
10	Queue	List
11	Queue	<code>std::list</code>
12	Deque	Vect
13	Deque	List
14	Deque	<code>std::list</code>
15	Set	динамический массив
16	Set	Vect
17	Set	List
18	Set	<code>std::vector</code>
19	Set	<code>std::list</code>
20	SparseArray	List

Семинар 4 Стандартные ПОТОКИ

Теоретический материал: с. 265–280.

Потоковые классы

Программа на С++ представляет ввод и вывод как поток байтов. При вводе она читает байты из потока ввода, при выводе вставляет байты в поток вывода. Понятие потока позволяет абстрагироваться от того, с каким устройством ввода/вывода идет работа. Например, байты потока ввода могут поступать либо с клавиатуры, либо из файла на диске, либо из другой программы. Аналогично, байты потока вывода могут выводиться на экран, в файл на диске, на вход другой программы.

Обычно ввод/вывод осуществляется через буфер — область оперативной памяти, накапливающую какое-то количество байтов, прежде чем они пересылаются по назначению. При обмене, например, с диском это значительно повышает скорость передачи информации. При вводе с клавиатуры буферизация обеспечивает другое удобство для пользователя — возможность исправления ошибок набора символов, пока они не отправлены из буфера в программу.

Программа на С++ обычно очищает буфер ввода при нажатии клавиши Enter. При выводе на экран очистка буфера вывода происходит либо при появлении в выходном потоке символа новой строки '\n', либо при выполнении программой очередной операции ввода с клавиатуры.

Язык С++ предоставляет возможности ввода/вывода как на низком уровне (*неформатированный* ввод/вывод), так и на высоком уровне (*форматированный* ввод/вывод). В первом случае передача информации осуществляется блоками байтов данных без какого-либо их преобразования. Во втором — байты группируются для представления таких элементов данных, как целые числа, числа с плавающей запятой, строки символов и т. д.

Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов — `ios` и `streambuf`. Класс `ios` содержит базовые средства управления потоками, являясь родительским для других классов ввода/вывода. Класс `streambuf` обеспечивает общие средства управления буферами потоков и их взаимодействие с физическими устройствами, являясь родительским для других буферных классов. Связь между этими двумя классами следующая: в классе `ios` есть поле `bp`, являющееся указателем на `streambuf`.

Классы стандартных потоков

Потоки, связанные с консольным вводом/выводом (клавиатура/экран), называются *стандартными*. Стандартному потоку ввода соответствует класс `istream`, стандартному потоку вывода — класс `ostream`. Оба класса являются производными от класса `ios`. Следующим в иерархии является класс `iostream`, наследующий классы `istream` и `ostream` и обеспечивающий общие средства потокового ввода/вывода.

Заголовочные файлы библиотеки ввода/вывода C++

На первом семинаре мы уже давали рекомендацию работать с версией стандартной библиотеки C++, соответствующей стандарту ISO/IEC 14882 (1998), если это позволяет ваш компилятор. Заголовочные файлы при этом указываются без расширения, например `<iostream>`, и кроме того, обычно используется директива `using namespace std;`, так как все имена в стандартной версии библиотеки принадлежат пространству `std`.

Если ваш компилятор не понимает директивы `#include <iostream>`, то это значит, что он работает с более старой версией библиотеки, и вам придется для всех заголовочных файлов указывать расширение `.h`, например: `#include <iostream.h >`¹. К сожалению, при этом не все примеры, приводимые в этой книге, будут у вас компилироваться².

Объекты и методы стандартных потоков ввода/вывода

При наличии директивы `#include <iostream>` в программе автоматически становятся доступными объекты:

- ❑ `cin` — объект класса `istream`, соответствующий стандартному потоку ввода;
- ❑ `cout` — объект класса `ostream`, соответствующий стандартному потоку вывода³.

¹ Директиву `using namespace std;` при этом следует убрать.

² Например, вы не сможете использовать строки класса `string`.

³ На самом деле, становятся доступными еще два объекта класса `ostream`: `clog` — соответствует стандартному буферизованному потоку вывода сообщений об ошибках, `cerr` — соответствует стандартному небуферизованному потоку вывода сообщений об ошибках.

- ❑ Оба потока являются буферизованными. Благодаря объектам `cin` и `cout` становятся доступными и методы соответствующих классов.
- ❑ Форматированный ввод/вывод реализуется через две перегруженные операции: операция сдвига влево '`<<`' перегружена для вывода в поток и называется *операцией вставки (помещения, включения)* в поток¹; операция сдвига вправо '`>>`' перегружена для ввода из потока и называется *операцией извлечения* из потока.
- ❑ Например, в классе `ostream` определены операции

```
ostream& operator<<(short);
ostream& operator<<(int);
ostream& operator<<(double);
// и так далее . . .
```

Рассмотрим, что произойдет, если в программе встретятся операторы

```
int x = 5;
cout << x;
```

Сначала компилятор определит, что левый аргумент имеет тип `ostream`, а правый — тип `int`. Вооруженный этими знаниями, он найдет прототип метода `ostream& operator<<(int)` в заголовочном файле `ostream`. В конечном итоге будет вызван метод `cout.operator<<(x)`, в результате работы которого мы увидим на экране число 5.

Заметим, что в случае вывода значений встроенных типов, таких как `int` или `double`, класс `ostream` обеспечивает их преобразование из внутреннего двоичного представления к изображению в виде строки символов. Вы можете не заботиться о форматировании, так как класс `ostream` поддерживает форматирование вывода по умолчанию, задаваемое соответствующими полями базового класса `ios`.

Аналогично происходит работа со стандартным потоком ввода через объект `cin`, надо только учесть направление движения информации: из потока — в программу.

Если форматирование по умолчанию вас не устраивает, можно воспользоваться либо соответствующими методами класса `ios`, либо так называемыми *манипуляторами* ввода/вывода. Последние представляют собой функции, которые можно включать прямо в поток. Они определены частично в файлах `istream` и `ostream` и частично в файле `iomanip`.

В табл. 4.1 приведены наиболее часто употребляемые манипуляторы².

Таблица 4.1. Основные манипуляторы и методы управления форматом

Манипулятор	Метод класса <code>ios</code>	Ввод	Вывод	Описание
<code>endl</code>	—		+	Включить в поток символ новой строки и выгрузить буфер
<code>dec</code>	<code>flags(ios::dec)</code>	+	+	Перейти в десятичную систему счисления

продолжение ↗

¹ Ранее мы использовали термин «операция помещения в поток», однако более короткий термин-синоним «операция вставки» не менее популярен в литературе по C++.

² Полный перечень манипуляторов и методов управления форматом см. в учебнике на с. 271–273.

Таблица 4.1 (продолжение)

Манипулятор	Метод класса ios	Ввод	Вывод	Описание
hex	flags(ios::hex)	+	+	Перейти в шестнадцатеричную систему счисления
oct	flags(ios::oct)	+	+	Перейти в восьмеричную систему счисления
setprecision(n)	precision(n)		+	Установить количество отображаемых знаков
setw(n)	width(n)		+	Установить ширину поля вывода
setfill(c)	fill(c)		+	Установить символ заполнения с

Например, вывести значение целой переменной x в шестнадцатеричной форме можно двумя способами:

```
cout.flags(ios::hex); cout << x;
```

или

```
cout << hex << x;
```

Очевидно, что код с использованием манипулятора в данном случае более лаконичен и выразителен.

Остановимся немного подробнее на манипуляторе `setprecision(n)` и соответствующем методе `precision(n)`, поскольку их воздействие на формат вывода подчиняется довольно сложным правилам. И тот, и другой изменяют значение поля `ios::x_precision` (по умолчанию это значение равно шести). Данное поле управляет точностью вывода вещественных чисел, причем его интерпретация зависит от значений других полей, управляющих форматом вывода:

- ❑ `ios::scientific` («научный» формат, или формат вывода с плавающей точкой);
- ❑ `ios::fixed` (формат вывода с фиксированной точкой).

Если установлен хотя бы один из этих форматов (например, с помощью метода `flags()`), то значение `x_precision` задает количество цифр *после десятичной точки*. Если не установлен ни тот, ни другой, и вывод идет в так называемом *автоматическом формате*, то значение `x_precision` задает *общее количество значащих цифр*.

Учтите, что большинство параметров форматирования сохраняют свое состояние вплоть до следующего вызова функции, изменяющей это состояние. Исключение — манипулятор `setw(n)` и соответствующий ему метод `width(n)`: их действие распространяется только на ближайшую операцию вывода, после чего восстанавливается ширина поля вывода по умолчанию.

По признаку наличия аргумента манипуляторы подразделяются на *простые* (без аргумента) и *параметризованные* (с аргументом). Для использования параметризованных манипуляторов необходимо подключить заголовочный файл `iomanip`.

С точки зрения *реализации* манипуляторы можно разделить на три группы:

- 1) манипуляторы без аргумента, выводящие в поток управляющий символ (`endl`, `ends`) или очищающие буфер потока (`flush`);

- 2) манипуляторы без аргумента, изменяющие значения тех полей базового класса `ios`, которые задают текущую систему счисления (`dec`, `hex`, `oct`);
- 3) манипуляторы с аргументом.

С техническими вопросами реализации манипуляторов мы столкнемся чуть позже, решая задачу 4.1.

Кроме рассмотренных, класс `ios` предоставляет и другие методы, обеспечивающие неформатированный ввод/вывод, а также связь с буфером потока. Наиболее часто употребляемые методы ввода/вывода приведены в табл. 4.2¹.

Таблица 4.2. Некоторые методы ввода/вывода класса `ios`

Метод	Описание
<code>get()</code>	Возвращает код извлеченного из потока символа или EOF, если достигнут конец файла
<code>get(c)</code>	Присваивает код извлеченного из потока символа аргументу <code>c</code>
<code>get(buf, num, lim='\\n')</code>	Читает из потока символы, пока не встретится символ <code>lim</code> (по умолчанию <code>'\\n'</code>), или пока не будет прочитано <code>num-1</code> символов. Извлеченные символы размещаются в символьный массив <code>buf</code> , после чего к ним добавляется нулевой байт. Символ <code>lim</code> остается в потоке
<code>getline(buf, num, lim='\\n')</code>	Выполняется аналогично предыдущему методу, но символ <code>lim</code> удаляется из потока (в массив <code>buf</code> он также не записывается)
<code>peek()</code>	Возвращает код следующего (готового для извлечения) символа в потоке, но не извлекает данный символ; или EOF, если достигнут конец файла
<code>read(buf, num)</code>	Считывает <code>num</code> символов из потока в символьный массив <code>buf</code>
<code>gcount()</code>	Возвращает количество символов, прочитанных последним вызовом функции неформатированного ввода
<code>rdbuf()</code>	Возвращает указатель на буфер типа <code>streambuf</code> , связанный с данным потоком
<code>put(c)</code>	Выводит в поток символ <code>c</code>
<code>write(buf, num)</code>	Выводит в поток <code>num</code> символов из массива <code>buf</code>

Обработка ошибок потоков

Библиотека ввода/вывода C++ обеспечивает гораздо более надежный ввод/вывод, чем старые функции библиотеки C. Это достигается перегрузкой операций извлечения и вставки для всех встроенных типов, что исключает путаницу с типами, которая была возможна при использовании функций `scanf()` и `printf()` и которая приносила массу неприятностей программистам. Ведь компилятор никак не реагировал на ошибки в спецификации формата, поэтому на этапе выполнения в самый неподходящий момент могли вылезти какие-то странные результаты. Теперь это исключено.

¹ Более подробную информацию см. в учебнике на с. 273.

И все же проблема ошибок нас практически не волнует только при выводе информации в поток `cout`. При вводе данных никто не может запретить пользователю ввести вместо ожидаемого программой целого числа какую-то произвольную строку символов, и если не принять специальных мер, то программа «сломается». Для отслеживания ошибок потоков в базовом классе `ios` определено поле `state`, отдельные биты (флаги) которого отображают состояние потока, как показано в табл. 4.3.

Таблица 4.3. Флаги состояния потока

Имя флага	Интерпретация
<code>ios::goodbit</code>	Нет ошибок
<code>ios::eofbit</code>	Достигнут конец файла
<code>ios::failbit</code>	Ошибка форматирования или преобразования
<code>ios::badbit</code>	Серьезная ошибка, после которой пользоваться потоком невозможно

Получить текущее состояние потока можно с помощью метода `rdstate()`, который возвращает значение типа `int`. Есть и другие методы, более удобные для анализа состояния потока:

`int eof()` — возвращает ненулевое значение, если установлен флаг `eofbit`;

`int fail()` — возвращает ненулевое значение, если случилась любая ошибка ввода/вывода (но не конец файла). Этому условию соответствует установка либо флага `badbit`, либо флага `failbit`;

`int bad()` — возвращает ненулевое значение, если случилась серьезная ошибка ввода/вывода (то есть установлен флаг `badbit`);

`int good()` — возвращает ненулевое значение, если сброшены все флаги ошибок.

Если произошла ошибка ввода, в результате которой установлен только флаг `failbit`, то после этого можно продолжать работу с потоком, но сначала нужно сбросить все флаги ошибок, для чего предназначен метод `void clear(int = 0)`. Для сброса флагов достаточно сделать вызов `clear()`, так как аргумент по умолчанию равен нулю. Этот же метод можно использовать для установки соответствующих флагов поля `state`, если передать ему ненулевой аргумент¹.

Есть и другие приемы диагностирования ошибочных ситуаций. Мы используем их при решении задачи 4.2.

Перегрузка операций извлечения и вставки для типов, определенных программистом

Одно из удобств C++ — возможность перегрузить операции извлечения и вставки для любого класса `MyClass`, созданного программистом. После этого для любого объекта `obj` класса `MyClass` можно записывать операторы вида `cin >> obj; cout << obj`. Удобно, не правда ли?

¹ Обычно это комбинация флагов из табл. 4.3, объединенных операцией `|` (ИЛИ).

Однако операции `>>` и `<<` не могут быть элементами класса `MyClass`. Причина в том, что у любой функции-операции класса левым операндом подразумевается объект этого класса, вызывающий данную функцию. Но для операций извлечения/вставки левый операнд должен быть потоком ввода/вывода. Поэтому эти функции всегда объявляются *дружественными* классу, для которого они создаются.

Общая форма пользовательской *функции извлечения*:

```
istream& operator >>(istream& is, MyClass& obj)
```

где `is` — ссылка на входной поток, а `obj` — ссылка на объект, к которому применяется операция. В теле функции последний оператор должен возвращать объект `is`.

Общая форма пользовательской *функции вставки*:

```
ostream& operator <<(ostream& os, MyClass& obj)
```

где `os` — ссылка на выходной поток, а `obj` — ссылка на объект, к которому применяется операция. В теле функции последний оператор должен возвращать объект `os`. В принципе, возможна и другая сигнатура операции вставки с передачей второго аргумента по значению (`MyClass obj`). Но вы должны понимать, что в этом случае, во-первых, при вызове функции в стеке будет создаваться копия объекта `obj`, а во-вторых, в классе `MyClass` должен быть предусмотрен конструктор копирования.

Примеры перегрузки операций `>>` и `<<` будут рассмотрены при решении задачи 4.2.

Перейдем теперь к решению задач.

О проблеме ввода/вывода кириллицы для консольных приложений в среде Visual C++ 6.0 мы говорили на семинаре 1. Там же было сказано о новом решении проблемы посредством использования разработанных для этого классов `CyrIstream` и `CyrOstream`. Процесс разработки излагается в виде решения следующей задачи.

Задача 4.1. Разработка потоковых классов, поддерживающих ввод/вывод кириллицы¹

Разработать два класса: `CyrIstream` и `CyrOstream`, объектами которых `Cin` и `Cout` можно подменять (путем макроподстановок) стандартные объекты `cin` и `cout`. Классы `CyrIstream` и `CyrOstream` должны обладать всеми свойствами стандартных классов `istream` и `ostream` за счет наследования. Дополнительно класс `CyrIstream` должен обеспечивать при вводе строк преобразование последних из кодировки ASCII в кодировку ANSI, а класс `CyrOstream` при выводе строк — обратное преобразование.

Интерфейс указанных классов должен размещаться в файле `CyrIOS.h`, реализация — в файле `CyrIOS.cpp`.

Использование классов:

□ подключить к проекту файлы `CyrIOS.h`, `CyrIOS.cpp`;

¹ Для консольных Windows-приложений.

□ в каждом файле проекта, использующем объекты *cin* и/или *cout* для ввода/вывода русского язычного текста, добавить директиву `#include «CyrIOS.h»`.

На первый взгляд задача не очень сложная: ну что стоит перегрузить, а точнее, *заместить*, пару операторов, отвечающих за ввод/вывод строк? Однако в процессе разработки указанных классов у нас будет много поводов убедиться в обратном. Поэтому мы решили показать вам весь процесс решения шаг за шагом, подробно разбирая возникающие проблемы и пути их разрешения.

ПРИМЕЧАНИЕ

В процессе решения мы будем неоднократно сталкиваться с сообщениями об ошибках как на стадии компиляции, так и на стадии выполнения. Кроме этого, нам понадобится помощь отладчика для пошагового выполнения программы. Все это, к сожалению, «завязано» на конкретную интегрированную среду, в которой ведется разработка. В данном случае изложение ведется применительно к IDE Microsoft Visual Studio 6.0 (Microsoft Visual C++ 6.0). Тем не менее подобная разработка может быть проведена в любой среде с соответствующими компилятором и отладчиком C++, так как и семантика сообщений и действия с отладчиком будут, скорее всего, аналогичны тем, которые мы приводим.

Начнем с разработки класса `CyrOstream`. Каким должен быть интерфейс класса? Из постановки задачи вытекает, что класс должен содержать, как минимум, функцию-операцию вставки для вывода строки, которая бы замещала операцию с аналогичной сигнатурой — `ostream& operator <<(const char* s)`: — в базовом классе `ostream`. Для начала было бы неплохо взглянуть на определение базового класса. Но как найти файл с этим определением? Есть два пути. Первый — пользуясь утилитой поиска файлов, имеющейся в каждой операционной системе, найти файл с именем `ostream`. Затем можно насладиться чтением более чем четырехсот строк исходного кода. Второй путь — использование отладчика. В ряде случаев он не только более эффективен, но и позволяет попутно определить, какие методы класса вызываются для реализации интересующих нас действий. Такое экспериментальное исследование поведения класса окажется очень важным для нас, так как иногда это — единственный способ определить сигнатуры тех методов, которые требуется заместить в производном классе. Сейчас мы покажем, как это сделать.

ПРИМЕЧАНИЕ

Работа с отладчиком в интегрированной среде Microsoft Visual C++ 6.0 описана в первой книге практикума. Напомним кратко назначение основных команд отладчика: `Insert/Remove Breakpoint` — установить/снять точку останова¹; `Go` — выполнить программу до следующей точки останова; `Step Over` — выполнить в пошаговом режиме следующий оператор программы (если это вызов функции, то выполнить функцию без трассировки ее тела); `Step Into` — выполнить в пошаговом режиме следующий оператор программы (если это вызов функции, то зайти в тело функции для последующей трассировки ее тела); `Stop Debugging` — прервать работу отладчика.

¹ Наименования команд здесь даны для IDE Visual C++ 6.0, однако в любой другой среде вы без труда найдете команды аналогичного назначения.

Создайте проект SearchIOS, добавьте в него файл SearchIOS.cpp и введите следующий текст:

```

////////////////////////////////////
// SearchIOS.cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Добро пожаловать в C++!";           // 1
    cout << endl;                                 // 2
    return 0;
}

```

Откомпилируйте проект.

Установите точку останова перед оператором 1 и запустите отладчик командой Go. Программа стартует и остановится в указанной точке.

Дайте команду STEP INTO. Отладчик — вот умница! — откроет файл ostream и остановится перед входом в тело функции

```

template<class _E, class _Tr> inline basic_ostream<_E, _Tr>& __cdecl operator<<
(basic_ostream<_E, _Tr>& _O, const _E *_X) { ... }

```

Но что это? Сигнатура найденной функции никак не похожа на ожидаемую ostream& operator <<(const char* s)! Вот здесь нам и пригодятся знания по теме «Шаблоны классов», добытые тяжким трудом на предыдущем семинаре! Видите знакомое словечко template? Как выясняется, класс ostream является инстанцированием (для конкретных видов символов) универсального шаблонного класса basic_ostream, определение которого и содержится в файле ostream!

Дальнейшее изучение текста файла прояснит ситуацию: операции вставки для встроенных числовых типов (short, int и т. д.) определены как методы класса, а операции вставки для символов и символьных строк — как внешние функции-шаблоны. Отладчик как раз и остановился перед входом в тело такой функции-шаблона.

Мы надеемся, что все составные элементы заголовка этой функции вам понятны, за исключением, может быть, спецификатора __cdecl, который характерен для реализации Microsoft и устанавливает соглашения по умолчанию для механизма вызова функций C и C++. А где же содержится собственно актуализация шаблона basic_ostream, порождающая класс ostream? Методом поиска файлов, содержащих заданный текст, можно найти определение

```
typedef basic_ostream<char, char_traits<char> > ostream;
```

которое находится в файле iosfwd.

Продолжим выполнение программы. Пройдите через тело функции последовательным выполнением команды Step Over. Отладчик остановится перед оператором 2 функции main(). После команды Step Into отладчик опять нырнет в файл ostream и остановится перед входом в тело функции

```
_Myt& operator<<(_Myt& (__cdecl *_F)(_Myt&)) { return ((*_F)(*this)); }
```


Обратим внимание на то, что значением аргумента `_F` в этот момент является адрес функции-шаблона `endl()`, которая определена тоже в файле `ostream`. А что такое `_Myt`? Это имя определено почти в самом начале файла оператором

```
typedef basic_ostream<_E, _Tr> _Myt;
```

Продолжим трассировку. После команды `Step Over` отладчик остановится уже перед оператором `return 0;`. В этот момент в консольном окне программы вы увидите что-нибудь вроде следующего:

```
-юсЁю яюцрюютрС№ т С++!
```

Все правильно. Для нормального вывода кириллицы мы еще ничего не сделали. Но зато заглянули на «кухню» объекта `cout` и теперь кое-что знаем о его реализации. Закончим отладку программы командой `Stop Debugging`.

Вернемся к вопросу о составе класса `CyrOstream`. Надо ли предусмотреть в нем конструктор? Для производных классов этот вопрос более сложен, чем для классов, которые ничего не наследуют. Если мы оставим класс без конструктора, то компилятор создаст его автоматически. Такой конструктор будет вызывать конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов. Значит, надо выяснить, есть ли в классе `basic_ostream` конструктор по умолчанию? Заглянем опять в файл `ostream`. Почти в самом начале определения класса мы найдем два конструктора:

```
explicit basic_ostream(basic_streambuf<_E, _Tr> *_S, bool _Isstd = false,
bool _Doinit = true) { ... }
basic_ostream(_Uninitialized) { ... }
```

Ни первый, ни второй не могут быть вызваны без параметров. Значит, нам не повезло, и класс `CyrOstream` должен содержать конструктор, который обеспечит передачу необходимых аргументов конструктору базового класса. Да, но которому из них — первому или второму?.. Очевидно, тому, который вызывается при создании объекта `cout`! А где определены объекты `cout` и `cout?` — Правильно, в файле `iostream.cpp`. Открыв файл, без особого труда находим объявления:

```
_CRTIMP2 istream cin(_Noinit);
_CRTIMP2 ostream cout(_Noinit);
```

Не обращайте внимания на странные спецификаторы `_CRTIMP2` — это всего лишь макросы¹, определяющие класс памяти для расширений C++, принятых Microsoft². Большого интереса заслуживает аргумент `_Noinit`. К какому типу он относится? Скорее всего, это какая-нибудь константа. Прибегнув к средствам поиска, находим в файле `xstdddef` объявление перечисляемого типа:

```
enum _Uninitialized {_Noinit};
```

Таким образом, константа `_Noinit` имеет тип `_Uninitialized`. Следовательно, при создании объекта `cout` вызывается *второй* из указанных выше конструкторов

¹ Определение макроса: `#define _CRTIMP2 __declspec(dllexport)` содержится в файле `xstdddef`.

² К стандарту C++ эти расширения никакого отношения не имеют и к рассматриваемой нами проблеме — тоже.

класса `basic_ostream`. Теперь мы знаем, *что* нужно передать конструктору базового класса.

Что же касается метода вставки для вывода строки типа `const char*`, с которого все и началось, то, к счастью, замещающий метод не обязан быть функцией-шаблоном, так что его сигнатура вполне ясна.

Итак, мы готовы предложить первую версию решения задачи. Создайте проект `Task4_1a` из трех файлов со следующим содержанием:

```

////////////////////////////////////
// Проект Task4_1a
////////////////////////////////////
// CyrIOS.h
#ifndef CYR_IOS_H
#define CYR_IOS_H

#include <iostream>
#include <iomanip>
#include <string>
#include <windows.h>
using namespace std;

#define MAX_STR_LEN 4096

// Класс CyrOstream
class CyrOstream : public ostream {
public:
    CyrOstream(_Uninitialized no_init) : ostream(no_init) {}
    CyrOstream& operator <<(const char*);

private:
    char buf_[MAX_STR_LEN];
};

extern CyrOstream Cout;

#endif /* CYR_IOS_H */

#ifndef CYR_IOS_IMPLEMENTATION // 1
#define cout Cout // 2
#endif // 3
////////////////////////////////////
// CyrIOS.cpp
#define CYR_IOS_IMPLEMENTATION // 4
#include "CyrIOS.h"
// Knacc CyrOstream
CyrOstream& CyrOstream::operator <<(const char* s) {
    int n = strlen(s);
    strncpy(buf_, s, n); buf_[n] = 0;
}

```

```

CharToOem(buf_, buf_);
cout << buf_;
return *this;
}

// Глобальный объект для вывода
CyrOstream Cout(_Noinit); // 5
////////////////////////////////////
// Task4_1a.cpp
#include "CyrIOS.h"

int main() {
    cout << "Добро пожаловать в C++!"; // 6
    cout << endl; // 7
    return 0;
}
//————— конец проекта Task4_1a —————
////////////////////////////////////

```

Сделаем необходимые пояснения.

Файл `CyrIOS.h`.

- Обратите внимание на передачу аргумента `no_init` конструктору базового класса в списке инициализаторов конструктора класса `CyrOstream`.
- В секции `private` класса объявлен символьный массив `buf_` размером `MAX_STR_LEN`, который будет использоваться для временного хранения преобразуемых строк.
- Операторы 1, 2, 3 обеспечивают макроподстановку имени `Cout` вместо имени `cout` только в тех файлах, в которых не определен макрос `CYR_IOS_IMPLEMENTATION`. Чуть ниже (строка 4) вы увидите, что этот макрос определен только в файле `CyrIOS.cpp`. Таким образом, в файле `CyrIOS.cpp` будет использоваться стандартный объект `cout`, а во всех остальных вместо него — объект `Cout`.

Файл `CyrIOS.cpp`.

- Алгоритм операции вставки для аргумента `const char* s` предельно прост. Функция `CharToOem` преобразует символы в буфере `buf_` из кодировки ANSI в кодировку ASCII, после чего преобразованная строка выводится в `cout`.
- В строке 4 определяется глобальный объект `Cout`.

Попробуем откомпилировать проект. Нас подстерегает неудача. Компилятор выругается чем-нибудь вроде

```
error: binary '<<' : no operator defined which takes a right-hand operand of type ''
```

указав в качестве «плохой» строки кода оператор 7. То есть, по мнению компилятора, отсутствует перегруженная функция-операция '<<' для манипулятора `endl`.

Но позвольте! Почему же здесь не вызывается метод базового класса `_Myt& operator<<(_Myt& (__cdecl * _F)(_Myt&))`, как это происходило в проекте `SearchIOS`? Здесь мы подошли к одному тонкому вопросу, освещаемому далеко не в каждом учебнике по C++.

Проблема состоит в следующем. Если в базовом классе некий метод перегружен более чем с одной сигнатурой, а в производном классе вы замещаете только одну версию этого метода, то для объекта производного класса окажутся *скрытыми* (*недоступными*) не только замещенная версия метода в базовом классе, но и *все* остальные версии базового класса!

ВНИМАНИЕ

Если возникла необходимость заместить в производном классе одну из версий перегруженного метода базового класса, это автоматически влечет за собой требование замещения всех остальных версий данного метода!

Таким образом, все версии функции-операции `<<`, имеющиеся в базовом классе `ostream`, должны быть переопределены в классе `CyrOstream`.

Давайте исправим допущенные недоработки. Добавьте в файл `CyrIOS.h` объявление метода:

```
CyrOstream& operator <<(_Myt& (__cdecl * _f)(_Myt&));
```

который, как мы выяснили в экспериментах с проектом `SearchIOS`, вызывается для манипулятора `endl`. Добавьте также определения еще двух методов:

```
CyrOstream& operator <<(int n) { cout << n; return *this; }  
CyrOstream& operator <<(double f) { cout << f; return *this; }
```

Заметим, что мы добавляем операцию вставки пока только для типов `int` и `double`, имея в виду, что для остальных встроенных типов аналогичные функции появятся в заключительном тексте проекта. Добавьте в файл `CyrIOS.cpp` реализацию метода:

```
CyrOstream& CyrOstream::operator <<(_Myt& (__cdecl * _f)(_Myt&)) {  
    cout << _f; return *this;  
}
```

Смысл переопределения (замещения) этих методов в производном классе заключается в том, что значение аргумента функции-операции выводится в стандартный поток `cout`.

И, наконец, добавьте в функцию `main()` следующие две строчки:

```
double y = 372.141526;  
cout << y << endl;
```

После компиляции (теперь она должна быть успешной) и запуска программа должна вывести на экран:

```
Добро пожаловать в C++!  
372.142
```

Вещественное число выводится здесь в соответствии с точностью, заданной по умолчанию¹. Результат совпадает с ожидаемым.

¹ `ios::x_precision` по умолчанию равно шести.

Продолжим тестирование. Так как по критерию реализации манипуляторы делятся на три группы (см. текст выше), давайте проверим работу представителя второй группы — манипулятора `hex`.

Удалите из функции `main()` две последние добавленные строки, заменив их на следующие пять строк:

```
int x = 1024;
cout << "x = " << x << endl;
cout << "x(hex) = ";
cout << hex;
cout << x << endl;
```

Попытка откомпилировать программу будет неудачной, причем со знакомым уже сообщением об ошибке:

```
error: binary '<<' : no operator defined which takes a right-hand operand of type '',
адресованным оператору cout << hex.
```

Но мы уже не «чайники», мгновенно ставим диагноз: некий метод базового класса `ostream`, отвечающий за работу данного манипулятора, оказался *не* замещенным в нашем производном классе. Для выяснения, какой это метод, воспользуемся уже отработанной технологией с применением отладчика. Надеемся, что вы еще не удалили с диска наш основной исследовательский инструмент — проект `SearchIOS`? Вот и чудесно — добавьте в функцию `main()` те же пять строк кода, установите точку останова перед оператором `cout << hex`, и в процессе трассировки нужный метод будет найден практически сразу:

```
_Myt& operator<<(ios_base& (__cdecl * _F)(ios_base&)) { ... }
```

Вернемся к проекту `Task4_1a` для внесения требуемых изменений.

Добавьте в определение класса `CyrOstream` (файл `CyrIOS.h`) объявление метода:

```
CyrOstream& operator <<(ios_base& (__cdecl * _f)(ios_base& ));
```

а в файл `CyrIOS.cpp` — реализацию метода:

```
CyrOstream& CyrOstream::operator <<(
ios_base& (__cdecl * _f)(ios_base& )) {
    cout << _f;    return *this;
}
```

Теперь программа должна выдать результат:

```
Добро пожаловать в C++!
x = 1024
x(hex) = 400
```

Отметив очередной успех разработки (чашечкой кофе), продолжим тестирование. Пришло время проверить работу манипуляторов третьей группы — параметризованных. Давайте возьмем наиболее часто используемый манипулятор этой группы: `setw(int)`, который задает максимальную ширину поля вывода. Замените в функции `main()` последние пять строк следующим кодом:

```
double y = 372.141526;
cout << setw(30) << y << endl;
```

После запуска на выполнение программа выведет:

```
Добро пожаловать в C++!
```

после чего она благополучно «заваливается», сообщая об ошибке типа «Access Violation»¹.

Попробуем локализовать место ошибки, разбив сложное выражение вывода на два изолированных звена:

```
cout << setw(30);
cout << y << endl;
```

В этом случае программа выведет:

```
Добро пожаловать в C++!
372.142
```

Результат более приятный: программа не ломается, — но все-таки неверный, так как манипулятор `setw(30)` свою задачу не выполнил: ширина поля вывода значительно меньше тридцати!

Итак, понятно, что параметризованные манипуляторы не «дружат» с нашим классом `CyrOstream`. Осталось выяснить — почему?

Вспомним, что параметризованные манипуляторы изменяют значения соответствующих полей базового класса `ios` у объекта `cout`. Но! Любезный читатель, вы не забыли, что в нашей программе в файле `CyrIOS.h` определен макрос, благодаря которому все имена `cout` заменяются на `Cout`? Так что если, к примеру, не перехватить операцию `<<` для манипулятора `setw(30)`, то результатом его работы будет новое значение поля `x_width` базового класса `ios` объекта `Cout`!.. А в то же время во всех перегруженных методах `<<` класса `CyrOstream` операция вставки *перенаправляется* в стандартный объект `cout`.

Очевидно, что для исправления ситуации *действие* параметризованных манипуляторов необходимо *перенаправить в стандартный объект* `cout` так же, как мы это делали, например, с манипулятором `endl`. Однако для параметризованных манипуляторов ситуация оказывается более сложной. Дело в том, что их определения и реализация содержатся в файлах `io manip` и `io manip.cpp`. Чтобы все-таки понять, как они работают, опять призовем на помощь отладчик, вернувшись к нашему исследовательскому проекту `SearchIOS`.

Добавьте в функцию `main()` (файл `SearchIOS.cpp`) следующие две строки:

```
cout << setw(30);
cout << y << endl;
```

Установите точку останова перед оператором

```
cout << setw(30);
```

¹ Нарушение доступа (к памяти).

После запуска отладчика выполняйте пошаговую трассировку до тех пор, пока не встретится вызов метода с сигнатурой

```
basic_ostream<...>& operator<<( ... )
```

И действительно, на одном из шагов вы увидите вызов функции-шаблона

```
template<class _E, class _Tr, class _Tm> inline basic_ostream<_E, _Tr>& __cdecl
operator<<(basic_ostream<_E, _Tr>& _O, const _Smanip<_Tm>& _M) { ... }
```

Это как раз то, что нужно, чтобы определиться с сигнатурой замещающего метода! Обратите внимание на то, что найденная функция-операция имеет *два* аргумента, то есть она не является членом класса.

Вернемся к проекту Task4_1a.

Добавьте в определение класса CyrOstream (файл CyrIOS.h) объявление следующей дружественной функции:

```
friend CyrOstream& operator <<(CyrOstream&, const _Smanip<int>&);
```

Добавьте в файл CyrIOS.cpp реализацию этой функции, которая перенаправляет действие параметризованного манипулятора в стандартный объект cout:

```
CyrOstream& operator <<(CyrOstream& os,
const _Smanip<int>& m) {
    cout << m;    return os;
}
```

После компиляции программы и запуска вы должны увидеть следующий результат:

```
Добро пожаловать в C++!
```

```
372.142
```

```
Press any key for continue.
```

То, что надо! Поздравляем!

Но не будем расслабляться. Ведь надо не забыть, что класс ostream обеспечивает также вывод в стандартный поток значений объектов класса string. Пользуясь уже проверенной методикой и проверенным инструментом (проект SearchIOS), вы легко установите, что вывод происходит благодаря вызову шаблонной функции-операции

```
template<class _E, class _Tr, class _A> inline
basic_ostream<_E, _Tr>& __cdecl operator<<(basic_ostream<_E, _Tr>& _O, const
basic_string<_E, _Tr, _A>& _X) { ... }.
```

определенной в файле string.

Чтобы обеспечить замещение в нашем классе операции вывода значений типа string, нужно предусмотреть в файле CyrOstream.h определение шаблонной функции-операции

```
template<class _E, class _Tr, class _A> inline
CyrOstream& operator <<(CyrOstream& os, const basic_string<_E, _Tr, _A>& _X) { ... }
```

Тело функции будет показано в заключительной версии проекта.

Вспомним также, что в базовом классе ios класса ostream есть еще и другие методы, которые, безусловно, должны быть переопределены в классе CyrOstream. Однако мы создаем не коммерческий продукт, а учебную программу, поэтому в приводимой ниже версии ограничимся замещением только наиболее употребляемых методов указанных классов.

Нам надо побеспокоиться еще об одном: о концептуальном единстве средств ввода/вывода, имея в виду возможность перегрузки операции вставки для типов данных, определяемых пользователем. Напомним, что сигнатура перегружаемой операции имеет вид:

```
friend ostream& operator << (ostream& out, MyClass& ob):
```

Поскольку основной объект вывода Cout имеет тип CyrOstream, то и при перегрузке операции '<<' для некоторого класса MyClass необходимо тип ostream заменить на CyrOstream. Проще всего эта проблема решается добавлением директивы #define ostream CyrOstream в составе блока:

```
#ifndef CYR_IOS_IMPLEMENTATION
#define cout Cout
#define ostream CyrOstream
#endif
```

Разработка класса CyrOstream завершена. Его окончательный вид представлен ниже в версии проекта Task4_1b. Для класса CyrIstream почти все решения «зеркально симметричные».

```
////////////////////////////////////
// Проект Task4_1b
////////////////////////////////////
// CyrIOS.h
#ifndef CYR_IOS_H
#define CYR_IOS_H

#include <iostream>
#include <iomanip>
#include <string>
#include <windows.h>
using namespace std;

#define MAX_STR_LEN 4096
////////////////////////////////////
// Класс CyrOstream
class CyrOstream : public ostream {
public:
    CyrOstream(_Uninitialized no_init) : ostream(no_init) {}
    CyrOstream& operator <<(_Myt& (__cdecl * _f)(_Myt&));
    CyrOstream& operator <<(ios_base& (__cdecl * _f)(ios_base& ));

    CyrOstream& operator <<(short n) { cout << n; return *this; }
    CyrOstream& operator <<(unsigned short n) {
        cout << n; return *this; }
```



```

CyrOstream& operator <<(int n) { cout << n; return *this; }
CyrOstream& operator <<(unsigned int n) {
    cout << n; return *this; }
CyrOstream& operator <<(long n) { cout << n; return *this; }
CyrOstream& operator <<(unsigned long n) {
    cout << n; return *this; }
CyrOstream& operator <<(float f) { cout << f; return *this; }
CyrOstream& operator <<(double f) { cout << f; return *this; }
CyrOstream& operator <<(long double f) {
    cout << f; return *this; }
CyrOstream& operator <<(const void* v) {
    cout << v; return *this; }

CyrOstream& operator <<(const char*);
CyrOstream& operator <<(const unsigned char* s) {
    operator <<((const char*)s); return *this; }
CyrOstream& operator <<(const signed char* s) {
    operator <<((const char*)s); return *this; }

CyrOstream& operator <<(char);
CyrOstream& operator <<(unsigned char);
CyrOstream& operator <<(signed char c) {
    operator <<((char)c); return *this; }

CyrOstream& put(char);
CyrOstream& write(const char*, int);
CyrOstream& write(const unsigned char* s, int len) {
    write((const char*)s, len); return *this; }

// Замещение методов класса ios
long setf(long lFlags) { return cout.setf(lFlags); }
void unsetf(long lFlags) { cout.unsetf(lFlags); }
char fill(char cFill) { return cout.fill(cFill); }
char fill() { return cout.fill(); }
int precision(int np) { return cout.precision(np); }
int precision() const { return cout.precision(); }
int width(int nw) { return cout.width(nw); }
int width() const { return cout.width(); }
int rdstate() const { return cout.rdstate(); }
long flags() const { return cout.flags(); }
long flags(long _l) { return cout.flags(_l); }
streambuf* rdbuf() const { return cout.rdbuf(); }

// Дружественная функция для поддержки параметризованных манипуляторов
friend CyrOstream& operator <<(CyrOstream&,
const _Smanip<int>&);
private:
    char buf_[MAX_STR-LEN];
};
////////////////////////////////////
// Шаблон для вывода типа string
template<class _E, class _Tr, class _A> inline

```

```

CyrOstream& operator <<(CyrOstream& os,
const basic_string<_E, _Tr, _A>& _X) {
    string temp(_X);
    unsigned char symb[2];
    symb[1] = 0;

    for (int i = 0; i < temp.size(); i++) {
        symb[0] = temp.at(i);
        if (symb[0] > 191)
            CharToOem((const char*)symb, (char*)symb);
        cout << symb;
    }
    return os;
}
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Класс CyrIstream
class CyrIstream : public istream {
public:
    CyrIstream(_Uninitialized no_init) : istream(no_init) {}
    CyrIstream& operator >>(ios_base& (_cdecl * _f)(ios_base& ));

    CyrIstream& operator >>(char*);
    CyrIstream& operator >>(unsigned char* s) {
        operator >>((char*)s); return *this; }
    CyrIstream& operator >>(signed char* s) {
        operator >>((char*)s); return *this; }

    CyrIstream& operator >>(char& c);
    CyrIstream& operator >>(unsigned char& c) {
        operator >>((char&)c); return *this; }
    CyrIstream& operator >>(signed char& c) {
        operator >>((char&)c); return *this; }

    CyrIstream& operator >>(short& n) { cin >> n; return *this; }
    CyrIstream& operator >>(unsigned short& n) {
        cin >> n; return *this; }
    CyrIstream& operator >>(int& n) { cin >> n; return *this; }
    CyrIstream& operator >>(unsigned int& n) {
        cin >> n; return *this; }
    CyrIstream& operator >>(long& n) { cin >> n; return *this; }
    CyrIstream& operator >>(unsigned long& n) {
        cin >> n; return *this; }
    CyrIstream& operator >>(float& f) { cin >> f; return *this; }
    CyrIstream& operator >>(double& f) { cin >> f; return *this; }
    CyrIstream& operator >>(long double& f) {
        cin >> f; return *this; }

    int get() { return cin.get(); }
    CyrIstream& get(char&);

```

```

CyrIstream& get(char*, int, char = '\n');
CyrIstream& get(unsigned char*, int, char = '\n');
CyrIstream& getline(char*, int, char = '\n');
CyrIstream& getline(unsigned char* pch, int nCount,
    char delim = '\n') {
    getline((char*)pch, nCount, delim); return *this; }
CyrIstream& read(char*, int);
CyrIstream& read(unsigned char* pch, int nCount) {
    read((char*)pch, nCount); return *this; }
CyrIstream& ignore(int nCount = 1, int delim = EOF) {
    cin.ignore(nCount, delim); return *this; }
int peek() { return cin.peek(); }
int gcount() const { return cin.gcount(); }
CyrIstream& putback(char ch) { cin.putback(ch); return *this; }

// Замещение методов класса ios
void clear(int nState = 0) { cin.clear(nState); }
long setf(long lFlags) { return cin.setf(lFlags); }
void unsetf(long lFlags) { cin.unsetf(lFlags); }
int rdstate() const { return cin.rdstate(); }
long flags() const { return cin.flags(); }
streambuf* rdbuf() const { return cin.rdbuf(); }

// Дружественная функция для поддержки параметризованных манипуляторов
friend CyrIstream& operator >>(CyrIstream&,
const _Smanip<int>&);
private:
    char buf_[MAX_STR_LEN];
};
////////////////////////////////////
// Шаблон для ввода типа string
template<class _E, class _Tr, class _A> inline
CyrIstream& operator >>(CyrIstream& is,
basic_string<_E, _Tr, _A>& _X) {
    string temp;
    cin >> temp;
    unsigned int n = temp.size();
    char* buf = new char[n+1];
    temp.copy(buf, n); buf[n] = 0;
    OemToChar(buf, (char*)buf);
    _X = string(buf);
    delete [] buf;
    return is;
}
////////////////////////////////////
extern CyrIstream Cin;
extern CyrOstream Cout;

#endif /* CYR_IOS_H */

#ifndef CYR_IOS_IMPLEMENTATION
#define cin Cin

```

```

#define cout Cout
#define istream CyrIstream
#define ostream CyrOstream
#endif
// Конец файла CyrIOS.h
/////////////////////////////////////////////////////////////////
// CyrIOS.cpp
#define CYR_IOS_IMPLEMENTATION
#include "CyrIOS.h"
/////////////////////////////////////////////////////////////////
// Класс CyrOstream

CyrOstream& CyrOstream::operator <<(
_Myt& (__cdecl * _f)(_Myt&)) {
    cout << _f;    return *this;
}

CyrOstream& CyrOstream::operator <<(
ios_base& (__cdecl * _f)(ios_base& )) {
    cout << _f;    return *this;
}

CyrOstream& CyrOstream::operator <<(const char* s) {
    int n = strlen(s);
    strncpy(buf_, s, n); buf_[n] = 0;
    CharToOem(buf_, buf_);
    cout << buf_;
    return *this;
}

CyrOstream& CyrOstream::operator <<(char c) {
    buf_[0] = c; buf_[1] = 0;
    CharToOem(buf_, buf_);
    cout << buf_;
    return *this;
}

CyrOstream& CyrOstream::operator <<(unsigned char c) {
    unsigned char buf[2];
    buf[0] = c; buf[1] = 0;
    if (c > 191)
        CharToOem((const char*)buf, (char*)buf);
    cout << buf;
    return *this;
}

CyrOstream& CyrOstream::put(char c) {
    buf_[0] = c; buf_[1] = 0;
    CharToOem(buf_, buf_);

```

```

    cout.put(buf_[0]);
    return *this;
}

CyrOstream& CyrOstream::write(const char* s, int len) {
    int n = strlen(s);
    strncpy(buf_, s, n); buf_[n] = 0;
    CharToOem(buf_, buf_);
    cout.write(buf_, len);
    return *this;
}

CyrOstream& operator <<(CyrOstream& os, const _Smanip<int>& m) {
    cout << m;    return os;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Knacc CyrIstream

CyrIstream& CyrIstream::operator >>(
ios_base& (__cdecl * _f)(ios_base& )) {
    cin >> _f;    return *this;
}

CyrIstream& CyrIstream::operator >>(char* s) {
    string temp;
    cin >> temp;
    unsigned int n = temp.size();
    temp.copy(buf_, n); buf_[n] = 0;
    OemToChar(buf_, buf_);
    strncpy(s, buf_, n + 1);
    return *this;
}

CyrIstream& CyrIstream::operator >>(char& c) {
    cin >> buf_[0];
    buf_[1] = 0;
    OemToChar(buf_, buf_);
    c = buf_[0];
    return *this;
}

CyrIstream& CyrIstream::get(char& symb) {
    cin.get(buf_[0]);
    buf_[1] = 0;
    OemToChar(buf_, buf_);
    symb = buf_[0];
    return *this;
}

CyrIstream& CyrIstream::get(char* pch, int nCount, char delim) {
    cin.get(pch, nCount, delim);
}

```

```

    OemToChar(pch, pch);
    return *this;
}

```

```

CyrIstream& CyrIstream::get(unsigned char* pch,
int nCount, char delim) {
    cin.get((char*)pch, nCount, delim);
    OemToChar((const char*)pch, (char*)pch);
    return *this;
}

```

```

CyrIstream& CyrIstream::getline(char* pch, int nCount, char delim) {
    cin.getline(pch, nCount, delim);
    OemToChar(pch, pch);
    return *this;
}

```

```

CyrIstream& CyrIstream::read(char* pch, int nCount) {
    cin.read(buf_, nCount);
    buf_[nCount] = 0;
    OemToChar(buf_, buf_);

    for(int i = 0; i < nCount; i++)
        pch[i] = buf_[i];
    return *this;
}

```

```

CyrIstream& operator >>(CyrIstream& is, const _Smanip<int>& m) {
    cin >> m;    return is;
}

```

```

////////////////////////////////////
// Глобальные объекты для ввода и вывода
CyrIstream Cin(_Noinit);
CyrOstream Cout(_Noinit);
// Конец файла CyrIOS.cpp
////////////////////////////////////
// Task4_1c.cpp
#include "CyrIOS.h"

```

```

int main() {
    // Тестирование класса CyrOstream
    char str[] = "++!\n";
    cout << "Добро пожаловать в ";
    cout.put('C');
    cout.write(str, strlen(str));

    double y = 372.141526;
    cout.width(20);
    cout << y << endl;
}

```

```

cout.fill('.');   cout.width(20);
cout << y << endl;
cout.precision(10);   cout.width(20);
cout << y << endl;
return 0;
}
//————— конец проекта Task4_1b —————

```

Заключительные замечания.

- Внимательный читатель наверняка обратил внимание на незнакомый доселе метод `copy()` класса `string`, использованный, например, в теле шаблонной функции-операции `>>` для типа `string`. Метод извлекает из объекта типа `string` значение традиционной C-строки, помещая ее по адресу, указанному первым аргументом. К такой манипуляции приходится прибегать в тех случаях, когда серверная функция (например, `0emToChar()`) не умеет работать с объектом типа `string`, требуя, чтобы ей подавали на вход именно традиционную C-строку.
- Заметим, что в методе `CyrOstream& operator <<(unsigned char)`, а также в шаблоне для вывода типа `string` обработке функцией `CharToOem()` подвергаются только те символы, которые имеют числовой код, превышающий значение 191. Это сделано для того, чтобы не исказить символы псевдографики, которые в таблице кодов ANSI размещены в интервале 128–191, в то время как символы кириллицы занимают часть таблицы с кодами 192–255.
- В функции `main()` находятся операторы для тестирования методов, добавленных в класс `CyrOstream`. Данная программа должна вывести¹:

```

Добро пожаловать в C++!
          372.142
.....372.142
.....372.141526

```

- Тестирование класса `CyrIstream` мы оставляем читателю в качестве упражнения. Рассмотрим теперь другую задачу, в которой используются стандартные средства ввода/вывода для потоков и, кроме этого, перегружаются операции извлечения и вставки для пользовательского типа данных.

Задача 4.2. Первичный ввод и поиск информации в базе данных

Написать программу, которая обеспечивает первичный ввод информации в базу данных отдела кадров предприятия с количеством сотрудников до 100 человек (без записи в файл) и поиск информации по заданному критерию.

¹ Заметим, что тестер-профессионал, конечно, проведет повторные испытания нашего класса на всех предыдущих тестах, а также проверит все замещенные методы.

Каждая запись базы данных содержит следующие сведения о сотруднике:

- фамилию и инициалы;
- год поступления на работу;
- оклад.

Критерий поиска: сотрудники с окладом, превышающим некоторую заданную величину.

Решение задачи начнем с выявления понятий/классов и их фундаментальных взаимосвязей.

В данном случае первым понятием является *база данных*, и, следовательно, для моделирования этого понятия нам понадобится класс, который естественно назвать `DBase`. Объект типа `DBase` (то есть сама база данных) должен содержать некоторую совокупность или коллекцию других объектов, соответствующих записям базы данных. Для моделирования понятия *запись базы данных* введем класс `Man`. Очевидно, что взаимоотношение между указанными классами относится к типу «`DBase has a Man`».

На втором этапе необходимо уточнить классы, определив основные поля и набор операций над ними.

Начнем с класса `DBase`. Вопрос первый, который нужно решить: какую структуру данных целесообразно использовать для хранения коллекции записей. Поскольку объем базы данных небольшой¹, выберем самое простое решение — массив объектов типа `Man`. Очевидно, что в конструкторе класса `DBase` необходимо предусмотреть динамическое выделение памяти для требуемого количества объектов типа `Man`, а в деструкторе — освобождение этой памяти. Адрес начала массива объектов будет представлен полем `Man* pMan`.

Из условия задачи выясняем также, что в классе необходимо иметь метод `InitInput()` для первичного ввода информации в базу данных и метод `SearchPayNotLess()` для поиска сотрудников с окладом, превышающим некоторую заданную величину. Для контроля правильности ввода исходных данных нам пригодится еще один метод — `Show()`, обеспечивающий вывод на экран содержимого базы данных.

Теперь разберемся с классом `Man`. Для хранения информации, относящейся к одному сотруднику, потребуются следующие поля:

- `char* pName` — адрес строки, содержащей фамилию и инициалы;
- `int come_year` — год поступления на работу;
- `double pay` — величина оклада.

Конструктор класса должен выделять память для хранения указанной строки, а деструктор — освобождать эту память. Для решения второй подзадачи (поиск информации) добавим в класс метод доступа `GetPay()`. И наконец, для класса `Man` нужно предусмотреть перегрузку операции извлечения, чтобы обеспечить первичный ввод информации с клавиатуры в методе `InitInput()` класса `DBase`, и операцию вставки, которая будет использована в методе `Show()` класса `DBase`. Обе операции будут реализованы как внешние дружественные функции.

¹ Это вытекает из условия задачи.

Иногда при решении задачи удобно использовать внешние функции, не являющиеся членами классов. Обычно эти функции выполняют какую-то рутинную работу и могут быть вызваны как из методов классов, так и из основной функции. Типичный пример — ввод значений из стандартного потока cin с защитой от непреднамеренных ошибок пользователя. Тему обработки ошибок потоков мы затрагивали в начале семинара; теперь пришло время показать возможное практическое решение проблемы. Начнем с «наивной» реализации перегруженной операции >> для класса Man:

```
istream& operator >> (istream& in, Man& obj) {
// . . . . .
    in >> obj.come_year;
    in >> obj.pay;
    return in;
}
```

Если в момент выполнения оператора `in >> obj.come_year;` пользователь введет вместо целого числа какую-то произвольную строку символов, то программа «сломается». Вашему заказчику наверняка не понравится такое поведение программы. Аналогичная проблема имеется и при вводе вещественного числа в следующем операторе.

Для решения этих проблем в программе будут использованы функции `GetInt()` и `GetDouble()`, обеспечивающие очень надежный ввод целых и вещественных чисел соответственно. Реализацию этих функций мы рассмотрим ниже. Так как эти функции универсальные и внеклассовые, то их код целесообразно разместить в отдельном модуле.

Решение задачи, в котором реализованы рассмотренные концепции, представляет собой многофайловый проект, содержащий файлы `DBase.h`, `DBase.cpp`, `Man.h`, `Man.cpp`, `GetFunc.h`, `GetFunc.cpp` и `Main.cpp`:

```
////////////////////////////////////
// Проект Task4_2
////////////////////////////////////
// DBase.h
class DBase {
public:
    DBase(int);
    ~DBase();
    void InitInput();
    void Show();
    void SearchPayNotLess(double);
private:
    Man* pMan;
    int nRecords;
};
////////////////////////////////////
// DBase.cpp
#include "Man.h"
#include "DBase.h"
```

```

DBase::DBase(int nRec) : nRecords(nRec),
pMan(new Man[nRec]) {}

DBase::~DBase() { if (pMan) delete [] pMan; }

void DBase::InitInput() {
    for (int i = 0; i < nRecords; i++)
        cin >> *(pMan + i); // 1
}

void DBase::Show() {
    cout << "======" << endl;
    cout << "Содержимое базы данных:" << endl;
    for (int i = 0; i < nRecords; i++)
        cout << *(pMan + i); // 2
}

void DBase::SearchPayNotLess(double anyPay) {
    bool not_found = true;
    for (int i = 0; i < nRecords; i++)
        if ((pMan + i)->GetPay() >= anyPay) {
            cout << *(pMan + i);
            not_found = false;
        }
    if (not_found) cout << "Таких сотрудников нет." << endl;
}

/////////////////////////////////////////////////////////////////
// Man.h
#include <iostream>
#include <iomanip>
using namespace std;
// #include "CyrIOS.h" // for Visual C++ 6.0

const int l_name = 30;

class Man {
public:
    Man(int lName = 30);
    ~Man();
    double GetPay() const;
    // Операция извлечения (ввода)
    friend istream& operator >>(istream&, Man&);
    // Операция вставки (вывода)
    friend ostream& operator <<(ostream&, Man&);

private:
    char* pName;
    int come_year;
    double pay;
};

```

```

////////////////////////////////////
// Man.cpp
#include "Man.h"
#include "GetFunc.h"

Man::Man(int lName) { pName = new char[lName + 1]; }
Man::~Man() { if (pName) delete [] pName; }
double Man::GetPay() const { return pay; }

// Операция извлечения (ввода)
istream& operator >> (istream& in, Man& ob) {
    cout << "\nВведите данные в формате" << endl;
    cout << "Фамилия И.О. <Enter> Год поступления <Enter>:";
    cout << " Оклад <Enter>:" << endl;
    in.getline(ob.pName, l_name);
    ob.come_year = GetInt(in); // 3
    ob.pay = GetDouble(in); // 4
    return in;
}

// Операция вставки (вывода)
ostream& operator << (ostream& out, Man& ob) {
    out << setw(30) << setiosflags(ios::left);
    out << ob.pName << " ";
    out << ob.come_year << " ";
    out << ob.pay << endl;
    return out;
}
////////////////////////////////////
// GetFunc.h
int GetInt(istream&); // Ввод целого числа
double GetDouble(istream&); // Ввод вещественного числа
////////////////////////////////////
// GetFunc.cpp
#include "Man.h"
#include "GetFunc.h"
// ----- ввод целого числа
int GetInt(istream& in) {
    int value;
    while (true) {
        in >> value; // 5
        if (in.peek() == '\n') { // 6
            in.get(); // 7
            break;
        }
    }
    else {
        cout << "Повторите ввод (ожидается целое число):"
            << endl; // 8
    }
}

```


Обратите внимание на следующие моменты.

- ❑ В реализации метода `InitInput()` перегруженная для класса `Map` операция извлечения применяется к объекту, задаваемому выражением `*(pMap + i)`, то есть к объекту, адрес которого есть `pMap + i` (оператор 1).
- ❑ Аналогичная адресация объекта для операции вставки использована в методе `Show()` — оператор 2.
- ❑ В реализации перегруженной операции извлечения (файл `Map.cpp`) обратите внимание на операторы 3 и 4, в которых вызываются функции `GetInt()` и `GetDouble()`, предназначенные для ввода из стандартного потока целых и вещественных чисел соответственно.
- ❑ Реализация функций `GetInt()` и `GetDouble()` находится в файле `GetFunc.cpp`. Рассмотрим подробно первую из них. Ввод целого числа организован внутри бесконечного цикла `while` следующим образом.
 - Информация читается из входного потока оператором 5. Если в буфере типа `streambuf`, связанном с потоком `in`, находится изображение целого числа, завершаемое символом перевода строки `'\n'`, то по завершении операции извлечения в буфере останется только символ `'\n'`. Оператор 6 проверяет это условие, используя метод `peek()`. Если проверка завершилась успешно, оператор 7 очищает входной буфер от символа `'\n'`, после чего происходит выход из цикла `while` с последующим возвратом из функции значения `value`. Если же введенная информация не является корректным изображением целого числа, то выполняются три действия:
 - оператор 8 выводит сообщение об этом, предлагая повторить ввод;
 - сбрасываются флаги ошибок для потока `in` (оператор 9);
 - с помощью внутреннего цикла `while` из входного буфера извлекаются все символы, вплоть до символа `'\n'` (оператор 10). Внешний цикл `while` повторяется сначала.
 - Функция `GetDouble()` работает аналогично.
- ❑ Функция `main()` (файл `Main.cpp`) ничего особенного собой не представляет. Алгоритм прозрачный и воспринимается без дополнительных пояснений благодаря выразительным именам методов.
- ❑ В заключение отметим, что рассмотренная задача очень похожа на задачу 1.1 из первого семинара¹. Поэтому мы советуем читателю сравнить решение этих двух задач, чтобы лучше прочувствовать, что дает полноценное использование технологии ООП. Особенно показательным сравнение кодов функции `main()`: положите рядом тексты из файлов `Main.cpp` для проектов `Task1_1` и `Task4_2` и вы будете поражены совершенством, лаконичностью и красотой второго решения по сравнению с первым.

Давайте повторим основные моменты этого семинара.

1. Ввод и вывод представляются в программе как поток байтов и обычно выполняются через буфер. Потоки поддерживаются в библиотеке C++ с по-

¹ Если абстрагироваться от способа ввода исходных данных: в задаче 1.1 данные читались из файла, а в задаче 4.2 вводятся с клавиатуры.

мощью иерархии классов, которая реализует безопасный ввод-вывод как стандартных, так и пользовательских типов данных.

2. Ввод-вывод бывает форматированный и неформатированный. Для форматированного ввода-вывода используются перегруженные операции << и >>, для неформатированного — методы стандартных классов.
3. Управление форматированием выполняется с помощью манипуляторов и методов стандартных классов.
4. Для вывода объектов пользовательских типов данных следует с помощью дружественных функций перегрузить операции чтения и извлечения.
5. Для обеспечения безопасного ввода необходимо диагностировать возможные ошибки. При этом используются флаги состояния потока и/или методы peek, get и clear.
6. Если необходимо заместить в производном классе одну из версий перегруженного метода базового класса, придется заместить и все остальные версии этого метода.

Задания

Вариант 1

1. Определить класс с именем STUDENT, содержащий следующие поля:

- фамилия и инициалы;
- номер группы;
- успеваемость (массив из пяти элементов).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа STUDENT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти объектов типа STUDENT; записи должны быть упорядочены по возрастанию номера группы;
- вывод на дисплей фамилий и номеров групп для всех студентов, включенных в массив, если средний балл студента больше 4.0;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 2

1. Определить класс с именем STUDENT, содержащий следующие поля:

- фамилия и инициалы;
- номер группы;
- успеваемость (массив из пяти элементов).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа STUDENT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти объектов типа STUDENT; записи должны быть упорядочены по возрастанию среднего балла;
- вывод на дисплей фамилий и номеров групп для всех студентов, имеющих оценки 4 и 5;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 3

1. Определить класс с именем STUDENT, содержащий следующие поля:

- фамилия и инициалы;
- номер группы;
- успеваемость (массив из пяти элементов).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа STUDENT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти объектов типа STUDENT; записи должны быть упорядочены по алфавиту;
- вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 4

1. Определить класс с именем AEROFLOT, содержащий следующие поля:

- название пункта назначения рейса;
- номер рейса;
- тип самолета.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа AEROFLOT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из семи объектов типа AEROFLOT; записи должны быть упорядочены по возрастанию номера рейса;
- вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры;
- если таких рейсов нет, выдать на дисплей соответствующее сообщение;

Вариант 5

1. Определить класс с именем AEROFLOT, содержащий следующие поля:
 - название пункта назначения рейса;
 - номер рейса;
 - тип самолета.Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа AEROFLOT.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из семи объектов типа AEROFLOT; записи должны быть размещены в алфавитном порядке по названиям пунктов назначения;
 - вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры;
 - если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Вариант 6

1. Определить класс с именем WORKER, содержащий следующие поля:
 - фамилия и инициалы работника;
 - название занимаемой должности;
 - год поступления на работу.Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа WORKER.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из десяти объектов типа WORKER; записи должны быть размещены по алфавиту;
 - вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры;
 - если таких работников нет, вывести на дисплей соответствующее сообщение.

Вариант 7

1. Определить класс с именем TRAIN, содержащий следующие поля:
 - название пункта назначения;
 - номер поезда;
 - время отправления.Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа TRAIN.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из восьми объектов типа TRAIN; записи должны быть размещены в алфавитном порядке по названиям пунктов назначения;

- вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени;
- если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 8

1. Определить класс с именем TRAIN, содержащий следующие поля:
 - название пункта назначения;
 - номер поезда;
 - время отправления.Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа TRAIN.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из шести объектов типа TRAIN; записи должны быть упорядочены по времени отправления поезда;
 - вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры;
 - если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 9

1. Определить класс с именем TRAIN, содержащий следующие поля:
 - название пункта назначения;
 - номер поезда;
 - время отправления.Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа TRAIN.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из восьми объектов типа TRAIN; записи должны быть упорядочены по номерам поездов;
 - вывод на экран информации о поезде, номер которого введен с клавиатуры;
 - если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 10

1. Определить класс с именем MARSH, содержащий следующие поля:
 - название начального пункта маршрута;
 - название конечного пункта маршрута;
 - номер маршрута.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа MARSH.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа MARSH; записи должны быть упорядочены по номерам маршрутов;
- вывод на экран информации о маршруте, номер которого введен с клавиатуры;
- если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 11

1. Определить класс с именем MARSH, содержащий следующие поля:

- название начального пункта маршрута;
- название конечного пункта маршрута;
- номер маршрута.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа MARSH.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа MARSH; записи должны быть упорядочены по номерам маршрутов;
- вывод на экран информации о маршрутах, которые начинаются или кончаются в пункте, название которого введено с клавиатуры;
- если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 12

1. Определить класс с именем NOTE, содержащий следующие поля:

- фамилия, имя;
- номер телефона;
- день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа NOTE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа NOTE; записи должны быть упорядочены по датам дней рождения;
- вывод на экран информации о человеке, номер телефона которого введен с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 13

1. Определить класс с именем NOTE, содержащий следующие поля:

- фамилия, имя;
- номер телефона;
- день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа NOTE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа NOTE; записи должны быть размещены по алфавиту;
- вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры;
- если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 14

1. Определить класс с именем NOTE, содержащий следующие поля:

- фамилия, имя;
- номер телефона;
- день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа NOTE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа NOTE; записи должны быть упорядочены по трем первым цифрам номера телефона;
- вывод на экран информации о человеке, чья фамилия введена с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 15

1. Определить класс с именем ZNAK, содержащий следующие поля:

- фамилия, имя;
- знак Зодиака;
- день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ZNAK.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ZNAK; записи должны быть упорядочены по датам дней рождения;

- вывод на экран информации о человеке, чья фамилия введена с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 16

1. Определить класс с именем ZNAK, содержащий следующие поля:
 - фамилия, имя;
 - знак Зодиака;
 - день рождения (массив из трех чисел).Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ZNAK.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ZNAK; записи должны быть упорядочены по датам дней рождения;
 - вывод на экран информации о людях, родившихся под знаком, наименование которого введено с клавиатуры;
 - если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 17

1. Определить класс с именем ZNAK, содержащий следующие поля:
 - фамилия, имя;
 - знак Зодиака;
 - день рождения (массив из трех чисел).Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ZNAK.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ZNAK; записи должны быть упорядочены по знакам Зодиака;
 - вывод на экран информации о людях, родившихся в месяце, значение которого введено с клавиатуры;
 - если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 18

1. Определить класс с именем PRICE, содержащий следующие поля:
 - название товара;
 - название магазина, в котором продается товар;
 - стоимость товара в рублях.Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа PRICE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа PRICE; записи должны быть размещены в алфавитном порядке по названиям товаров;
- вывод на экран информации о товаре, название которого введено с клавиатуры;
- если таких товаров нет, выдать на дисплей соответствующее сообщение.

Вариант 19

1. Определить класс с именем PRICE, содержащий следующие поля:

- название товара;
- название магазина, в котором продается товар;
- стоимость товара в рублях.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа PRICE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа PRICE; записи должны быть размещены в алфавитном порядке по названиям магазинов;
- вывод на экран информации о товарах, продающихся в магазине, название которого введено с клавиатуры;
- если такого магазина нет, выдать на дисплей соответствующее сообщение.

Вариант 20

1. Определить класс с именем ORDER, содержащий следующие поля:

- расчетный счет плательщика;
- расчетный счет получателя;
- перечисляемая сумма в рублях.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ORDER.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ORDER; записи должны быть размещены в алфавитном порядке по расчетным счетам плательщиков;
- вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры;
- если такого расчетного счета нет, выдать на дисплей соответствующее сообщение.

Семинар 5 Файловые и строковые потоки. Строки класса string

Теоретический материал: с. 280–294.

Файловые потоки

Для поддержки файлового ввода и вывода стандартная библиотека C++ содержит классы, указанные в табл. 5.1.

Таблица 5.1. Классы файловых потоков

Класс	Инстанцирован из шаблона	Базовый шаблонный класс	Назначение
ifstream	basic_ifstream	basic_istream	Входной файловый поток
ofstream	basic_ofstream	basic_ostream	Выходной файловый поток
fstream	basic_fstream	basic_iostream	Двунаправленный файловый поток

Так как классы файловых потоков являются производными от классов `istream`, `ostream` и `iostream` соответственно, то они наследуют все методы указанных классов, перегруженные операции вставки и извлечения, манипуляторы, состояние потоков и т. д. Так же, как и в случае стандартных потоков, файловые потоки обеспечивают гораздо более надежный ввод/вывод, чем старые функции библиотеки C, работающие с потоками типа `FILE*`.

Для использования файловых потоков необходимо подключить к программе заголовочный файл `<fstream>`.

Работа с файлом обычно предполагает следующие операции:

- ❑ создание потока (потокowego объекта);
- ❑ открытие потока и связывание его с файлом;
- ❑ обмен с потоком (ввод/вывод);
- ❑ закрытие файла.

Классы файловых потоков содержат несколько конструкторов, позволяющих варьировать способы создания потокowych объектов.

Конструкторы с параметрами создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:

```
ifstream(const char* name, int mode = ios::in);
ofstream(const char* name, int mode = ios::out | ios::trunc);
fstream(const char* name, int mode = ios::in | ios::out);
```

Второй параметр конструктора задает режим открытия файла. Если значение по умолчанию вас не устраивает, можно указать другое, выбрав одно или несколько значений (объединенных операцией |) из указанных в табл. 5.2.

Таблица 5.2. Значения аргумента mode

Флаг	Назначение
ios::in	Открыть файл для ввода
ios::out	Открыть файл для вывода
ios::ate	Установить указатель на конец файла
ios::app	Открыть в режиме добавления в конец файла
ios::trunc	Если файл существует, то обрезать его до нулевой длины
ios::binary	Открыть в двоичном режиме (по умолчанию используется текстовый режим)
ios::nocreate ¹	Если файл не существует, зафиксировать ошибку (установкой потокowego объекта в нулевое значение)
ios::noreplace ¹	Если файл существует, зафиксировать ошибку

Конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом. В этом случае связь потока с конкретным файлом осуществляется позже — вызовом метода open(), который имеет параметры, аналогичные параметрам рассмотренных выше конструкторов.

Приведем примеры создания потокowych объектов и связывания их с конкретными файлами:

```
// Файлы для вывода
ofstream flog("flog.txt");
ofstream fout1, fout2;
```

¹ Поддерживается не всеми компиляторами.

```
fout1.open("test1", ios::app);
fout2.open("test2", ios::binary);
// Файл для ввода
ifstream finpl("data.txt");
// Файл для ввода и вывода
fstream myfile;
myfile.open("mf.dat");
```

Если в качестве параметра `name` задано *краткое имя файла* (без указания полного пути), то подразумевается, что файл открывается в текущем каталоге, в противном случае требуется задавать *полное имя файла*, например:

```
ifstream finpl("D:\\VCwork\\Task1\\data.txt");
```

ВНИМАНИЕ

Если файл по какой-либо причине не удастся открыть (например, входной файл в указанном каталоге не найден или нет свободного места на диске для выходного файла), то независимо от способа его открытия — конструктором или методом `open()` — потоковый объект принимает значение, равное нулю. Поэтому рекомендуется всегда проверять, чем завершилась попытка открытия файла, например:

```
ifstream finpl("data.txt");
if (!finpl) {
    cerr << "Файл data.txt не найден." << endl;
    throw "Ошибка открытия файла ";
}
```

После того как файловый поток открыт, работа с ним чрезвычайно проста: с входным потоком можно обращаться так же, как со стандартным объектом `cin`, а с выходным — так же, как со стандартным объектом `cout`.

При чтении данных из входного файла иногда требуется контролировать, были ли достигнут конец файла после очередной операции ввода. Это позволяет делать метод `eof()`, возвращающий нулевое значение, если конец файла еще не достигнут, и ненулевое значение — если уже достигнут¹.

Если в процессе выполнения операций ввода/вывода фиксируется некоторая ошибочная ситуация, то потоковый объект также принимает значение, равное нулю. Рекомендуется особо следить за состоянием потокового объекта во время выполнения операций вывода, так как диски «не резиновые» и имеют тенденцию переполняться.

Когда программа покидает область видимости потокового объекта, он уничтожается. При этом перестает существовать связь между потоковым объектом и физическим файлом, а физический файл закрывается. Если алгоритм требует более раннего закрытия файла, вы можете воспользоваться методом `close()`.

¹ Учтите, что в C++ после чтения из файла *последнего элемента* условие конца файла *не возникает!* Оно возникает при следующем чтении, когда программа пытается считать данные за последним элементом в файле.

В качестве примера работы с файловыми потоками приведем программу копирования одного файла в другой. Имена файлов берутся из аргументов командной строки:

```
// MyCopy.cpp
#include <iostream>
#include <fstream>
using namespace std;

void error(const char* text1, const char* text2 = "") {
    cerr << text1 << ' ' << text2 << endl;
    exit(1);
}

int main(int argc, char* argv[]) {
    if (argc != 3) error("Неверное число аргументов");

    ifstream from(argv[1]); // открываем входной файл
    if (!from) error("Входной файл не найден:", argv[1]);

    ofstream to(argv[2]); // открываем выходной файл
    if (!to) error("Выходной файл не открыт:", argv[2]);

    char ch;
    while (from.get(ch)) {
        to.put(ch);
        if (!to) error("Ошибка записи (диск переполнен).");
    }
    cout << "Копирование из " << argv[1] << " в " << argv[2] << " завершено." << endl;
    return 0;
}
```

Другой пример — программа вывода содержимого текстового файла на экран (имя файла задается аргументом командной строки):

```
// MyType.cpp
#include <iostream>
#include <fstream>
using namespace std;

// Определение функции error() — из файла MyCopy.cpp
// ...

int main(int argc, char* argv[]) {
    if (argc != 2) error("Неверное число аргументов");

    ifstream tfile(argv[1]); // открываем входной файл
    if (!tfile) error("Входной файл не найден:", argv[1]);

    char buf[1024];
```

```
while (!tfile.eof()) {
    tfile.getline(buf, sizeof(buf));
    cout << buf << endl;
}
return 0;
}
```

В приведенной программе предполагается, что длина строки в текстовом файле не превышает 1024 символа. При необходимости можно увеличить размер буфера `buf` до требуемой величины.

Строковые потоки

Работу со строковыми потоками обеспечивают классы `istream`, `ostream` и `stringstream`, которые являются производными от классов `istream`, `ostream` и `iostream` соответственно. Для использования строковых потоков необходимо подключить к программе заголовочный файл `<sstream>`.

Применение строковых потоков аналогично применению файловых потоков. Отличие заключается в том, что физически информация потока размещается в оперативной памяти, а не в файле на диске. Кроме того, классы строковых потоков содержат метод `str()`, возвращающий копию строки типа `string` или присваивающий потоку значение такой строки:

```
string str() const;
void str(const string& s);
```

Строковые потоки являются некоторыми аналогами функций `sscanf()` и `sprintf()` библиотеки C, которые также работают со строками в памяти, имитируя консольный ввод/вывод. Например, с помощью `sprintf()` можно сформировать в памяти некоторую символьную строку, которая затем отображается на экране. Эта же проблема легко решается с помощью объекта типа `ostringstream`.

В качестве примера приведем модифицированную версию предыдущей программы, которая выводит содержимое текстового файла на экран, предваряя каждую строку текстовой меткой «Line N:», где N — номер строки:

```
// AdvType.cpp
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
using namespace std;

// Определение функции error() — из файла MyCopy.cpp
// ...

int main(int argc, char* argv[]) {
    if (argc != 2) error("Неверное число аргументов.");
```

```
ifstream tfile(argv[1]);
if (!tfile) error("Входной файл не найден:", argv[1]);

int n = 0;
char buf[1024];
while (!tfile.eof()) {
    n++;
    tfile.getline(buf, sizeof(buf));
    ostringstream line;
    line << "Line " << setw(3) << n << ": " << buf << endl;
    cout << line.str();
}
return 0;
}
```

Строки класса string

Мы уже пользовались объектами класса `string` начиная со второго семинара и успели оценить те удобства, которые обеспечивает этот класс в сравнении с традиционными C-строками¹. В этом разделе строки типа `string` будут рассмотрены более подробно.

Одной из важнейших для удобства программиста возможностей класса `string`² является то, что он берет на себя управление памятью как при первоначальном размещении строки, так и при всех ее модификациях, увеличивающих или уменьшающих длину строки. Таким образом, вы можете «забыть» об операциях `new` и `delete`, неаккуратное обращение с которыми является источником трудно диагностируемых ошибок.

Кроме этого, строки типа `string` защищены от ошибочных обращений к памяти, связанных с выходом за их границы, чего не скажешь про C-строки. Но за все надо платить: строки типа `string` значительно проигрывают C-строкам в эффективности. Поэтому, если от программы требуется максимальное быстродействие при обработке строк, то, возможно, лучше воспользоваться старым инструментарием. В большинстве же программ на C++ строки типа `string` обеспечивают необходимую скорость обработки, поэтому их применение предпочтительней.

Чтобы использовать строки типа `string`, необходимо подключить к программе заголовочный файл `<string>`.

Для понимания определений методов класса `string` необходимо знать назначение некоторых имен. Так, в пространстве `std` определен идентификатор `size_type`, яв-

¹ Массивами символов типа `char`, завершаемыми нулевым байтом.

² На самом деле класс `string` инстанцирован из шаблонного класса `basic_string` путем объявления `typedef basic_string<char> string;`.

ляющийся синонимом типа unsigned int. В классе string определена константа `npos`, задающая максимально возможное число, которое в зависимости от контекста означает либо «все элементы» строки, либо отрицательный результат поиска. Так как максимально возможное число имеет вид `0xFFFF...FFFF`, то в случае присваивания его переменной типа `int` получится значение `-1`.

В классе string имеется несколько конструкторов. Ниже в упрощенном виде приведены наиболее употребительные из них¹:

```
string(); // создает пустой объект класса string
string(const char*); // создает объект, инициализируя его значением C-строки
```

Класс содержит три операции присваивания:

```
string& operator=(const string& str);
string& operator=(const char* s); // присвоить значение C-строки
string& operator=(char c); // присвоить значение символа
```

В табл. 5.3 приведены допустимые для объектов класса string операции.

Таблица 5.3. Операции класса string

Операция	Действие	Операция	Действие
=	Присваивание	>	Больше
+	Конкатенация	>=	Больше или равно
= =	Равенство	[]	Индексация
!=	Неравенство	<<	Вывод
<	Меньше	>>	Ввод
<=	Меньше или равно	+=	Добавление

Использование операций очевидно. Размеры строк устанавливаются автоматически так, чтобы объект мог содержать присваиваемое ему значение.

Кроме операции индексации для доступа к элементу строки определен метод `at(size_type n)`, который можно использовать как для чтения, так и для записи `n`-го элемента строки `s`²:

```
cout << s.at(2); // Будет выведен 2-й символ строки
s.at(5) = 'W'; // 5-й символ заменяется символом W
```

Заметим, что в операции индексации не проверяется выход за диапазон строки. Метод `at()`, напротив, такую проверку содержит, и если индекс превышает длину строки, то порождается исключение `out_of_range`.

В табл. 5.4 приведены некоторые наиболее употребительные методы класса string.

¹ Более полную информацию о методах класса string см. в учебнике (глава 11).

² Нумерация элементов начинается с нуля.

Таблица 5.4. Методы класса string

Метод	Назначение
<code>size_type size() const:</code>	Возвращает размер строки
<code>size_type length() const:</code>	То же, что и <code>size()</code>
<code>insert(size_type pos1, const string& str):</code>	Вставляет строку <code>str</code> в вызывающую строку, начиная с позиции <code>pos1</code>
<code>replace(size_type pos1, size_type n1, const string& str):</code>	Заменяет <code>n1</code> элементов, начиная с позиции <code>pos1</code> вызывающей строки, элементами строки <code>str</code>
<code>string substr(size_type pos=0, size_type n=npos) const:</code>	Возвращает подстроку длиной <code>n</code> , начиная с позиции <code>pos</code>
<code>size_type find(const string& str, size_type pos=0) const:</code>	Ищет самое левое вхождение строки <code>str</code> в вызывающую строку, начиная с позиции <code>pos</code> . Возвращает позицию вхождения, или <code>npos</code> , если вхождение не найдено
<code>size_type find(char c, size_type pos=0) const:</code>	Ищет самое левое вхождение символа <code>c</code> , начиная с позиции <code>pos</code> . Возвращает позицию вхождения или <code>npos</code> , если вхождение не найдено
<code>size_type rfind(const string& str, size_type pos=0) const:</code>	Ищет самое правое вхождение строки <code>str</code> , начиная с позиции <code>pos</code> ¹
<code>size_type rfind(char c, size_type pos=0) const:</code>	Ищет самое правое вхождение символа <code>c</code> , начиная с позиции <code>pos</code> ¹
<code>size_type find_first_of(const string& str, size_type pos=0) const:</code>	Ищет самое левое вхождение любого символа строки <code>str</code> , начиная с позиции <code>pos</code> ¹
<code>size_type find_last_of(const string& str, size_type pos=0) const:</code>	Ищет самое правое вхождение любого символа строки <code>str</code> , начиная с позиции <code>pos</code> ¹
<code>swap(string& str):</code>	Обменивает содержимое вызывающей строки и строки <code>str</code>
<code>erase(size_type pos=0, size_type n=npos):</code>	Удаляет <code>n</code> элементов, начиная с позиции <code>pos</code>
<code>const char* c_str() const:</code>	Возвращает указатель на C-строку, содержащую копию вызывающей строки. Полученную C-строку нельзя пытаться изменить
<code>size_type copy(char* s, size_type n, size_type pos=0) const:</code>	Копирует в символьный массив <code>s</code> <code>n</code> элементов вызывающей строки, начиная с позиции <code>pos</code> . Ноль-символ в результирующий массив не заносится. Метод возвращает количество скопированных элементов

Поясним применение метода `find()`. Допустим, что вы работаете над программой для игры в шахматы с компьютером, а в данный момент пишете функцию для

¹ Возвращаемые значения такие же, как и у метода `find()`.

ввода обозначения колонки шахматной доски. Эти колонки, как известно, обозначаются начальными символами латинского алфавита: А, В, С, D, E, F, G, H. Желательно, чтобы ваша функция не допускала ввод некорректных символов. Очевидно, что существуют десятки решений этой задачи. Мы приведем лишь один возможный вариант:

```
char GetColumn() {
    string goodChar = "ABCDEFGH";
    char symb;
    cout << "Введите обозначение колонки: ";
    while (1) {
        cin >> symb;
        if (-1 == goodChar.find(symb)) {
            cout << "Ошибка. Введите корректный символ:\n";
            continue;
        }
        return symb;
    }
}
```

Обратите внимание на то, что метод `find()` используется здесь для проверки, принадлежит ли введенный с клавиатуры символ множеству символов, заданному с помощью строки `goodChar`.

Перейдем к рассмотрению задач.

Задача 5.1. Подсчет количества вхождений слова в текст

Написать программу, которая определяет, сколько раз встретилось заданное слово в текстовом файле. Текст не содержит переиосов слов. Максимальная длина строки в файле неизвестна¹.

Определим слово в тексте как последовательность алфавитно-цифровых символов, после которых следует либо знак пунктуации², либо разделитель. В качестве разделителей могут выступать один или несколько пробелов, один или несколько символов табуляции `'\t'` и символ конца строки `'\n'`.

Для хранения заданного слова (оно вводится с клавиатуры) определим переменную `word` типа `string`.

Поскольку максимальная длина строки в файле неизвестна, будем читать файл не построчно, а пословно, размещая очередное прочитанное слово в переменной `curword` типа `string`. Такое чтение можно реализовать с помощью операции `>>`, которая в случае операнда типа `string` игнорирует все разделители, предваряющие текущее слово, и считывает символы текущего слова в переменную `curword`, пока не встретится очередной разделитель.

¹ Аналогичная задача решалась в первой книге практикума (задача 5.2), но с упрощающим ограничением: длина строки в файле не более 80 символов.

² Один из знаков: «точка», «запятая», «вопросительный знак», «восклицательный знак».

Очевидно, что «опознание» текущего слова должно осуществляться с учетом возможного наличия после него одного из знаков пунктуации. Для решения этой задачи определим глобальную функцию

```
bool equal(const string& cw, const string& w):
```

которая возвращает значение true, если текущее слово cw совпадает с заданным словом w с точностью до знака пунктуации, или false — в противном случае.

Имея такую функцию, очень просто составить алгоритм основного цикла:

- прочитать очередное слово;
- если оно совпадает с заданным словом w (с точностью до знака пунктуации), то увеличить на единицу значение счетчика count.

Теперь приведем текст решения:

```
// Main.cpp
#include <fstream>
#include <string>
#include "CyrIOS.h" // for Visual C++ 6.01
using namespace std;

bool equal(const string& cw, const string& w) {
    char punct[] = {'.', '.', '?', '!'};

    if (cw == w) return true;
    for (int i=0; i < sizeof(punct); ++i)
        if (cw == w + punct[i]) return true;
    return false;
}

int main() {
    string word, curword;

    cout << " Введите слово для поиска: ";
    cin >> word;

    ifstream fin("infile.txt", ios::in|ios::nocreate);
    if (!fin) { cout << "Ошибка открытия файла." << endl; return 1; }

    int count = 0;
    while (!fin.eof()) {
        fin >> curword;
        if (equal(curword, word)) count++;
    }
    cout << "Количество вхождений слова: " << count << endl;
    return 0;
}
```

¹ Если вы не читали семинар 1, рекомендуем вам посмотреть комментарии к задаче 1.1 по вопросу ввода/вывода кириллицы.

Обратите внимание на реализацию функции `equal()` и, в частности, на использование операции сложения для добавления в конец строки `w` одного из знаков пунктуации.

Задача 5.2. Вывод вопросительных предложений

Написать программу, которая считывает текст из файла и выводит на экран только вопросительные предложения из этого текста¹.

Итак, мы имеем текстовый файл неизвестного размера, состоящий из неизвестного количества предложений. Предложение может занимать несколько строк, поэтому читать файл построчно неудобно — это привело бы к неоправданно сложному алгоритму. При решении аналогичной задачи в первой книге практикума было принято решение выделить буфер, в который поместится *весь* файл. Такое решение тоже нельзя признать идеальным — ведь файл может иметь сколь угодно большие размеры, и тогда программа окажется неработоспособной.

Поискем более удачное решение, используя новые средства языка C++, с которыми мы познакомимся на этом семинаре.

Попробуем читать файл пословно, как и в предыдущей программе, в переменную `word` типа `string`, и отправлять каждое прочитанное слово в строковый поток `sentence` типа `ostringstream`, который, как вы уже догадались, будет хранилищем очередного предложения.

При таком подходе, однако, есть проблема, связанная с потерей разделителей при чтении файла операцией `fin >> word`. Чтобы ее решить, будем «заглядывать» в следующую позицию файлового потока `fin` с помощью метода `peek()`. При обнаружении символа-разделителя его нужно отправить в поток `sentence` и переместиться на следующую позицию в потоке `fin`, используя метод `seekg()`. Подробности обнаружения символа-разделителя инкапсулируем в глобальную функцию `isLimit()`.

Осталось решить подзадачи:

- обнаружить конец предложения, то есть один из символов '.', '!', '?';
- если это вопросительное предложение, то вывести его в поток `cout`, в противном случае очистить поток `sentence` для накопления следующего предложения.

Рассмотренный алгоритм реализуется в следующем коде:

```
#include <fstream>
#include <sstream>
#include <string>
#include "CyrIOS.h" // for Visual C++ 6.0
using namespace std;

bool isLimit(char c) {
    char lim[] = { ' ', '\t', '\n' };

```

¹ Аналогичная задача рассмотрена в первой книге практикума (задача 5.3).


```

    for (int i = 0; i < sizeof(lim); ++i)
        if (c == lim[i]) return true;
    return false;
}

int main() {
    ifstream fin("infile.txt", ios::in|ios::nocreate);
    if (!fin) { cout << "Ошибка открытия файла." << endl; return 1; }

    int count = 0;
    string word;
    ostringstream sentence;
    while(!fin.eof()) {
        char symb;
        while(isLimit(symb = fin.peek())) {
            sentence << symb;
            if (symb == '\n') break;
            fin.seekg(1, ios::cur);
        }

        fin >> word;
        sentence << word;
        char last = word[word.size() - 1];
        if ((last == '.' || last == '!')) {
            sentence.str(""); // очистка потока
            continue;
        }
        if (last == '?') {
            cout << sentence.str();
            sentence.str("");
            count++;
        }
    }
    if (!count) cout << "Вопросительных предложений нет.";
    cout << endl;
    return 0;
}

```

Протестируйте приведенные программы. Не забудьте поместить в один каталог с программой текстовый файл `infile.txt`.

Давайте повторим наиболее важные моменты этого семинара.

1. Для поддержки файлового ввода и вывода стандартная библиотека C++ содержит классы `ifstream`, `ofstream`, `fstream`.
2. Работа с файлом предполагает следующие операции: создание потока, открытие потока и связывание его с файлом, обмен с потоком (ввод/вывод), закрытие файла.
3. Рекомендуется всегда проверять, чем завершилась попытка открытия файла.

4. Если в процессе ввода/вывода фиксируется ошибочная ситуация, то потоковый объект принимает значение, равное нулю.
5. Следите за состоянием выходного потока после каждой операции вывода, так как на диске может не оказаться свободного места.
6. Классы `istringstream`, `ostream`, `stringstream` обеспечивают работу со строковыми потоками. Использование строковых потоков аналогично применению файловых потоков. Отличие в том, что физически информация потока размещается в оперативной памяти, а не в файле на диске.
7. Класс `string` стандартной библиотеки C++ предоставляет программисту очень удобные средства работы со строками. Класс берет на себя управление памятью как при первоначальном размещении строки, так и при всех ее модификациях.

Задания

Вариант 1

Написать программу, которая считывает из текстового файла три предложения и выводит их в обратном порядке.

Вариант 2

Написать программу, которая считывает текст из файла и выводит на экран только предложения, содержащие заданное с клавиатуры слово.

Вариант 3

Написать программу, которая считывает текст из файла и выводит на экран только строки, содержащие двузначные числа.

Вариант 4

Написать программу, которая считывает английский текст из файла и выводит на экран слова, начинающиеся с гласных букв.

Вариант 5

Написать программу, которая считывает текст из файла и выводит его на экран, меняя местами каждые два соседних слова.

Вариант 6

Написать программу, которая считывает текст из файла и выводит на экран только предложения, не содержащие запятых.

Вариант 7

Написать программу, которая считывает текст из файла и определяет, сколько в нем слов, состоящих не более чем из четырех букв.

Вариант 8

Написать программу, которая считывает текст из файла и выводит на экран только цитаты, то есть предложения, заключенные в кавычки.

Вариант 9

Написать программу, которая считывает текст из файла и выводит на экран только предложения, состоящие из заданного количества слов.

Вариант 10

Написать программу, которая считывает английский текст из файла и выводит на экран слова текста, начинающиеся с гласных букв и оканчивающиеся гласными буквами.

Вариант 11

Написать программу, которая считывает текст из файла и выводит на экран только строки, не содержащие двузначные числа.

Вариант 12

Написать программу, которая считывает текст из файла и выводит на экран только предложения, начинающиеся с тире, перед которым могут следовать только пробельные символы.

Вариант 13

Написать программу, которая считывает английский текст из файла и выводит его на экран, заменив каждую первую букву слов, начинающихся с гласной буквы, на прописную.

Вариант 14

Написать программу, которая считывает текст из файла и выводит его на экран, заменив цифры от 0 до 9 на слова «ноль», «один», ..., «девять», начиная каждое предложение с новой строки.

Вариант 15

Написать программу, которая считывает текст из файла, находит самое длинное слово и определяет, сколько раз оно встретилось в тексте.

Вариант 16

Написать программу, которая считывает текст из файла и выводит на экран сначала вопросительные, а затем восклицательные предложения.

Вариант 17

Написать программу, которая считывает текст из файла и выводит его на экран, после каждого предложения добавляя, сколько раз встретилось в нем заданное с клавиатуры слово.

Вариант 18

Написать программу, которая считывает текст из файла и выводит на экран все его предложения в обратном порядке.

Вариант 19

Написать программу, которая считывает текст из файла и выводит на экран сначала предложения, начинающиеся с однобуквенных слов, а затем все остальные.

Вариант 20

Написать программу, которая считывает текст из файла и выводит на экран предложения, содержащие максимальное количество знаков пунктуации.

Семинар 6 Стандартная библиотека шаблонов

Теоретический материал: с. 295–368.

Основные концепции STL

Стандартная библиотека шаблонов STL¹ состоит из двух основных частей: набора контейнерных классов и набора обобщенных алгоритмов.

Контейнеры — это объекты, содержащие другие однотипные объекты. Контейнерные классы являются шаблонными, поэтому хранимые в них объекты могут быть как встроенных, так и пользовательских типов. Эти объекты должны допускать *копирование* и *присваивание*. Встроенные типы этим требованиям удовлетворяют; то же самое относится к классам, если конструктор копирования или операция присваивания не объявлены в них закрытыми или защищенными. В контейнерных классах реализованы такие типовые структуры данных, как стек, список, очередь и т. д.

Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам — например, поиск, сортировку, слияние и т. п. Однако они не являются методами контейнерных классов. Наоборот, алгоритмы представлены в STL в форме глобальных шаблонных функций. Благодаря этому достигается их универсальность: эти функции можно применять не только к объектам различных контейнерных классов, но также и к массивам. Независимость от типов контейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара адресов *first*, *last*, задающая диапазон обрабатываемых элементов.

Реализация указанного механизма взаимодействия базируется на использовании так называемых итераторов. *Итераторы* — это обобщение концепции указате-

¹ Standard Template Library.

лей: они ссылаются на элементы контейнера. Их можно инкрементировать, как обычные указатели, для последовательного продвижения по контейнеру, а также разыменовывать для получения или изменения значения элемента.

Контейнеры

Контейнеры STL можно разделить на два типа: последовательные и ассоциативные.

Последовательные контейнеры обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым последовательным контейнерам относятся *векторы* (vector), *списки* (list) и *двусторонние очереди* (deque). Есть еще специализированные контейнеры (или *адаптеры* контейнеров), реализованные на основе базовых — *стеки* (stack), *очереди* (queue) и *очереди с приоритетами* (priority_queue).

Между прочим, обычный встроенный массив C++ также может рассматриваться как последовательный контейнер. Проблема с массивами заключается в том, что их размеры нужно указывать в исходном коде, а часто бывает не известно заранее, сколько элементов придется хранить. Если же выделять память для массива динамически (оператором new), то алгоритм усложняется из-за необходимости отслеживать время жизни массива и вовремя освобождать память. Использование контейнера *вектор* вместо динамического массива резко упрощает жизнь программиста, в чем вы уже могли убедиться на семинаре 2.

Для использования контейнера в программе необходимо включить в нее соответствующий заголовочный файл. Тип объектов, сохраняемых в контейнере, задается с помощью аргумента шаблона, например:

```
vector<int> aVect; // создать вектор aVect целых чисел (типа int)
list<Man> department; // создать список department типа Man
```

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

Итераторы

Чтобы понять, зачем нужны итераторы, давайте посмотрим, как можно реализовать шаблонную функцию для поиска значения value в обычном массиве, хранящем объекты типа T. Например, возможно следующее решение:

```
template <class T> T* Find(T* ar, int n, const T& value) {
    for (int i = 0; i < n; ++i)
        if (ar[i] == value) return &ar[i];
    return 0;
}
```

Функция возвращает адрес найденного элемента или 0, если элемент с заданным значением не найден. Цикл `for` может быть записан и в несколько иной форме:

```
for (int i = 0; i < n; ++i)
    if (*(ar + i) == value) return ar + i;
```

Работа функции при этом останется прежней. Обратите внимание, что при продвижении по массиву адрес следующего элемента вычисляется и в первом, и во втором случаях с использованием арифметики указателей, то есть он отличается от адреса предыдущего элемента на некоторое фиксированное число байтов, требуемое для хранения одного элемента.

Попытаемся теперь расширить сферу применения нашей функции — хорошо бы, чтобы она решала задачу поиска заданного значения среди объектов, хранящихся в виде линейного списка! Однако, к сожалению, ничего не выйдет: адрес следующего элемента в списке нельзя вычислить, пользуясь арифметикой указателей. Элементы списка могут размещаться в памяти самым причудливым образом, а информация об адресе следующего объекта хранится в одном из полей текущего объекта.

Авторы STL решили эту проблему, введя понятие *итератора* как более абстрактной сущности, чем указатель, но обладающей похожим поведением. Для *всех* контейнерных классов STL определен тип `iterator`, однако реализация его в разных классах разная. Например, в классе `vec`, где объекты размещаются один за другим, как в массиве, тип итератора определяется посредством `typedef T* iterator`. А вот в классе `list` тип итератора реализован как встроенный класс `iterator`, поддерживающий основные операции с итераторами.

К *основным операциям*, выполняемым с любыми итераторами, относятся:

- Разыменованное итератора: если `p` — итератор, то `*p` — значение объекта, на который он ссылается.
- Присваивание одного итератора другому.
- Сравнение итераторов на равенство и неравенство (`==` и `!=`).
- Перемещение его по всем элементам контейнера с помощью префиксного (`++p`) или постфиксного (`p++`) инкремента.

Так как реализация итератора специфична для каждого класса, то при объявлении объектов типа итератор всегда указывается область видимости в форме `имя_шаблона::`, например:

```
vector<int>::iterator iter1;
list<Man>::iterator iter2;
```

Организация циклов просмотра элементов контейнеров тоже имеет некоторую специфику. Так, если `i` — некоторый итератор, то вместо привычной формы

```
for (i = 0; i < n; ++i)
```

используется следующая:

```
for (i = first; i != last; ++i)
```

где `first` — значение итератора, указывающее на первый элемент в контейнере, а `last` — значение итератора, указывающее на воображаемый элемент, который следует за последним элементом контейнера. Операция сравнения `<` здесь заменена на операцию `!=`, поскольку операции `<` и `>` для итераторов в общем случае не поддерживаются.

Для всех контейнерных классов определены унифицированные методы `begin()` и `end()`, возвращающие адреса `first` и `last` соответственно.

Вообще, все типы итераторов в STL принадлежат одной из пяти категорий: *входные*, *выходные*, *прямые*, *двунаправленные* итераторы и *итераторы произвольного доступа*.

Входные итераторы (*InputIterator*) используются алгоритмами STL для чтения значений из контейнера, аналогично тому, как программа может вводить данные из потока `cin`.

Выходные итераторы (*OutputIterator*) используются алгоритмами для записи значений в контейнер, аналогично тому, как программа может выводить данные в поток `cout`.

Прямые итераторы (*ForwardIterator*) используются алгоритмами для навигации по контейнеру только в прямом направлении, причем они позволяют и читать, и изменять данные в контейнере.

Двунаправленные итераторы (*BidirectionalIterator*) имеют все свойства прямых итераторов, но позволяют осуществлять навигацию по контейнеру и в прямом, и в обратном направлениях (для них дополнительно реализованы операции префиксного и постфиксного декремента).

Итераторы произвольного доступа (*RandomAccessIterator*) имеют все свойства двунаправленных итераторов плюс операции (наподобие сложения указателей) для доступа к произвольному элементу контейнера.

В то время как значения прямых, двунаправленных и итераторов произвольного доступа могут быть сохранены, значения входных и выходных итераторов сохраняться не могут (аналогично тому, как не может быть гарантирован ввод тех же самых значений при вторичном обращении к потоку `cin`). Следствием является то, что любые алгоритмы, базирующиеся на входных или выходных итераторах, должны быть однопроходными.

Вернемся к функции `find()`, которую мы безуспешно пытались обобщить для любых типов контейнеров. В STL аналогичный алгоритм имеет следующий прототип:

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Для двунаправленных итераторов и итераторов произвольного доступа определены разновидности, называемые *адаптерами итераторов*. Адаптер, просматривающий последовательность в обратном направлении, называется `reverse_iterator`. Другие специализированные итераторы-адаптеры мы рассмотрим ниже.

Общие свойства контейнеров

В табл. 6.1 приведены имена типов, определенные с помощью `typedef` в большинстве контейнерных классов.

Таблица 6.1. Унифицированные типы, определенные в STL

Поле	Пояснение
<code>value_type</code>	Тип элемента контейнера
<code>size_type</code> ¹	Тип индексов, счетчиков элементов и т. д.
<code>iterator</code>	Итератор
<code>const_iterator</code>	Константный итератор (значения элементов изменять запрещено)
<code>reference</code>	Ссылка на элемент
<code>const_reference</code>	Константная ссылка на элемент (значение элемента изменять запрещено)
<code>key_type</code>	Тип ключа (для ассоциативных контейнеров)
<code>key_compare</code>	Тип критерия сравнения (для ассоциативных контейнеров)

В табл. 6.2 представлены некоторые общие для всех контейнеров операции.

Таблица 6.2. Операции и методы, общие для всех контейнеров

Операция или метод	Пояснение
Операции равенства (<code>==</code>) и неравенства (<code>!=</code>)	Возвращают значение <code>true</code> или <code>false</code>
Операция присваивания (<code>=</code>)	Копирует один контейнер в другой
<code>clear</code>	Удаляет все элементы
<code>insert</code>	Добавляет один элемент или диапазон элементов
<code>erase</code>	Удаляет один элемент или диапазон элементов
<code>size_type size() const</code>	Возвращает число элементов
<code>size_type max_size() const</code>	Возвращает максимально допустимый размер контейнера
<code>bool empty() const</code>	Возвращает <code>true</code> , если контейнер пуст
<code>iterator begin()</code>	Возвращают итератор на начало контейнера (итерации будут производиться в прямом направлении)
<code>iterator end()</code>	Возвращают итератор на конец контейнера (итерации в прямом направлении будут закончены)
<code>reverse_iterator begin()</code>	Возвращают реверсивный итератор на конец контейнера (итерации будут производиться в обратном направлении)
<code>reverse_iterator end()</code>	Возвращают реверсивный итератор на начало контейнера (итерации в обратном направлении будут закончены)

¹ Эквивалентен `unsigned int`.

Алгоритмы

Алгоритм — это функция, которая производит некоторые действия над элементами контейнера (контейнеров). Чтобы использовать обобщенные алгоритмы, нужно подключить к программе заголовочный файл `<algorithm>`.

В табл. 6.3 приведены наиболее популярные алгоритмы STL¹.

Таблица 6.3. Некоторые типичные алгоритмы STL

Алгоритм	Назначение
<code>accumulate</code>	Вычисление суммы элементов в заданном диапазоне
<code>copy</code>	Копирование последовательности, начиная с первого элемента
<code>count</code>	Подсчет количества вхождений значения в последовательность
<code>count_if</code>	Подсчет количества выполнений условия в последовательности
<code>equal</code>	Попарное равенство элементов двух последовательностей
<code>fill</code>	Замена всех элементов заданным значением
<code>find</code>	Нахождение первого вхождения значения в последовательность
<code>find_first_of</code>	Нахождение первого значения из одной последовательности в другой
<code>find_if</code>	Нахождение первого соответствия условию в последовательности
<code>for_each</code>	Вызов функции для каждого элемента последовательности
<code>merge</code>	Слияние отсортированных последовательностей
<code>remove</code>	Перемещение элементов с заданным значением
<code>replace</code>	Замена элементов с заданным значением
<code>search</code>	Нахождение первого вхождения в первую последовательность второй последовательности
<code>sort</code>	Сортировка
<code>swap</code>	Обмен двух элементов
<code>transform</code>	Выполнение заданной операции над каждым элементом последовательности

В списках параметров всех алгоритмов первые два параметра задают диапазон обрабатываемых элементов в виде полуинтервала $[first, last)^2$, где `first` — итератор, указывающий на начало диапазона, а `last` — итератор, указывающий на выход за границы диапазона.

Например, если имеется массив

```
int arr[7] = {15, 2, 19, -3, 28, 6, 8};
```

то его можно отсортировать с помощью алгоритма `sort`:

```
sort(arr, arr + 7);
```

¹ Более подробное описание обобщенных алгоритмов см. в учебнике.

² Полуинтервал $[a, b)$ — это промежуток, включающий a , но не включающий b — последний элемент полуинтервала предшествует элементу b .

Здесь имя массива `arr` имеет тип указателя `int*` и используется как итератор.

Примеры использования некоторых алгоритмов будут даны ниже.

Использование последовательных контейнеров

К основным последовательным контейнерам относятся *вектор* (`vector`), *список* (`list`) и *двусторонняя очередь* (`deque`).

Чтобы использовать последовательный контейнер, нужно включить в программу соответствующий заголовочный файл:

```
#include <vector>
#include <list>
#include <deque>
using namespace std;
```

Контейнер *вектор* является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации `[]` или метода `at`¹. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

Контейнер *список* организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки *n*-го элемента нужно последовательно выбрать предыдущие *n*-1 элементов.

Контейнер *двусторонняя очередь* (*дек*) во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора двусторонняя очередь эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

Существует пять способов определить объект для последовательного контейнера.

1. Создать пустой контейнер:

```
vector<int> vec1;
list<string> list1;
```

2. Создать контейнер заданного размера и инициализировать его элементы значениями по умолчанию:

```
vector<string> vec1(100);
list<double> list1(20);
```

¹ Метод `at()` аналогичен операции индексации, но в отличие от последней проверяет выход за границу вектора, и если такое нарушение обнаруживается, то метод генерирует исключение `out_of_range`.

3. Создать контейнер заданного размера и инициализировать его элементы указанным значением:

```
vector<string> vec1(100, "Hello!");
deque<int> dec1(300, -1);
```

4. Создать контейнер и инициализировать его элементы значениями диапазона [first, last) элементов другого контейнера:

```
int arr[7] = {15, 2, 19, -3, 28, 6, 8};
vector<int> v1(arr, arr + 7);
list<int> lst(v1.begin() + 2, v1.end());1
```

5. Создать контейнер и инициализировать его элементы значениями элементов другого *однотипного* контейнера:

```
vector<int> v1;
// добавить в v1 элементы
vector<int> v2(v1);
```

6. Для вставки и удаления последнего элемента контейнера любого из трех рассматриваемых классов предназначены методы `push_back()` и `pop_back()`. Кроме того, список и очередь (но не вектор) поддерживают операции вставки и удаления первого элемента контейнера `push_front()` и `pop_front()`. Учтите, что методы `pop_back()` и `pop_front()` *не возвращают* удаленное значение. Чтобы считать первый элемент, используется метод `front()`, а для считывания последнего элемента — метод `back()`.

Кроме этого, все типы контейнеров имеют более общие операции вставки и удаления, перечисленные в табл. 6.4.

Таблица 6.4. Методы `insert()` и `erase()`

Метод	Пояснение
<code>insert(iterator position, const T& value)</code>	Вставка элемента со значением <code>value</code> в позицию, заданную итератором <code>position</code>
<code>insert(iterator position, size_type n, const T& value)</code>	Вставка <code>n</code> элементов со значением <code>value</code> , начиная с позиции <code>position</code>
<code>template <class InputIter> void insert(iterator position, InputIter first, InputIter last)</code>	Вставка диапазона элементов, заданного итераторами <code>first</code> и <code>last</code> , начиная с позиции <code>position</code>
<code>erase(iterator position)</code>	Удаление элемента, на который указывает итератор <code>position</code>
<code>erase(iterator first, iterator last)</code>	Удаление диапазона элементов, заданного позициями <code>first</code> и <code>last</code>

¹ К сожалению, компилятор Microsoft Visual C++ 6.0 не поддерживает инициализацию *двусторонней очереди* диапазоном элементов *контейнера другого типа*, то есть определение

```
deque<int> dec(v1.begin(), v1.end())
```

Задача 6.1. Сортировка вектора

В файле находится произвольное количество целых чисел. Вывести их на экран в порядке возрастания.

Программа считывает числа в вектор, сортирует по возрастанию и выводит на экран:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    ifstream in ("innum.txt");
    if (!in) { cerr << "File not found\n"; exit(1); }
    vector<int> v;
    int x;
    while (in >> x) v.push_back(x);
    sort(v.begin(), v.end());
    vector<int>::const_iterator i;
    for (i = v.begin(); i != v.end(); ++i)
        cout << *i << " ";
    return 0;
}
```

В данном примере вместо вектора можно было использовать любой последовательный контейнер путем простой замены слова `vector` на `deque` или `list`. При этом изменилось бы внутреннее представление данных, но результат работы программы остался бы таким же.

Приведем еще один пример работы с векторами, демонстрирующий использование методов `swap()`, `empty()`, `back()`, `pop_back()`:

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    double arr[] = {1.1, 2.2, 3.3, 4.4 };
    int n = sizeof(arr)/sizeof(double);
    // Инициализация вектора массивом
    vector<double> v1(arr, arr + n);
    vector<double> v2; // пустой вектор

    v1.swap(v2); // обменять содержимое v1 и v2

    while (!v2.empty()) {
        cout << v2.back() << ' '; // вывести последний элемент
    }
}
```

```

        v2.pop_back(); // и удалить его
    }
    return 0;
}

```

Результат выполнения программы:

```
4.4 3.3 2.2 1.1
```

Шаблонная функция print() для вывода содержимого контейнера

В процессе работы над программами, использующими контейнеры, часто приходится выводить на экран их текущее содержимое. Приведем шаблон функции, решающей эту задачу для любого типа контейнера:

```

template <class T> void print(T& cont) {
    typename T::const_iterator p = cont.begin();
    if (cont.empty())
        cout << "Container is empty.";
    for (p; p != cont.end(); ++p)
        cout << *p << ' ';
    cout << endl;
}

```

Обратите внимание на служебное слово `typename`, с которого начинается объявление итератора `p`. Дело в том, что библиотека STL «знает», что `T::iterator` — это некоторый тип, а компилятор C++ таким знанием не обладает. Поэтому без `typename` *нормальные компиляторы*¹ фиксируют ошибку.

Теперь можно пользоваться функцией `print()`, включая ее определение в исходный файл с программой, как, например, в следующем эксперименте с очередью:

```

#include <iostream>
#include <deque>
using namespace std;
/* ... определение функции print ... */
int main() {
    deque<int> dec;    print(dec); // Container is empty
    dec.push_back(4); print(dec); // 4
    dec.push_front(3); print(dec); // 3 4
    dec.push_back(5); print(dec); // 3 4 5
    dec.push_front(2); print(dec); // 2 3 4 5
    dec.push_back(6); print(dec); // 2 3 4 5 6
    dec.push_front(1); print(dec); // 1 2 3 4 5 6
    return 0;
}

```

¹ Компилятору Microsoft Visual C++ 6.0 все равно — есть в данном контексте `typename` или нет.

Адаптеры контейнеров

Специализированные последовательные контейнеры — *стек*, *очередь* и *очередь с приоритетами* — не являются самостоятельными контейнерными классами, а реализованы на основе рассмотренных выше классов, поэтому они называются *адаптерами контейнеров*.

Стек

Шаблонный класс `stack` (заголовочный файл `<stack>`) определен как

```
template <class T, class Container = deque<T> >
class stack { /* ... */ };
```

где параметр `Container` задает класс-прототип. По умолчанию для стека прототипом является класс `deque`. Смысл такой реализации заключается в том, что специализированный класс просто переопределяет интерфейс класса-прототипа, ограничивая его только теми методами, которые нужны новому классу. В табл. 6.5 показано, как сформирован интерфейс класса `stack` из методов класса-прототипа.

Таблица 6.5. Интерфейс класса `stack`

Методы класса <code>stack</code>	Методы класса-прототипа
<code>push()</code>	<code>push_back()</code>
<code>pop()</code>	<code>pop_back()</code>
<code>top()</code>	<code>back()</code>
<code>empty()</code>	<code>empty()</code>
<code>size()</code>	<code>size()</code>

В соответствии со своим назначением стек не только не позволяет выполнить произвольный доступ к своим элементам, но даже не дает возможности пошагового перемещения, в связи с чем концепция итераторов в стеке не поддерживается. Напоминаем, что метод `pop()` *не возвращает* удаленное значение. Чтобы считать значение на вершине стека, используется метод `top()`.

Пример работы со стеком — программа вводит из файла числа и выводит их на экран в обратном порядке¹:

```
int main() {
    ifstream in ("inpnun.txt");
    stack<int> s;
    int x;
    while (in >> x) s.push(x);

    while (!s.empty()) {
        cout << s.top() << ' ';
        s.pop();
    }
}
```

¹ В дальнейших примерах мы будем опускать директивы `#include` и объявление `using namespace std`, полагая, что они очевидны.

```

    }
    return 0;
}

```

Объявление `stack<int> s` создает стек на базе двусторонней очереди (по умолчанию). Если по каким-то причинам нас это не устраивает и мы хотим создать стек на базе списка, то объявление будет выглядеть следующим образом:

```
stack<int, list<int> > s;1
```

Очередь

Шаблонный класс `queue` (заголовочный файл `<queue>`) является адаптером, который может быть реализован на основе двусторонней очереди (реализация по умолчанию) или списка. Класс `vector` в качестве класса-прототипа не подходит, поскольку в нем нет выборки из начала контейнера. Очередь использует для проталкивания данных один конец, а для выталкивания — другой. В соответствии с этим ее интерфейс образуют методы, представленные в табл. 6.6.

Таблица 6.6. Интерфейс класса `queue`

Методы класса <code>queue</code>	Методы класса-прототипа
<code>push()</code>	<code>push_back()</code>
<code>pop()</code>	<code>pop_front()</code>
<code>front()</code>	<code>front()</code>
<code>back()</code>	<code>back()</code>
<code>empty()</code>	<code>empty()</code>
<code>size()</code>	<code>size()</code>

Очередь с приоритетами

Шаблонный класс `priority_queue` (заголовочный файл `<queue>`) поддерживает такие же операции, как и класс `queue`, но реализация класса возможна либо на основе вектора (реализация по умолчанию), либо на основе списка. Очередь с приоритетами отличается от обычной очереди тем, что для извлечения выбирается максимальный элемент из хранимых в контейнере. Поэтому после каждого изменения состояния очереди максимальный элемент из оставшихся сдвигается в начало контейнера. Если очередь с приоритетами организуется для объектов класса, определенного программистом, то в этом классе должна быть определена операция `<`.

Пример работы с очередью с приоритетами:

```

int main() {
    priority_queue<int> P;
    P.push(17); P.push(5); P.push(400);
    P.push(2500); P.push(1);
    while (!P.empty()) {
        cout << P.top() << ' ';
        P.pop();
    }
}

```

¹ Не забывайте ставить пробел между угловыми скобками `> > !`


```

    }
    return 0;
}

```

Результат выполнения программы:

```
2500 400 17 5 1
```

Использование алгоритмов

Вернемся к изучению алгоритмов. Не забывайте включать заголовочный файл `<algorithm>` и добавлять определение нашей функции `print()`, если она используется.

Алгоритмы `count` и `find`

Алгоритм `count` подсчитывает количество вхождений в контейнер (или его часть) значения, заданного его третьим аргументом. Алгоритм `find` выполняет поиск заданного значения и возвращает итератор на самое первое вхождение этого значения. Если значение не найдено, то возвращается итератор, соответствующий возврату метода `end()`. В следующей программе показано использование этих алгоритмов.

```

int main() {
    int arr[] = {1, 2, 3, 4, 5, 2, 6, 2, 7};
    int n = sizeof(arr) / sizeof(int);
    vector<int> v1(arr, arr + n);
    int value = 2;           // искомая величина

    int how_much = count(v1.begin(), v1.end(), value);
    cout << "how_much << endl; // вывод: 3

    list<int> loc_list;     // список позиций искомой величины
    vector<int>::iterator location = v1.begin();
    while (1) {
        location = find(location, v1.end(), value);
        if (location == v1.end()) break;
        loc_list.push_back(location - v1.begin());
        location++;
    }
    print(loc_list);       // вывод: 1 5 7
    return 0;
}

```

В приведенной программе создается вектор `v1`, наполняясь при инициализации значениями из массива `arr`. Затем с помощью алгоритма `count` подсчитывается количество вхождений в вектор значения `value`, равного двум. В цикле `while` выясняется, на каких позициях в векторе размещена эта величина. Обратите внимание на то, что первый аргумент алгоритма `find` (переменная `location`) первоначально — перед входом в цикл — принимает значение итератора, указывающего на нулевой¹ элемент контейнера. Затем `location` получает значение итератора, указывающего на найденный элемент.

¹ Нумерация элементов в векторе, так же как и в массиве, начинается с нуля.

Если поиск завершился успешно, то, во-первых, вычисляется позиция найденного элемента как разность значений `location` и адреса нулевого элемента и полученное значение заносится в список `loc_list`. Во-вторых, итератор `location` сдвигается операцией инкремента на следующую позицию в контейнере, чтобы обеспечить (на следующей итерации цикла) продолжение поиска в оставшейся части контейнера. Если поиск завершился неудачей, то `break` приведет к выходу из цикла.

Алгоритмы `count_if` и `find_if`

Алгоритмы `count_if` и `find_if` отличаются от алгоритмов `count` и `find` тем, что в качестве третьего аргумента они требуют некоторый предикат. *Предикат* — это функция или функциональный объект, возвращающие значение типа `bool`. Например, если в предыдущей программе добавить определение глобальной функции

```
bool isMyValue(int x) { return ((x > 2) && (x < 5)); }
```

и заменить инструкцию с вызовом `count` на

```
int how_much = count_if(v1.begin(), v1.end(), isMyValue);
```

то программа определит, что контейнер содержит два числа, значение которых больше двух, но меньше пяти.

Аналогично, замена инструкции с вызовом `find` на

```
location = find_if(location, v1.end(), isMyValue);
```

будет иметь следствием наполнение списка `loc_list` двумя значениями: 2 и 3 (номера позиций вектора `v1`, на которых находятся числа, удовлетворяющие предикату `isMyValue`).

Алгоритм `for_each`

Этот алгоритм позволяет выполнить некоторое действие над каждым элементом диапазона `[first, last)`. Чтобы определить, какое именно действие должно быть выполнено, нужно написать соответствующую функцию с одним аргументом типа `T` (`T` — тип данных, содержащихся в контейнере). Функция не имеет права модифицировать данные в контейнере, но может их использовать в своей работе. Имя этой функции передается в качестве третьего аргумента алгоритма. Например, в следующей программе `for_each` используется для перевода всех значений массива из дюймов в сантиметры и вывода их на экран.

```
void InchToCm(double inch) {  
    cout << (inch * 2.54) << ' ';
```

```
}  
  
int main() {  
    double inches[] = {0.5, 1.0, 1.5, 2.0, 2.5};  
    for_each(inches, inches + 5, InchToCm);  
    return 0;  
}
```

Алгоритм search

Некоторые алгоритмы оперируют одновременно двумя контейнерами. Таков и алгоритм `search`, находящий первое вхождение в первую последовательность `[first1, last1)` второй последовательности `[first2, last2)`. Например:

```
int main() {
    int arr[] = {11, 77, 33, 11, 22, 33, 11, 22, 55};
    int pattern[] = { 11, 22, 33 };

    int* ptr = search(arr, arr + 9, pattern, pattern + 3);
    if (ptr == arr + 9)
        cout << "Pattern not found" << endl;
    else
        cout << "Found at position " << (ptr - arr) << endl;

    list<int> lst(arr, arr + 9);
    list<int>::iterator ifound;
    ifound = search(lst.begin(), lst.end(), pattern, pattern + 3);
    if (ifound == lst.end())
        cout << "Pattern not found" << endl;
    else
        cout << "Found." << endl;
    return 0;
}
```

Результат выполнения программы:

```
Found at position 3
Found.
```

Отметим, что список не поддерживает произвольного доступа к своим элементам и соответственно не допускает операций «+» и «-» с итераторами. Поэтому мы можем только зафиксировать факт вхождения последовательности `pattern` в контейнер `lst`.

Алгоритм sort

Назначение алгоритма очевидно из его названия. Алгоритм можно применять только для тех контейнеров, которые обеспечивают произвольный доступ к элементам, — этому требованию удовлетворяют *массив*, *вектор* и *двусторонняя очередь*, но не удовлетворяет *список*. В связи с этим класс `list` содержит метод `sort()`, решающий задачу сортировки.

Алгоритм `sort` имеет две сигнатуры:

```
template<class RandomAccessIt> void sort(RandomAccessIt first, RandomAccessIt last);
template<class RandomAccessIt> void sort(RandomAccessIt first, RandomAccessIt last,
Compare comp);
```

Первая форма алгоритма обеспечивает сортировку элементов из диапазона `[first, last)`, причем для упорядочения по умолчанию используется операция `<`, которая

должна быть определена для типа T^1 . Таким образом, сортировка по умолчанию — это сортировка по возрастанию значений. Например:

```
int main() {
    double arr[6] = {2.2, 0.0, 4.4, 1.1, 3.3, 1.1};
    vector<double> v1(arr, arr + 6);
    sort(v1.begin(), v1.end());
    print(v1);
    return 0;
}
```

Результат выполнения программы:

```
0 1.1 1.1 2.2 3.3 4.4
```

Вторая форма алгоритма `sort` позволяет задать произвольный критерий упорядочения. Для этого нужно передать через третий аргумент соответствующий предикат, то есть функцию или функциональный объект, возвращающие значение типа `bool`. Использование функции в качестве предиката было показано выше². Использованию функциональных объектов посвящен следующий раздел.

Функциональные объекты

На семинаре 3 было показано, как можно использовать функциональные объекты для настройки шаблонных классов, поэтому рекомендуем вам еще раз просмотреть этот материал. *Функциональным объектом* называется объект некоторого класса, для которого определена единственная операция вызова функции `operator()`.

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения, встроенных в язык C++. Они возвращают значение типа `bool`, то есть являются предикатами (табл. 6.7).

Таблица 6.7. Предикаты стандартной библиотеки

Операция	Эквивалентный предикат (функциональный объект)
<code>==</code>	<code>equal_to</code>
<code>!=</code>	<code>not_equal_to</code>
<code>></code>	<code>greater</code>
<code><</code>	<code>less</code>
<code>>=</code>	<code>greater_equal</code>
<code><=</code>	<code>less_equal</code>

Очевидно, что при подстановке в качестве аргумента алгоритма требуется инстанцирование этих шаблонов, например: `equal_to<int>()`.

Вернемся к последней программе, где с помощью алгоритма `sort` был отсортирован вектор `v1`. Заменяем вызов `sort` на следующий:

```
sort(v1.begin(), v1.end(), greater<double>());
```

¹ T — тип данных, содержащихся в контейнере.

² См. описание алгоритмов `count_if` и `find_if`.

В результате вектор будет отсортирован по убыванию значений его элементов. Несколько сложнее обстоит дело, когда сортировка выполняется для контейнера с объектами пользовательского класса. В этом случае программисту нужно самому позаботиться о наличии в классе предиката, задающего сортировку по умолчанию, а также (при необходимости) определить функциональные классы, объекты которых позволяют изменять настройку алгоритма `sort`.

В приведенной ниже программе показаны варианты вызова алгоритма `sort` для вектора `men`, содержащего объекты класса `Man`. В классе `Man` определен предикат — операция `operator<()`, — благодаря которому сортировка по умолчанию будет происходить по возрастанию значений поля `name`. Кроме этого, в программе определен функциональный класс `LessAge`, использование которого позволяет осуществить сортировку по возрастанию значений поля `age`.

```
class Man {
public:
    Man (string _name, int _age) :
        name(_name), age(_age) {}
    // предикат, задающий сортировку по умолчанию
    bool operator< (const Man& m) const {
        return name < m.name;
    }
    friend ostream& operator<< (ostream&, const Man&);
    friend struct LessAge;
private:
    string name;
    int age;
};

ostream& operator<<(ostream& os, const Man& m) {
    return os << endl << m.name << ",\t age: " << m.age;
}

// Функциональный класс для сравнения по возрасту
struct LessAge {
    bool operator() (const Man& a, const Man& b) {
        return a.age < b.age;
    }
};

int main() {
    Man ar[] = {
        Man("Mary Poppins", 36),
        Man("Count Basie", 70),
        Man("Duke Ellington", 90),
        Man("Joy Amore", 18)
    };
    int size = sizeof(ar) / sizeof(Man);
    vector<Man> men(ar, ar + size);
```

```

// Сортировка по имени (по умолчанию)
sort(men.begin(), men.end());
print(men);
// Сортировка по возрасту
sort(men.begin(), men.end(), LessAge());
print(men);
return 0;
}

```

Обратные итераторы

Эта разновидность итераторов (`reverse_iterator`) очень удобна для прохода по контейнеру от конца к началу. Например, если в программе имеется контейнер `vector<double> v1`, то для вывода содержимого вектора в обратном порядке можно написать:

```

vector<double>::reverse_iterator ri;
ri = v1.rbegin();
while (ri != v1.rend())
    cout << *ri++ << ' ';

```

Обратите внимание на то, что операция инкремента для такого итератора перемещает указатель на предыдущий элемент контейнера.

Итераторы вставки и алгоритм `soru`

Мы можем использовать алгоритм `soru` для копирования элементов одного контейнера в другой, причем источником может быть, например, вектор, а приемником — список, как показывает следующая программа:

```

int main() {
    int a[4] = {10, 20, 30, 40};
    vector<int> v(a, a + 4);
    list<int> L(4); // список из 4 элементов
    copy(v.begin(), v.end(), L.begin());
    print(L);
    return 0;
}

```

Алгоритм `soru` при таком использовании, как в этом примере, работает в *режиме замещения*. Это означает, что i -й элемент контейнера-источника замещает i -й элемент контейнера-приемника¹. Однако этот же алгоритм может работать и в *режиме вставки*, если в качестве третьего аргумента использовать так называемый *итератор вставки*.

Итераторы вставки `front_inserter()`, `back_inserter()`, `inserter()` предназначены для добавления новых элементов в начало, конец или произвольное место контейнера.

¹ Это напоминает режим замещения при вводе текста с клавиатуры в текстовом редакторе.

Покажем использование этих итераторов на следующем примере.

```
int main() {
    int a[4] = {40, 30, 20, 10};
    vector<int> va(a, a + 4);
    int b[3] = {80, 90, 100};
    vector<int> vb(b, b + 3);
    int c[3] = {50, 60, 70};
    vector<int> vc(c, c + 3);

    list<int> L;    // пустой список

    copy(va.begin(), va.end(), front_inserter(L));
    print(L);
    copy(vb.begin(), vb.end(), back_inserter(L));
    print(L);
    list<int>::iterator from = L.begin();
    advance(from, 4);
    copy(vc.begin(), vc.end(), inserter(L, from));
    print(L);
    return 0;
}
```

Результат выполнения программы:

```
10 20 30 40
10 20 30 40 80 90 100
10 20 30 40 50 60 70 80 90 100
```

Обратите внимание на следующие моменты:

- ❑ Первый вызов функции `copy` осуществляет копирование (вставку) вектора `va` в список `L`, причем итератор вставки `front_inserter` обеспечивает размещение очередного элемента вектора `va` в начале списка — поэтому порядок элементов в списке изменяется на обратный.
- ❑ Второй вызов `copy` пересылает элементы вектора `vb` в конец списка `L` благодаря итератору вставки `back_inserter`, поэтому порядок копируемых элементов не меняется.
- ❑ Третий вызов `copy` копирует вектор `vc` в заданное итератором `from` место списка `L`, а именно после четвертого элемента списка. Чтобы определить нужное значение итератора `from`, мы предварительно устанавливаем его в начало списка, а затем обеспечиваем приращение на 4 — вызовом функции `advance()`.

Алгоритм merge

Алгоритм `merge` выполняет слияние отсортированных последовательностей для любого типа последовательного контейнера, более того — все три участника алгоритма могут представлять различные контейнерные типы. Например, вектор `a` и массив `b` могут быть слиты в список `c`:

```
int main() {
    int arr[5] = {2, 3, 8, 20, 25};
    vector<int> a(arr, arr + 5);
    int b[6] = {7, 9, 23, 28, 30, 33};
    list<int> c;    // Список с сначала пуст
    merge(a.begin(), a.end(), b, b + 6, back_inserter(c));
    print(c);
    return 0;
}
```

Результат выполнения программы:

2 3 7 8 9 20 23 25 28 30 33

Использование ассоциативных контейнеров

В ассоциативных контейнерах элементы не выстроены в линейную последовательность. Они организованы в более сложные структуры, что дает большой выигрыш в скорости поиска. Поиск производится с помощью *ключей*, обычно представляющих собой одно числовое или строковое значение. Рассмотрим две основные категории ассоциативных контейнеров в STL: множества и словари¹.

В *множестве* (set) хранятся объекты, упорядоченные по некоторому ключу, являющемуся атрибутом самого объекта. Например, множество может хранить объекты класса Map, упорядоченные в алфавитном порядке по значению ключевого поля name. Если в множестве хранятся значения одного из встроженных типов, например int, то ключом является сам элемент.

Словарь (map) можно представить себе как своего рода таблицу из двух столбцов, в первом из которых хранятся объекты, содержащие ключи, а во втором — объекты, содержащие значения. Похожая организация данных рассматривалась нами в задаче 3.1 (шаблонный класс для разреженных массивов).

И в множествах, и в словарях все ключи являются уникальными (только одно значение соответствует ключу). *Мультимножества* (multiset) и *мультисловари* (multimap) аналогичны своим родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

Ассоциативные контейнеры имеют много общих методов с последовательными контейнерами. Тем не менее некоторые методы, а также алгоритмы характерны только для них.

Множества

Шаблон множества имеет два параметра: тип ключа и тип функционального объекта, определяющего отношение «меньше»:

¹ Английский термин map переводится в литературе по C++ либо как *словарь*, либо как *отображение*.


```
template <class Key, class Compare = less<Key> >
class set{ /* ... */ };
```

Таким образом, если объявить некоторое множество `set<int> s1` с опущенным вторым параметром шаблона, то по умолчанию для упорядочения членов множества будет использован предикат `less<int>`. Точно так же можно опустить второй параметр при объявлении множества `set<MyClass> s2`, если в классе `MyClass` определена операция `operator<>`.

Для использования контейнеров типа `set` необходимо подключить заголовочный файл `<set>`.

Имеется три способа определить объект типа `set`:

```
set<int> set1;           // создается пустое множество
int a[5] = { 1, 2, 3, 4, 5 };
set<int> set2(a, a + 5); // инициализация копированием массива
set<int> set3(set2);    // инициализация другим множеством
```

Для вставки элементов в множество можно использовать метод `insert()`, для удаления — метод `erase()`. Также к множествам применимы общие для всех контейнеров методы, указанные в табл. 6.2.

Во всех ассоциативных контейнерах есть метод `count()`, возвращающий количество объектов с заданным ключом. Так как и в множествах, и в словарях все ключи уникальны, то метод `count()` возвращает либо 0, если элемент не обнаружен, либо 1.

Для множеств библиотека содержит некоторые специальные алгоритмы, в частности, реализующие традиционные теоретико-множественные операции. Эти алгоритмы перечислены ниже.

Алгоритм `includes` выполняет проверку включения одной последовательности в другую. Результат равен `true` в том случае, когда каждый элемент последовательности `[first2, last2)` содержится в последовательности `[first1, last1)`.

Алгоритм `set_intersection` создает отсортированное *пересечение множеств*, то есть множество, содержащее только те элементы, которые одновременно входят и в первое, и во второе множество.

Алгоритм `set_union` создает отсортированное *объединение множеств*, то есть множество, содержащее элементы первого и второго множества без повторяющихся элементов.

В следующей программе показано использование этих алгоритмов:

```
int main() {
    const int N = 5;
    string s1[N] = {"Bill", "Jessica", "Ben", "Mary", "Monica"};
    string s2[N] = {"Sju", "Monica", "John", "Bill", "Sju"};
    typedef set<string> SetS;
    SetS A(s1, s1 + N);
    SetS B(s2, s2 + N);
    print(A); print(B);
}
```

```

SetS prod, sum;
set_intersection(A.begin(), A.end(), B.begin(), B.end(),
    inserter(prod, prod.begin()));
print(prod);
set_union(A.begin(), A.end(), B.begin(), B.end(),
    inserter(sum, sum.begin()));
print(sum);

if (includes(A.begin(), A.end(), prod.begin(), prod.end()))
    cout << "Yes" << endl;
else cout << "No" << endl;
return 0;
}

```

Результат выполнения программы:

```

Ben Bill Jessica Mary Monica
Bill John Monica Sju
Bill Monica
Ben Bill Jessica John Mary Monica Sju
Yes

```

ВНИМАНИЕ

Если вы работаете с компилятором Microsoft Visual C++ 6.0, то в случае использования *множеств* или *словарей* добавляйте в начало программы директиву `#pragma warning (disable:4786)`, которая подавляет вывод предупреждений компилятора типа «идентификатор был усечен до 255 символов в отладочной информации»¹.

Словари

В определении класса `map` используется тип `pair`, который описан в заголовочном файле `<utility>` следующим образом:

```

template <class T1, class T2> struct pair{
    T1 first;
    T2 second;
    pair(const T1& x, const T2& y):
        ...
};

```

¹ Непонятны причины, по которым разработчики компилятора Visual C++ 6.0 решили вводить это сообщение, но для только что рассмотренного примера компилятор выводит 86 (!) предупреждений, *каждое из них* примерно следующего содержания:

```

warning C4786: 'std:: Tree<std::basic_string<char, std::char_traits<char>,
std::allocator<char> >, std::basic_string<char, std::char_traits<char>, std::allocator<char> >,
std::set<std::basic_string<char, std::char_traits<char>, std::allocator<char> >,
std::less<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >,
std::allocator<std::basic_string<char, std::char_traits<char>, std::allocator<char> > > >::_Kfn,
std::less<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >,
std::allocator<std::basic_string<char, std::char_traits<char>, std::allocator<char> > > >':
identifier was truncated to '255' characters in the debug information

```

Шаблон `pair` имеет два параметра, представляющих собой типы элементов пары. Первый элемент пары имеет имя `first`, второй — `second`. В этом же файле определены шаблонные операции `==`, `!=`, `<`, `>`, `<=`, `>=` для двух объектов типа `pair`.

Шаблон словаря имеет три параметра: тип ключа, тип элемента и тип функционального объекта, определяющего отношение «меньше»:

```
template <class Key, class T, class Compare = less<Key> >
class map {
public:
    typedef pair <const Key, T> value_type;
    explicit map(const Compare& comp = Compare());
    map(const value_type* first, const value_type* last,
        const Compare& comp = Compare());
    map(const map <Key, T, Compare>& x);
};
```

Обратите внимание на то, что тип элементов словаря `value_type` определяется как пара элементов типа `Key` и `T`.

Первый конструктор класса `map` создает пустой словарь. Второй — создает словарь и записывает в него элементы, определяемые диапазоном `[first, last)`. Третий конструктор является конструктором копирования.

Для доступа к элементам по ключу определена операция `[]`:

```
T& operator[](const Key & x);
```

С помощью нее можно не только получать значения элементов, но и добавлять в словарь новые.

Для использования контейнеров типа `map` необходимо подключить заголовочный файл `<map>`.

Задача 6.2. Формирование частотного словаря

Написать программу формирования частотного словаря появления отдельных слов в некотором тексте. Исходный текст читается из файла `prose.txt`, результат — частотный словарь — записывается в файл `freq_map.txt`.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <map>
#include <set>
#include <string>
using namespace std;
```

```
int main() {
    char punct[6] = {'.', ',', '?', '!', ':', ';'};
```

```

set<char> punctuation(punct, punct + 6);
ifstream in("prose.txt");
if (!in) { cerr << "File not found\n"; exit(1); }

map<string, int> wordCount;
string s;
while (in >> s) {
    int n = s.size();
    if (punctuation.count(s[n - 1]))
        s.erase(n - 1, n);
    ++wordCount[s];
}
ofstream out("freq_map.txt");
map<string, int>::const_iterator it = wordCount.begin();
for (it; it != wordCount.end(); ++it)
    out << setw(20) << left << it->first
        << setw(4) << right << it->second << endl;
return 0;
}

```

Определяя в этой программе объект `wordCount` как словарь типа `map<string, int>`, мы тем самым показываем наше намерение связать каждое прочитанное слово с целочисленным счетчиком.

В цикле `while` разворачиваются следующие события:

- ❑ В строку `s` пословно считываются данные из входного файла.
- ❑ Определяется длина `n` строки `s`.
- ❑ С помощью метода `count()` проверяется, принадлежит ли последний символ строки `s` множеству `punctuation`, содержащему знаки препинания, которыми может завершаться слово. Если да, то последний символ удаляется из строки (метод `erase()`).
- ❑ Заслуживает особого внимания лаконичная инструкция `++wordCount[s]`. Здесь мы как бы «заглядываем» в объект `wordCount`, используя только что считанное слово в качестве ключа. Результат выражения `wordCount[s]` представляет собой некоторое целочисленное значение, обозначающее, сколько раз слово `s` уже встречалось ранее. Затем операция инкремента увеличивает это целое значение на единицу. А что будет, если мы встречаем некоторое слово в первый раз? Если в словаре нет элемента с таким ключом, то он будет создан с инициализацией поля типа `int` значением по умолчанию, то есть нулем. Следовательно, после операции инкремента это значение будет равно единице.

Завершив считывание входных данных и формирование словаря `wordCount`, мы должны вывести в выходной файл `freq_map.txt` значения обнаруженных слов и соответствующих им счетчиков. Вывод результатов реализуется здесь практически так же, как и для последовательных контейнеров — с помощью соответствующего итератора. Однако есть одна тонкость, связанная с тем, что при разменовании итератора `map`-объекта мы получаем значение, которое имеет тип `pair`, соответствующий данному `map`-объекту. Так как `pair` — это структура, то доступ

к полям структуры через «указатель» `it` осуществляется посредством выражений `it->first`, `it->second`.

Рассмотрим теперь более сложную задачу, чем предыдущие, чтобы продемонстрировать, с одной стороны, применение принципов ООП на практике, а с другой — те удобства, которые дает применение STL.

Задача 6.3. Морской бой

Написать программу, реализующую упрощенную версию игры «Морской бой» между двумя игроками: пользователем и компьютером. Упрощения данной версии: а) все корабли размещаются только вертикально; б) размещение кораблей — случайное у обоих игроков.

Для тех, у кого было тяжелое детство, напомним правила.

- Имеются два игровых поля: «свое» и «противника», каждое 10×10 клеток.
- У каждого игрока по 10 кораблей: один четырехпалубный (состоящий из четырех клеток), два трехпалубных (из трех клеток), три двухпалубных (из двух), четыре однопалубных (из одной клетки). При расстановке корабли не должны касаться друг друга (находиться в соседних клетках).
- Каждый игрок видит размещение кораблей на своем игровом поле, но не имеет информации о размещении кораблей на поле противника.
- После расстановки кораблей игроки начинают «стрелять» друг в друга. Для этого стреляющий выбирает клетку на поле противника и объявляет ему ее координаты (А1, Е5 и т. д.). Противник смотрит на своем поле, находится ли по указанным координатам его корабль, и сообщает результат выстрела:
 - *промах* — на данной клетке нет корабля противника;
 - *ранен (поврежден)* — на данной клетке есть корабль противника с хотя бы еще одной непораженной клеткой (палубой);
 - *убит* — на данной клетке есть корабль противника, и все его клетки (палубы) уже поражены.
- В случае попадания в корабль противника игроку дается право на внеочередной выстрел, в противном случае ход переходит к противнику.
- Стрельба ведется до тех пор, пока у одного из игроков не окажутся «убитыми» все корабли (в этом случае он признается проигравшим, а его противник — победителем).

Так как мы пишем программу для консольного приложения, то доступные нам графические средства сильно ограничены — это текстовые символы и символы псевдографики. Примем решение, что после некоторого хода играющих картинка в консольном окне будет иметь примерно такой вид, как на рис. 6.1.

Изображенные на рисунке игровые поля «Мой флот» и «Флот неприятеля» отображают текущее состояние игры со стороны пользователя. Изначальное размещение кораблей на поле пользователя — в клетках, помеченных символом

«заштрихованный прямоугольник»¹. Символом «.» (точка) обозначены те свободные клетки, по которым еще не было произведено выстрела, символом «o» — промахи стреляющих, символом «X» — пораженные клетки (палубы) кораблей. Пробелами обозначены те клетки, в которых согласно правилам размещения кораблей уже не могут находиться корабли противника. Эти «мертвые зоны» являются после гибели очередного корабля.

```

C:\VCwork\Task6_1\Debug\Task6_1.exe
31. Мой выстрел: R6
*** Есть попадание? ***

Выстрел неприятеля: пропускается...

Мой флот          Флот неприятеля
 1 2 3 4 5 6 7 8 9 10  1 2 3 4 5 6 7 8 9 10
A . . o . . o . o . .  A o . . o . . . .
B . . o o o . . . . .  B X . . . . o . .
C . . o . . . . . X   C X . . o . o . .
D . . o . . . . . X   D X . . o . . . .
E o . . o o . . . . X  E o . . o . o . o
F . . . . . . . . . .  F . o . . X . . . .
G . X . . . . o . . .  G . . . . o . . . .
H o X o . . o . . . .  H . . . . o . . . .
I . X o o . . o . . .  I . o . . . . . . .
J . X . . . . . . . .  J . . . . o . . . o
=====
32. Мой выстрел:

```

Рис. 6.1. Возможный вид консольного окна после i -го хода ($i = 31$)

После сделанных разъяснений мы можем приступить к решению задачи, и начнем, как всегда, с выявления понятий/классов и их взаимосвязей.

Итак, мы имеем двух игроков: первый — пользователь (User), второй — компьютер, выступающий в роли робота (Robot). Каждый игрок «управляет» своим собственным флотом и поэтому будет логичным создать два класса: UserNavy (флот пользователя) и RobotNavy (флот робота). Очевидно, что эти классы обладают различным поведением — например, метод FireOff() (выстрел по неприятелю) в первом классе должен пригласить пользователя ввести координаты выстрела, в то время как во втором классе аналогичный метод должен автоматически сформировать координаты выстрела, сообразуясь с искусственным интеллектом робота. В то же время в этих классах есть и общие атрибуты, такие, например, как игровые поля (свое и неприятеля), корабли своего флота и т. д. Поэтому выделим все общие атрибуты (поля и методы) в базовый класс Navy, который будут наследовать классы UserNavy и RobotNavy.

Каждый флот состоит из кораблей, отсюда вытекает потребность в классе Ship, объекты которого инкапсулируют такую информацию, как координаты размещения корабля, имя корабля, общее количество палуб, количество неповрежденных палуб.

Для описания размещения кораблей здесь предлагается воспользоваться классом Rect, который позволяет задать любой прямоугольник в двумерном дискретном пространстве. Конечно, наши прямоугольники вырождаются в линию², но

¹ Символ 176 в кодовой таблице ср866 (MS DOS).

² В формализме двумерного дискретного пространства.

Ниже приводится текст программы.

```

////////////////////////////////////
// Проект Task6_3
////////////////////////////////////
// Ship.h
#ifndef SHIP_H
#define SHIP_H

#include <set>
#include <map>
#include <string>
#include "CyrIOS.h" // for Visual C++ 6.0

#define N 10 // размер поля для размещения флота
// (N * N клеток)

struct Cell;
typedef std::set<Cell> CellSet; // множество клеток

// Клетка (ячейка) на игровом поле
struct Cell {
    Cell(int _r = 0, int _c = 0) : row(_r), col(_c) {}
    bool InSet(const CellSet&) const; // определяет
    // принадлежность клетки множеству
    // типа CellSet

    bool operator<(const Cell&) const;
    int row; // ряд
    int col; // колонка
};

// Прямоугольная область (размещение кораблей и их "оболочек")
struct Rect {
    Rect() {}
    Rect(Cell _lt, Cell _rb) : lt(_lt), rb(_rb) { FillCset(); }
    void FillCset(); // наполнить cset клетками
    // данной области

    bool Intersect(const CellSet& cs) const; // определить наличие
    // непустого пересечения
    // прямоугольника с множеством cs

    Cell lt; // left-top-клетка
    Cell rb; // right-bottom-клетка
    CellSet cset; // множество клеток, принадлежащих
    // прямоугольнику
};

// Класс Ship (для представления корабля)
class Ship {
    friend class UserNavy;
    friend class RobotNavy;
};

```



```

public:
    Ship() : nDeck(0), nLiveDeck(0) {}
    Ship(int, std::string, Rect):
protected:
    Rect place;           // координаты размещения
    std::string name;    // имя корабля
    int nDeck;           // количество палуб
    int nLiveDeck;      // количество неповрежденных палуб
};

#endif /* SHIP_H */
//////////////////////////////////////////////////////////////////
// Ship.cpp
#include <string>
#include <algorithm>
#include "Ship.h"
using namespace std;
//////////////////////////////////////////////////////////////////
// Класс Cell
bool Cell::InSet(const CellSet& cs) const {
    return (cs.count(Cell(row, col)) > 0);
}
bool Cell::operator<(const Cell& c) const {
    return ((row < c.row) || ((row == c.row) && (col < c.col)));
}
//////////////////////////////////////////////////////////////////
// Класс Rect
void Rect::FillCset() {
    for (int i = lt.row; i <= rb.row; i++)
        for (int j = lt.col; j <= rb.col; j++)
            cset.insert(Cell(i, j));
}
bool Rect::Intersect(const CellSet& cs) const {
    CellSet common_cell;
    set_intersection(cset.begin(), cset.end(), cs.begin(), cs.end(),
        inserter(common_cell, common_cell.begin()));
    return (common_cell.size() > 0);
}
//////////////////////////////////////////////////////////////////
// Класс Ship
Ship::Ship(int _nDeck, string _name, Rect _place) :
place(_place), name(_name), nDeck(_nDeck), nLiveDeck(_nDeck) {}
//////////////////////////////////////////////////////////////////
// Navy.h
#include "Ship.h"
#define DECK 176 // исправная клетка-палуба
#define DAMAGE 'X' // разрушенная клетка-палуба
#define MISS 'o' // пустая клетка, в которую упал снаряд

```



```

// Класс RobotNavy
class RobotNavy : public Navy {
public:
    RobotNavy();
    void Allocation();
    void FireOff();           // выстрел по неприятелю
    void ResultAnalys();     // анализ результатов выстрела
    void GetFire();          // "прием" огня противника
private:
    bool isCrushContinue;    // предыдущий выстрел был успешным
    bool upEmpty;           // у поврежденного корабля противника
                            // нет "живых" клеток в верхнем направлении
};
/////////////////////////////////////////////////////////////////
// Navy.cpp
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <algorithm>
#include "Navy.h"
using namespace std;

Cell Space::u_fire;
Cell Space::r_fire;
CurrentState Space::u_state = Miss;
CurrentState Space::r_state = Miss;
int Space::step = 1;

// Функция gap(n) возвращает строку из n пробелов
string gap(int n) { return string(n, ' '); }

/////////////////////////////////////////////////////////////////
// Класс Navy
Navy::Navy() : nLiveShip(10) {
    // Заполняем игровые поля символом "точка"
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            ownField[i][j] = '.';
            enemyField[i][j] = '.';
        }
}

Rect Navy::Shell(Rect r) const {
    Rect sh(r);
    sh.lt.row = (-sh.lt.row < 0) ? 0 : sh.lt.row;
    sh.lt.col = (-sh.lt.col < 0) ? 0 : sh.lt.col;
    sh.rb.row = (++sh.rb.row > (N - 1)) ? (N - 1) : sh.rb.row;
    sh.rb.col = (++sh.rb.col > (N - 1)) ? (N - 1) : sh.rb.col;
    return sh;
}

```

```

void Navy::AddToVetoSet(Rect r) {
    for (int i = r.lt.row; i <= r.rb.row; i++)
        for (int j = r.lt.col; j <= r.rb.col; j++)
            vetoSet.insert(Cell(i, j));
}

void Navy::AllocShip(int indShip, int nDeck, string name) {
    int i, j;
    Cell lt, rb;
    // Генерация случайно размещенной начальной клетки корабля
    // с учетом недопустимости "пересечения" нового корабля
    // с множеством клеток vetoSet
    while(1) {
        lt.row = rand() % (N + 1 - nDeck);
        lt.col = rb.col = rand() % N;
        rb.row = lt.row + nDeck - 1;
        if (!Rect(lt, rb).Intersect(vetoSet)) break;
    }
    // Сохраняем данные о новом корабле
    ship[indShip] = Ship(nDeck, name, Rect(lt, rb));

    // Наносим новый корабль на игровое поле (символ DECK).
    // Добавляем соответствующие элементы в словарь ассоциаций
    for (i = lt.row; i <= rb.row; i++)
        for (j = lt.col; j <= rb.col; j++) {
            ownField[i][j] = DECK;
            shipMap[Cell(i, j)] = indShip;
        }
    // Добавляем в множество vetoSet клетки нового корабля
    // вместе с пограничными клетками
    AddToVetoSet(Shell(Rect(lt, rb)));
}

void Navy::Show() const {
    char rowName[10] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'};
    string colName("1 2 3 4 5 6 7 8 9 10");
    int i, j;
    cout << "_____\\n";
    cout << gap(3) << "Мой флот" << gap(18) << "Флот неприятеля" << endl;
    cout << gap(3) << colName << gap(6) << colName << endl;

    for (i = 0; i < N; i++) {
        // Own
        string line = gap(1) + rowName[i];
        for (j = 0; j < N; j++)
            line += gap(1) + (char)ownField[i][j];
        // Enemy
        line += gap(5) + rowName[i];
        for (j = 0; j < N; j++)

```

```

        line += gap(1) + (char)enemyField[i][j];
        cout << line << endl;
    }
    cout << endl;
    cout << "=====\n";
    cout << step << ". " << "Мой выстрел:      ";
    step++;
}

int Navy::GetInt() {
    int value;
    while (true) {
        cin >> value;
        if ('\n' == cin.peek()) { cin.get(); break; }
        else {
            cout << "Повторите ввод колонки (ожидается целое число):" << endl;
            cin.clear();
            while (cin.get() != '\n') {};
        }
    }
    return value;
}

////////////////////////////////////
// Класс UserNavy
void UserNavy::Allocation() {
    srand((unsigned)time(NULL));
    AllocShip(0, 4, "Авианосец 'Варяг'");
    AllocShip(1, 3, "Линкор 'Муромец'");
    AllocShip(2, 3, "Линкор 'Никитич'");
    AllocShip(3, 2, "Крейсер 'Чудный'");
    AllocShip(4, 2, "Крейсер 'Добрый'");
    AllocShip(5, 2, "Крейсер 'Справедливый'");
    AllocShip(6, 1, "Миноносец 'Храбрый'");
    AllocShip(7, 1, "Миноносец 'Ушлый'");
    AllocShip(8, 1, "Миноносец 'Проворный'");
    AllocShip(9, 1, "Миноносец 'Смелый'");
    vetoSet.clear();
}

void UserNavy::FillDeadZone(Rect r, Field& field) {
    int i, j;
    Rect sh = Shell(r);
    for (i = sh.lt.row, j = sh.lt.col; j <= sh.rb.col; j++)
        if (sh.lt.row < r.lt.row) field[i][j] = ' ';
    for (i = sh.rb.row, j = sh.lt.col; j <= sh.rb.col; j++)
        if (sh.rb.row > r.rb.row) field[i][j] = ' ';
    for (j = sh.lt.col, i = sh.lt.row; i <= sh.rb.row; i++)
        if (sh.lt.col < r.lt.col) field[i][j] = ' ';
}

```

```

    for (j = sh.rb.col, i = sh.lt.row; i <= sh.rb.row; i++)
        if (sh.rb.col > r.rb.col) field[i][j] = ' ';
}

void UserNavy::FireOff() {
    string capital_letter = "ABCDEFGHIJ";
    string small_letter = "abcdefghij";
    unsigned char rowName; // обозначение ряда (A, B, ... , J)
    int colName;          // обозначение колонки (1, 2, ..., 10)
    int row;              // индекс ряда (0, 1, ..., 9)
    int col;              // индекс колонки (0, 1, ..., 9)

    bool success = false;
    while (!success) {
        cin >> rowName;
        row = capital_letter.find(rowName);
        if (-1 == row) row = small_letter.find(rowName);
        if (-1 == row) { cout << "Ошибка. Повторите ввод.\n"; continue; }
        colName = GetInt();
        col = colName - 1;
        if ((col < 0) || (col > 9)) {
            cout << "Ошибка. Повторите ввод.\n"; continue;
        }
        success = true;
    }
    u_fire = Cell(row, col);
}

void UserNavy::ResultAnalys() {
    // r_state - сообщение робота о результате выстрела
    // пользователя по клетке u_fire
    switch(r_state) {
    case Miss:
        enemyField[u_fire.row][u_fire.col] = MISS;
        break;
    case Damage:
        enemyField[u_fire.row][u_fire.col] = DAMAGE;
        crushSet.insert(u_fire);
        break;
    case Kill:
        enemyField[u_fire.row][u_fire.col] = DAMAGE;
        crushSet.insert(u_fire);
        Rect kill;
        kill.lt = *crushSet.begin();
        kill.rb = *(-crushSet.end());
        // Заполняем "обрамление" пробелами
        FillDeadZone(kill, enemyField);
        crushSet.clear();
    }
}

```

```

void UserNavy::GetFire() {
    // выстрел робота - по клетке r_fire
    string capital_letter = "ABCDEFGHIJ";
    char rowName = capital_letter[r_fire.row];
    int colName = r_fire.col + 1;
    cout << "\nВыстрел неприятеля: " << rowName << colName << endl;
    if (DECK == ownField[r_fire.row][r_fire.col]) {
        cout << "*** Есть попадание! ***";
        ownField[r_fire.row][r_fire.col] = DAMAGE;
        u_state = Damage;
        // индекс корабля, занимающего клетку r_fire
        int ind = shipMap[r_fire];
        ship[ind].nLiveDeck--;

        if (!ship[ind].nLiveDeck) {
            u_state = Kill;
            cout << gap(6) << "O ужас! Погиб " << ship[ind].name << " !!!";
            nLiveShip--;
            Rect kill = ship[ind].place;
            FillDeadZone(kill, ownField);
        }
    }
    else {
        u_state = Miss;
        cout << "*** Мимо! ***";
        ownField[r_fire.row][r_fire.col] = MISS;
    }
    cout << endl;
}
////////////////////////////////////
// Класс RobotNavy
RobotNavy::RobotNavy() {
    Allocation();
    isCrushContinue = false;
    upEmpty = false;
}

void RobotNavy::Allocation() {
    AllocShip(0, 4, "Авианосец 'Алькаида'");
    AllocShip(1, 3, "Линкор 'БенЛаден'");
    AllocShip(2, 3, "Линкор 'Хусейн'");
    AllocShip(3, 2, "Крейсер 'Подлый'");
    AllocShip(4, 2, "Крейсер 'Коварный'");
    AllocShip(5, 2, "Крейсер 'Злой'");
    AllocShip(6, 1, "Миноносец 'Гадкий'");
    AllocShip(7, 1, "Миноносец 'Мерзкий'");
    AllocShip(8, 1, "Миноносец 'Пакостный'");
    AllocShip(9, 1, "Миноносец 'Душный'");
    vetoSet.clear();
}

```

```
void RobotNavy::FireOff() {
    Cell c, cUp;
    if (!isCrushContinue) {
        // случайный выбор координат выстрела
        while(1) {
            c.row = rand() % N;
            c.col = rand() % N;
            if (!c.InSet(vetoSet)) break;
        }
    }
    else {
        // "пляшем" от предыдущего попадания
        c = cUp = r_fire;
        cUp.row--;
        if ((!upEmpty) && c.row && (!cUp.InSet(vetoSet)))
            c.row--;
        else {
            c = *(-crushSet.end());
            c.row++;
        }
    }
    r_fire = c;
    vetoSet.insert(r_fire);
}
```

```
void RobotNavy::ResultAnalys() {
    // u_state - сообщение пользователя о результате
    // выстрела робота по клетке r_fire
    switch(u_state) {
    case Miss:
        if (isCrushContinue) upEmpty = true;
        break;
    case Damage:
        isCrushContinue = true;
        crushSet.insert(r_fire);
        break;
    case Kill:
        isCrushContinue = false;
        upEmpty = false;
        crushSet.insert(r_fire);
        Rect kill;
        kill.lt = *crushSet.begin();
        kill.rb = *(-crushSet.end());

        AddToVetoSet(Shell(kill));
        crushSet.clear();
    }
}
```



```

void RobotNavy::GetFire() {
    // выстрел пользователя - по клетке u_fire
    if (DECK == ownField[u_fire.row][u_fire.col]) {
        cout << "*** Есть попадание! ***";
        r_state = Damage;
        // индекс корабля, занимающего клетку u_fire
        int ind = shipMap[u_fire];
        ship[ind].nLiveDeck--;

        if (!ship[ind].nLiveDeck) {
            r_state = Kill;
            cout << gap(6) << "Уничтожен " << ship[ind].name << " !!!";
            nLiveShip--;
        }
    }
    else {
        r_state = Miss;
        cout << "*** Мимо! ***";
    }
    cout << endl;
}

```

```

////////////////////////////////////

```

```

// Main.cpp

```

```

#include <iostream>

```

```

#include "Navy.h"

```

```

using namespace std;

```

```

int main() {

```

```

    // Начальная позиция

```

```

    UserNavy userNavy;

```

```

    RobotNavy robotNavy;

```

```

    userNavy.Show();

```

```

    while (userNavy.IsLive() && robotNavy.IsLive()) {

```

```

        // Выстрел пользователя

```

```

        if (Space::u_state != Miss) {

```

```

            cout << "пропускается...: <Enter>" << endl;

```

```

            cin.get();

```

```

        }

```

```

        else {

```

```

            userNavy.FireOff();

```

```

            robotNavy.GetFire();

```

```

            userNavy.ResultAnalys();

```

```

            if (!robotNavy.IsLive()) {

```

```

                userNavy.Show();

```

```

                break;

```

```

            }

```

```

        }

```

```

        // Выстрел робота

```

```

        if (Space::r_state != Miss)

```

```

        cout << "\nВыстрел неприятеля: пропускается..." << endl;
    else {
        robotNavy.FireOff();
        userNavy.GetFire();
        robotNavy.ResultAnalysis();
    }
    userNavy.Show();
}
if (userNavy.IsLive())
    cout << "\n:-))) Ура! Победа!!! :-)))" << endl;
else {
    cout << "\n:-((( Увы. Непрятель оказался сильнее." << endl;
    cout << ":-((( Но ничего, в следующий раз мы ему покажем!!!" << endl;
}
cin.get();
return 0;
}
//————— конец проекта Task6_3 —————
////////////////////////////////////

```

Читая код программы, обратите особое внимание на использование объектов контейнерных классов:

- объект `cset` типа `set<Cell>` в классе `Rect`;
- объекты `vetoSet` и `crushSet` типа `set<Cell>` в классе `Navy`;
- объект `shipMap` типа `map<Cell, int>` в классе `Navy`.

В качестве самостоятельного упражнения рекомендуем вам доработать приведенную программу, сняв ограничение «только вертикальное расположение кораблей» (решение об ориентации размещаемого корабля — горизонтальное или вертикальное — принимается случайным образом).

Давайте повторим наиболее важные моменты этого семинара.

1. Стандартная библиотека шаблонов содержит общецелевые классы и функции, которые реализуют широко используемые алгоритмы и структуры данных.
2. STL построена на основе шаблонных классов, поэтому входящие в нее алгоритмы и структуры могут настраиваться на различные типы данных.
3. Использование STL позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки.
4. Везде, где это возможно, используйте классы и алгоритмы STL!

Задания

Вариант 1

Написать программу для моделирования Т-образного сортировочного узла на железной дороге с использованием контейнерного класса `stack` из STL.

Программа должна разделять на два направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность ввода исходных данных с клавиатуры и из файла.

Вариант 2

Написать программу, отыскивающую проход по лабиринту, с использованием контейнерного класса `stack` из STL.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Программа находит проход через лабиринт, двигаясь от заданного входа. После отыскания прохода программа выводит найденный путь в виде координат квадратов.

Вариант 3

Написать программу, моделирующую управление каталогом в файловой системе.

Для каждого файла в каталоге содержатся следующие сведения: имя файла, дата создания, количество обращений к файлу.

Программа должна обеспечивать:

- начальное формирование каталога файлов;
- вывод каталога файлов;
- удаление файлов, дата создания которых раньше заданной;
- выборку файла с наибольшим количеством обращений.

Выбор моделируемой функции должен осуществляться с помощью меню.

Для представления каталога использовать контейнерный класс `list` из STL.

Вариант 4

Написать программу моделирования работы автобусного парка.

Сведения о каждом автобусе содержат: номер автобуса, фамилию и инициалы водителя, номер маршрута.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- начальное формирование данных о всех автобусах в парке в виде списка (ввод с клавиатуры или из файла);
- имитация выезда автобуса из парка: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;

- имитация въезда автобуса в парк: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте.

Для представления необходимых списков использовать контейнерный класс `list`.

Вариант 5

Написать программу учета заявок на авиабилеты.

Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- добавление заявок в список;
- удаление заявок;
- вывод заявок по заданному номеру рейса и дате вылета;
- вывод всех заявок.

Для хранения данных использовать контейнерный класс `list`.

Вариант 6

Написать программу учета книг в библиотеке.

Сведения о книгах содержат: фамилию и инициалы автора, название, год издания, количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- добавление данных о книгах, вновь поступающих в библиотеку;
- удаление данных о списываемых книгах;
- выдача сведений о всех книгах, упорядоченных по фамилиям авторов;
- выдача сведений о всех книгах, упорядоченных по годам издания.

Хранение данных организовать с применением контейнерного класса `multimap`, в качестве ключа использовать «фамилию и инициалы автора».

Вариант 7

Написать программу «Моя записная книжка».

Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Хранение данных организовать с применением контейнерного класса `map` или `multimap`.

Вариант 8

Написать программу учета заявок на обмен квартир и поиска вариантов обмена. Каждая заявка содержит сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- ввод заявки на обмен;
- поиск в картотеке подходящего варианта: при совпадении требований и предложений по количеству комнат и этажности и различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;
- вывод всей картотеки.

Для хранения данных картотеки использовать контейнерный класс `list`.

Вариант 9

Написать программу «Автоматизированная информационная система на железнодорожном вокзале».

Информационная система содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается: номер поезда, станция назначения, время отправления.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- первоначальный ввод данных в информационную систему (с клавиатуры или из файла);
- вывод сведений по всем поездам;
- вывод сведений по поезду с запрошенным номером;
- вывод сведений по тем поездам, которые следуют до запрошенной станции назначения.

Хранение данных организовать с применением контейнерного класса `vector`.

Вариант 10

Написать программу «Англо-русский и русско-английский словарь».

«База данных» словаря должна содержать синонимичные варианты перевода слов.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- Загрузка «базы данных» словаря (из файла).
- Выбор режима работы:
 - англо-русский;
 - русско-английский.

- Вывод вариантов перевода заданного английского слова.
 - Вывод вариантов перевода заданного русского слова.
- Базу данных словаря реализовать в виде двух контейнеров типа `map`.

Вариант 11

Написать программу, реализующую игру «Крестики-нолики» между двумя игроками: пользователем и компьютером (роботом). В программе использовать контейнерные классы STL.

Вариант 12

Написать программу, решающую игру-головоломку «Игра в 15». Начальное размещение номеров — случайное. Предусмотреть два режима демонстрации решения: непрерывный (с некоторой задержкой визуализации) и пошаговый (по нажатию любой клавиши). В программе использовать контейнерные классы STL.

Вариант 13

Составить программу формирования списка кандидатов, участвующих в выборах губернатора.

Каждая заявка от кандидата содержит: фамилию и инициалы, дату рождения, место рождения, индекс популярности.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- Добавление заявки в список кандидатов. Для ввода индекса популярности (значение указано в скобках) предусмотреть выбор с помощью подменю одного из следующих вариантов:
 - поддержан президентом (70);
 - поддержан оппозиционной партией (15);
 - оппозиционный кандидат, который снимет свою кандидатуру в пользу кандидата № 1 (10);
 - прочие (5).
- Удаление заявки по заявлению кандидата.
- Формирование и вывод списка для голосования.

Хранение данных организовать с применением контейнерного класса `priority_queue` из STL. Для надлежащего функционирования очереди с приоритетами по беспокоиться о надлежащем определении операции `<` (меньше) в классе, описывающем заявку кандидата. Формирование и вывод списка для голосования реализовать посредством выборки заявок из очереди.

Вариант 14

Составить программу моделирования работы автобусного парка.

Сведения о каждом автобусе содержат: номер автобуса, фамилию и инициалы водителя, номер маршрута.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- начальное формирование данных о всех автобусах в парке в виде списка (ввод с клавиатуры или из файла);
- имитация выезда автобуса из парка: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- имитация въезда автобуса в парк: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте, упорядоченных по номерам автобусов;
- вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте, упорядоченных по номерам маршрутов.

Хранение всех необходимых списков организовать с применением контейнерного класса `map`, в качестве ключа использовать «номер автобуса».

Вариант 15

Составить программу учета заявок на авиабилеты.

Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- добавление заявок в список;
- удаление заявок;
- вывод заявок по заданному номеру рейса и дате вылета;
- вывод всех заявок, упорядоченных по пунктам назначения;
- вывод всех заявок, упорядоченных по датам вылета.

Хранение данных организовать с применением контейнерного класса `multimap`, в качестве ключа использовать «пункт назначения».

Вариант 16

Написать программу учета книг в библиотеке.

Сведения о книгах содержат: фамилию и инициалы автора, название, год издания, количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- добавление данных о книгах, вновь поступающих в библиотеку;
- удаление данных о списываемых книгах;
- выдача сведений о всех книгах, упорядоченных по фамилиям авторов;
- выдача сведений о всех книгах, упорядоченных по годам издания.

Хранение данных организовать с применением контейнерного класса `vector`.

Вариант 17

Написать программу «Моя записная книжка».

Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Хранение данных организовать с применением контейнерного класса `list`.

Вариант 18

Написать программу учета заявок на обмен квартир и поиска вариантов обмена.

Каждая заявка содержит фамилию и инициалы заявителя, а также сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- ввод заявки на обмен;
- поиск в картотеке подходящего варианта: при совпадении требований и предположений по количеству комнат и этажности и различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;
- вывод всей картотеки.

Хранение данных организовать с применением контейнерного класса `set`.

Вариант 19

Написать программу «Автоматизированная информационная система на железнодорожном вокзале».

Информационная система содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается: номер, станция назначения, время отправления.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- первоначальный ввод данных в информационную систему (с клавиатуры или из файла);

- вывод сведений по всем поездам;
- вывод сведений по поезду с запрошенным номером;
- вывод сведений по тем поездам, которые следуют до запрошенной станции назначения.

Хранение данных организовать с применением контейнерного класса `set`.

Вариант 20

Написать программу «Англо-русский и русско-английский словарь».

«База данных» словаря должна содержать синонимичные варианты перевода слов.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

- Загрузка «базы данных» словаря (из файла).
- Выбор режима работы:
 - англо-русский;
 - русско-английский.
- Вывод вариантов перевода заданного английского слова.
- Вывод вариантов перевода заданного русского слова.

Базу данных словаря реализовать в виде двух контейнеров типа `set`.

Приложение Паттерны проектирования

При создании объектно-ориентированной программы требуется разработать классы, соответствующие предметам, процессам и понятиям предметной области¹, а также определить взаимоотношения этих классов и протоколы взаимодействия объектов во время выполнения программы. Задача эта весьма сложная, поскольку есть великое множество различных правильных способов объектной декомпозиции, не говоря уже о неправильных.

Архитектура системы должна, с одной стороны, соответствовать решаемой задаче, с другой — быть достаточно общей, чтобы сделать возможным внесение изменений, которые могут потребоваться в будущем. Новички испытывают шок от количества возможных вариантов и нередко возвращаются к привычным, не объектно-ориентированным методикам, а опытным проектировщикам, как правило, удается создать хороший дизайн системы.

Причина прежде всего в том, что они стремятся повторно использовать те решения, которые оказались удачными в прошлом. Именно благодаря накопленному опыту проектировщик и становится экспертом в своей области. Во многих объектно-ориентированных системах вы встретите повторяющиеся решения (паттерны), состоящие из классов и взаимодействующих объектов.

Паттерн — это описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Паттерны выявляются по мере накопления опыта разработок, когда программист использует одну и ту же схему организации и взаимодействия объектов в разных контекстах.

Применение паттернов (шаблонов) наряду с объектной технологией позволяет вывести процесс проектирования из области интуиции и представить систему на более высоком уровне абстракции — на уровне схем взаимодействия объектов, в результате чего система становится более прозрачной и понятной.

¹ Часто в программе используют и классы, не имеющие аналогов в предметной области — например, окно для вывода информации или массив для ее хранения.

Все сведения о паттернах в этом приложении приведены из ставшей классической книги [7]. В этой работе все паттерны в соответствии с их назначением разделены на три группы: порождающие, структурные и паттерны поведения. *Порождающие паттерны* описывают способы создания объектов, *структурные* — схемы организации классов и объектов, а *паттерны поведения* определяют типичные схемы взаимодействия классов и объектов.

Таблица П.1. Классификация паттернов проектирования

Назначение/ Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер (класса)	Интерпретатор Шаблонный метод
Объект	Абстрактная фабрика	Адаптер (объекта)	Итератор
	Одиночка	Декоратор	Команда
	Прототип	Заместитель	Наблюдатель
	Строитель	Компоновщик	Посетитель
		Мост	Посредник
		Приспособленец	Состояние
		Фасад	Стратегия
		Хранитель	
		Цепочка обязанностей	

В каждой группе можно, в свою очередь, выделить два уровня, определяющих, применяется паттерн к классам или объектам. В табл. П.1 приведена классификация паттернов. Ниже приводится определение каждого паттерна, а в следующих разделах — описание двух избранных паттернов для того, чтобы вы получили конкретное представление о наиболее простых паттернах и могли более успешно изучать специальную литературу, например [7], [10] и [15].

Описание каждого паттерна в [7] выполнено по одной и той же схеме, включающей его назначение, область применения, структурную схему, описание входящих в него объектов и их взаимодействий, а также пример реализации. Такой уровень документирования делает возможным использование паттерна в различных конкретных случаях, возникающих при проектировании программных систем.

Некоторые паттерны часто используются совместно. Например, **компоновщик** применяется с **итератором** или **посетителем**. Некоторые паттерны представляют собой альтернативные решения — так, **прототип** нередко можно использовать вместо **абстрактной фабрики**. Применение различных паттернов может привести к одному и тому же проектному решению, хотя изначально их назначение отличается. Например, **защищающий заместитель** может быть реализован точно так же, как **декоратор**.

Паттерны позволяют повысить степень повторной используемости и улучшить качество документирования программного проекта. Поскольку ничто не дается

даром, за это часто приходится платить усложнением программного кода и ухудшением производительности, поэтому применять паттерны следует только в тех случаях, когда гибкость действительно необходима. Впрочем, практически любой сколько-нибудь успешный программный продукт требует сопровождения, модификации и, следовательно, перепроектирования.

Порождающие паттерны

Порождающие паттерны проектирования абстрагируют процесс создания классов и объектов с целью обеспечить гибкость системы, то есть помогают сделать систему независимой от способа создания, композиции и представления объектов. Эти паттерны инкапсулируют знания о конкретных классах и скрывают детали того, как эти классы создаются и стыкуются. Таким образом, можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

Паттерн, порождающий *классы*, использует наследование, чтобы варьировать создаваемый класс, а паттерны, порождающие *объекты*, передают инстанцирование другим объектам.

Фабричный метод (Factory Method) определяет интерфейс для создания объектов, при этом требуемый класс создается с помощью подклассов. В подклассе определяется метод (то есть собственно *фабричный метод*), который возвращает экземпляр конкретного подкласса.

Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом надкласса, поэтому он может работать с любыми определенными пользователями классами конкретных продуктов.

Паттерн используется в следующих случаях:

- ❑ классу заранее не известно, объекты каких классов ему нужно создавать;
- ❑ класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- ❑ класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс принимает эти обязанности на себя.

Фабричные методы часто применяются в инструментальных библиотеках и каркасах.

Абстрактная фабрика (Abstract Factory) предоставляет интерфейс для создания семейств, связанных между собой, или независимых объектов, конкретные классы которых неизвестны.

От класса «абстрактная фабрика» наследуются классы конкретных фабрик, которые содержат методы создания конкретных объектов-продуктов, являющихся наследниками класса «абстрактный продукт», объявляющего интерфейс для их создания. Клиент пользуется только интерфейсами, заданными в классах «абстрактная фабрика» и «абстрактный продукт».

Паттерн изолирует клиента от деталей реализации классов, упрощает замену семейств продуктов и гарантирует их сочетаемость. Паттерн применяется, если:

- ❑ система не должна зависеть от того, как создаются, komponуются и представляются входящие в нее объекты;
- ❑ входящие в семейство взаимосвязанные объекты должны использоваться вместе, и вам необходимо обеспечить выполнение этого ограничения;
- ❑ система должна конфигурироваться одним из семейств составляющих ее объектов;
- ❑ вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Одиночка (Singleton) гарантирует, что некоторый класс может иметь только один экземпляр, и предоставляет глобальную точку доступа к нему.

Для некоторых классов важно, чтобы существовал только один экземпляр. При этом сам класс контролирует то, что у него есть только один экземпляр, может запретить создание дополнительных экземпляров, перехватывая запросы на создание новых объектов, и он же способен предоставить доступ к своему экземпляру. Одиночка определяет операцию Instance, которая позволяет клиентам получать доступ к единственному экземпляру. Клиенты имеют доступ к одиночке только через эту операцию.

Паттерн позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры. От класса Singleton можно порождать подклассы, а приложение легко сконфигурировать экземпляром расширенного класса.

Реализуется одиночка обычно с помощью статического метода класса, который имеет доступ к переменной, хранящей уникальный экземпляр, и гарантирует инициализацию переменной этим экземпляром перед возвратом ее клиенту.

Прототип (Prototype) описывает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем его копирования.

Класс «прототип» задает интерфейс для клонирования себя (например, с помощью метода, возвращающего копию себя). От этого класса наследуются конкретные классы, реализующие ту же операцию. Клиент обращается к прототипу, чтобы тот создал свою копию.

У прототипа те же самые результаты, что у **абстрактной фабрики и строителя**: он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен. Кроме того, все эти паттерны позволяют клиентам единообразно работать со специфичными для приложения классами.

Некоторые дополнительные преимущества паттерна:

- ❑ добавление и удаление продуктов во время выполнения. Прототип позволяет включать новый конкретный класс продуктов в систему, просто сообщив клиенту о новом экземпляре-прототипе;
- ❑ уменьшение числа подклассов. При использовании этого паттерна нет необходимости запрашивать фабричный метод создать новый объект — просто клонируется прототип.

Строитель (Builder) отделяет конструирование сложного объекта от его представления, позволяя использовать один и тот же процесс конструирования для создания различных внутренних представлений объекта.

Паттерн позволяет строить сложные объекты, процесс создания которых состоит из нескольких этапов. Класс «строитель» задает абстрактный интерфейс для создания частей объекта, а его подклассы («конкретные строители») реализуют этот интерфейс, определяют порядок сборки и предоставляют интерфейс к создаваемому объекту.

Клиент для создания объекта создает объект-распорядитель, конфигурируя его конкретным строителем. Распорядитель конструирует объект из частей, пользуясь интерфейсом строителя и вызывая методы конкретного строителя.

Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид конкретного строителя. Данный паттерн улучшает модульность, инкапсулируя способ конструирования и представления сложного объекта. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта.

В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя, что дает более тонкий контроль над процессом конструирования.

Структурные паттерны

Структурные паттерны описывают, как из классов и объектов образуются более крупные структуры. Структурные паттерны *уровня класса* используют наследование для составления композиций из интерфейсов и реализаций. Паттерны *уровня объекта* компонуют объекты для получения новой функциональности. Дополнительная гибкость в этом случае связана с тем, что можно изменять композицию объектов во время выполнения, что при статической композиции классов невозможно.

Адаптер (Adapter) преобразует интерфейс класса в некоторый другой интерфейс, ожидаемый клиентами. Обеспечивает совместную работу классов, которая была бы невозможна без данного паттерна из-за несовместимости интерфейсов.

Адаптер *уровня класса* использует множественное наследование (интерфейс наследуется от одного класса, а реализация — от другого), а в адаптере *уровня объекта* применяется композиция объектов (как правило, в объекте определяется ссылка на другой объект).

Паттерн применяется, если требуется использовать существующий класс с интерфейсом, не подходящим к нашим требованиям, а также если требуется создать повторно используемый класс, который должен взаимодействовать с заранее не известными или не связанными с ним классами, имеющими несовместимые интерфейсы.

Декоратор (Decorator) динамически возлагает на объект новые функции. Декораторы применяются для расширения имеющейся функциональности и являются гибкой альтернативой порождению подклассов.

Декоратор следует интерфейсу декорируемого компонента, поэтому его присутствие прозрачно для клиентов компонента. Например, таким образом можно добавить рамку к графическому объекту. Декоратор переадресует запросы внутреннему компоненту, а также может выполнять дополнительные действия. Декораторы могут вкладываться друг в друга, это позволяет сочетать дополнительные обязанности произвольным образом. Компоненты и их декораторы должны быть наследниками одного класса, который следует делать максимально легким, то есть задавать в нем в основном не данные, а интерфейс.

Заместитель (Proxy) замещает другой объект для обеспечения контроля доступа к нему. Используются следующие разновидности заместителей:

- *удаленный заместитель* предоставляет локальный представитель вместо объекта, находящегося в другом адресном пространстве;
- *виртуальный заместитель* создает «тяжелые» объекты по требованию (например, когда вместо изображения появляется рамка);
- *защищающий заместитель* контролирует доступ к исходному объекту (когда для разных объектов определены различные права доступа);
- *«умная» ссылка* — это замена обычного указателя. Она позволяет выполнить дополнительные действия при доступе к объекту (например, подсчет числа ссылок на реальный объект, загрузку объекта в память при первом обращении к нему или проверку и установку блокировки на реальный объект при обращении к нему, чтобы никакой другой объект не смог в это время изменить его).

Заместитель хранит ссылку на реальный объект и предоставляет идентичный ему интерфейс. Таким образом, он ведет себя аналогично указателю, то есть определяет дополнительный уровень косвенности при обращении к объекту.

Компоновщик (Composite) группирует объекты в древовидные структуры для представления иерархий типа «часть-целое». Позволяет клиентам работать с единичными объектами так же, как с группами объектов. Паттерн описан на с. 255.

Мост (Bridge) отделяет абстракцию от реализации, благодаря чему появляется возможность независимо изменять и то и другое. Такое разделение облегчает разбиение системы на слои и тем самым позволяет улучшить ее структуру.

Примером может служить абстракция окна в библиотеке для разработки пользовательских интерфейсов. Написанные с ее помощью приложения должны работать в разных средах: клиенты должны иметь возможность создавать окно, не привязываясь к конкретной реализации. Только сама реализация окна должна зависеть от платформы, на которой работает приложение, поэтому в клиентском коде не может быть никаких упоминаний о платформах. Для этого абстракция окна и ее реализация помещаются в две отдельные иерархии классов. Класс абстракции содержит ссылку на класс реализации. Все операции класса абстракции реализованы через абстрактные операции класса реализации.

Паттерн дает возможность конфигурировать реализацию абстракции во время выполнения, а также изменять класс реализации без перекомпиляции класса абстракции и его клиентов.

Приспособленец (Flyweight) использует разделение для эффективной поддержки большого числа мелких объектов и экономии ресурсов. Приспособленец является разделяемым объектом, который можно использовать одновременно в нескольких контекстах. В каждом контексте он выглядит как независимый объект. Ключевая идея здесь — различие между внутренним и внешним состояниями. Внутреннее состояние хранится в самом приспособленце и состоит из информации, не зависящей от его контекста. Именно поэтому он может разделяться. Внешнее состояние зависит от контекста и изменяется вместе с ним, поэтому не подлежит разделению. Объекты-клиенты отвечают за передачу внешнего состояния приспособленцу, когда в этом возникает необходимость.

Приспособленцы моделируют концепции или сущности, число которых слишком велико для представления объектами. Например, редактор документов мог бы создать по одному приспособленцу для каждой буквы алфавита. Каждый приспособленец хранит код символа, но координаты положения символа в документе и стиль его начертания определяются алгоритмами размещения текста и командами форматирования, действующими в том месте, где символ появляется. Код символа — это внутреннее состояние, а все остальное — внешнее.

Паттерн используется, когда приложение содержит большое число объектов, значительную часть состояния которых можно вынести вовне. При этом приложение не должно зависеть от идентичности объекта (поскольку объекты-приспособленцы могут разделяться, то проверка на идентичность возвратит «истину» для концептуально различных объектов).

Фасад (Facade) предоставляет унифицированный интерфейс к множеству интерфейсов в некоторой подсистеме. Определяет интерфейс более высокого уровня, облегчающий работу с подсистемой.

Фасад позволит отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степени независимости и переносимости. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Клиенты общаются с подсистемой, посылая запросы фасаду. Фасад делегирует запросы клиентов подходящим объектам внутри подсистемы. Классы подсистемы ничего не «знают» о существовании фасада, то есть не хранят ссылок на него.

Фасад может также упростить процесс переноса системы на другие платформы, поскольку уменьшается вероятность того, что в результате изменения одной подсистемы понадобится изменять и все остальные.

Паттерны поведения

В паттернах поведения внимание сконцентрировано на способах взаимодействия классов и объектов и распределении обязанностей между ними при реализации сложных алгоритмов поведения системы. Паттерны поведения *уровня класса* для распределения функций между разными классами используют наследование. В паттернах поведения *уровня объектов* используется композиция.

Часть паттернов описывает, как множество равноправных объектов сообща справляется с задачей, которая ни одному из них не под силу. Для того чтобы каждый объект не хранил информацию обо всех своих «коллегах», используются разные стратегии: например, паттерн **посредник**, находящийся между объектами-коллегами, обеспечивает косвенность ссылок, необходимую для разрывания лишних связей, а паттерн **цепочка обязанностей** дает возможность посылать запросы объекту не напрямую, а по цепочке «объектов-кандидатов». Таким образом уменьшается степень связанности системы.

Интерпретатор (Interpreter) для заданного языка определяет представление его грамматики, а также интерпретатор предложений языка, использующий это представление.

Необходимость в применении этого паттерна может возникнуть, если решение некоторой задачи можно представить в виде набора простых правил. Интерпретатор определяет грамматику языка, описывая класс для каждого правила. Клиент строит предложение на этом языке в виде синтаксического дерева, а затем для каждого узла вызывается интерпретирующая операция. Например, таким образом можно интерпретировать булевские выражения, форматы файлов, регулярные выражения.

Шаблонный метод (Template Method) определяет в некотором классе скелет алгоритма, перекадывая ответственность за некоторые его шаги на подклассы. Подклассы могут переопределять шаги алгоритма, не меняя его общей структуры, а класс-предок локализует общее для всех подклассов поведение, что позволяет не дублировать код. Операции, которые обязательно требуют замещения в подклассах, реализуются в предке как чисто виртуальные функции.

Шаблонные методы — один из основных приемов повторного использования кода. Они широко применяются в библиотеках, поскольку позволяют вынести общее поведение в библиотечные классы.

Итератор (Iterator) дает возможность последовательно обойти все элементы составного объекта, не раскрывая его внутреннего представления.

Основная идея состоит в том, чтобы за доступ к элементам и способ обхода отвечал не сам составной объект (контейнер), а отдельный объект-итератор, который имеет интерфейс для доступа к элементам контейнера. Различные итераторы могут обеспечивать различные стратегии обхода контейнера, при этом нет необходимости перечислять их в интерфейсе контейнера. Итераторы широко используются в большинстве библиотек, содержащих коллекции классов, — например, в STL и OWL.

Команда (Command) инкапсулирует запрос в виде объекта, позволяя тем самым параметризовать клиентов типом запроса, устанавливать очередность запросов, протоколировать их и поддерживать отмену выполнения операций.

Паттерн отделяет объект, инициирующий операцию, от объекта, который «знает», как ее выполнить. Это позволяет добиться высокой гибкости. Конкретные команды являются подклассами класса `Command`. Их действие состоит в вызове некоторой операции объекта-получателя запроса. Команды широко используются в библиотеках классов.

Наблюдатель (Observer) определяет между объектами зависимость типа «один-ко-многим», так что при изменении состояния одного объекта все зависящие от него получают извещение и автоматически обновляются.

В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных объектов. При этом желательно избежать жесткой связанности классов, так как это часто уменьшает возможность повторного использования.

Паттерн описывает два ключевых объекта: *субъект* и *наблюдатель*. У субъекта может быть несколько зависимых от него наблюдателей, которые уведомляются об изменении его состояния. Получив уведомление, наблюдатель опрашивает субъекта, чтобы синхронизировать с ним свое состояние. Примером реализации паттерна может служить связь между объектом и его различными представлениями — например, между электронной таблицей и связанными с ней диаграммами.

Посетитель (Visitor) представляет операцию, которую надо выполнить над элементами объекта. Позволяет определить новую операцию, не меняя классы элементов, к которым он применяется.

Паттерн применяется в следующих случаях:

- ❑ в структуре присутствуют объекты многих классов с различными интерфейсами и вы хотите выполнять над ними операции, зависящие от конкретных классов;
- ❑ над объектами, входящими в состав структуры, надо выполнять разнообразные, не связанные между собой операции, и вы не хотите «засорять» классы такими операциями. Паттерн позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн позволит включить в каждое приложение только относящиеся к нему операции;
- ❑ классы, устанавливающие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто.

Структура паттерна приведена на рис. П.1.

Паттерн состоит из следующих классов:

- ❑ **Visitor** (посетитель) объявляет операцию *Visit* для каждого класса *ConcreteElement* в структуре объектов. Имя и сигнатура этой операции идентифицируют класс, который посылает посетителю запрос *Visit*. Это позволяет посетителю определить, элемент какого конкретного класса он посещает. Владея такой информацией, посетитель может обращаться к элементу напрямую через его интерфейс.
- ❑ **ConcreteVisitor** (конкретный посетитель) реализует все операции, объявленные в классе *Visitor*. Каждая операция реализует фрагмент алгоритма, определенного для класса соответствующего объекта в структуре. Класс *ConcreteVisitor* предоставляет контекст для этого алгоритма и сохраняет его локальное состояние. Часто в этом состоянии аккумулируются результаты, полученные в процессе обхода структуры.

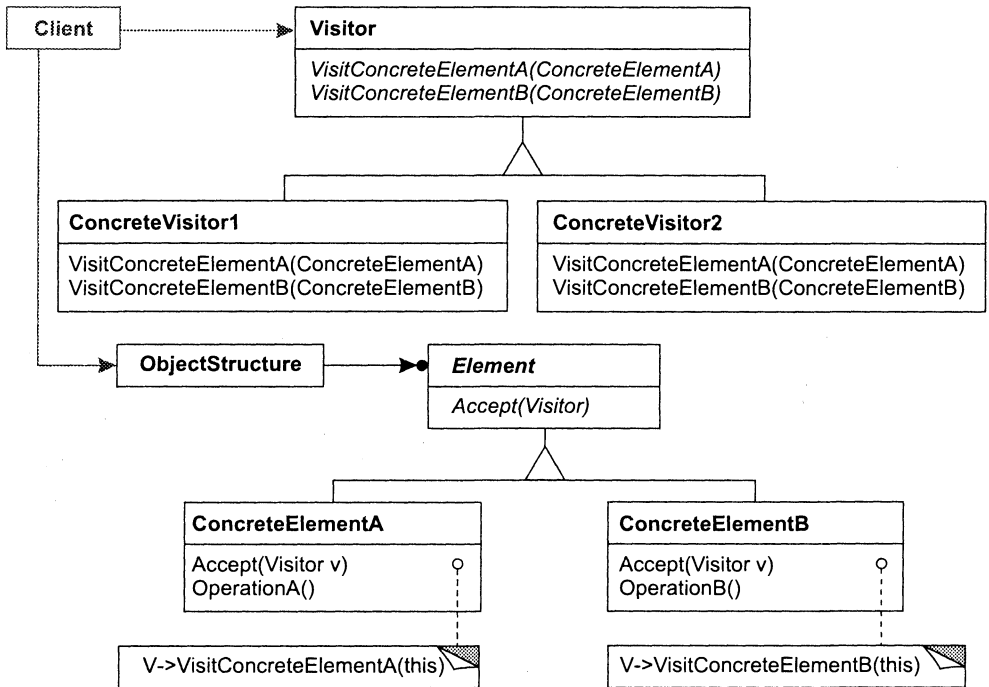


Рис. П. 1. Структура паттерна поведения

- ❑ Element (элемент) определяет операцию Accept, которая принимает посетителя в качестве аргумента.
- ❑ ConcreteElement (конкретный элемент) реализует операцию Accept, принимающую посетителя как аргумент.
- ❑ ObjectStructure (структура объектов):
 - может перечислить свои элементы;
 - может предоставить посетителю высокоуровневый интерфейс для посещения своих элементов;
 - может быть как составным объектом (см. паттерн **компоновщик**), так и коллекцией, например списком или множеством.

Клиент, использующий паттерн **посетитель**, должен создать объект класса ConcreteVisitor, а затем обойти всю структуру, посетив каждый ее элемент.

При посещении элемента последний вызывает операцию посетителя, соответствующую своему классу. Элемент передает этой операции себя в качестве аргумента, чтобы посетитель мог при необходимости получить доступ к его состоянию.

Основные *достоинства паттерна*:

1. Упрощает добавление новых операций над структурой объектов: для этого достаточно просто ввести нового посетителя. Напротив, если функциональность распределена по нескольким классам, то для определения новой операции придется изменить каждый класс.

2. Локализует в посетителе родственные операции и отсекает те, которые не имеют к ним отношения. Не связанные друг с другом функции распределяются по отдельным подклассам класса *Visitor*. Это способствует упрощению как классов, определяющих элементы, так и алгоритмов, инкапсулированных в посетителях.
3. Может посещать различные иерархии классов, в отличие от паттерна **итератор**, который не способен работать со структурами, состоящими из объектов разных типов.

Недостатком паттерна является то, что добавление новых классов *ConcreteElement* затруднено: каждый новый конкретный элемент требует объявления новой абстрактной операции в классе *Visitor*, которую нужно реализовать в каждом из существующих классов *ConcreteVisitor*.

Поэтому при решении вопроса о том, стоит ли использовать паттерн **посетитель**, нужно прежде всего посмотреть, что будет изменяться чаще: алгоритм, применяемый к объектам структуры, или классы объектов, составляющих эту структуру. Вполне вероятно, что сопровождать иерархию классов *Visitor* будет нелегко, если новые классы *ConcreteElement* добавляются часто. В таких случаях проще определить операции прямо в классах, представленных в структуре.

Посредник (Mediator) определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Паттерн способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия.

Посредник отвечает за координацию взаимодействий между группой объектов. Он определяет интерфейс для обмена информацией и избавляет входящие в группу объекты от необходимости явно ссылаться друг на друга. Все объекты общаются между собой только через посредника, поэтому количество взаимосвязей сокращается и поведение системы легче поддается настройке.

Посредники используются, например, для обмена информацией между виджетами в диалоговых окнах, в графических редакторах для поддержки визуальной связанности графических объектов, в системах проектирования электронных схем и т. п.

Состояние (State) позволяет объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что изменился класс объекта.

Основная идея этого паттерна заключается в том, чтобы ввести абстрактный класс для представления различных состояний объекта. Этот класс объявляет интерфейс, общий для всех классов, описывающих различные рабочие состояния. В подклассах реализуется поведение, специфичное для конкретного состояния. Паттерн используется, когда поведение объекта зависит от его состояния и должно изменяться во время выполнения или если в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Паттерн предлагает поместить каждую ветвь в отдельный класс.

Паттерн можно использовать, например, в графических редакторах для реализации абстрактного инструмента рисования, в подклассах которого задаются правила отрисовки различных примитивов.

Стратегия (Strategy) определяет семейство алгоритмов, делая их взаимозаменяемыми. Алгоритм можно менять независимо от клиента, который им пользуется. Паттерн описан на с. 252.

Хранитель (Memento) позволяет, не нарушая инкапсуляции, получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии.

Паттерн используется, когда необходимо зафиксировать внутреннее состояние объекта — например, при реализации контрольных точек и механизмов отката, позволяющих пользователю отменить операцию или восстановить состояние после ошибки. Хранитель — объект, хранящий состояние объекта-хозяина. Только хозяин может помещать в хранитель информацию и извлекать ее оттуда, для других объектов хранитель непрозрачен и предоставляет только «узкий» интерфейс. Это сохраняет инкапсуляцию и упрощает структуру хозяина.

Цепочка обязанностей (Chain of Responsibility) позволяет избежать привязки отправителя запроса к его получателю. Объекты-получатели связываются в цепочку, и запрос передается по ней до тех пор, пока он не будет обработан, при этом обработать запрос могут несколько объектов.

Паттерн используется, когда есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее не известен и должен быть найден автоматически; когда требуется отправить запрос одному из нескольких объектов, не указывая явно, какому именно; когда набор объектов, способных обработать запрос, должен задаваться динамически.

Примеры применения паттерна: в библиотеках классов для обработки событий пользователя; в графических редакторах для обработки запросов на выполнение операций (например, на обновление графического изображения, представляющего собой контейнер).

Паттерн Стратегия (Strategy)

Этот паттерн инкапсулирует семейство алгоритмов, делая их взаимозаменяемыми. Применять его целесообразно в следующих случаях:

- имеются родственные классы, отличающиеся только поведением. Стратегия позволяет гибко конфигурировать класс, задавая одно из возможных поведений;
- требуется иметь несколько разных вариантов алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой — больше памяти. Кроме того, с помощью стратегии легко определить поведение класса по умолчанию. Варианты алгоритмов могут быть реализованы в виде иерархии классов, что позволяет вычленив общую для всех классов функциональность. Инкапсулированные алгоритмы можно затем применять в разных контекстах;

- ❑ в алгоритме хранятся данные, о которых клиент не должен «знать». Стратегия позволяет не раскрывать сложные, специфичные для алгоритма структуры данных;
- ❑ в классе определено много вариантов поведения, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Структура паттерна приведена на рис. П.2.

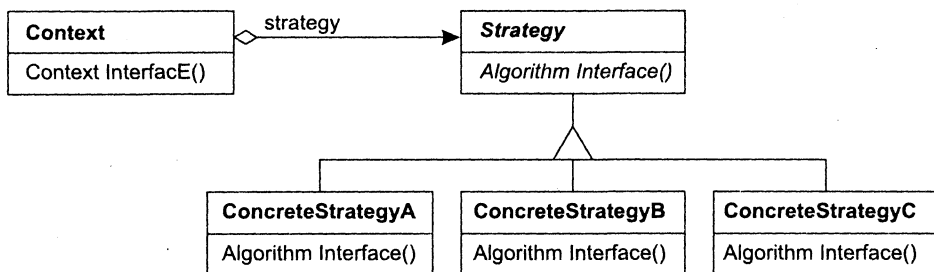


Рис. П.2. Структура паттерна Стратегия

Паттерн состоит из следующих классов:

- ❑ Strategy (стратегия) объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy;
- ❑ ConcreteStrategy (конкретная стратегия) — наследник класса Strategy. Реализует алгоритм, использующий интерфейс, объявленный в классе Strategy;
- ❑ Context (контекст) конфигурируется объектом класса ConcreteStrategy. Хранит ссылку на объект класса Strategy и может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.

Таким образом, объект Context делегирует реализацию конкретного алгоритма поведения объекту Strategy, что дает возможность гибко изменять поведение объекта. Если поведение объекта не требуется варьировать во время выполнения программы, стратегию можно реализовать как параметр шаблона. При этом необходимость в абстрактном классе Strategy отпадает, например:

```

template <class AStrategy> class Context {
    void Operation() { theStrategy.DoAlgorithm(); }
    // ...
private:
    AStrategy theStrategy;
};
// ...
class MyStrategy {
public:
    void DoAlgorithm();
};
// ...
Context<MyStrategy> aContext;
    
```

В качестве примера реализации паттерна приведем решение, аналогичное применяемому в Object Windows Library фирмы Borland в диалоговых окнах для проверки правильности введенных пользователем данных. Например, можно контролировать, что в одном поле могут присутствовать только буквы, а введенное в другое поле число принадлежит заданному диапазону.

Поле ввода данных (контекст) при необходимости присоединяет к себе «контролер» — объект класса Validator (стратегия). При закрытии диалогового окна поля запрашивают у своих контролеров проверку правильности данных. В библиотеке имеются классы контролеров (конкретные стратегии) для наиболее распространенных случаев, например RangeValidator для проверки принадлежности числа диапазону. Кроме того, клиент легко может определить собственные стратегии проверки, порождая подклассы от класса Validator.

Таким образом, поле ввода делегирует стратегию контроля объекту Validator, конкретный тип которого определяется на этапе выполнения программы при конструировании полей ввода.

Приводимый ниже текст представляет собой «скелет» приложения, в котором функциональность классов заменена выводом диагностических сообщений (например, вместо проверки текста выводится сообщение о том, что она выполняется).

```
#include "iostream"
using namespace std;

class Validator { // абстрактный класс стратегии
                // проверки ввода
public:
    virtual bool validate(char * text) = 0; // чисто виртуальный метод.
}; // замещается в подклассах

class RangeValidator : public Validator { // проверка на допустимый диапазон
public:
    RangeValidator();
    bool validate(char * text);
};

class FilterValidator : public Validator { // проверка на допустимые символы
public:
    FilterValidator();
    bool validate(char * text);
};

class InputLine { // поле ввода
public:
    InputLine(Validator *):
    void checkInput();
protected:
    Validator * validator; // ссылка на проверяющий эту строку класс
    char text[100]; // данные, введенные пользователем в поле
};
```

```

RangeValidator::RangeValidator() {
cout << "RangeValidator created" << endl;
}

bool RangeValidator::validate(char * text){
    cout << "RangeValidator working under the text " << text << endl;
    return true;
}

FilterValidator::FilterValidator(){
cout << "FilterValidator created" << endl;
}

bool FilterValidator::validate(char * text){
    cout << "FilterValidator working under the text " << text << endl;
    return true;
}

InputLine::InputLine(Validator * _validator) {
validator = _validator; // привязка конкретного проверяющего класса к полю ввода
}

void InputLine::checkInput(){
    cout << "checkInput working " << endl;
    validator -> validate(text); // вызов виртуального метода.
} // соответствующего классу, связанному с полем ввода

int main() {
// при создании полей ввода им передается конкретный класс для проверки:
InputLine * rangeInputLine = new InputLine(new RangeValidator);
InputLine * filterInputLine = new InputLine(new FilterValidator);

// ... здесь пользователь вводит данные в поля ввода

// пример запросов на проверку:
rangeInputLine->checkInput();
filterInputLine->checkInput();
return 0;
}

```

В реальных классах критерии проверки полей ввода задаются через параметры конструкторов объектов `RangeValidator` и `FilterValidator`.

Паттерн Компоновщик (Composite)

Этот паттерн группирует объекты в древовидные структуры. Позволяет клиентам работать с единичными объектами так же, как с группами объектов.

Применяется для представления иерархий типа «часть-целое», а также для того, чтобы клиенты могли единообразно работать как с простыми объектами, так и с контейнерами.

Структура паттерна приведена на рис. П.3.

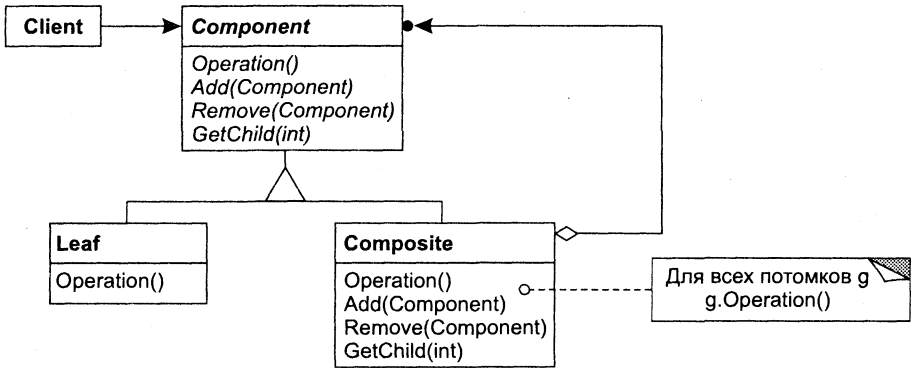


Рис. П.3. Структура паттерна Компоновщик

Паттерн состоит из следующих классов.

□ Component (компонент):

- объявляет интерфейс для компокуемых объектов;
- предоставляет подходящую реализацию операций по умолчанию, общую для всех классов;
- объявляет интерфейс для доступа к потомкам и управления ими;
- определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его (необязательная возможность).

□ Leaf (примитивный объект, лист):

- представляет листовые узлы композиции и не имеет потомков;
- определяет поведение примитивных объектов в композиции.

□ Composite (составной объект, контейнер):

- определяет поведение компонентов, у которых есть потомки;
- хранит компоненты-потомки;
- реализует относящиеся к управлению потомками операции в интерфейсе класса Component.

□ Client (клиент):

- манипулирует объектами композиции через интерфейс Component.

Основой паттерна является абстрактный класс Component, который представляет одновременно и примитивы, и контейнеры. В нем объявлены операции, специфичные для каждого вида графического объекта (например, функция рисования для графических примитивов) и общие для всех составных объектов, например операции для работы с потомками.

Каждый примитивный объект Leaf переопределяет реализуемые им специфичные операции, а контейнер Composite — операции для работы с потомками. Контейнер, в свою очередь, может включать другие составные объекты.

Клиент Client использует интерфейс класса Component для взаимодействия с объектами в составной структуре. Если получателем запроса является примитивный объект Leaf, то он и обрабатывает запрос. Когда же получателем является составной объект Composite, то обычно он перенаправляет запрос своим потомкам, возможно, выполняя некоторые дополнительные операции до или после перенаправления.

Таким образом, применение компоновщика упрощает архитектуру клиента и добавление новых видов примитивов и контейнеров.

Компоновщик в том или ином виде используется во многих объектно-ориентированных системах: при построении интерфейсов пользователя, электрических схем, деревьев синтаксического разбора, в финансовых пакетах и т. д. Приводимый ниже пример кода дает представление об использовании паттерна в калькуляторе для вычисления арифметических выражений.

В реальном калькуляторе имеется процедура синтаксического разбора исходного выражения (Parser), результатом которой является структура данных типа дерева. Узлы дерева представляют собой либо примитивные операнды типа Value (аналог объектов класса Leaf), либо составные операнды одного из четырех типов, предназначенные для реализации арифметических операций над двумя операндами. Они являются потомками класса DoubleOperand (аналог класса Composite). Класс Expression в рассматриваемом примере исполняет роль класса Component. Он содержит общий для всех классов иерархии метод getValue(), который и составляет интерфейс компонуемых объектов.

С целью упрощения примера здесь не рассматривается процедура синтаксического разбора и, более того, результат этого разбора представлен в упрощенном виде как объект класса ClientExpression.

```
// expr.h
class Expression {
public:
    virtual double getValue() = 0;
};

class DoubleOperand: public Expression {
protected:
    Expression* e1;
    Expression* e2;
public:
    DoubleOperand(Expression* e1, Expression* e2): e1(e1), e2(e2) {}
    virtual double getValue() = 0;
};

class Addition: public DoubleOperand {
public:
    Addition(Expression* e1, Expression* e2): DoubleOperand(e1, e2) {}
    virtual double getValue() { return e1->getValue() + e2->getValue(); }
};
```

```

class Subtraction: public DoubleOperand {
public:
    Subtraction(Expression* e1, Expression* e2): DoubleOperand(e1, e2) {}
    virtual double getValue() { return e1->getValue() - e2->getValue(); }
};

class Multiplication: public DoubleOperand {
public:
    Multiplication(Expression* e1, Expression* e2): DoubleOperand(e1, e2) {}
    virtual double getValue() { return e1->getValue() * e2->getValue(); }
};

class Division: public DoubleOperand {
public:
    Division(Expression* e1, Expression* e2): DoubleOperand(e1, e2) {}
    virtual double getValue() { return e1->getValue() / e2->getValue(); }
};

class Value: public Expression {
private:
    double v;
public:
    Value(double v): v(v) {}
    virtual double getValue() { return v; }
};

// client.h
#include <string>
#include "expr.h"
using namespace std;

class ClientExpression {
private:
    // результат синтаксического разбора выражения expression
    Expression* v1, v2, v3, v4, v5;
    Expression* add, sub, mul, div, result;
    string      expression;

public:
    ClientExpression();
    ~ClientExpression();
    double getValue();
    string getString() { return expression; }
};

// client.cpp
#include "client.h"

ClientExpression::ClientExpression() {
    v1 = new Value(5);
    v2 = new Value(100);

```

```
v3 = new Value(3);
v4 = new Value(4);
v5 = new Value(2);
mul = new Multiplication(v4, v5);
div = new Division(v2, v3);
add = new Addition(v1, div);
sub = new Subtraction(add, mul);
result = sub;
expression = "5 + 100 / 3 - 4 * 2";
}

ClientExpression::~ClientExpression() {
    delete v1; delete v2; delete v3; delete v4; delete v5;
    delete add; delete sub; delete mul; delete div;
}

double ClientExpression::getValue() {
    return result->getValue();
}

// main.cpp
#include <iostream>
#include "client.h"
using namespace std;

int main() {
    ClientExpression e;
    cout << "expression: " << e.getString() << " = " << e.getValue() << endl;
    return 0;
}
```

Литература

1. *Павловская Т.* С/С++. Программирование на языке высокого уровня. — СПб.: Питер, 2001 — 460 с.
2. *Павловская Т., Щупак Ю.* С/С++. Структурное программирование: Практикум. — СПб.: Питер, 2002 — 240 с.
3. *Элджер Д.* С++: библиотека программиста. — СПб.: Питер, 2000 — 320 с.
4. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами на С++. — М.: Бином, 1998 — 560 с.
5. *Страуструп Б.* Язык программирования С++. — СПб.: Бином, 1999 — 991 с.
6. *Ласло М.* Вычислительная геометрия и компьютерная графика на С++. — М.: Бином, 1997 — 304 с.
7. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2001 — 368 с.
8. *Саттер Г.* Решение сложных задач на С++. Серия С++ In-Depth. Т. 4. — М.: Издательский дом «Вильямс», 2002 — 400 с.
9. *Либерти Д.* Освой самостоятельно С++ за 21 день. — М.: Издательский дом «Вильямс», 2000 — 816 с.
10. *Влиссидес Дж.* Применение шаблонов проектирования. Дополнительные штрихи. — М.: Издательский дом «Вильямс», 2003 — 144 с.
11. *Кениг Э., Му Б.* Эффективное программирование на С++. Серия С++ In-Depth. Т. 2. — М.: Издательский дом «Вильямс», 2002 — 384 с.
12. *Штерн В.* Основы С++. Методы программной инженерии. — М.: Лори, 2003 — 860 с.
13. *Лафоре Р.* Объектно-ориентированное программирование в С++. Классика Computer Science. 4-е изд. — СПб.: Питер, 2003 — 928 с.
14. *Аммерааль Л.* STL для программистов на С++. — М.: ДМК, 1999 — 240 с.
15. *Шаллоуэй А., Гротт Д.* Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию. — М.: Издательский дом «Вильямс», 2002 — 288 с.
16. *Александреску А.* Современное проектирование на С++. Серия С++ In-Depth. Т. 3. — М.: Издательский дом «Вильямс», 2002 — 336 с.

Алфавитный указатель

A

abort(), 122
at, string, 187

B

back, 203, 204
back_inserter, итератор вставки, 213
bad, ios, 148
badbit, ios, 148

C

c_str, string, 188
catch, 120
cin, 144
clear, ios, 148
close, 183
const, 24
copy, string, 188
copy, алгоритм, 213
count, 216
count, алгоритм, 208
count_if, алгоритм, 209
cout, 144

D

dec, манипулятор, 145
deque, 202

E

empty, 204
endl, манипулятор, 145
ends, 146
eof, ios, 148

eofbit, ios, 148
erase, 203, 216
erase, string, 188

F

fail, ios, 148
failbit, ios, 148
find, string, 188
find, алгоритм, 208
find_first_of, string, 188
find_if, алгоритм, 209
find_last_of, string, 188
flush, 146
for_each, алгоритм, 209
front, 203
front_inserter, итератор вставки, 213
fstream, 181

G

gcount, ios, 147
get, ios, 147
getline, ios, 147
good, ios, 148
goodbit, ios, 148

H

hex, манипулятор, 146

I

ifstream, 181
includes, алгоритм, 216
insert, 203, 216
insert, string, 188
inserter, итератор вставки, 213

iomanip, 145
ios, 144, 146
iostream, 144
istream, 144
istringstream, 185

L

length, string, 188
list, 202

M

map, 215, 217
merge, алгоритм, 214
multimap, 215
multiset, 215

O

ост, манипулятор, 146
ofstream, 181
open, 182
ostream, 144
ostringstream, 185
out_of_range, string, 187

P

pair, 217
peek, ios, 147
pop, 206
pop_back, 203
pop_front, 203
priority_queue, 207
private, 15, 59
protected, 15, 59
public, 15, 59
push_back, 203
push_front, 203
put, ios, 147

Q

queue, 207

R

rdbuf, ios, 147
rdstate(), 148
read, ios, 147
replace, string, 188
rfind, string, 188
RTTI, 102

S

search, алгоритм, 210
set, 215, 216
set_intersection, алгоритм, 216
set_terminate(), 123
set_union, алгоритм, 216
setfill, 146
setprecision, 146
setw, 146
size, string, 188
sort, алгоритм, 210
sstream, 185
stack, 206
state, 148
static, 32
STL, 196
streambuf, 144
string, 73, 186
stringstream, 185
substr, string, 188
swap, 204
swap, string, 188

T

template, 106
terminate(), 122
throw, 120
top, 206
try, 120
typename, 106, 205

U

unexpected(), 125
using, 81

V

vector, 74, 202
virtual, 63

W

write, ios, 147

B

ввод/вывод кириллицы, 23, 27

З

заголовочные файлы, 24

И

- итератор, 196, 197
 - вставки, 213
 - входной, 199
 - выходной, 199
- двунаправленный, 199
- обратный, 213
- объявление объектов типа итератор, 198
- организация цикла просмотра
 - элементов контейнера, 198
- основные операции, 198
- произвольного доступа, 199
- прямой, 199

К

- класс, 14
 - встроенные функции (методы), 18
 - деструктор, 17
 - инициализаторы конструктора, 27
 - инкапсуляция, 15
 - интерфейс, 15
 - константные методы, 24
 - конструктор, 17
 - конструктор копирования, 28
 - конструктор по умолчанию, 18
 - метод, 15
 - методы доступа к полям, 19
 - методы, определенные вне класса, 19
 - объявление массива объектов, 18
 - отладочная печать в конструкторе и деструкторе, 24
 - перегрузка операций, 29
 - операция инкремента, 30
 - операция присваивания, 31
 - поле, 15
 - статические элементы, 32
 - элемент, 15
- классы стандартных потоков
 - манипуляторы ввода/вывода, 145
 - параметризованные, 146
 - простые, 146
 - обработка ошибок, 147
 - объекты и методы, 144
 - операция вставки в поток, 145
 - операция извлечения из потока, 145
 - перегрузка операций извлечения
 - и вставки, 148
 - поток ввода, 144
 - поток вывода, 144

- контейнер, 196
 - ассоциативный, 197
 - двусторонняя очередь (deque), 202
 - методы
 - back, 203, 204
 - begin, 199
 - count, 216
 - empty, 204
 - end, 199
 - erase, 205, 216
 - front, 203
 - insert, 205, 216
 - pop_back, 203, 204
 - pop_front, 203
 - push_back, 203
 - push_front, 203
 - swap, 204
 - общие для всех контейнеров, 200
 - множество (set), 215
 - мультимножество (multiset), 215
 - очередь (queue), 207
 - очередь с приоритетами
 - (priority_queue), 207
 - последовательный, 197
 - словарь (map), 215, 218
 - список (list), 202
 - стек (stack), 206
 - унифицированные типы, 200
 - шаблон функции для вывода
 - содержимого, 205
- критерии качества декомпозиции объекта, 13
- критерии качества декомпозиции проекта
 - связанность между компонентами, 14
 - сцепление внутри компонента, 14

Н

- наследование классов, 58
 - базовый класс, 59
 - виртуальные методы, 63
 - виртуальный деструктор, 64
 - доступ к объектам иерархии, 62
 - конструкторы и деструкторы
 - в производном классе, 60
 - закрытое, 59
 - замещение функций базового
 - класса, 59
 - защищенное, 59
 - иерархия, 59
 - множественное, 59
 - устранение неоднозначности, 61
 - открытое, 59

наследование классов (*продолжение*)
 полиморфизм, 64
 производный класс, 59
 простое, 59

О

обобщенный алгоритм, 196, 201
обработка исключений, 119
 блок try, 120
 исключения в деструкторе, 129
 исключения в конструкторе, 125
 классы исключений, 123
 неперехваченные исключения, 122
 обработчик catch, 120
 оператор throw, 120
 перехват исключения, 121
 спецификации исключений, 124
объект класса, 15
отношения между классами
 агрегация, 67
 ассоциация, 66
 зависимость (использование), 68
 композиция, 68
 наследование, 67

П

паттерн, 241
паттерны
 поведения, 247
 порождающие, 243
 структурные, 245
поточковый класс, 143
предикат, 209
предикаты стандартной библиотеки, 211

С

стандартная библиотека шаблонов, 196
строки класса string, 186
 конструкторы, 187
 метод at(), 187
 операции, 187
строковые потоки, 185
 метод str(), 185
 применение, 185

У

унифицированный язык
 моделирования UML, 65
 диаграмма видов деятельности, 89
 диаграмма классов, 65

Ф

файловые потоки, 181
 конструкторы, 182
 метод close(), 183
 метод open(), 182
функциональный объект, 211

Ш

шаблон класса, 105
 инстанцирование, 105, 107
 использование функциональных
 объектов, 110
 определение, 106
 организация исходного кода, 107
 параметры, 108
 специализация, 110
шаблоны проектирования, 241
 шаблон DoubleSwitch, 81
 шаблон Switch, 71

*Павловская Татьяна Александровна,
Щупак Юрий Абрамович*

**С++. Объектно-ориентированное программирование
Практикум**

Заведующий редакцией
Ведущий редактор
Литературный редактор
Художник
Иллюстрации
Корректоры
Верстка

*А. Кривцов
Ю. Суркис
А. Пасечник
Н. Биржаков
М. Жданова
С. Беляева, И. Смирнова
Р. Гришинов*

Лицензия ИД № 05784 от 07.09.01.

Подписано к печати 09.03.06. Формат 70×100/16. Усл. п. л. 21,93.

Доп. тираж 3500. Заказ 575

ООО «Питер Принт», 196105, Санкт-Петербург, ул. Благодатная, д. 67в.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2;
95 3005 — литература учебная.

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»
190005, Санкт-Петербург, Измайловский пр., 29.