ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

ИСПОЛЬЗОВАНИЕ ПРОЦЕДУРНОЙ И ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ ПАРАДИГМ ПРОГРАММИРОВАНИЯ

Методические указания к лабораторным работам

Легалов А.И., Легалов И.А.

2017

ВВЕДЕНИЕ

Парадигма (от греческого παράδειγμα – пример, модель, образец) – в философии, социологии исходная концептуальная схема, модель постановки проблем и их решения, методов исследования, господствующих в течение определенного исторического периода в научном сообществе. Смена парадигм представляет собой научную революцию или эволюционный переход.

Парадигма программирования – это парадигма, определяющая некоторый цельный набор идей и рекомендаций, формирующих стиль и технику написания программ. Например, в объектно-ориентированном программировании программист рассматривает программу как набор взаимодействующих объектов, тогда как в функциональном программировании программа представляется в виде цепочки вычисления функций.

Существуют различные стили написания программ, каждый из которых имеет свои преимущества и недостатки. Каждая из парадигм программирования может успешно применяться в некоторой предметной области и быть менее удобной, если ее попытаться использовать при решении других задач. Одним из важных факторов, определяющих использование того или иного стиля, является стремление к достижению заданных критериев качества разрабатываемой программы, среди которых можно выделить:

Корректность (правильность). Программа обеспечивает правильную обработку на правильных данных.

Устойчивость. Программа «элегантно» завершает обработку ошибок.

Расширяемость. Программа может легко адаптироваться к изменяющимся требованиям.

Многократность использования. Программа может использоваться и в других системах, а не только в той, для которой была создана.

Совместимость. Может легко использоваться с другим программным обеспечением.

Эффективность. Эффективное использование времени, компьютерной памяти, дискового пространства и т. д.

Переносимость (мобильность). Программу можно легко перенести на другие аппаратные и программные средства.

Верификация. Простота проверки, легкость разработки тестов для обнаружения ошибок, легкость обнаружения мест, где программа потерпела неудачу и т. д.

Поддержка целостности. Наличие внутренней защиты от неправильного обращения и неправильного употребления.

Легкость использования. Не возникает проблем в восприятии написанного кода и интерфейса для пользователей и будущих программистов.

Для достижения тех или иных критериев существуют различные приемы написания кода, которые иногда могут противоречить друг другу. Вместе

с тем имеется ряд рекомендаций, которые должны соблюдаться при написании любой программы независимо от избранной парадигмы программирования. К ним относятся: модульность, функциональная и структурная декомпозиция, отсутствие прямых связей через глобальные переменные и др.

Постепенное расширение программной системы – один из основных критериев, обуславливающих ее успешное создание, эксплуатацию и развитие. Невозможно за один проектный цикл построить большую программу, удовлетворяющую всем предъявляемым требованиям, что объясняется следующими факторами:

Требования к программному продукту могут меняться не только во время разработки, но и во время эксплуатации. Выявляется потребность в новых функциях, появляются новые условия использования. Все это ведет к необходимости вносить дополнительные расширения в уже написанный код.

Разработка больших программных систем – длительный и кропотливый процесс, требующий тщательной проработки и отладки. При попытках создать всю систему сразу разработчики сталкиваются с проблемами, обусловленными большой размерностью решаемой задачи. Поэтому, даже при хорошо известном наборе реализуемых функций, целесообразно вести инкрементальную разработку программ, постепенно добавляя и отлаживая новые функции на каждом витке итеративного цикла разработки. Сдача программы в эксплуатацию при этом осуществляется итеративно.

Лабораторные работы по дисциплине «Технология программирования» направлены на практическое закрепление методов эволюционной разработки программ с применением модульного программирования, процедурного и объектно-ориентированного стилей.

Отличие парадигм программирования проявляется в разнообразии реализаций моделей состояния и поведения, а также отношений между этими понятиями, осуществляемых через такие элементарные программные объекты, как данные и операции. В процедурном подходе абстрагирование от конкретных экземпляров достигается за счет введения понятий ***«абстрактный тип данных»*** и «***процедура»*** (понятие «***функция»*** используется как синоним процедуры). В объектно-ориентированном программировании «базовыми кирпичиками» являются ***класс*** и ***метод класса***. Эти элементарные понятия используются для построения составных программных объектов путем объединения ***в агрегаты*** и разделения ***по категориям***. Категорию Г. Буч [2] называет иерархией типа «is-a». Она также трактуется как ***обобщение*** программных объектов [6]. ***Агрегаты*** и ***обобщения*** используются при конструировании композиций данных и процедур. В каждой из существующих парадигм программирования вопросы такого конструирования композиций решаются по-своему, что и вносит определенные отличительные черты.

Цель работы

Изучение основных приемов модульной и эволюционной разработки программ с применением процедурной и объектно-ориентированной парадигм программирования. Анализ взаимосвязи между артефактами (программными объектами, разработанными программистом). Описание зависимостей между артефактами.

Порядок выполнения

1. Ознакомиться с описанием лабораторной работы.
2. Получить вариант задания у преподавателя. Содержание заданий выбирается на основе данных, представленных в разд. 1.
3. Разработать процедурную и объектно-ориентированную программы в соответствии с условием задания.
4. Провести отладку и тестирование разработанных программ на одинаковых, заранее подготовленных наборах данных. Количество тестовых наборов данных – не менее пяти. Число уникальных элементов в тестовых наборах должно варьироваться от нуля до 100 и более. При необходимости, программа должна правильно обрабатывать переполнение по излишним данным.
5. Описать зависимость между артефактами процедурной и объектно-ориентированной программ.
6. Зафиксировать для отчета основные характеристики процедурной и объектно-ориентированной программ, такие как: общее количество модулей, число интерфейсных модулей и модулей реализации, количество артефактов, общий размер исходных текстов.
7. Провести сравнительный анализ процедурной и объектно-ориентированной программ по полученным характеристикам.

Содержание отчета

Полный отчет предоставляется в электронном виде, целиком размещаясь в специально выделенном каталоге. Он должен содержать:

1. Пояснительную записку, содержащую:
   a) описание полученного задания;
   b) описание зависимостей между артефактами и модулями процедурной и объектно-ориентированной программ;
   c) основные характеристики процедурной и объектно-ориентированной программ, разработанных в ходе выполнения лабораторной работы;

d) выводы, полученные в ходе сравнительного анализа процедурной и объектно-ориентированной программ.

2. Исходные тексты процедурной и объектно-ориентированной программ, каждая из которых должна находиться в своем подкаталоге.

3. Тестовые наборы данных.

4. Результаты работы процедурной и объектно-ориентированной программ для различных тестовых наборов.

1. ВАРИАНТЫ ЛАБОРАТОРНЫХ РАБОТ

В ходе выполнения лабораторной работы необходимо написать процедурную и объектно-ориентированную программы, которые должны быть оформлены в виде консольных приложений, удовлетворяющих следующим требованиям:

1. Запуск программы осуществляется из командной строки, в которой указываются: имя запускаемой программы; имя файла с исходными данными; имя файла с выходными данными.

2. Для каждого программного объекта, загружаемого в контейнер, исходный файл должен содержать, признак, а также список необходимых параметров. Этот список должен быть представлен в формате, удобном для обработки компьютером.

3. Функция, формирующая выходной файл, должна выдавать в него программные объекты и их текущие параметры. Помимо этого необходимо вывести информацию об общем количестве объектов, содержащихся в контейнере. Информация должна быть представлена в форме, удобной для восприятия пользователем.

4. Программа должна иметь модульную структуру, соответствующую выданному варианту задания.

5. Процедурная программа должна иметь организацию обобщений, соответствующую выданному варианту задания.

1.1. Выбор варианта задания

Каждый из вариантов собирается из четырех независимых компонент: условия задачи, типа контейнера, вида модульной структуры программы, способа построения обобщения в процедурной программе. Выбор этих составляющих осуществляется на основе номера варианта задания (от 1 до 648), выданного преподавателем.

Пусть N_{var} – номер варианта задания, div – операция целочисленного деления, mod – операция выделения остатка от целочисленного деления. Тогда номер условия задачи N_{task} вычисляется следующим образом:

$$N_{task} = ((((N_{var} - 1) \text{ div } 3) \text{ div } 3) \text{ div } 6) \text{ mod } 12 + 1.$$

Номер контейнера N_{cont} определяется по формуле:

$$N_{cont} = (((N_{var} - 1) \text{ div } 3) \text{ div } 3) \bmod 6 + 1.$$

Номер модульной структуры N_{mod} определяется как:

$$N_{mod} = ((N_{var} - 1) \text{ div } 3) \bmod 3 + 1.$$

Номер обобщения N_{union} вычисляется следующим образом:

$$N_{union} = (N_{var} - 1) \bmod 3 + 1.$$

1.2. Условие задачи

Условие задачи определяет основное задание для составления программы. Множество различных заданий представлено в табл. 1.

Таблица 1

Условия задач на лабораторные работы

Обобщенный артефакт, используемый в задании	Базовые альтернативы (уникальные параметры, задающие отличительные признаки альтернатив)	Параметры, общие для всех альтернатив
1. Плоская геометрическая фигура.	1. Круг (целочисленные координата центра окружности, радиус) 2. Прямоугольник (целочисленные координаты левого верхнего и правого нижнего углов)	Цвет фигуры (перечислимый тип) = {красный, оранжевый, желтый, зеленый, голубой, синий, фиолетовый}
2. Объемная геометрическая фигура.	1. Шар (целочисленный радиус) 2. Параллелепипед (три целочисленных ребра)	Плотность материала фигуры (действительное число)
3. Квадратные матрицы с целочисленными элементами	1. Обычный двумерный массив 2. Диагональная (на основе одномерного массива)	Размерность – целое
4. Транспорт	1. Самолеты (дальность полета – целое, грузоподъемность – целое) 2. Поезда (количество вагонов – целое)	Скорость – целое; Расстояние между пунктами отправления и назначения – целое
5. Фильмы	1. Игровой (режиссер – строка символов) 2. Мультфильм (способ создания – перечислимый тип = рисованный, кукольный,	Название фильма – строка символов

	пластилиновый…)	
6. Языки программирования	1. Процедурные (наличие, отсутствие абстрактных типов данных – булевская величина) 2. Объектно-ориентированные (наследование: одинарное, множественное, интерфейса – перечислимый тип)	Год разработки – короткое целое
7. Тексты, состоящие из цифр и латинских букв, зашифрованные различными способами.	1. Шифрование заменой символов (указатель на массив пар: [текущий символ, замещающий символ]; зашифрованный текст – строка символов) 2. Шифрование циклическим сдвигом кода каждого символа на n (целое число, определяющее сдвиг; зашифрованный текст – строка символов)	Открытый текст – строка символов.
8. Кладезь мудрости.	1. Афоризмы (один из авторов – строка символов) 2. Пословицы и поговорки (страна – строка символов)	Содержание – строка символов
9. Различные числа	1. Комплексные (действительная и мнимая части – пара действительных чисел) 2. Простые дроби (числитель, знаменатель – пара целых чисел)	–
10. Животные	1. Рыбы (место проживания – перечислимый тип: река, море, озеро…) 2. Птицы (отношение к перелету: перелетные, остающиеся на зимовку – булевская величина)	Название – строка символов
11. Растения	1. Деревья (возраст – длинное целое) 2. Кустарники (месяц цветения – перечислимый тип)	Название – строка символов

12. Автомобильный транспорт	1. Грузовик (грузоподъемность кг – целое) 2. Автобус (пассажировместимость – короткое целое)	Мощность двигателя – целое

1.3. Организация контейнера

Тип контейнера определяет способ группировки альтернативных элементов, для каждой из задач, в список. Среди всего многообразия способов группировки выделим шесть следующих вариантов.

1. Контейнер на основе одномерного массива элементов с проверкой на переполнение.
2. Контейнер на основе однонаправленного линейного списка.
3. Контейнер на основе однонаправленного кольцевого списка.
4. Контейнер на основе двунаправленного линейного списка.
5. Контейнер на основе двунаправленного кольцевого списка.
6. Хеш массив с разрешением конфликтов через дополнительные списки для элементов с одинаковым значением ключа.

При выполнении лабораторной работы необходима поддержка следующих операций:

• Заполнение контейнера данными, поступающими из входного потока. Полученный элемент должен быть размещен в контейнере любым из способов, выбранных программистом.

• Вывод значений всех элементов в выходной поток. Выводятся параметры элементов, размещенных в контейнере. Порядок вывода элементов определяется порядком размещения элементов в контейнере. Вывод осуществляется в стандартный поток и в файл, указанный в командной строке.

1.4. Организация модульной структуры программы

Разрабатываемая программа может быть разбита на модули различными способами. Например, самый простой вариант – это использование одного модуля (файла) для размещения всей программы. Он же является и самым неудобным при разработке больших, эволюционно расширяемых программ. В связи с небольшим размером программ, создаваемых в ходе выполнения лабораторной работы, разбиение на модули больше несет методическую, нежели практическую нагрузку. Вместе с тем, это разбиение позволяет выявить зависимость между логической структурой программы и ее физическим размещением. В качестве альтернативных вариантов предлагаются следующие способы разбиения:

1. Клиентская часть программы (тестовая функция) размещается в отдельном модуле. Основная часть кода, определяющего реализацию функций

или методов, содержится в другом (едином) модуле. Программные объекты, обеспечивающие связь между основной и клиентской частями (абстрактные типы данных, сигнатуры функций, описания классов), размещается в отдельном интерфейсном модуле (заголовочном файле).

2. Разбиение на модули осуществляется по объектному принципу. То есть описание абстракций и функций их обработки группируются по виду абстракций. При этом проводится разделение описания типов, сигнатур и классов от их реализации. Клиентский модуль связан с основной частью программы только минимально необходимым интерфейсом.

3. Каждый абстрактный тип, функция, класс, метод размещается в своей единице компиляции. При этом описание артефакта по возможности отделено от его реализации (размещены в разных файлах).

1.5. Организация обобщений в процедурной программе

Использование процедурного программирования позволяет использовать разнообразные варианты для построения одних и тех же программных объектов. С одной стороны, такая гибкость позволяет выбирать более эффективные технические решения, но с другой стороны затрудняет реализацию и ведет к менее надежным программам за счет использования структур, удовлетворяющих другим критериям качества. Для изучения этих особенностей процедурного подхода в лабораторной работе предлагаются следующие варианты организации обобщений:

1. Обобщение, построенное на основе непосредственного включения специализаций.

2. Обобщение, построенное на основе косвенного связывания через универсальный указатель (***void****** в языке C++).

3. Обобщение, использующее общую основу (базовую структуру), которая повторяется в каждой из отдельно прописанных специализаций.

Использование непосредственного включения обеспечивается использованием следующей структуры обобщения:

```
struct обобщение1 {
    ключ;
    union {специализация1; специализация2; …};
}
```

Специализации являются абстрактными типами, непосредственно включенными в указанную структуру. Ключ построен на основе целочисленного или перечислимого типа.

Использование косвенного связывания через универсальный указатель позволяет подключать любые программные объекты в качестве специализаций, определяя тип конкретного объекта только с помощью ключа. Структура обобщения для этого варианта выглядит следующим образом:

```
struct обобщение2 {ключ; void* на что угодно;}
```

Третий из вариантов предполагает использовать в качестве обобщения базовую структуру, содержащую только ключ:

```
struct обобщение 3 {ключ;}
```

Тогда любая специализация, подключаемая к указателю на эту структуру, будет иметь следующий вид:

```
struct специализация из 3 {
    ключ; основа специализации i
}
```

Реализация того или иного варианта обобщения в процедурной программе позволит оценить гибкость используемого технического решения и сопоставить его с возможностями, предоставляемыми объектно-ориентированным подходом.

2. ПРИМЕР ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Использование процедурной и объектно-ориентированной парадигм программирования рассмотрим на примере простой программы, осуществляющей различные манипуляции с заданным набором геометрических фигур, хранимых в едином контейнере, определяющем некоторую «картину». Исходными данными для задачи будут следующие:

1. Условие задачи приведено в табл. 2. Исходный комплект геометрических фигур состоит из треугольников и прямоугольников.
2. Контейнер для хранения геометрических фигур построен на основе одномерного массива. Контроль переполнения контейнера отсутствует.
3. Программа должна иметь модульную структуру с размещением каждого артефакта в отдельной единице компиляции.
4. Обобщение реализовано на основе непосредственного включения.

Таблица 2

Обобщенный артефакт, используемый в задании	Базовые альтернативы (уникальные параметры, задающие отличительные признаки альтернатив)	Параметры, общие для всех альтернатив
0. Плоских геометрические фигуры различного размера и типа.	1. Прямоугольник (две целочисленные стороны) 2. Треугольник (три целочисленные стороны)	–

Вполне естественно, что представленная простая задача не позволяет рассмотреть все особенности проектирования сложных программных систем. Однако, даже при решении подобных простых задач можно сравнить изучаемые парадигмы программирования и выявить ряд типичных особенностей и приемов, возникающих при написании реальных программ. В примере от-

сутствует код, контролирующий корректное поведение программы в ряде ситуаций. Это сделано для сокращения объема исходных текстов. Предполагается, что в ходе выполнения лабораторных работ этот код должен быть написан, так как информация об его применении была рассмотрена в ранее изучаемых дисциплинах. В рамках данного примера программа разбита на модули таким образом, что каждый из программных объектов находится в своей единице компиляции и в отдельном интерфейсном модуле.

2.1. Разработка и анализ процедурной программы

Процедурная программа базируется на абстрактных типах данных и независимых процедурах. Каждое из этих понятий обеспечивает независимую трактовку для артефактов состояния и поведения. При этом для построения агрегатов и обобщений используются свои средства. Демонстрационный пример написан на языке программирования C++, позволяющем использовать как процедурный, так и объектно-ориентированный стиль. Исходные тексты процедурной программы представлены в приложении А.

2.1.1. Процедурный контейнер геометрических фигур

В рамках разрабатываемой программы по агрегатному принципу строится контейнер, используемый для хранения геометрических фигур. Контейнер содержит следующие поля:

• переменную, определяющую текущее количество элементов, размещенных в контейнере;

• одномерный массив указателей на геометрические фигуры.

Следует отметить, что одномерный массив, используемый при построении контейнера, может хранить геометрические фигуры, используя для этого непосредственное включение [4]. Выбор для хранения косвенного связывания обусловлен в данном случае субъективными факторами и стремлением сделать контейнер независимым от особенностей геометрических фигур. Предоставление информации, необходимой для обработки только своих данных, не является прерогативой только объектно-ориентированных программ. Хорошо структурированные процедурные программы, построенные с использованием модульного программирования, тоже следуют этому принципу. Это в дальнейшем облегчает эволюционное расширение программ. Использование в данной ситуации непосредственного включения элементов в контейнер привело бы к необходимости подключения в каждую из функций его обработки дополнительных заголовочных файлов, содержащих всю информацию о геометрических фигурах. Выбранные решения позволяют следующим образом представить абстрактный тип данных, описывающий контейнер:

```
//-------------------------------------------------
// Ссылка на описание геометрической фигуры.
// Знание структуры самой фигуры для представленной
// реализации контейнера не требуется
struct shape;

//-------------------------------------------------
// Простейший контейнер на основе одномерного массива
struct container {
    enum {max_len = 100}; // максимальная длина
    int len;              // текущая длина
    shape *cont[max_len];
};
```

этот абстрактный тип хранится в отдельном заголовочном файле ***container_atd.h***, выполняющем роль интерфейсного модуля.

Функции, обеспечивающие обработку контейнера, располагаются в отдельном модуле реализации. Их состав определяется из условия задачи. Помимо этого добавляются вспомогательные функции, осуществляющие инициализацию и очистку контейнера. В целом их состав выглядит следующим образом:

• функция ***void Init(container &c)*** осуществляет инициализацию контейнера, заключающуюся в установке его начального состояния (просто обнуляется количество элементов);

• функция утилизации ***void Clear(container &c)***, осуществляет удаление фигур, размещенных в контейнере и установку его в начальное состояние;

• функция ***In(container &c, ifstream &ifst)*** осуществляет ввод геометрических фигур в контейнер из указанного потока, который может быть связан с файлом или стандартным вводом;

• функция ***void Out(container &c, ofstream &ofst)*** осуществляет вывод содержимого контейнера в заданный выходной поток;

В соответствии с выбранной модульной структурой, реализация функций прописывается в отдельных файлах, являющихся независимыми единицами компиляции ***container_Constr.cpp***, ***container_Out.cpp***, ***container_Out.cpp*** соответственно:

```
//-------------------------------------------------
// Инициализация контейнера
void Init(container &c) { c.len = 0; }

//-------------------------------------------------
// Очистка контейнера от элементов (освобождение памя-
ти)
```

```
void Clear(container &c) {
    for(int i = 0; i < c.len; i++){
      delete c.cont[i];
    }
    c.len = 0;
}

//-----------------------------------------------------
// Ввод содержимого контейнера из указанного потока
void In(container &c, ifstream &ifst) {
    while(!ifst.eof()) {
      if((c.cont[c.len] = In(ifst)) != 0) { c.len++; }
    }
}

//-----------------------------------------------------
// Вывод содержимого контейнера в указанный поток
void Out(container &c, ofstream &ofst) {
    ofst << "Container contains " << c.len
         << " elements." << endl;
    for(int i = 0; i < c.len; i++) {
      ofst << i << ": ";
      Out(*(c.cont[i]), ofst);
    }
}
```

2.1.2. Процедурная реализация геометрических фигур

Обобщенное понятие геометрической фигуры в данном конкретном случае формируется с использованием непосредственного включения в объединение основ специализации. Основы специализации (прямоугольник и треугольник) представлены в отдельных заголовочных файлах следующим образом:

```
//-----------------------------------------------------
// прямоугольник
struct rectangle {
    int x, y; // ширина, высота
};

//-----------------------------------------------------
// треугольник
struct triangle {
    int a, b, c; // стороны
```

```
};
```

Эти заголовочные файлы подключаются к файлу, содержащему обобщенную фигуру, в которой представленные основы выступают в роли конкретных специализаций:

```
//-------------------------------------------------
#include "rectangle_atd.h"
#include "triangle_atd.h"
//-------------------------------------------------
// структура, обобщающая все имеющиеся фигуры
struct shape {
    // значения ключей для каждой из фигур
    enum key {RECTANGLE, TRIANGLE};
    key k; // ключ
    // используемые альтернативы
    union { // используем прямое включение
      rectangle r;
      triangle t;
    };
};
```

Обработка геометрической фигуры в лабораторной работе связана с реализацией функций ее ввода ***shape* In(ifstream &ifst)*** и вывода ***void Out(shape &s, ofstream &ofst)***, размещенных в отдельных единицах компиляции ***shape_In.cpp*** и ***shape_Out.cpp*** соответственно:

```
//-------------------------------------------------
void In(rectangle &r, ifstream &ist);
void In(triangle  &t, ifstream &ist);

//-------------------------------------------------
// Ввод параметров обобщенной фигуры из файла
shape* In(ifstream &ifst) {
    shape *sp;
    int k;
    ifst >> k;
    switch(k) {
    case 1:
      sp = new shape;
      sp->k = shape::key::RECTANGLE;
      In(sp->r, ifst);
      return sp;
    case 2:
      sp = new shape;
      sp->k = shape::key::TRIANGLE;
      In(sp->t, ifst);
```

```
      return sp;
    default:
      return 0;
    }
}

//-----------------------------------------------------
void Out(rectangle &r, ofstream &ofst);
void Out(triangle  &t, ofstream &ofst);

//-----------------------------------------------------
// Вывод параметров текущей фигуры в поток
void Out(shape &s, ofstream &ofst) {
    switch(s.k) {
    case shape::key::RECTANGLE:
      Out(s.r, ofst);
      break;
    case shape::key::TRIANGLE:
      Out(s.t, ofst);
      break;
    default:
      ofst << "Incorrect figure!" << endl;
    }
}
```

При этом обобщенные функции обращаются за исходными данными и для вывода результатов к обработчикам основ специализаций, непосредственно манипулирующим исходными геометрическими фигурами:

```
//-----------------------------------------------------
// Ввод параметров прямоугольника из файла
void In(rectangle &r, ifstream &ifst) {
    ifst >> r.x >> r.y;
}

//-----------------------------------------------------
// Ввод параметров треугольника из потока
void In(triangle &t, ifstream &ifst)
{
    ifst >> t.a >> t.b >> t.c;
}

//-----------------------------------------------------
// Вывод параметров прямоугольника в форматируемый по-
ток
```

```
void Out(rectangle &r, ofstream &ofst) {
    ofst << "It is Rectangle: x = " << r.x
        << ", y = " << r.y << endl;
}

//-----------------------------------------------------
// Вывод параметров треугольника в поток
void Out(triangle &t, ofstream &ofst) {
    ofst << "It is Triangle: a = "
        << t.a << ", b = " << t.b
        << ", c = " << t.c << endl;
}
```

2.1.3. Реализация клиентской части процедурной программы

Клиентская часть приложения, независимо от заданной модульной структуры программы, находится в отдельной единице компиляции и содержит функции, обеспечивающие тестирование разработанного приложение. В данном случае это выполнение ввода в контейнер геометрических фигур, вывода содержимого контейнера и его очистка:

```
int main(int argc, char* argv[]) {
    if(argc !=3) {
        cout << "incorrect command line! "
                "Waited: command in_file out_file"
                << endl;
        exit(1);
    }
    ifstream ifst(argv[1]);
    ofstream ofst(argv[2]);
    cout << "Start"<< endl;

    container c;
    Init(c);
    In(c, ifst);
    ofst << "Filled container. " << endl;
    Out(c, ofst);

    Clear(c);
    ofst << "Empty container. " << endl;
    Out(c, ofst);

    cout << "Stop"<< endl;
    return 0;
```

```
}
```

Исходные данные размещаются в файле, имеющем формат данных удобный для обработки компьютером. Он содержит признак геометрической фигуры и, в зависимости от типа фигуры, ее данные. Например:

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

В первой строке задан признак прямоугольник, во второй – его стороны. Третья строка содержит признак треугольника, а четвертая размеры его сторон.

Результаты должны формироваться в виде удобном для чтения пользователем. Для приведенных в качестве примера исходных данных получим следующий их вывод:

```
Filled container.
Container contains 2 elements.
0: It is Rectangle: x = 3, y = 4
1: It is Triangle: a = 1, b = 1, c = 1
Empty container.
Container contains 0 elements.
```

2.1.4. Зависимости между артефактами в процедурной программе

К основным артефактам разработанной процедурной программы относятся абстрактные типы данных и функции (процедуры). Между ними могут существовать прямые и косвенные зависимости. Прямые зависимости между артефактами необходимы для использования информации об их внутренней структуре. Например, при обработке полей агрегата или обобщения необходимо знать, как организован используемый абстрактный тип данных. Косвенные зависимости требуют знания только имени артефакта (для абстрактного типа данных) или сигнатуры используемой функции. Наличие косвенных связей ослабляет взаимосвязь между артефактами и облегчает модификацию программы. Зачастую, модули, имеющие только косвенные связи, даже не перекомпилируются при изменении влияющих на них модулей.

Рассмотрим зависимости в разработанной процедурной программе. Выделим для анализа следующие артефакты:

- абстрактные типы данных ***rectangle***, ***triangle***, ***shape***, ***container***;
- функции ***In(triangle&…)***, ***Out(triangle&…)***, ***In(rectangle&…)***, ***Out(rectangle&…)***, ***shape* In(…)***, ***Out(shape&…)***, ***Init(container&)***, ***Clear(container &c)***, ***In(container&…)***, ***Out(container&…)***, ***main(…)***.

Сведения об этих зависимостях представлены в табл. 3.

Таблица 3

Артефакт		Непосредственно связан с	Косвенно связан с
rectangle			
triangle			
shape		rectangle, triangle	
container			shape
In(rectangle&…)	сигнатура		rectangle
	тело	rectangle	
Out(rectangle&…)	сигнатура		rectangle
	тело	rectangle	
In(triangle&…)	сигнатура		triangle
	тело	triangle	
Out(triangle&…)	сигнатура		triangle
shape* In(…)	сигнатура		shape
	тело	shape[rectangle, triangle], In(rectangle&…), In(triangle&…)	rectangle, triangle
Out(shape&…)	сигнатура		shape
	тело	shape[rectangle, triangle], Out (rectangle&…), Out (triangle&…)	rectangle, triangle
Init(container&)	сигнатура		container
	тело	container	
Clear(container&)	сигнатура		container
	тело	shape[rectangle, triangle], container	
In(container&…)	сигнатура		container
	тело	container, shape* In(…)	shape
Out(container&…)	сигнатура		container
	тело	container, Out(shape&…)	shape
main(…)	сигнатура		
	тело	container, Init(container&), Clear(container&), In(container&…), Out(container&…)	

Использование в обобщении shape непосредственного включения основ специализаций triangle и rectangle ведет к его прямой зависимости от этих артефактов. Поэтому, любая модификация исходных структур связана с изменением характеристик обобщения и, следовательно, с его перекомпиляцией. Это, в частности, отражается на телах функций ***shape* In(…)***, ***Out(shape&…)*** и ***Clear(container&)***. В них специализации ***rectangle*** и ***triangle*** не используются. Однако их структура доступна из-за прямого включения в

shape. Использование косвенного связывания внутри ***shape*** возможно и привело бы в данном случае к меньшей зависимости артефактов.

2.1.5. Модульная структура процедурной программы

Модульная структура должна сохранять ранее установленные логические зависимости между разработанными артефактами. Разнесение артефактов по модулям способствует раздельной компиляции программы и облегчает ее групповую разработку. Грамотное разбиение на модули способствует также эволюционному расширению программы и позволяет, при добавлении в нее новых программных объектов, избавляться от модификации уже существующих единиц компиляции. Это уменьшает количество ошибок, вносимых разработчиков и особенно важно при создании и расширении больших программных систем.

Зависимости между модулями демонстрационного примера представлены в табл. 4.

Таблица 4

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое модуля
rectangle_atd.h			rectangle
triangle_atd.h			triangle
shape_atd.h	rectangle_atd.h triangle_atd.h		shape
container_atd.h			container, shape
rectangle_In.cpp	rectangle_atd.h		In(rectangle&…)
rectangle_Out.cpp	rectangle_atd.h		Out(rectangle&…)
triangle_In.cpp	triangle_atd.h		In(triangle&…)
triangle_Out.cpp	triangle_atd.h		Out (triangle&…)
shape_In.cpp	shape_atd.h In(rectangle&…) In(triangle&…)	rectangle_atd.h triangle_atd.h	shape *In(…)
shape_Out.cpp	shape_atd.h Out(rectangle&…) Out (triangle&…)	rectangle_atd.h triangle_atd.h	Out(shape&…)
container_Constr.cpp	container_atd.h shape_atd.h	rectangle_atd.h triangle_atd.h	Init(container&) Clear(container&)
container_In.cpp	container_atd.h shape *In(…)	shape*	In(container&…)

container_Out.cpp	container_atd.h Out(shape&…)		Out(container&…)
main.cpp	container_atd.h Init(container&) Clear(container&) In(container&…) Out(container&…)	shape*	main(…)

Использование модульной структуры может привести к дополнительным зависимостям между программными объектами, избыточным с точки зрения их логической взаимосвязи. В разработанной программе такая зависимость проявляется между функциями ***Init(container&)*** и ***Clear(container&)*** из-за их размещения в одном модуле. Это также ведет к избыточной зависимости функции ***Init(container&)*** от содержимого ***shape_atd.h***, ***rectangle_atd.h***, ***triangle_atd.h***. Следует отметить, что не всегда следует избавляться от излишних зависимостей, так как это ведет к большому числу очень мелких модулей, что также трудно поддается контролю.

2.2. Разработка и анализ объектно-ориентированной программы

Основные концепции объектно-ориентированной парадигмы программирования требуют, чтобы программа состояла из автономных объектов, взаимодействующих между собой посредством передачи сообщений [1]. В большинстве современных ОО языков программирования объекты являются экземплярами классов, а обобщения строятся на основе наследования и виртуализации. Язык программирования C++, используемый для написания демонстрационных примеров, поддерживает эти концепции. Механизм передачи сообщений реализован в нем через вызовы методов классов. Применение наследования и виртуализации обеспечивают инструментальную поддержку полиморфизма. Использование ОО подхода, по сравнению с процедурным, также уменьшает число разнообразных вариантов реализации одних и тех же программных объектов и «отрывает» реализацию программы от особенностей компьютерной архитектуры. Это уменьшение гибкости языков средств упрощает процесс разработки за счет меньшего анализа возможных альтернативных методов построения программных объектов, используемых при написании кода. Исходные тексты ОО демонстрационной программы приведены в приложении Б. Как и в случае с процедурной программой, используется размещение каждого класса и каждой реализации метода в отдельном модуле.

2.2.1. Объектно-ориентированный контейнер геометрических фигур

Реализация данного контейнера мало чем отличается от процедурной версии. Вместе с тем следует отметить, что инструментальная поддержка конструкторов и деструкторов позволяет упростить инициализацию и разрушение программных объектов. Программисту не нужно явно включать в программу вызовы соответствующих функций. Вместе с тем, размещение методов внутри класса увеличивает его интерфейсную часть, которая выглядит следующим образом:

```
//-------------------------------------------------
// Простейший контейнер на основе одномерного массива
class container  {
    enum {max_len = 100}; // максимальная длина
    int len; // текущая длина
    shape *cont[max_len];
public:
    void In(ifstream &ifst);  // ввод фигур
    void Out(ofstream &ofst);  // вывод фигур
    void Clear();  // очистка контейнера от фигур
    container();   // инициализация контейнера
    ~container() {Clear();} // утилизация контейнера
};
```

Контейнер содержит массив указателей на хранимые в нем разнотипные фигуры. Использование в подобных ситуациях указателей обеспечивает гибкое подключение программных объектов, наследующих свойства базового класса и поддерживается на уровне языковых средств.

Реализации методов класса размещаются в отдельных единицах компиляции. Они обеспечивают ввод данных в контейнер из подключенного входного потока, вывод содержимого контейнера в заданный выходной поток, очистку контейнера:

```
//-------------------------------------------------
// Ввод содержимого контейнера
void container::In(ifstream &ifst) {
    while(!ifst.eof()) {
      if((cont[len] = shape::In(ifst)) != 0) {
        len++;
      }
    }
}

//-------------------------------------------------
// Вывод содержимого контейнера
void container::Out(ofstream &ofst)
{
    ofst << "Container contents " << len
```

```
        << " elements." << endl;
    for(int i = 0; i < len; i++) {
      ofst << i << ": ";
      cont[i]->Out(ofst);
    }
}

//-----------------------------------------------------
// Очистка контейнера от элементов
void container::Clear() {
    for(int i = 0; i < len; i++) {
      delete cont[i];
    }
    len = 0;
}
```

Инициализация контейнера осуществляется в момент его создания конструктором класса:

```
container::container(): len(0) { }
```

2.2.2. Объектно-ориентированная реализация геометрических фигур

Основным способом построения обобщений в ОО подходе является использование в качестве основы базового класса, который в данном случае выступает в роли геометрической фигуры. Тогда производные классы выступают в качестве специализаций и могут реализовывать необходимые геометрические фигуры. Для преемственности интерфейса базового класса в производных классах используется механизм виртуальных функций. Описание базового класса, определяющего геометрическую фигуру, выглядит следующим образом:

```
//-----------------------------------------------------
// Класс, обобщающает все имеющиеся фигуры.
// Является абстрактным, обеспечивая, тем самым,
// проверку интерфейса
class shape {
public:
    // иденитфикация, порождение и ввод фигуры из пото-
ка
    static  shape* In(ifstream &ifst);
    virtual void InData(ifstream &ifst) = 0; // ввод
    virtual void Out(ofstream &ofst) = 0;    // вывод
protected:
    shape() {};
};
```

Он является абстрактным, что показывает на отсутствие порождаемых от него экземпляров (невозможно существование обобщенной фигуры). Чистые виртуальные функции ***virtual void InData(ifstream &ifst)*** и ***virtual void Out(ofstream &ofst)*** в дальнейшем должны быть обязательно переопределены в производных классах, от которых предполагается порождение конкретных геометрических фигур.

Следует также отметить наличие статического метода ***static shape* In(ifstream &ifst)***, порождающего на выходе экземпляр геометрической фигуры, прочитанной из файла с исходными данными. Этот метод доступен в любой точке программы вне зависимости от того могут или нет порождаться экземпляры данного класса:

```
//-----------------------------------------------------
// Ввод параметров обобщенной фигуры
// из стандартного потока ввода
shape* shape::In(ifstream &ifst) {
    shape *sp;
    int k;
    ifst >> k;
    switch(k) {
    case 1:
      sp = new rectangle; break;
    case 2:
      sp = new triangle;  break;
    default:
      return 0;
    }
    sp->InData(ifst);
    return sp;
}
```

Для того чтобы порождать экземпляры различных фигур, данный метод должен иметь информацию о структуре их классов. Поэтому, в любом случае, его целесообразно разместить в отдельной единице компиляции, так как для любых других методов базового класса информация о производных классах не нужна. Производные классы ***rectangle*** и ***triangle*** используются для порождения экземпляров конкретных геометрических фигур. Поэтому они содержат все необходимые данные:

```
//-----------------------------------------------------
// прямоугольник
class rectangle: public shape {
    int x, y; // ширина, высота
public:
    // переопределяем интерфейс класса
```

```
    void InData(ifstream &ifst);  // ввод
    void Out(ofstream &ofst);     // вывод
    rectangle() {} // создание без инициализации.
};

//-----------------------------------------------------
// треугольник
class triangle: public shape {
    int a, b, c; // стороны
public:
    // переопределяем интерфейс класса
    void InData(ifstream &ifst);  // ввод
    void Out(ofstream &ofst);     // вывод
    triangle() {} // создание без инициализации.
};
```

В каждом из этих классов также переопределены виртуальные методы ввода и вывода данных, выполняющие операции в соответствии с внутренней структурой каждого из классов. Описания указанных классов размещены в заголовочных файлах ***rectangle_atd.h*** и ***triangle_atd.h*** соответственно. Реализации методов классов разнесены по разным единицам компиляции в соответствии с указанным вариантом их размещения. Методы производных классов реализуются следующим кодом:

```
//-----------------------------------------------------
// Ввод параметров прямоугольника
void rectangle::InData(ifstream &ifst) {
    ifst >> x >> y;
}

//-----------------------------------------------------
// Вывод параметров прямоугольника
void rectangle::Out(ofstream &ofst) {
    ofst << "It is Rectangle: x = " << x
         << ", y = " << y << endl;
}

//-----------------------------------------------------
// Ввод параметров треугольника
void triangle::InData(ifstream &ifst) {
    ifst >> a >> b >> c;
}

//-----------------------------------------------------
// Вывод параметров треугольника
```

```
void triangle::Out(ofstream &ofst) {
    ofst << "It is Triangle: a = "
      << a << ", b = " << b
      << ", c = " << c << endl;
}
```

В отличие от процедурного подхода, при котором обработка альтернативных данных обычно концентрируется в одной процедуре, объектно-ориентированная парадигма программирования позволяет распределить обработку альтернатив по отдельным классам. При этом каждая специализация обрабатывается своими собственными методами и некоим образом не зависит от других альтернативных объектов.

2.2.3. Реализация клиентской части объектно-ориентированного приложения

Клиентская часть выполняет те же функции, что и ранее написанная процедурная программа. Отличие заключается только в вызове методов вместо используемого ранее вызова процедур:

```
int main(int argc, char* argv[])
{
    if(argc !=3)
    {
        cout << "incorrect command line! "
                "Waited: command in_file out_file"
                << endl;
        exit(1);
    }
    ifstream ifst(argv[1]);
    ofstream ofst(argv[2]);

    cout << "Start"<< endl;

    container c;
    c.In(ifst);
    ofst << "Filled container. " << endl;
    c.Out(ofst);

    c.Clear();
    ofst << "Empty container. " << endl;
    c.Out(ofst);

    cout << "Stop"<< endl;
    return 0;
```

}

Разработанная программа использует те же файлы с исходными данным, что и процедурная программа, а также записывает результат в выходной файл аналогичным образом.

2.2.4. Зависимости между артефактами в объектно-ориентированной программе

К основным артефактам объектно-ориентированной программы относятся классы и методы. Как и в процедурной программе, между ними существуют прямые и косвенные зависимости. Однако, эти зависимости имеют несколько иной характер, что определяется другим взаимным расположением программных объектов.

Размещение методов внутри классов оказывает прямое воздействие на сами классы. Для уменьшения зависимостей между методами и классами в языке программирования C++ реализации методов (их тела) часто выносятся в отдельные единицы компиляции. Помимо этого, существенную роль на зависимость между классами оказывает использование механизма наследования.

В разработанной объектно-ориентированной программе реализованы следующие классы: ***rectangle***, ***triangle***, ***shape***, ***container***. Зависимости между ними и методами, с учетом возможного независимого размещения тел методов, представлены в табл. 5.

Таблица 5

Артефакт		Непосредственно связан с	Косвенно связан с
shape::In(…)	сигнатура		
shape::Out(…)=0	сигнатура		
shape::InData(…)=0	сигнатура		
shape::shape() {}			
Shape		shape ::In(…), shape ::Out(…), shape::InData(…), shape:: shape() {}	
rectangle::Out(…)	сигнатура	shape	
rectangle::InData(…)	сигнатура	shape	
rectangle::rectangle() {}		shape	
rectangle		shape, rectangle::Out(…), rectangle::InData(…), rectangle::rectangle() {}	
rectangle::Out(…)	тело	rectangle[shape[…]]	
rectangle::InData(…)	тело	rectangle[shape[…]]	
triangle::Out(…)	сигнатура	shape	

triangle::InData(…)	сигнатура	shape	
triangle::triangle() {}		shape	
triangle		shape, triangle::Out(…), triangle::InData(…), triangle::triangle() {}	
triangle::Out(…)	тело	triangle[shape[…]]	
triangle::InData(…)	тело	triangle[shape[…]]	
shape ::In(…)	тело	shape, rectangle, triangle	
container::In(…)	сигнатура		
container::Out(…)	сигнатура		
container::Clear(…)	сигнатура		
container::container()	сигнатура		
container::~container(){…}			Clear(…)
container		container::In(…), container::Out(…), container::container()	shape
container::In(…)	тело	container, shape	
container::Out(…)=0	тело	container, shape	
container::Clear(…)	тело	container, shape	
container::container()	тело	container	
main(…)	сигнатура		
	тело	container	

Видно, что отличие от процедурной программы, существует зависимость данных, размещаемых в классах от методов их обработки. Для ослабления этой зависимости реализации методов вынесены в другие единицы компиляции. Вместе с тем, использование наследования позволяет сделать обобщение независимым от специализации. Инструментальная поддержка конструкторов и деструкторов позволяет избавиться от явного вызова соответствующих методов и функций, что сокращает размер программы.

2.2.5. Модульная структура объектно-ориентированной программы

В связи с изменением зависимостей между артефактами, модульная структура ОО программы отличается от модульной структуры процедурной программы. В данном случае независимым становится заголовочный файл ***shape_atd.h***, который подключается практически ко всем остальным модулям основной программы. Необходимость подобного подключения обуславливается тем, что класс shape содержит интерфейс, обеспечивающий доступ к необходимым методам обработки программных объектов, размещаемых в контейнере. Вынесение реализаций методов в отдельные файлы способствует раздельной компиляции.

Зависимости между модулями демонстрационного примера представлены в табл. 6.

Таблица 6

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
shape_atd.h			shape
rectangle_atd.h	shape_atd.h		shape, rectangle
triangle_atd.h	shape_atd.h		shape, triangle
container_atd.h	shape_atd.h		shape, container
shape_In.cpp	shape_atd.h rectangle_atd.h triangle_atd.h		shape, rectangle, triangle
rectangle_In.cpp	rectangle_atd.h	shape_atd.h	shape, rectangle
rectangle_Out.cpp	rectangle_atd.h	shape_atd.h	shape, rectangle
triangle_In.cpp	triangle_atd.h	shape_atd.h	shape, triangle
triangle_Out.cpp	triangle_atd.h	shape_atd.h	shape, triangle
container_Constr.cpp	container_atd.h	shape_atd.h	shape, container
container_In.cpp	container_atd.h	shape_atd.h	shape, container
container_Out.cpp	container_atd.h	shape_atd.h	shape, container
main.cpp	container_atd.h	shape_atd.h	shape, container, main(…)

Упрощение модульной структуры данной программы осуществляется за счет использование подключений, которые устанавливают избыточные связям между разработанными артефактами. В целом это ведет дополнительным расходам во время компиляции программы и излишней перекомпиляции модулей при модификациях программы. Однако в ряде случаев такие расходы могут быть оправданы, так как упрощение модульной структуры облегчает понимание программы. В целом же следует отметить, что процесс разбиения программы на отдельные единицы компиляции в большей степени является субъективным и во многом зависит от того, какие критерии качества являются предпочтительными в той или иной ситуации.

3. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №2

Добавить в процедурную и объектно-ориентированную программы, разработанные при выполнении лабораторной работы №1, дополнительные программные объекты, расширяющие номенклатуру обрабатываемых данных (абстрактных типов или классов). Добавление осуществлять в соответствии с вариантами заданий, выданными на первую работу. Необходимые сведения представлены в табл. 7.

Таблица 7

Вариант	Добавляемая альтернатива и ее признак
1	Треугольник (три точки, задающие целочисленные координаты вершин)
2	Тетраэдр (длина стороны – целое)
3	Нижняя треугольная матрица (одномерный массив с формулой пересчета)
4	Корабли (водоизмещение – целое; вид судна – перечислимый тип = лайнер, буксир, танкер…)
5	Документальный фильм (год выпуска – целое)
6	Функциональные языки (типизация – перечислимый тип = строгая, динамическая; поддержка «ленивых» вычислений – булевский тип)
7	Шифрование заменой символов на числа (пары: текущий символ, целое число – подстановка при шифровании кода символа в виде короткого целого; зашифрованный текст – целочисленный массив)
8	Загадки (ответ – строка символов)
9	Полярные координаты (угол [радиан] – действительное; расстояние до точки – целое)
10	Звери (хищники, травоядные, насекомоядные… – перечислимый тип)
11	Цветы (домашние, садовые, дикие… – перечислимый тип)
12	Легковой автомобиль (максимальная скорость – короткое целое)

Результаты работы

1. Разработанная программа, реализующая, дополнительные программные объекты.
2. Письменный отчет по лабораторной работе, представленный в виде электронного документа в текстовом формате. Отчет должен содержать сведения об изменениях, вносимых в исходные тексты программ в связи с условием выполняемого задания:
   - Общий объем исходных текстов, реализующих заданные функции (байт).
   - Количество и состав абстрактных типов данных (АТД).
   - Количество переменных, соответствующих различным АТД.
   - Количество и состав процедур (функций, используемых для решения поставленной задачи).
   - Модифицированный граф зависимостей между модулями, процедурами (функциями) и АТД, отражающий новые связи и изменения.

- Объем исполняемого модуля.

3.1. Пример выполнения лабораторной работы №2. Процедурная реализация.

Пусть требуется добавить в программу новый тип фигур – ромб. Ромб будет характеризоваться двумя параметрами: шириной и высотой его диагоналей.

Для добавления ромба в первую очередь необходимо создать описывающую его структуру и поместить ее в отдельном заголовочном файле:

```
//-----------------------------------------------------
// ромб
struct rhombus {
  int x, y; // ширина и высота диагоналей
};
```

Функции ввода-вывода размещаются в отдельных файлах исходного кода rhombus_In.cpp и rhombus_Out.cpp.

```
//-----------------------------------------------------
// Ввод параметров ромба из файла
void In(rhombus &r, ifstream &ifst)
{
  ifst >> r.x >> r.y;
}

//-----------------------------------------------------
// Вывод параметров ромба в форматируемый поток
void Out(rhombus &r, ofstream &ofst)
{
  ofst << "It is Rhombus: x = " << r.x << ", y = "
       << r.y << endl;
}
```

После создания, новую основу специализации необходимо подключить в уже существующее обобщение. Для этого необходимо внести изменения в структуру shape: добавить новый ключ RHOMBUS, идентифицирующий ромб, и прописать новую переменную b в объединение фигур.

```
//-----------------------------------------------------
// структура, обобщающая все имеющиеся фигуры
struct shape {
  // значения ключей для каждой из фигур
  enum key {RECTANGLE, TRIANGLE, RHOMBUS};
  key k; // ключ
  // используемые альтернативы
```

```
  union { // используем простейшую реализацию
    rectangle r;
    triangle t;
    rhombus b;
  };
};
```

Все обобщенные функции, обрабатывающие тип shape также нуждаются в модификации. Необходимо добавить обработку нового ключа в каждый имеющийся условный оператор:

```
//-----------------------------------------------
// заголовок функции для чтения данных ромба
void In(rhombus  &b, ifstream &ist);

//-----------------------------------------------
// Ввод параметров обобщенной фигуры из файла
shape* In(ifstream &ifst)
{
  shape *sp;
  int k;
  ifst >> k;
  switch(k) {
  case 1:
    sp = new shape;
    sp->k = shape::key::RECTANGLE;
    In(sp->r, ifst);
    return sp;
  case 2:
    sp = new shape;
    sp->k = shape::key::TRIANGLE;
    In(sp->t, ifst);
    return sp;
  case 3:
    sp = new shape;
    sp->k = shape::key::RHOMBUS;
    In(sp->b, ifst);
    return sp;
  default:
    return 0;
  }
}

//-----------------------------------------------
// заголовок функции для вывода ромба
```

```
void Out(rhombus &b, ofstream &ofst);

//-----------------------------------------------------
// Вывод параметров текущей фигуры в поток
void Out(shape &s, ofstream &ofst)
{
  switch(s.k) {
  case shape::key::RECTANGLE:
    Out(s.r, ofst);
    break;
  case shape::key::TRIANGLE:
    Out(s.t, ofst);
    break;
  case shape::key::RHOMBUS:
    Out(s.b, ofst);
    break;
  default:
    ofst << "Incorrect figure!" << endl;
  }
}
```

Таким образом, в проект было добавлено три файла с новым кодом и внесены изменения в три уже существующих. Итог представлен в таблице 8.

Таблица 8

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
rhombus_atd.h			rhombus
rhombus_In.cpp	rhombus_atd.h		In(rhombus&…)
rhombus_Out.cpp	rhombus_atd.h		Out(rhombus&…)
shape_atd.h	rectangle_atd.h triangle_atd.h rhombus_atd.h		Shape
shape_In.cpp	shape_atd.h In(rectangle&…) In(triangle&…) In(rhombus&…)	rectangle_atd.h triangle_atd.h rhombus_atd.h	In(shape&…)
shape_Out.cpp	shape_atd.h Out(rectangle&…) Out(triangle&…) Out(rhombus&…)	rectangle_atd.h triangle_atd.h rhombus_atd.h	Out(shape&…)

3.2. Пример выполнения лабораторной работы №2. Объектно-ориентированная реализация.

Расширение объектно-ориентированной программы осуществляется при помощи добавления нового класса rhombus, наследующего от базового класса shape:

```
//-----------------------------------------------------
// ромб, аналогичен прямоугольнику
class rhombus: public shape
{
  int x, y; // ширина, высота
public:
  // переопределяем интерфейс класса
  void InData(ifstream &ifst);
  void Out(ofstream &ofst);
  rhombus() {}
};
```

В отдельных файлах прописываются методы ввода-вывода:

```
//-----------------------------------------------------
// Ввод параметров ромба
void rhombus::InData(ifstream &ifst) {
  ifst >> x >> y;
}

//-----------------------------------------------------
// Вывод параметров ромба
void rhombus::Out(ofstream &ofst) {
  ofst << "It is Rhombus: x = " << x << ", y = "
       << y << endl;
}
```

Для окончательного включения нового типа фигур в программу необходимо добавить код по созданию объекта в функцию чтения из файла. Это единственная модификация существующего кода из лабораторной работы №1.

```
//-----------------------------------------------------
// Ввод параметров обобщенной фигуры из стандартного
// потока ввода
shape* shape::In(ifstream &ifst) {
  shape *sp;
  int k;
  ifst >> k;
  switch(k) {
```

```
  case 1:
    sp = new rectangle;
    break;
  case 2:
    sp = new triangle;
    break;
  case 3:
    sp = new rhombus;
    break;
  default:
    return 0;
  }
  sp->InData(ifst);
  return sp;
}
```

Финальный список добавленных и изменившихся единиц компиляции представлен в таблице 9.

Таблица 9

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
rhombus_atd.h	shape_atd.h		shape, rhombus
rhombus_In.cpp	rhombus_atd.h		shape, rhombus
rhombus_Out.cpp	rhombus_atd.h		shape, rhombus
shape_In.cpp	shape_atd.h rectangle_atd.h triangle_atd.h rhombus_atd.h		shape, rectangle, triangle, rhombus

4. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №3

В программу, разработанную при выполнении **лабораторной работы №1**, добавить дополнительные процедуры для обработки данных (абстрактных типов или классов). Добавление осуществлять в соответствии с вариантами заданий, выданными на первую работу. Необходимые сведения представлены в таблице 10.

Таблица 10

Вариант	Добавляемая процедура и возвращаемый ею результат
1	Вычисление периметра для каждой из фигур (действительное число)
2	Вычисление объема для каждого из тел (действительное число)

3	Сумма всех элементов матрицы (целое число)
4	Идеальное время прохождения пути (действительное число)
5	Количество гласных букв в названии фильма (целое число)
6	Количество лет, прошедших с года создания языка (целое число)
7	Количество символов в исходном тексте (целое число)
8	Количество знаков препинания в содержательной строке
9	Приведение каждого значения к действительному числу, эквивалентному записанному. Например, для комплексного числа осуществляется по формуле: sqrt(d^2+i^2)).
10	Количество символов в названии животного (целое число)
11	Количество согласных букв в названии растения (целое число
12	Отношение веса груза к единице мощности (действительное число). Вес пассажира считать равным 75 кг.

Примечание. *Для отображения результатов вычисления использовать прямой вывод полученного числа без изменения существующей процедуры вывода.*

Результаты работы.

1. Разработанная программа, реализующая, дополнительные программные объекты.
2. Письменный отчет по лабораторной работе, представленный в виде электронного документа в текстовом формате. Отчет должен содержать сведения об изменениях, вносимых в исходные тексты программ в связи с условием выполняемого задания:
   - Общий объем исходных текстов, реализующих заданные функции (байт).
   - Количество и состав абстрактных типов данных (АТД).
   - Количество переменных, соответствующих различным АТД.
   - Количество и состав процедур (функций, используемых для решения поставленной задачи).
   - Модифицированный граф зависимостей между процедурами (функциями и АТД), отражающий новые связи и изменения.
   - Объем исполняемого модуля.

4.1. Пример выполнения лабораторной работы №3. Процедурная реализация.

Рассмотрим процесс добавления функции вычисления периметра. В случае процедурной программы сначала прописываются обработчики специализаций:

```
//-----------------------------------------------
// Вычисление периметра прямоугольника
int Perimeter(rectangle &r)
{
  return r.x + r.y;
}

//-----------------------------------------------
// Вычисление периметра треугольника
int Perimeter(triangle &t)
{
  return t.a + t.b + t.c;
}
```

Затем добавляется обработчик обобщения:

```
//-----------------------------------------------
// Сигнатуры требуемых функций
int Perimeter(rectangle &r);
int Perimeter(triangle  &t);

//-----------------------------------------------
// Вычисление периметра фигуры
int Perimeter(shape &s)
{
  switch(s.k) {
  case shape::key::RECTANGLE:
    return Perimeter(s.r);
  case shape::key::TRIANGLE:
    return Perimeter(s.t);
  default:
    return -1;
  }
}
```

Вызов функции вычисления периметра и вывод его значения осуществляется внутри специализированной функции контейнера:

```
//-----------------------------------------------
void Out(shape &s, ofstream &ofst);
int Perimeter(shape &s);

//-----------------------------------------------
// Вывод содержимого контейнера в указанный поток
```

```
void Out(container &c, ofstream &ofst) {
  ofst << "Container contains " << c.len
       << " elements." << endl;
  for(int i = 0; i < c.len; i++) {
    ofst << i << ": ";
    Out(*(c.cont[i]), ofst);
    ofst << "perimeter = "
         << Perimeter(*(c.cont[i])) << endl;
  }
}
```

В результате, добавление новой процедуры повлекло за собой только создание новых модулей без изменения старых. Эти модули описаны в таблице 11.

Таблица 11

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
rectangle_Perimeter.cpp	rectangle_atd.h		Perimeter(rectangle&…)
triangle_Perimeter.cpp	triangle_atd.h		Perimeter(triangle&…)
shape_Perimeter.cpp	shape_atd.h	rectangle_atd.h triangle_atd.h	Perimeter(shape&…)
container_Perimeter.cpp	container_atd.h	shape_atd.h rectangle_atd.h triangle_atd.h	Perimeter(container&…)

4.2. Пример выполнения лабораторной работы №3. Объектно-ориентированная реализация.

Добавление новой функции, в случае объектно-ориентированной программы, влечет за собой изменение базового класса и последующие модификации классов-потомков:

```
// Класс, обобщающает все имеющиеся фигуры.
class shape {
public:
  static  shape* In(ifstream &ifst);
  virtual void InData(ifstream &ifst) = 0;
  virtual void Out(ofstream &ofst) = 0;
  virtual int Perimeter() = 0; // вычисление периметра
};
```

```
//-----------------------------------------------------
// прямоугольник
class rectangle: public shape
{
  int x, y; // ширина, высота
public:
  void InData(ifstream &ifst);
  void Out(ofstream &ofst);
  int Perimeter();  // вычисление периметра
  rectangle() {}
};

//-----------------------------------------------------
// треугольник
class triangle: public shape
{
  int a, b, c; // стороны
public:
  void InData(ifstream &ifst
  void Out(ofstream &ofst);
  int Perimeter();  // вычисление периметра
  triangle() {}
};
```

Тела методов прописываются в отдельных модулях:

```
//-----------------------------------------------------
// Вычисление периметра прямоугольника
int rectangle::Perimeter() {
  return x + y;
}

//-----------------------------------------------------
// Вычисление периметра треугольника
int triangle::Perimeter() {
  return a + b + c;
}
```

Непосредственный вывод данных по фигурам и их периметру осуществляется в методе Perimeter контейнера:

```
class container
{
  enum {max_len = 100}; // максимальная длина
  int len; // текущая длина
  shape *cont[max_len];
public:
```

```
  void In(ifstream &ifst);
  void Out(ofstream &ofst);
// вывод фигур и периметра
  void Perimeter(ofstream &ofst);
  void Clear();
  container();
  ~container() {Clear();}
};

//---------------------------------------------------
// Вывод содержимого контейнера
void container::Perimeter(ofstream &ofst) {
  ofst << "Container contents " << len
       << " elements." << endl;
  for(int i = 0; i < len; i++) {
    ofst << i << ": ";
    cont[i]->Out(ofst);
    ofst << "perimeter = "
         << cont[i]->Perimeter() << endl;
  }
}
```

Таким образом, с алгоритмической точки зрения разница между двумя вариантами реализации минимальна. Отличие заключается в представлении и способах организации программных объектов.

Изменения в модульной структуре приведены в таблице12.

Таблица 12

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
shape_atd.h			shape
rectangle_atd.h	shape_atd.h		shape, rectangle
triangle_atd.h	shape_atd.h		shape, triangle
rectangle_Perimeter.cpp	rectangle_atd.h		shape, rectangle
triangle_Perimeter.cpp	triangle_atd.h		shape, triangle
container_atd.h	shape_atd.h		shape, container
rectangle_Perimeter.cpp	container_atd.h		shape, container

5. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №4

В абстрактные типы данных (классы) программы, полученной в ходе выполнения **лабораторной работы №1**, добавить поля для хранения допол-

нительных данных. Местоположение полей определяется из условия задачи. они могут быть добавлены как в обобщение, так и в специализации. **Изменить функции ввода-вывода** разработанных программных объектов с учетом проведенных добавлений. Необходимые сведения представлены в таблице 13.

Таблица 13

Вариант	Добавляемые поля
1	Плотность материала, из которого вырезаются эти геометрические фигуры (действительное число)
2	Температура плавления (горения) материала (в град Цельсия), из которого «сделана» объемная фигура (целое число)
3	Способ вывода матрицы на печать (перечислимый тип = построчно, по столбцам, в виде одномерного массива)
4	Масса груза, перевозимого в данный момент (действительное число)
5	Страна, в которой выпущен фильм (строка символов)
6	Количество упоминаний о данном языке в сети Интернет (целое число)
7	Информация о владельце текста (строка символов)
8	Субъективная оценка изречения по десятибалльной шкале (целое число)
9	Единица измерения (строка символов)
10	Возраст (целое число)
11	Где произрастает (перечислимый тип = тундра, пустыня, степь, …
12	Расход топлива в литрах на 100 км (действительное)

Результаты работы.

1. Разработанная программа, реализующая, дополнительные программные объекты.
2. Письменный отчет по лабораторной работе, представленный в виде электронного документа в текстовом формате. Отчет должен содержать сведения об изменениях, вносимых в исходные тексты программ в связи с условием выполняемого задания:
   - Общий объем исходных текстов, реализующих заданные функции (байт).
   - Количество и состав абстрактных типов данных (АТД).
   - Количество переменных, соответствующих различным АТД.
   - Количество и состав процедур (функций, используемых для решения поставленной задачи).

- Модифицированный граф зависимостей между процедурами (функциями и АТД), отражающий новые связи и изменения.
- Объем исполняемого модуля.

5.1. Пример выполнения лабораторной работы №4. Процедурная реализация.

К имеющимся фигурам необходимо добавить новое поле – угол поворота данной фигуры относительно центра координат. Поскольку данное поле является общим для всех фигур, оно будет размещено в обобщении.

```
// структура, обобщающая все имеющиеся фигуры
struct shape {
  // значения ключей для каждой из фигур
  enum key {RECTANGLE, TRIANGLE};
  key k; // ключ
  int angle;
  // используемые альтернативы
  union { // используем простейшую реализацию
    rectangle r;
    triangle t;
  };
};
```

Теперь необходимо изменить процедуры ввода-вывода обобшения shape таким образом, чтобы они дополнительно могли считывать и выводить значение угла:

```
// Ввод параметров обобщенной фигуры из файла
shape* In(ifstream &ifst)
{
  shape *sp;
  int k;
  ifst >> k;
  switch(k) {
  case 1:
    sp = new shape;
    sp->k = shape::key::RECTANGLE;
    In(sp->r, ifst);
    break;
  case 2:
    sp = new shape;
    sp->k = shape::key::TRIANGLE;
    In(sp->t, ifst);
    break;
```

```
  default:
    return 0;
  }
  ifst >> sp->angle;
  return sp;
}

//-----------------------------------------------------
// Вывод параметров текущей фигуры в поток
void Out(shape &s, ofstream &ofst)
{
  switch(s.k) {
  case shape::key::RECTANGLE:
    Out(s.r, ofst);
    break;
  case shape::key::TRIANGLE:
    Out(s.t, ofst);
    break;
  default:
    ofst << "Incorrect figure!" << endl;
  }
  ofst << "angle = " << s.angle << endl;
}
```

Список изменившихся файлов представлен в таблице 14.

Таблица 14

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
shape_atd.h	rectangle_atd.h triangle_atd.h		shape
shape_In.cpp	shape_atd.h	rectangle_atd.h triangle_atd.h	In(shape&…)
shape_Out.cpp	shape_atd.h	rectangle_atd.h triangle_atd.h	Out(shape&…)

5.2. Пример выполнения лабораторной работы №4. Объектно-ориентированная реализация.

Поле angle в объектно-ориентированной реализации также будет размещено в обобщении. В данном случае это базовый класс shape:

```
// Класс, обобщающает все имеющиеся фигуры.
```

```
class shape {
  int angle;
public:
  static  shape* In(ifstream &ifst);
  virtual void InData(ifstream &ifst);
  virtual void Out(ofstream &ofst);
protected:
  shape() {};
};
```

Следует отметить, что поле angle было размещено в области видимости private. Для того, чтобы обеспечить доступа к этому полю из классов-потомков достаточно прописать алгоритм ввода-вывода внутри методов InData и OutData.

Поскольку методы InData и OutData больше не будут являться абстрактными, пользователи при желании смогут создавать объекты класса shape. Избежать этого можно разместив конструктор shape в защищенной области protected.

```
//-----------------------------------------------------
// Ввод параметров фигуры
void shape::InData(ifstream &ifst) {
  ifst >> angle;
}

//-----------------------------------------------------
// Вывод параметров фигуры
void shape::Out(ofstream &ofst) {
  ofst << "angle = " << angle;
}
```

Теперь для работы с полем angle в классах-потомках достаточно просто вызвать соответствующие методы класса shape:

```
//-----------------------------------------------------
// Ввод параметров прямоугольника
void rectangle::InData(ifstream &ifst) {
  ifst >> x >> y;
  shape::InData(ifst);
}

//-----------------------------------------------------
// Вывод параметров прямоугольника
void rectangle::Out(ofstream &ofst) {
  ofst << "It is Rectangle: x = " << x << ", y = "
       << y << ", ";
  shape::Out(ofst);
```

```
  ofst << endl;
}

//-----------------------------------------------------
// Ввод параметров треугольника
void triangle::InData(ifstream &ifst) {
  ifst >> a >> b >> c;
  shape::InData(ifst);
}

//-----------------------------------------------------
// Вывод параметров треугольника
void triangle::Out(ofstream &ofst) {
  ofst << "It is Triangle: a = "
    << a << ", b = " << b
    << ", c = " << c << ", ";
  shape::Out(ofst);
  ofst << endl;
}
```

Список изменившихся файлов представлен в таблице 15.

Таблица 15

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
shape_atd.h			shape
shape_In.cpp	shape_atd.h		shape
shape_Out.cpp	shape_atd.h		shape
rectangle_In.cpp	rectangle_atd.h		shape, rectangle
rectangle_Out.cpp	rectangle_atd.h		shape, rectangle
triangle_In.cpp	triangle_atd.h		shape, triangle
triangle_Out.cpp	triangle_atd.h		shape, triangle

6. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №5

В программу, разработанную в **лабораторной работе №3**, добавить дополнительную процедуру, обеспечивающую решение требуемой задачи. Дополнительная процедура (метод) должна выполнять следующие функции:

- **Сортировка элементов контейнера по заданному ключу.**

Ключ – Значение, вычисленное процедурой, введенной в лабораторной работе №3.

Примечание: *Сравнение ключей двух программных объектов должно быть оформлено в виде вспомогательной процедуры, проверяющей, какой из ключей меньше (больше…). Процедура должна использовать в качестве параметров два обобщения, определяя внутри истинную структуру сравниваемых артефактов с использованием механизмов, присущих применяемым парадигмам программирования.*

Результаты работы.

1. Разработанная программа, реализующая, дополнительные операции (описанные выше в документе).
2. Письменный отчет по лабораторной работе, представленный в виде электронного документа в текстовом формате. Отчет должен содержать сведения об изменениях, вносимых в исходные тексты программ в связи с условием выполняемого задания:
   - Общий объем исходных текстов, реализующих заданные функции (байт).
   - Количество и состав абстрактных типов данных (АТД).
   - Количество переменных, соответствующих различным АТД.
   - Количество и состав процедур (функций, используемых для решения поставленной задачи).
   - Модифицированный граф зависимостей между процедурами (функциями и АТД), отражающий новые связи и изменения.
   - Объем исполняемого модуля.

6.1. Пример выполнения лабораторной работы №5. Процедурная реализация.

Вспомогательная процедура по сравнению двух объектов расположена в отдельном файле и выполняет сравнение значений, возвращаемых функцией Perimeter:

```
// Сигнатуры требуемых функций
int Perimeter(shape &s);

//-----------------------------------------------------
// Сравнение ключей двух программных объектов
bool Compare(shape *first, shape *second) {
  return Perimeter(*first) < Perimeter(*second);
}
```

Сортировка осуществляется внутри функции Sort при помощи алгоритма пузырька:

```
// Сигнатуры требуемых функций
int Compare(shape *first, shape *second);

//-----------------------------------------------------
// Сортировка содержимого контейнера
void Sort(container &c) {
  for(int i = 0; i < c.len - 1; i++) {
    for(int j = i + 1; j < c.len; j++) {
      if(Compare(c.cont[i], c.cont[j])) {
        shape *tmp = c.cont[i];
        c.cont[i] = c.cont[j];
        c.cont[j] = tmp;
      }
    }
  }
}
```

В результате в программу добавилось всего два новых модуля. Они описаны в таблице 16.

Таблица 16

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
container_Compare.cpp	container_atd.h	shape_atd.h rectangle_atd.h triangle_atd.h	Compare(…)
container_Sort.cpp	container_atd.h	shape_atd.h rectangle_atd.h triangle_atd.h	Sort(…)

6.2. Пример выполнения лабораторной работы №5. Объектно-ориентированная реализация.

Объектно-ориентированная программа в целом аналогична своей процедурной версии. Функция Compare размещена в классе shape по аналогии со стандартным размещением перегрузки оператора сравнения. При желании ее можно прописать в самом контейнере.

```
// Класс, обобщающает все имеющиеся фигуры.
class shape {
public:
  static  shape* In(ifstream &ifst);
  virtual void InData(ifstream &ifst) = 0;
```

```
  virtual void Out(ofstream &ofst) = 0;
  // сравнение двух объектов
  bool Compare(shape &other);
};

//-----------------------------------------------------
// сравнение двух объектов
bool shape::Compare(shape &other) {
  return Perimeter() < other.Perimeter();
}
```

Метод Sort осуществляет непосредственно сортировку контейнера:

```
class container {
  enum {max_len = 100};
  int len;
  shape *cont[max_len];
public:
  void In(ifstream &ifst);
  void Out(ofstream &ofst);
  void Clear();
  void Sort();   // сортировка контейнера
  container();
  ~container() {Clear();}
};

//-----------------------------------------------------
// Сортировка содержимого контейнера
void container::Sort() {
  for(int i = 0; i < len - 1; i++) {
    for(int j = i + 1; j < len; j++) {
      if(cont[i]->Compare(*cont[j])) {
        shape *tmp = cont[i];
        cont[i] = cont[j];
        cont[j] = tmp;
      }
    }
  }
}
```

Список изменений проекта представлен в таблице 17.

Таблица 17

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое

shape_atd.h			shape
shape_Compare.cpp	shape_atd.h		shape
container_atd.h	shape_atd.h		shape, container
container_Sort.cpp	container_atd.h		shape, container

7. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №6

В программу, разработанную в **лабораторной работе №1**, добавить дополнительную процедуру, реализующую выборочный вывод из контейнера в файл сведений только о первом (по порядку описания в задании на лабораторную работу №1) из двух видов объектов. Информация об объектах другого типа, расположенных в контейнере выводиться этой процедурой не должна.

Результаты работы.

1. Разработанная программа, реализующая, дополнительные операции (описанные выше в документе).
2. Письменный отчет по лабораторной работе, представленный в виде электронного документа в текстовом формате. Отчет должен содержать сведения об изменениях, вносимых в исходные тексты программ в связи с условием выполняемого задания:
   - Общий объем исходных текстов, реализующих заданные функции (байт).
   - Количество и состав абстрактных типов данных (АТД).
   - Количество переменных, соответствующих различным АТД.
   - Количество и состав процедур (функций, используемых для решения поставленной задачи).
   - Модифицированный граф зависимостей между процедурами (функциями и АТД), отражающий новые связи и изменения.
   - Объем исполняемого модуля.

7.1. Пример выполнения лабораторной работы №6. Процедурная реализация.

Реализация процедурной версии очень проста: поскольку вся информация о типах хранится в открытом виде, то нам достаточно просто проанализировать ключ и вывести только нужные объекты. Данный анализ осуществляется функцией OutRect:

```
// Сигнатуры требуемых функций
void Out(shape &s, ofstream &ofst);
```

```
//-----------------------------------------------------
// Вывод только прямоугольников
void OutRect(container &c, ofstream &ofst) {
  ofst << "Only rectangles." << endl;
  for(int i = 0; i < c.len; i++) {
    ofst << i << ": ";
    if(c.cont[i]->k == shape::RECTANGLE)
      Out(*(c.cont[i]), ofst);
    else
      ofst << endl;
  }
}
```

Относительно лабораторной работы №1 был добавлен всего один файл – container_OutRect.cpp.

7.2. Пример выполнения лабораторной работы №6. Объектно-ориентированная реализация.

Чистая объектно-ориентированная реализация запрещает непосредственный анализ типа во время исполнения программы. В связи с этим решение получается более сложным, с использованием виртуальных функций.

В базовом классе shape объявляется виртуальная функция OutRect:

```
// Класс, обобщающает все имеющиеся фигуры.
class shape {
public:
  static  shape* In(ifstream &ifst);
  virtual void InData(ifstream &ifst) = 0;
  virtual void Out(ofstream &ofst) = 0;
  // вывод только прямоугольников
  virtual void OutRect(ofstream &ofst);
};

//-----------------------------------------------------
// Вывод данных только для прямоугольника
void shape::OutRect(ofstream &ofst) {
  ofst << endl;  // пустая строка
}
```

Результатом ее работы является вывод пустой строки. Данная функция по наследству без изменений переходит ко всем классам-потомкам shape, кроме класса rectangle. Он переопределяет функцию OutRect таким образом, что вывод пустой строки изменяется на вызов функции Out.

```
// прямоугольник
class rectangle: public shape
{
  int x, y; // ширина, высота
public:
  void InData(ifstream &ifst);
  void Out(ofstream &ofst);
  // вывод только прямоугольников
  void OutRect(ofstream &ofst);
  rectangle() {}
  };

//-----------------------------------------------------
// Вывод данных только для прямоугольника
void rectangle::OutRect(ofstream &ofst) {
  Out(ofst);
}
```

Теперь достаточно пройти по всем элементам контейнера и вызвать у каждого метод OutRect:

```
// Вывод содержимого контейнера
void container::OutRect(ofstream &ofst) {
  ofst << "Only rectangles." << endl;
  for(int i = 0; i < len; i++) {
    ofst << i << ": ";
    cont[i]->OutRect(ofst);
  }
}
```

Результат работы программы аналогичен процедурной версии. Список изменений проекта представлен в таблице 18.

Таблица 18

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
shape_atd.h			shape
shape_OutRect.cpp	shape_atd.h		shape
rectangle_atd.h	shape_atd.h		shape, rectangle
rectangle_OutRect.cpp	rectangle_atd.h		shape, rectangle
container_atd.h	shape_atd.h		shape, container
container_OutRect.cpp	container_atd.h		shape, container

8. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №7

В программу, разработанную **в лабораторной работе №1**, добавить дополнительную процедуру, реализующую мультиметод с двумя аргументами. Мультиметод должен выводить в файл информацию о своей работе и типе объектов используемых в подставленных комбинациях, а также печатать содержимое принятых значений.

Комбинации объектов порождаются путем перебора всех пар элементов, расположенных в контейнере.

Результаты работы.

1. Разработанная программа, реализующая, дополнительные операции (описанные выше в документе).
2. Письменный отчет по лабораторной работе, представленный в виде электронного документа в текстовом формате. Отчет должен содержать сведения об изменениях, вносимых в исходные тексты программ в связи с условием выполняемого задания:
   - Общий объем исходных текстов, реализующих заданные функции (байт).
   - Количество и состав абстрактных типов данных (АТД).
   - Количество переменных, соответствующих различным АТД.
   - Количество и состав процедур (функций, используемых для решения поставленной задачи).
   - Модифицированный граф зависимостей между процедурами (функциями и АТД), отражающий новые связи и изменения.
   - Объем исполняемого модуля.

8.1. Пример выполнения лабораторной работы №7. Процедурная реализация.

Процедурная реализация мультиметода представляет собой набор условных операторов, осуществляющих анализ типа объекта и вывод информации о полученной комбинации:

```
// Сигнатуры требуемых
void Out(shape &s, ofstream &ofst);

//-------------------------------------------------
// Мультиметод
void MultiMethod(container &c, ofstream &ofst) {
  ofst << "Multimethod." << endl;
```

```
  for(int i = 0; i < c.len - 1; i++) {
    for(int j = i + 1; j < c.len; j++) {
      switch(c.cont[i]->k) {
      case shape::RECTANGLE:
        switch(c.cont[j]->k) {
        case shape::RECTANGLE:
          ofst << "Rectangle and Rectangle." << endl;
          break;
        case shape::TRIANGLE:
          ofst << "Rectangle and Triangle." << endl;
          break;
        default:
          ofst << "Unknown type" << endl;
        }
        break;
      case shape::TRIANGLE:
        switch(c.cont[j]->k) {
        case shape::RECTANGLE:
          ofst << "Triangle and Rectangle." << endl;
          break;
        case shape::TRIANGLE:
          ofst << "Triangle and Triangle." << endl;
          break;
        default:
          ofst << "Unknown type" << endl;
        }
        break;
      default:
        ofst << "Unknown type" << endl;
      }
      Out(*(c.cont[i]), ofst);
      Out(*(c.cont[j]), ofst);
    }
  }
}
```

Эта функция расположена в модуле container_MM.cpp.

8.2. Пример выполнения лабораторной работы №7. Объектно-ориентированная реализация.

Объектно-ориентированная реализация опирается на использование виртуальных функций для анализа типа аргумента. Сначала первая функция оп-

ределяет тип первого аргумента, а затем происходит перенаправление анализа в виртуальную функцию второго аргумента.

```
// Класс, обобщает все имеющиеся фигуры.
class shape {
public:
  static  shape* In(ifstream &ifst);
  virtual void InData(ifstream &ifst) = 0;
  virtual void Out(ofstream &ofst) = 0;
  // мультиметод
  virtual void MultiMethod(shape *other,
                           ofstream &ofst) = 0;
  virtual void MMRect(ofstream &ofst) = 0;
  virtual void MMTrian(ofstream &ofst) = 0;
};

//-----------------------------------------------------
// прямоугольник
class rectangle: public shape
{
  int x, y; // ширина, высота
public:
  // переопределяем интерфейс класса
  void InData(ifstream &ifst);
  void Out(ofstream &ofst);
  // мультиметод
  void MultiMethod(shape *other, ofstream &ofst);
  void MMRect(ofstream &ofst);
  void MMTrian(ofstream &ofst);
  rectangle() {} // создание без инициализации.
};

//-----------------------------------------------------
// треугольник
class triangle: public shape
{
  int a, b, c; // стороны
public:
  void InData(ifstream &ifst);
  void Out(ofstream &ofst);
  // мультиметод
  void MultiMethod(shape *other, ofstream &ofst);
  void MMRect(ofstream &ofst);
  void MMTrian(ofstream &ofst);
```

```
  triangle() {}
};
```

Реализация всех элементов мультиметода разбросана по нескольким модулям. Особое внимание необходимо обратить на функции-диспетчеры, осуществляющие переход к анализу второго аргумента:

```
// мультиметод
void rectangle::MultiMethod(shape *other,
                            ofstream &ofst) {
  other->MMRect(ofst);
}

//-------------------------------------------------
// мультиметод
void triangle::MultiMethod(shape *other,
                           ofstream &ofst) {
  other->MMTrian(ofst);
}
```

Тип первого аргумента определяется автоматически, после чего вызывается один из обработчиков специализации второго аргумента. При этом информация о типе первого аргумента неявно сохраняется в названии вызываемого метода.

```
// Вывод двух прямоугольников
void rectangle::MMRect(ofstream &ofst) {
  ofst << "Rectangle and Rectangle" << endl;
}

//-------------------------------------------------
// Вывод треугольника и прямоугольника
void rectangle::MMTrian(ofstream &ofst) {
  ofst << "Triangle and Rectangle" << endl;
}

//-------------------------------------------------
// Вывод прямоугольника и треугольника
void triangle::MMRect(ofstream &ofst) {
  ofst << "Rectangle and Triangle" << endl;
}

//-------------------------------------------------
// Вывод двух треугольников
void triangle::MMTrian(ofstream &ofst) {
  ofst << "Triangle and Triangle" << endl;
}
```

Перебор всех пар осуществляется функцией MultiMethod контейнера.

```
// Вызов мультиметода для элементов контейнера
void container::MultiMethod(ofstream &ofst) {
  ofst << "Multimethod." << endl;
  for(int i = 0; i < len - 1; i++) {
    for(int j = i + 1; j < len; j++) {
      cont[i]->MultiMethod(cont[j], ofst);
      cont[i]->Out(ofst);
      cont[j]->Out(ofst);
    }
  }
}
```

Список добавленных и измененных файлов можно увидеть в таблице 19.

Таблица 19

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
shape_atd.h			shape
rectangle_atd.h	shape_atd.h		shape
triangle_atd.h	shape_atd.h		shape, rectangle
container_atd.h	shape_atd.h		shape, container
rectangle_MM.cpp	rectangle_atd.h		shape, rectangle
rectangle_MMRect.cpp	rectangle_atd.h		shape, rectangle
rectangle_MMTrian.cpp	rectangle_atd.h		shape, rectangle
triangle_MM.cpp	triangle_atd.h		shape, triangle
triangle_MMRect.cpp	triangle_atd.h		shape, triangle
triangle_MMTrian.cpp	triangle_atd.h		shape, triangle
container_MM.cpp	container_atd.h		shape, container

9. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №8

Добавить в программы, разработанные при выполнении лабораторной работы №7, дополнительные программные объекты, расширяющие номенклатуру обрабатываемых данных (абстрактных типов или классов). Добавление осуществлять в соответствии с вариантами заданий, выданными на первую работу и используемыми в работе №2. Необходимые сведения представлены в табл. 20.

Таблица 20

Вариант	Добавляемая альтернатива и ее признак
1	Треугольник (три точки, задающие целочисленные координаты вершин)

2	Тетраэдр (длина стороны – целое)
3	Нижняя треугольная матрица (одномерный массив с формулой пересчета)
4	Корабли (водоизмещение – целое; вид судна – перечислимый тип = лайнер, буксир, танкер…)
5	Документальный фильм (год выпуска – целое)
6	Функциональные языки (типизация – перечислимый тип = строгая, динамическая; поддержка «ленивых» вычислений – булевский тип)
7	Шифрование заменой символов на числа (пары: текущий символ, целое число – подстановка при шифровании кода символа в виде короткого целого; зашифрованный текст – целочисленный массив)
8	Загадки (ответ – строка символов)
9	Полярные координаты (угол [радиан] – действительное; расстояние до точки – целое)
10	Звери (хищники, травоядные, насекомоядные… – перечислимый тип)
11	Цветы (домашние, садовые, дикие… – перечислимый тип)
12	Легковой автомобиль (максимальная скорость – короткое целое)

Результаты работы.

1. Разработанная программа, реализующая, дополнительные операции (описанные выше в документе).
2. Письменный отчет по лабораторной работе, представленный в виде электронного документа в текстовом формате. Отчет должен содержать сведения об изменениях, вносимых в исходные тексты программ в связи с условием выполняемого задания:
   - Общий объем исходных текстов, реализующих заданные функции (байт).
   - Количество и состав абстрактных типов данных (АТД).
   - Количество переменных, соответствующих различным АТД.
   - Количество и состав процедур (функций, используемых для решения поставленной задачи).
   - Модифицированный граф зависимостей между процедурами (функциями и АТД), отражающий новые связи и изменения.
   - Объем исполняемого модуля.

9.1. Пример выполнения лабораторной работы №8. Процедурная реализация.

Добавление нового типа данных происходит аналогично лабораторной работе №2. Появление ромба потребует добавления новых обработчиков в уже существующий мультиметод. В случае процедурной реализации необходимо дописать дополнительные проверки в каждый условный оператор, анализирующий тип объекта:

```
case shape::RECTANGLE:
  switch(c.cont[j]->k) {
  // ...
  case shape::RHOMBUS:
    ofst << "Rectangle and Rhombus." << endl;
    break;
  // ...

case shape::TRIANGLE:
  switch(c.cont[j]->k) {
  // ...
  case shape::RHOMBUS:
    ofst << "Triangle and Rhombus." << endl;
    break;
  // ...

case shape::RHOMBUS:
  switch(c.cont[j]->k) {
  case shape::RECTANGLE:
    ofst << "Rhombus and Rectangle." << endl;
    break;
  case shape::TRIANGLE:
    ofst << "Rhombus and Triangle." << endl;
    break;
  case shape::RHOMBUS:
    ofst << "Rhombus and Rhombus." << endl;
    break;
  default:
    ofst << "Unknown type" << endl;
  }
```

Все изменения производятся в файле container_MM.cpp.

9.2. Пример выполнения лабораторной работы №8. Объектно-ориентированная реализация.

В объектной реализации в первую очередь необходимо добавить новый класс rhomb, наследующий от базового класса shape.

```
// ромб, аналогичен прямоугольнику
class rhombus: public shape
{
  int x, y; // ширина, высота
public:
  void InData(ifstream &ifst);
  void Out(ofstream &ofst);
  // мультиметод
  void MultiMethod(shape *other, ofstream &ofst);
  void MMRect(ofstream &ofst);
  void MMTrian(ofstream &ofst);
  void MMRhomb(ofstream &ofst);
  rhombus() {} // создание без инициализации.
};
```

В базовый класс необходимо добавить новую функцию - обработчик специализации ромба, а также прописать его у всех классов-потомков.

```
// Класс, обобщающает все имеющиеся фигуры.
class shape {
public:
  static  shape* In(ifstream &ifst);
  virtual void InData(ifstream &ifst) = 0;
  virtual void Out(ofstream &ofst) = 0;
  // мультиметод
  virtual void MultiMethod(shape *other,
                           ofstream &ofst) = 0;
  virtual void MMRect(ofstream &ofst) = 0;
  virtual void MMTrian(ofstream &ofst) = 0;
  virtual void MMRhomb(ofstream &ofst) = 0;
};

//-----------------------------------------------------
// прямоугольник
class rectangle: public shape
{
  int x, y; // ширина, высота
public:
  void InData(ifstream &ifst);
```

```
  void Out(ofstream &ofst);
  // мультиметод
  void MultiMethod(shape *other, ofstream &ofst);
  void MMRect(ofstream &ofst);
  void MMTrian(ofstream &ofst);
  void MMRhomb(ofstream &ofst);
  rectangle() {} // создание без инициализации.
};

//-----------------------------------------------------
// треугольник
class triangle: public shape
{
  int a, b, c; // стороны
public:
  void InData(ifstream &ifst);
  void Out(ofstream &ofst);
  // мультиметод
  void MultiMethod(shape *other, ofstream &ofst);
  void MMRect(ofstream &ofst);
  void MMTrian(ofstream &ofst);
  void MMRhomb(ofstream &ofst);
  triangle() {} // создание без инициализации.
};
```

Затем нужно прописать тела функций, обрабатывающих новые комбинации:

```
// Мультиметод
void rhombus::MultiMethod(shape *other, ofstream &ofst)
{
  other->MMRhomb(ofst);
}

//-----------------------------------------------------
// Вывод ромба и прямоугольника
void rectangle::MMRhomb(ofstream &ofst) {
  ofst << "Rhombus and Rectangle" << endl;
}

//-----------------------------------------------------
// Вывод ромба и треугольника
void triangle::MMRhomb(ofstream &ofst) {
  ofst << "Rhombus and Triangle" << endl;
}
```

```
//-----------------------------------------------------
// Вывод треугольника и ромба
void rhombus::MMTrian(ofstream &ofst) {
  ofst << "Triangle and Rhombus" << endl;
}

//-----------------------------------------------------
// Вывод прямоугольника и ромба
void rhombus::MMRect(ofstream &ofst) {
  ofst << "Rectangle and Rhombus" << endl;
}

//-----------------------------------------------------
// Вывод двух ромбов
void rhombus::MMRhomb(ofstream &ofst) {
  ofst << "Rhombus and Rhombus" << endl;
}
```

Список изменений проекта представлен в таблице 21.

Таблица 21

Название модуля	Непосредственно подключенные модули и артефакты	Косвенно подключенные модули и артефакты	Содержимое
shape_atd.h			shape
triangle_atd.h	shape_atd.h		shape, triangle
rectangle_atd.h	shape_atd.h		shape, rectangle
rhombus_atd.h	shape_atd.h		shape, rhombus
rectangle_MMRhomb.cpp	shape_atd.h		shape, container
triangle_MMRhomb.cpp	container_atd.h		shape, container
rhombus_MM.cpp	rhombus_atd.h		shape, rhombus
rhombus_MMRect.cpp	rhombus_atd.h		shape, rhombus
rhombus_MMRhomb.cpp	rhombus_atd.h		shape, rhombus
rhombus_MMTrian.cpp	rhombus_atd.h		shape, rhombus

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что понимается под парадигмой программирования?
2. Назовите основные особенности процедурной парадигмы программирования.
3. Назовите основные особенности объектно-ориентированной парадигмы программирования.

4. Какое влияние на модульную структуру программы, представленной в примере, произвела бы замена передачи параметров по ссылке и указателем на передачу по значению?

5. Назовите основные способы модульной организации программы. Назовите достоинства и недостатки основных модульных структур.

6. Сравните сходство и различие в организации модульных структур для двух известных Вам языков программирования.

7. Каковы достоинства и недостатки использования условной компиляции?

8. В разработанных примерах программ в подключаемых заголовочных файлах используется условная компиляция. Каким образом изменится модульная структура программы, если условная компиляция будет исключена?

9. Сравните размеры исходных текстов программ, написанных с применением процедурного и объектного стилей. Какая из программ оказалась больше по объему? Почему?

10. В каких случаях многообразие методов реализации обобщений при процедурном подходе является достоинством, а в каких – недостатком?

11. Назовите достоинства и недостатки каждого из трех приведенных метода построения обобщений, используемых при процедурном подходе.

12. Какой из приведенных процедурных способов построения обобщений наиболее близок к методу, используемому при объектно-ориентированном подходе?

13. Почему в программировании используются разнообразные методы создания контейнеров? Каковы их достоинства и недостатки?

14. Опишите зависимости между артефактами демонстрационной процедурной программы, полученные при замене косвенного связывания элементов в контейнере на прямое включение. Каковы достоинства и недостатки прямого включения элементов в контейнер?

15. Повлияет ли на стратегию разработки программы использование прямого включения элементов в контейнер вместо косвенного связывания?

16. Можно ли с помощью процедурного подхода написать объектно-ориентированную программу? Если да, то каким образом это можно осуществить?

17. Расскажите об одном из известных вам методов преобразования объектно-ориентированной программы в процедурную. Используются ли на практике подобные преобразования?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Бадд, Т. Объектно-ориентированное программирование. / Т. Бадд, СПб: Питер Паблишинг, 1997. 464 с.

2. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. / Г. Буч. М.: Бином, СПб.: Невский диалект, 1998. 560 с.

3. Вирт, Н. Алгоритмы и структуры данных. / Н. Вирт. М.: Мир, 1989. 360 с.

4. Легалов, А.И. Разнорукое программирование. А. И. Легалов. Материал расположен в сети Интернет по адресу: http://www.softcraft.ru/paradigm/dhp/index.shtml

5. Непейвода, Н. Н. Основания программирования. / Н. Н. Непейвода, И. Н. Скопин. Москва-Ижевск, РХД, 2003. 880 с.

6. Цикритзис, Д. Модели данных. / Д. Цикритзис, Ф. Лоховски. М.: Финансы и статистика, 1985. 344 с.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЕ ТЕКСТЫ ПРОЦЕДУРНОЙ ПРОГРАММЫ (обязательное)

Файл ***rectangle_atd.h***, содержащий описание прямоугольника.

```
//-------------------------------------------------
#ifndef __rectangle_atd__
#define __rectangle_atd__
namespace simple_shapes {
  // прямоугольник
  struct rectangle {
    int x, y; // ширина, высота
  };
} // end simple_shapes namespace
#endif
```

Файл ***rectangle_In.cpp***, содержащий функцию ввода параметров для уже существующего прямоугольника.

```
//-------------------------------------------------
#include <fstream>
#include "rectangle_atd.h"
using namespace std;
namespace simple_shapes {
  // Ввод параметров прямоугольника из файла
  void In(rectangle &r, ifstream &ifst) {
    ifst >> r.x >> r.y;
  }
} // end simple_shapes namespace
```

Файл ***rectangle_Out.cpp***, содержащий функцию вывода параметров прямоугольника.

```
//---------------------------------------------------
#include <fstream>
#include "rectangle_atd.h"
using namespace std;
namespace simple_shapes {
  // Вывод параметров прямоугольника в поток
  void Out(rectangle &r, ofstream &ofst) {
    ofst << "It is Rectangle: x = " << r.x
         << ", y = " << r.y << endl;
  }
} // end simple_shapes namespace
```

Файл ***triangle_atd.h***, содержащий описание треугольника.

```
//---------------------------------------------------
#ifndef __triangle_atd__
#define __triangle_atd__
namespace simple_shapes {
  // треугольник
  struct triangle {
    int a, b, c; // стороны
  };
} // end simple_shapes namespace
#endif
```

Файл ***triangle_In.cpp***, содержащий функцию ввода параметров для уже существующего треугольника.

```
//---------------------------------------------------
#include <fstream>
#include "triangle_atd.h"
using namespace std;
namespace simple_shapes {
  // Ввод параметров треугольника из потока
  void In(triangle &t, ifstream &ifst)
  {
    ifst >> t.a >> t.b >> t.c;
  }
} // end simple_shapes namespace
```

Файл ***triangle_Out.cpp***, содержащий функцию вывода параметров треугольника.

```
//---------------------------------------------------
```

```
#include <fstream>
#include "triangle_atd.h"
using namespace std;
namespace simple_shapes {
  // Вывод параметров треугольника в поток
  void Out(triangle &t, ofstream &ofst)
  {
    ofst << "It is Triangle: a = "
      << t.a << ", b = " << t.b
      << ", c = " << t.c << endl;
  }
} // end simple_shapes namespace
```

Файл ***shape_atd.h***, содержащий описание обобщенной фигуры.

```
//-------------------------------------------------
#ifndef __shape_atd__
#define __shape_atd__
// Подключение необходимых типов данных
#include "rectangle_atd.h"
#include "triangle_atd.h"
namespace simple_shapes {
  // структура, обобщающая все имеющиеся фигуры
  struct shape {
    // значения ключей для каждой из фигур
    enum key {RECTANGLE, TRIANGLE};
    key k; // ключ
    // используемые альтернативы
    union { // используем включение
      rectangle r;
      triangle t;
    };
  };
} // end simple_shapes namespace
#endif
```

Файл ***shape_In.cpp***, содержащий функцию создания и ввода обобщенной фигуры.

```
//-------------------------------------------------
#include <fstream>
#include "shape_atd.h"
using namespace std;
namespace simple_shapes {
  // Сигнатуры требуемых внешних функций
```

```
  void In(rectangle &r, ifstream &ist);
  void In(triangle  &t, ifstream &ist);

  // Ввод параметров обобщенной фигуры из файла
  shape* In(ifstream &ifst)
  {
    shape *sp;
    int k;
    ifst >> k;
    switch(k) {
    case 1:
      sp = new shape;
      sp->k = shape::key::RECTANGLE;
      In(sp->r, ifst);
      return sp;
    case 2:
      sp = new shape;
      sp->k = shape::key::TRIANGLE;
      In(sp->t, ifst);
      return sp;
    default:
      return 0;
    }
  }
} // end simple_shapes namespace
```

Файл ***shape_Out.cpp***, содержащий функцию вывода обобщенной фигуры.

```
//---------------------------------------------------
#include <fstream>
#include "shape_atd.h"
using namespace std;
namespace simple_shapes {
  // Сигнатуры требуемых внешних функций.
  void Out(rectangle &r, ofstream &ofst);
  void Out(triangle  &t, ofstream &ofst);

  // Вывод параметров текущей фигуры в поток
  void Out(shape &s, ofstream &ofst) {
    switch(s.k) {
    case shape::key::RECTANGLE:
      Out(s.r, ofst);
      break;
```

```
    case shape::key::TRIANGLE:
      Out(s.t, ofst);
      break;
    default:
      ofst << "Incorrect figure!" << endl;
    }
  }
} // end simple_shapes namespace
```

Файл ***container_atd.h*** - содержит тип данных, представляющий простейший контейнер.

```
//-------------------------------------------------
#ifndef __container_atd__
#define __container_atd__
namespace simple_shapes {
// Ссылка на описание геометрической фигуры.
// Знание структуры самой фигуры для представленной
// реализации контейнера не требуется
struct shape;
// Простейший контейнер на основе одномерного массива
  struct container
  {
    enum {max_len = 100}; // максимальная длина
    int len; // текущая длина
    shape *cont[max_len];
  };
} // end simple_shapes namespace
#endif
```

Файл ***container_Constr.cpp***, содержащий функции начальной инициализации и очистки (утилизации) контейнера.

```
//-------------------------------------------------
#include "container_atd.h"
#include "shape_atd.h"
namespace simple_shapes {
  // Инициализация контейнера
  void Init(container &c) { c.len = 0; }
  // Очистка контейнера от элементов
  // (освобождение памяти)
  void Clear(container &c) {
    for(int i = 0; i < c.len; i++){
      delete c.cont[i];
    }
```

```
    c.len = 0;
  }
} // end simple_shapes namespace
```

Файл ***container_In.cpp***, содержащий функцию ввода фигур из заданного потока.

```
//-----------------------------------------------------
#include <fstream>
#include "container_atd.h"
using namespace std;
namespace simple_shapes {
  // Сигнатуры требуемых внешних функций
  shape *In(ifstream &ifdt);

  // Ввод содержимого контейнера из указанного потока
  void In(container &c, ifstream &ifst) {
    while(!ifst.eof()) {
      if((c.cont[c.len] = In(ifst)) != 0){ c.len++; }
    }
  }
} // end simple_shapes namespace
```

Файл ***container_Out.cpp***, содержащий функцию ввода фигур из заданного потока.

```
//-----------------------------------------------------
#include <fstream>
#include "container_atd.h"
using namespace std;
namespace simple_shapes {
  // Сигнатуры требуемых внешних функций
  void Out(shape &s, ofstream &ofst);
  // Вывод содержимого контейнера в указанный поток
  void Out(container &c, ofstream &ofst) {
    ofst << "Container contains " << c.len
         << " elements." << endl;
    for(int i = 0; i < c.len; i++) {
      ofst << i << ": ";
      Out(*(c.cont[i]), ofst);
    }
  }
} // end simple_shapes namespace
```

Файл ***main.cpp***, содержащий главную функцию программы.

```
//---------------------------------------------------
#include <iostream>
#include <fstream>
#include "container_atd.h"
using namespace std;
namespace simple_shapes {
// Сигнатуры требуемых внешних функций
    void Init(container &c) ;
    void Clear(container &c);
    void In(container &c, ifstream &ifst) ;
    void Out(container &c, ofstream &ofst) ;
}
using namespace simple_shapes;
int main(int argc, char* argv[]) {
    if(argc !=3) {
        cout << "incorrect command line! "
             "Waited: command infile outfile" << endl;
        exit(1);
    }
    ifstream ifst(argv[1]);
    ofstream ofst(argv[2]);
    cout << "Start"<< endl;
    container c;
    Init(c);
    In(c, ifst);
    ofst << "Filled container. " << endl;
    Out(c, ofst);
    Clear(c);
    ofst << "Empty container. " << endl;
    Out(c, ofst);
    cout << "Stop"<< endl;
    return 0;
}
```

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЕ ТЕКСТЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ ПРОГРАММЫ (обязательное)

Файл ***shape_atd.h***, содержащий обобщенной фигуры.

```
//---------------------------------------------------
#ifndef __shape_atd__
#define __shape_atd__
```

```
#include <fstream>
using namespace std;
namespace simple_shapes {
  // Класс, обобщающает все имеющиеся фигуры.
  // Является абстрактным, обеспечивая, тем самым,
  // проверку интерфейса
  class shape {
  public:
    // иденитфикация, порождение и ввод фигуры из пото-
ка
    static  shape* In(ifstream &ifst);
    virtual void InData(ifstream &ifst) = 0;  // ввод
    virtual void Out(ofstream &ofst) = 0;     // вывод
  };
} // end simple_shapes namespace
#endif
```

Файл ***rectangle_atd.h***, содержащий описание прямоугольника.

```
//-------------------------------------------------
#ifndef __rectangle_atd__
#define __rectangle_atd__
// Требеутся описание класса shape
#include "shape_atd.h"
namespace simple_shapes {
  // прямоугольник
  class rectangle: public shape {
    int x, y; // ширина, высота
  public:
    // переопределяем интерфейс класса
    void InData(ifstream &ifst);  // ввод
    void Out(ofstream &ofst);    // вывод
    rectangle() {} // создание без инициализации.
  };
} // end simple_shapes namespace
#endif
```

Файл ***rectangle_In.cpp***, содержащий метод ввода параметров прямоугольника из заданного входного потока.

```
//-------------------------------------------------
#include "rectangle_atd.h"
using namespace std;
namespace simple_shapes {
  // Ввод параметров прямоугольника
```

```
  void rectangle::InData(ifstream &ifst) {
    ifst >> x >> y;
  }
} // end simple_shapes namespace
```

Файл ***rectangle_Out.cpp***, содержащий метод вывода параметров прямоугольника в заданный выходной поток.

```
//-------------------------------------------------
#include "rectangle_atd.h"
using namespace std;
namespace simple_shapes {
  // Вывод параметров прямоугольника
  void rectangle::Out(ofstream &ofst) {
    ofst << "It is Rectangle: x = " << x
         << ", y = " << y << endl;
  }
} // end simple_shapes namespace
```

Файл ***triangle_atd.h***, содержащий описание треугольника.

```
//-------------------------------------------------
#ifndef __triangle_atd__
#define __triangle_atd__
// Требуется описание класса shape
#include "shape_atd.h"
namespace simple_shapes {
  // треугольник
  class triangle: public shape {
    int a, b, c; // стороны
  public:
    // переопределяем интерфейс класса
    void InData(ifstream &ifst);  // ввод
    void Out(ofstream &ofst);     // вывод
    triangle() {} // создание без инициализации.
  };
} // end simple_shapes namespace
#endif
```

Файл ***triangle_In.cpp***, содержащий метод ввода параметров треугольника из заданного входного потока.

```
//-------------------------------------------------
#include "triangle_atd.h"
using namespace std;
namespace simple_shapes {
```

```
  // Ввод параметров треугольника
  void triangle::InData(ifstream &ifst) {
    ifst >> a >> b >> c;
  }
} // end simple_shapes namespace
```

Файл ***triangle_Out.cpp***, содержащий метод вывода параметров треугольника в заданный выходной поток.

```
//-------------------------------------------------
#include "triangle_atd.h"
using namespace std;
namespace simple_shapes {
  // Вывод параметров треугольника
  void triangle::Out(ofstream &ofst) {
    ofst << "It is Triangle: a = "
      << a << ", b = " << b
      << ", c = " << c << endl;
  }
} // end simple_shapes namespace
```

Файл ***shape_In.cpp***, содержащий статический метод ***shape::In***, предназначенный для создания и ввода из файла заданной геометрической фигуры.

```
//-------------------------------------------------
#include "shape_atd.h"
// Необходима подключить информацию обо всех имеющихся
// геометрических фигурах
#include "rectangle_atd.h"
#include "triangle_atd.h"
using namespace std;
namespace simple_shapes {
  // Ввод параметров обобщенной фигуры
  shape* shape::In(ifstream &ifst) {
    shape *sp;
    int k;
    ifst >> k;
    switch(k) {
    case 1:
      sp = new rectangle;
      break;
    case 2:
      sp = new triangle;
      break;
    default:
```

```
      return 0;
    }
    sp->InData(ifst);
    return sp;
  }
} // end simple_shapes namespace
```

Файл ***container_atd.h***, содержащий описание контейнерного класса.

```
//-----------------------------------------------
#ifndef __container_atd__
#define __container_atd__
#include "shape_atd.h"
namespace simple_shapes {
  // Простейший контейнер на основе одномерного массива
  class container  {
    enum {max_len = 100}; // максимальная длина
    int len; // текущая длина
    shape *cont[max_len];
  public:
    void In(ifstream &ifst);     // ввод
    void Out(ofstream &ofst);    // вывод
    void Clear();  // очистка контейнера от фигур
    container();    // инициализация контейнера
    ~container() {Clear();} // утилизация контейнера
  };
} // end simple_shapes namespace
#endif
```

Файл ***container_Constr.cpp***, содержащий конструктор и метод, обеспечивающий очистку контейнера от данных.

```
//-----------------------------------------------
#include "container_atd.h"
namespace simple_shapes {
  // Инициализация контейнера
  container::container(): len(0) { }
  // Очистка контейнера от элементов
  void container::Clear()  {
    for(int i = 0; i < len; i++) {
      delete cont[i];
    }
    len = 0;
  }
} // end simple_shapes namespace
```

Файл ***container_In.cpp***, содержащий метод ввода геометрических фигур в контейнер из заданного входного потока.

```
//-------------------------------------------------
#include "container_atd.h"
using namespace std;
namespace simple_shapes {
  // Ввод содержимого контейнера
  void container::In(ifstream &ifst) {
    while(!ifst.eof()) {
      if((cont[len] = shape::In(ifst)) != 0) {
        len++;
      }
    }
  }
} // end simple_shapes namespace
```

Файл ***container_Out.cpp***, содержащий метод вывода параметров геометрических фигур, размещенных в контейнере.

```
//-------------------------------------------------
#include "container_atd.h"
using namespace std;
namespace simple_shapes {
  // Вывод содержимого контейнера
  void container::Out(ofstream &ofst) {
    ofst << "Container contents " << len
         << " elements." << endl;
    for(int i = 0; i < len; i++) {
      ofst << i << ": ";
      cont[i]->Out(ofst);
    }
  }
} // end simple_shapes namespace
```

ОГЛАВЛЕНИЕ