МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. Р.Е. АЛЕКСЕЕВА»

Институт радиоэлектроники и информационных технологий

Кафедра «Прикладная математика»

Дисциплина: «Технологии программирования»

Курсовая работа на тему: «Иерархии классов»

Выполнил:

Студент группы 19-ПМ-1

Шамшетдинов Р.Р

Проверил:

Демкин В.М.

Нижний Новгород

2020

Содержание:

Введение....3

Глава 1. Иерархии абстрактных типов данных....6

1.1 Агрегирование («целое – часть») 6

1.2 Одиночное наследование («потомок – родитель»)....7

1.3 Множественное виртуальное наследование («потомок – родитель»)....8

Глава 2. Простейшие формы классов....11

2.1 Проект структуры 11

2.2 Проект переменной структуры....12

Глава 3. Класс....16

3.1 Проект класса 17

3.2 Проект класса с захватом ресурса в виде свободной памяти....19

3.3 Проект класса с захватом ресурса в виде текстового файла 23

Глава 4. Открытое одиночное наследование классов 26

4.1. Иерархия для двух классов 27

4.2 Проект иерархии для двух классов 27

Заключение 30

Приложение 31

Библиографический список 52

Введение

Центральным свойством C++ является поддержка объектно-ориентированного программирования (ООП). Именно ООП послужило стимулом к созданию C++. По определению авторитета в области объектно-ориентированных методов разработки программ Гради Буча «объектно-ориентированное программирование (ООП) – это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса, а классы образуют иерархию на принципах наследуемости. Объект - это базовое понятие ООП. Любой объект принадлежит одному или нескольким классам, которые в свою очередь определяют, описывают поведение объекта. Каждый объект характеризуется свойствами, методами и событиями. Свойства — описание объекта. Метод - это действие объекта, изменяющее его состояние или реализующее другое его поведение. Объект, класс, метод, свойства, события -— это базовые понятия ООП. Действие в ООП инициируется посредством передачи сообщений объекту, ответственному за действия. Сообщение содержит запрос на осуществление действия и сопровождается дополнительной информацией, необходимой для его выполнения.

Объектно-ориентированное программирование вобрало в себя лучшие идеи структурного программирования и скомбинировало их с некоторыми новыми концепциями. В результате возник новый и лучший способ организации программы. Вообще программу можно организовывать двумя способами, положив во главу угла либо коды (описывающие, что происходит), либо данные (над которыми выполняются действия). Если программа использует только методы структурного программирования, то обычно в основе в ее организации лежат коды. При таком подходе можно сказать, что «коды воздействуют на данные».

Объектно-ориентированные программы работают как раз наоборот. В основе их организации лежат данные, и их принцип заключается в том, что «данные управляют доступом к коду». Использую объектно-ориентированный язык, пользователь определяет данные и процедуры, которым разрешается обрабатывать эти данные. В результате тип данного однозначно определяет, какого рода операции допустимо выполнять над этим данным.

С целью поддержки принципов объектно-ориентированного программирования все объектно-ориентированные языки, включая C++, обеспечивает три характерных принципа: инкапсуляцию, полиморфизм и наследование. Совокупность принципов проектирования, разработки и реализации программ базируется на абстракции данных. Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

В C ++ классы обеспечивают отличный уровень **абстракции данных**. Они обеспечивают достаточные общественные методы внешнему миру, чтобы играть с функциональностью объекта и манипулировать объектными данными, то есть состоянием, не зная, каким образом класс был реализован внутри страны.

Актуальность выбранного метода ООП обуславливается упрощением управления процессом разработки, гибкостью применения и перенастраиваемостью.

В данной курсовой работе был выбран абстрактный тип данных под названием «Катер». Целью курсовой работы является рассмотрение таких методик как агрегирование, одиночное наследование и множественное виртуальное наследование. Целью также является создание структуры,

переменной структуры, класса, класса с захватом ресурса в виде свободной памяти, класса с захватом ресурса в виде текстового файла, а также рассмотрение одиночного наследование классов.

Основными задачами курсовой работы для выполнения поставленной цели являются разработка схем агрегирования, одиночного и множественного виртуального наследования, реализация программного кода для простейших форм класса (структуры и переменной структуры), а также для класса, класса с захватом ресурса в виде свободной памяти, класса с захватом ресурса в виде текстового файла и реализация одиночного наследования класса в соответствии с разработанной иерархии классов.

Глава 1. Иерархии абстрактных типов данных

Абстракция - вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает умственные возможности пользователя. Инкапсуляция позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Иерархия - это упорядочение абстракций, расположение их по уровням.

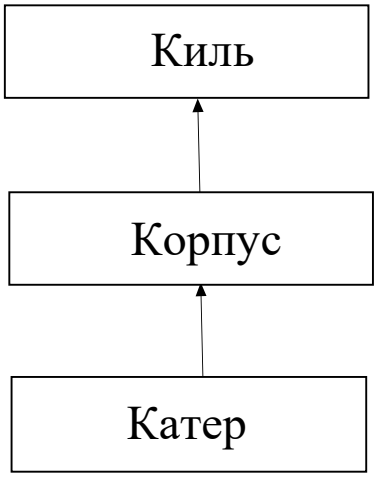
Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия "is-a") и структура объектов (иерархия "part of"). Наследование означает такое отношение между классами (отношение родитель/потомок), когда один класс заимствует структурную или функциональную часть одного или нескольких других классов. Например, для выбранного АТД «Катер» родителем будет более широкая область «Водный транспорт».

1.1 Агрегирование («целое – часть»)

В объектно-ориентированном программировании под агрегированием (также называемом композицией или включением) подразумевают методику создания нового класса из уже существующих классов путём их включения. Об агрегировании также часто говорят, как об «отношении принадлежности» по принципу «у машины есть корпус, колёса и двигатель».

На базе агрегирования реализуется методика делегирования, когда поставленная перед внешним объектом задача перепоручается внутреннему объекту, специализирующемуся на решении задач такого рода.

Для выбранного АТД «Катер» агрегирование выглядит следующим образом:



У катера есть составная часть корпус, а корпус включает в себя киль.

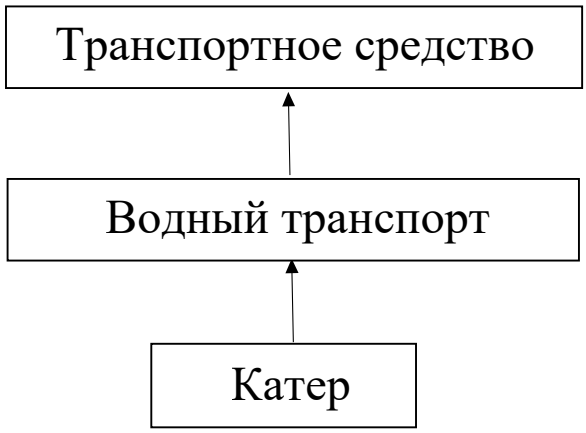
1.2 Одиночное наследование («потомок – родитель»)

Механизм наследования позволяет определять новые классы на основе уже имеющихся. Класс, на основе которого создается новый класс, называют базовым (родительским) классом, а новый – производным (наследником). Любой производный класс сможет в свою очередь стать базовым для других создаваемых классов. Таким образом формируется иерархия классов.

Существует одиночное и множественное наследование. Главным отличием является то, что при одиночном наследовании базовым является один класс, а при множественном наследовании базовыми классами должны быть несколько классов.

Таким образом, одиночное наследование — это отношение между классами, в котором абстрактный тип данных может наследовать переменные и методы уже существующего типа данных, способствуя повторному использованию компонентов.

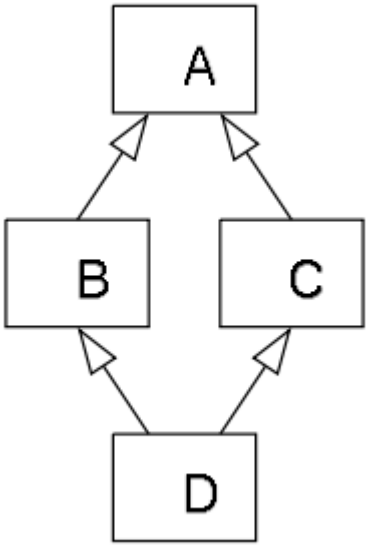
Для выбранного АТД «Катер» одиночное наследование выглядит следующим образом:



Для «Катера» «родителем» будет являться более широкая область «Водный транспорт», а для «Водного транспорта» «родителем» будет более широкая область «Транспортное средство»

1.3 Множественное виртуальное наследование («потомок – родитель»)

В большинстве реальных приложений на C++ используется открытое наследование от одного базового класса. Но иногда одиночного наследования не хватает, потому что с его помощью либо нельзя адекватно смоделировать абстракцию предметной области, либо получающаяся модель чересчур сложна и неинтуитивна. В таких случаях следует предпочесть множественное наследование или его частный случай – виртуальное наследование. Как уже отмечалось выше, множественное наследование — наследование от нескольких базовых классов одновременно. При множественном наследовании базовые классы могут иметь компоненты с одинаковым именем. Рассмотрим более подробно в чем проблема использования множественного наследования. Эта проблема носит название **«проблема ромба»**. Рассмотрим обобщенный пример множественного наследования без привязки к каким-либо абстрактным типам данных.

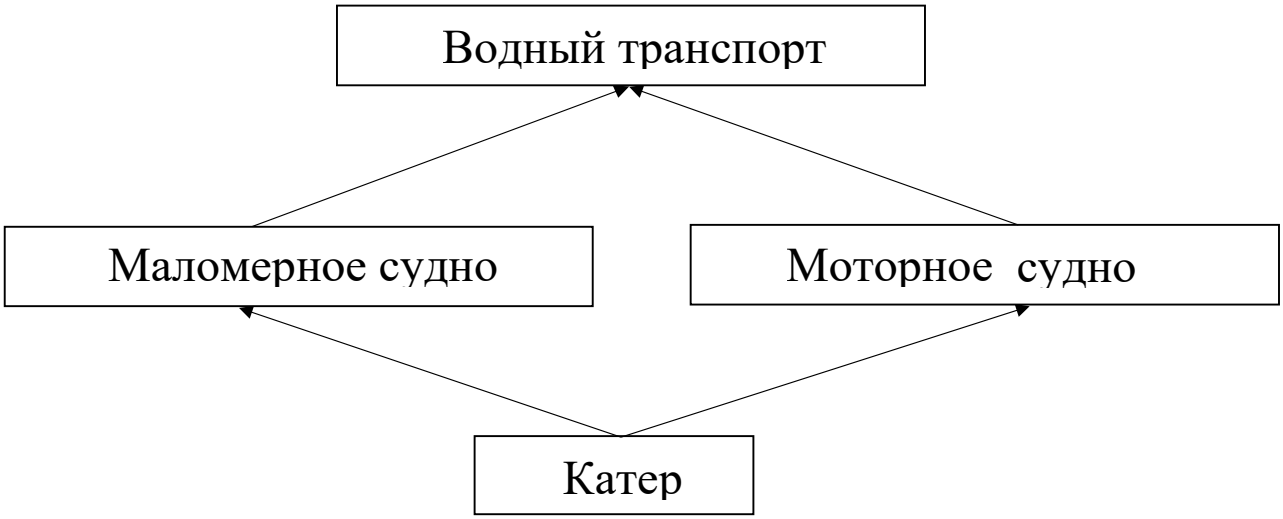


Проблема ромба - классическая проблема в языках, которые поддерживают возможность множественного наследования. Эта проблема возникает, когда классы B и C наследуют A, а класс D наследует B и C.

К примеру, классы A, B и C определяют какой – либо метод класса. Если этот метод будет вызываться классом D, неясно какой метод должен быть вызван — метод класса A, B или C. Проблема ромба также затрагивает конструкторы и деструкторы. Поскольку в C++ при инициализации объекта дочернего класса вызываются конструкторы всех родительских классов, возникает и другая проблема: конструктор базового класса будет вызван дважды.

Разные языки по-разному подходят к решению ромбовидной проблем. Ромбовидная проблема — прежде всего проблема дизайна, и она должна быть предусмотрена на этапе проектирования. Одним из вариантов решения этой проблемы является использование виртуального наследования. Виртуальное наследование предотвращает появление множественных объектов базового класса в иерархии наследования. Таким образом, конструктор базового класса будет вызван только единожды, а обращение к методу базового класса без его переопределения в дочернем классе не будет вызывать ошибку при компиляции. При этом базовый класс, наследуемый множественно, определяется виртуальным с помощью ключевого слова **virtual**.

Для выбранного АТД «Катер» множественное виртуальное наследование выглядит следующим образом:



«Катер» одновременно наследует компоненты областей «Маломерное судно» и «Моторное судно», а они, в свою очередь, наследуют компоненты области «Водный транспорт».

Глава 2. Простейшие формы классов

Базовые процедуры большинства языков программирования возможно дополнить применением абстрактных типов данных, понятий класса и объекта, что уменьшает количество кода, а структуру программы делает более понятной.

В языке C++ структурированный код по сути выполняет такое же назначение, как и основные типы данных языка Си. Традиционно они носят название абстрактных типов данных. В дальнейшем будут подробно рассмотрены такие абстрактные типы данных, как структуры, объединения и классы.

2.1 Проект структуры

Структура - это совокупность переменных, объединенных одним именем, предоставляющая общепринятый способ совместного хранения информации. Объявление структуры приводит к образованию шаблона, используемого для создания объектов структуры. Переменные, образующие структуру, называются членами структуры. Когда объявлена структурная переменная, компилятор автоматически выделяет необходимый участок памяти для размещения всех ее членов.

На данном этапе работы создадим структуру для АТД «Катер», которая содержит информацию о названии, скорости, времени путешествия и варианте движения катера.

```
struct Motorboat
{
    char *name;             // the name of the motorboat
    unsigned int speed;     // the speed of the motorboat(km per
hour)
    unsigned int movement; // the movement of the motorboat (0-
start moving, 1-moving forward, 2 - moving back, 3 - moving right,
4 - moving left, 5 - stop moving)
    unsigned int time;      // motorboat travel time (hour)
};
```

Перед использованием структура должна быть инициализирована. Создадим функцию инициализации структуры. Инициализация структуры

осуществляется вручную при помощи установки начальных значений всех полей структуры.

```
//initialization of the motorboat
void init(Motorboat*
          char *name = "Yamaha 12X",
          unsigned int speed = 60,
          unsigned int movement = 0,
          unsigned int time = 3);
```

Также создадим функции визуализации и полезную функцию, которая определяет направление движения катера. В данных функциях в качестве параметра передается указатель на объект структуры.

```
//visualization of the motorboat
void view(Motorboat*);
//options of motorboat traffic
void moving(Motorboat*, int movement);
```

В полезной функции void moving(Motorboat*, int movement) используется оператор **switch()**, который в качестве параметра принимает значение переменной movement. В зависимости от значения переменной в операторе switch() описаны 6 возможных вариантов.

Программная реализация структуры представлена в Приложении в Главе 2.1.

2.2 Проект переменной структуры

Объединения - это объект, позволяющий нескольким переменным различных типов занимать один участок памяти. Для определения объединений применяется ключевое слово **union.** Когда объявлено объединение, компилятор автоматически создает переменную достаточного размера для хранения наибольшей переменной. Таким образом, размер объединения определяется размером наибольшего элемента, а размер элементов объединения соответствует размерам своих типов. Фактически объединение является структурой, в которой все элементы имеют нулевое смещение от ее начала. Над объединениями

разрешено выполнять те же операции, что и над структурами: присваивать или копировать как единое целое, брать адрес и обращаться к отдельным элементам. Использование объединений помогает создавать машинно-независимый код. Поскольку компилятор отслеживает настоящие размеры переменных, образующих объединение, уменьшается зависимость от компьютера. Не нужно беспокоиться о размере целых или вещественных чисел, символов или чего-либо еще. Инициализация объединений производится также, как и инициализация структур, за исключением того, что инициализировать можно только первый член.

Можно сделать вывод о том, что основное достоинство объединений — возможность разных трактовок одного и того же содержимого участка памяти, т.е. возможность доступа к одному и тому же участку памяти с помощью объектов разных типов.

Создадим структуру для АТД «Катер», которая содержит объединение. При этом в самом объединении хранится информация о дополнительном оборудовании для катера, а именно: имя дополнительного оборудования, значение радиодиапазона и наличие навигационной системы.

```
struct Motorboat
{
    char *name;                // the name of the motorboat
    unsigned int speed;        // the speed of the motorboat(km per
hour)
    unsigned int movement;     // the movement of the motorboat (0-
start moving, 1-moving forward, 2 - moving back, 3 - moving right,
4 - moving left, 5 - stop moving)
    unsigned int time;         // motorboat travel time (hour)
    int additional_equipment; // active component mark
    union {
        char name_equipment[256];  // mark =0
        int radio_range;           // mark =1
        bool navigation_equipment; // mark =2
    };
};
```

Функция инициализации структуры (инициализируем структуру вручную при помощи установки начальных значений всех полей структуры):

```
//initialization of the motorboat
```

```
void init(Motorboat*
          char *name = "Yamaha 12X",
          unsigned int speed = 60,
          unsigned int movement = 0,
          unsigned int time = 3,
          int additional_equipment = 0);
```

В выдержке кода, представленной ниже, значение элемента **additional_equipment** объекта структуры Motorboat является инициализатором объединения. В соответствии со значением инициализатора объединения, содержащегося в объекте структуры, в операторе **switch()**, который в качестве параметра принимает значение элемента additiona_equipment, описаны четыре возможных варианта.

```
     switch (additional_equipment)
{
case 0:
    strcpy(motorboat->name_equipment, "sounder");
    break;
case 1:
    motorboat->radio_range = 400;
    break;
case 2:
    motorboat->navigation_equipment = true;
    break;
default:
    motorboat->additional_equipment = 0;
    strcpy(motorboat->name_equipment, "sounder");
    break;
}
```

Также создадим функции визуализации и полезную функцию, которая определяет направление движения катера.

```
//visualization of the motorboat
void view(Motorboat*);
/* options of motorboat traffic */
void moving(Motorboat *motorboat, int movement);
```

В полезной функции void moving(Motorboat*, int movement) используется оператор **switch()**, который в качестве параметра принимает значение

переменной movement. В зависимости от значения переменной в операторе switch() описаны 6 возможных вариантов.

Программная реализация переменной структуры представлена в Приложении в Главе 2.2.

Глава 3. Класс

Класс – это ключевое понятие в объектно-ориентированном программировании. Класс – это шаблон, который определяет форму объекта. Объекты являются экземплярами класса. Таким образом, класс по существу представляет собой комплект планов, по которым строится объект. Важно понимать, что класс – это логическая абстракция. Лишь после того, как будет создан объект класса, в памяти появится и начнет существовать физическое представление этого класса. Определяя класс, объявляются данные, которые будут в него входить, а также код, который будет работать с этими данными. Данные содержатся в переменных экземплярах, определенных в классе, а код содержится в функциях. Код и данные, составляющие класс, называются членами класса. В ходе выполнения работы был поставлен вопрос, в чем причина использования класса, а не продолжение усовершенствования структуры? Ключевой причиной стало то, что в структуре все члены класса по умолчанию открытые, а в классе — закрытые. В связи с началом использования класса изменяется тип переменной «name» с указателя на char на string, так как в данном случае string становится более удобным в использовании. Для инициализации переменных в классе предусмотрен конструктор по умолчанию. В конструкторе по умолчанию задаем аргументам параметры, которые будут явлениями по умолчанию. Конструктор копирования – это специальный конструктор, который позволяет получить идентичный к заданному объект. То есть, с помощью конструктора копирования можно получить копию уже существующего объекта. Для освобождения захваченного переменными ресурсов вызывается деструктор. Для вывода информации на экран используем дружественную функцию для перегрузки операторов вывода. Дружественной функцией класса называется функция, которая, не являясь его компонентом, имеет доступ к его собственным (private) и защищенным (protected) компонентам. Одним из основных свойств этих специфических функций является то, что с их помощью можно осуществить

перегрузку операторов (в нашем случае перегрузка оператора вывода). Возникает закономерный вопрос: «Зачем вообще нужна перегрузка операторов?». При помощи перегрузки операторов можно добиться того, чтобы ввод-вывод значений объектов через потоки можно было использовать при помощи стандартных в этом случае операторов "<<" и ">>".

3.1 Проект класса

Для АТД «Катер» создадим класс, который содержит информацию о названии, скорости, времени путешествия и варианте движения катера.

```
Class Motorboat
{
    string name;            // the name of the motorboat
    unsigned int speed;    // the speed of the motorboat(km per hour)
    unsigned int movement; // the movement of the motorboat (0-start moving, 1-moving forward, 2 - moving back, 3 - moving right, 4 - moving left, 5 - stop moving)
    unsigned int time;     // motorboat travel time (hour)

public:
// default constructor
Motorboat(string name = "Yamaha 12X",
          unsigned int speed = 60,
          unsigned int movement = 0,
          unsigned int time = 3) : name(name),
                                   speed(speed),
                                   movement(movement),
                                   time(time)
{
}
// copy constructor
Motorboat(const Motorboat& motorboat) : name(motorboat.name),
                                        speed(motorboat.speed),
                                        movement(motorboat.movement),
```

```
                        time(motorboat.time)
{
}
// usefull function
void moving();
// friend operator function
friend ostream& operator<<(ostream&, Motorboat);
};
```

В случае использования класса полезная функция для рассматриваемого АТД «Катер» остается такой же, как и в случае использования структуры – определение варианта движения катера. В полезной функции void moving() используется оператор **switch()**, который в качестве параметра принимает значение переменной movement. В зависимости от значения переменной в операторе switch() описаны 6 возможных вариантов. Ниже представлена выдержка из кода с реализацией полезной функции:

```
void Motorboat::moving()
{
    switch (movement)
    {
    case 0:
        cout << "motorboat start moving" << endl;
        break;
    case 1:
        cout << "motorboat moving forward" << endl;
        break;
    case 2:
        cout << "motorboat moving back" << endl;
        break;
    case 3:
        cout << "motorboat moving right" << endl;
        break;
    case 4:
        cout << "motorboat moving left" << endl;
```

```
            break;
        case 5:
            cout << "motorboat stop moving" << endl;
            break;
        }
    }
```

Также создадим дружественную функцию для перегрузки операторов вывода для класса Motorboat:

```
// friend operator function
friend ostream& operator<<(ostream&, Motorboat);
```

Программная реализация для проекта «Класс» представлена в Приложении в Главе 3.1.

3.2 Проект класса с захватом ресурса в виде свободной памяти

Динамическое выделение памяти — это способ запроса памяти из операционной системы запущенными программами по мере необходимости. Эта память не выделяется из ограниченной памяти стека программы, а выделяется из гораздо большего хранилища, управляемого операционной системой — кучи. Для динамического выделения памяти одной переменной используется оператор **new.** Для того, чтобы получить доступ к выделенной памяти необходимо создать указатель. Когда динамически выделяется память, то операционная система должна зарезервировать часть этой памяти для использования программой. Если ОС может выполнить этот запрос, то возвращается адрес этой памяти обратно в программу. С этого момента и в дальнейшем программа сможет использовать эту память, как только пожелает. Когда пользователь уже выполнил с этой памятью всё, что было необходимо, то её нужно вернуть обратно в операционную систему, для распределения между другими запросами. Это выполняется с помощью оператора **delete.** Оператор delete на самом деле ничего не удаляет.

Он просто возвращает память, которая была выделена ранее, обратно в операционную систему. Затем операционная система может переназначить эту память другому приложению (или этому же снова).

В проекте «Класс с захватом ресурса в виде свободной памяти» добавляется новая переменная **count of name**, отвечающая за количество имен катера, а строковая переменная **name** становится строковым массивом с именами катеров.

В конструкторе по умолчанию выделяется память для строкового массива с помощью оператора new[] на определенное количество строковых переменных, которые за тем инициализируются. В деструкторе с помощью оператора delete[] возвращаем использованную память обратно в операционную систему.

```
class Motorboat
{
    unsigned int movement; /* the movement of the motorboat (0-
start moving, 1-
moving forward, 2 - moving back, 3 - moving right, 4 - moving left
, 5 - stop moving) */
    unsigned int speed;    /* the speed of the motorboat(km per ho
ur) */
    unsigned int time;     /* motorboat travel time (hour) */
    int count_of_name;
    string *name; /* the name of the motorboat */
public:
    /* default constructor */
    Motorboat(unsigned int movement = 1,
              unsigned int speed = 120,
              unsigned int time = 2,
              int count_of_name = 2,
              string name_1 = "Yamaha 12X",
              string name_2 = "Yamaha 12Y") : movement(movement),
                                              speed(speed),
                                              time(time),
                                              count_of_name(count_
of_name)
    {
        name = new string[count_of_name];
        switch (count_of_name)
        {
        case 1:
```

```
        {
            name[0] = name_1;
            break;
        }
        case 2:
        {
            name[0] = name_1;
            name[1] = name_2;
            break;
        }
        }
    }
    /*copy constructor */
    Motorboat(const Motorboat &motorboat) : movement(motorboat.mov
ement),
                                            speed(motorboat.speed)
,
                                            time(motorboat.time),
                                            count_of_name(motorboa
t.count_of_name)

    {
        name = new string[motorboat.count_of_name];
        switch (motorboat.count_of_name)
        {
        case 1:
        {
            name[0] = motorboat.name[0];
            break;
        }
        case 2:
        {
            name[0] = motorboat.name[0];
            name[1] = motorboat.name[1];
            break;
        }
        }
    }
    /* destructor */
    ~Motorboat()
    {
        delete[] name;
    }
    /* options of the motorboat traffic */
    void moving();
    /* visualization of the motorboat */
    friend ostream &operator<<(ostream &, Motorboat);
};
```

Создадим полезную функцию, которая определяет направление движения катера:

```
    /* options of the motorboat traffic */
    void moving();
```

Также создадим дружественную функцию для перегрузки операторов вывода для класса Motorboat. В этой функции добавляется оператор switch(), который в качестве параметра принимает значение переменной count_of_name. В зависимости от значения этой переменной в операторе switch() описаны 2 варианта. Ниже представлен фрагмент кода с реализацией дружественной функции для перегрузки операторов вывода:

```
ostream &operator<<(ostream &stream, Motorboat motorboat)
{
    switch (motorboat.count_of_name)
    {
    case 1:
        stream << "the motorboat name = " << motorboat.name[0] <<
"\n";
        break;
    case 2:
        stream << "the motorboat name = " << motorboat.name[0] <<
" and " << motorboat.name[1] << "\n";
        break;
    }
    stream << "the motorboat speed = " << motorboat.speed << " (km
 per hour)\n";
    stream << "the motorboat travel time = " << motorboat.time <<
" (hour)\n";
    if (motorboat.movement > 5)
    {
        stream << "The motorboat cant be moving\n";
    }
    else
    {
        stream << "The motorboat can be moving\n";
    }
    return stream;
}
```

Однако, на данном этапе работы была допущена серьезная ошибка – отсутствие обработки исключения, если память не выделится. Исключение

обрабатывается с помощью ключевого слова **throw**. Оно используется ввиду неэффективности использования стандартного исключения **std::bad_alloc**. Так как оператор new всегда будут возвращать действительный указатель, даже если технически нет свободной памяти слева или её недостаточно - поэтому std::bad_alloc не является надежным признаком исчерпания памяти. Эта ошибка была замечена проверяющим экспертом, поэтому программный код с исправлением данной ошибки представлена в Приложении в Главе 3.2.2. Программная реализация для проекта «Класс с захватом ресурса в виде свободной памяти» без исправленной ошибки представлена в Приложении в Главе 3.2.1.

3.3 Проект класса с захватом ресурса в виде текстового файла

В C++ файл открывается путем связывания его с потоком. Существует три типа потоков: ввода, вывода и ввода – вывода. В проекте «Класс с захватом ресурса в виде текстового файла» мы используем только поток ввода. Для того, чтобы открыть поток ввода, объявим поток класса **ifstream**. Создав поток необходимо связать его с файлом с помощью функции **open()**. Эта функция входит как член в потоковый класс ifstream. Перед тем, как пытаться обратиться к открываемому файлу, необходимо проверить результат выполнения функции open(). Сделаем это с помощью функции **is_open()**, которая является членом класса ifstream. Для закрытия файла используем функцию-член **close()**. Это как раз выполняется в деструкторе. Если файл был открыт, то в деструкторе он закрывается. В конструкторе по умолчанию задается параметр по умолчанию, отвечающий за имя файла, из которого захватывается ресурс. Параметр по умолчанию — это параметр функции, который имеет определенное (по умолчанию) значение. Если пользователь не передает в функцию значение для параметра, то используется значение по умолчанию. Для того, чтобы считать переменную name типа string используется

функция getline(). Функция getline () извлекает символы из входного потока и добавляет его к строковому объекту, пока не встретится символ-разделитель.

```
Class Motorboat
{
    string name;            // the name of the motorboat
    unsigned int speed;    // the speed of the motorboat(km per
hour)
    unsigned int movement; // the movement of the motorboat (0-
start moving, 1-moving forward, 2 - moving back, 3 - moving right,
4 - moving left, 5 - stop moving)
    unsigned int time;     // motorboat travel time (hour)
    ifstream stream;      // input file stream

public:
// default constructor
Motorboat(const string file_name ="SportMotorboat.txt")
{
    stream.open(file_name);
    if (stream.is_open())
    {
     //data member initialization;
     getline(stream, name);
     stream >> speed;
     stream >> movement;
     stream >> time;
    }
    else
    {
     //data member initialization
     name = "Yamaha 12X";
     speed = 120;
```

```
     movement = 1;
     time = 2;
    }
}
// copy constructor
Motorboat(const Motorboat& motorboat) : name(motorboat.name),
                          speed(motorboat.speed),
                          movement(motorboat.movement),
                          time(motorboat.time)
{
}
// destructor
~Motorboat()
{
     if (stream.is_open())
     stream.close();
}
// usefull function
void moving();
// friend operator function
friend ostream& operator<<(ostream&, Motorboat);
};
```

Создадим полезную функцию, которая определяет направление движения катера, и дружественную функцию для перегрузки операторов вывода:

```
// usefull function
void moving();
// friend operator function
friend ostream& operator<<(ostream&, Motorboat);
```

Программная реализация для проекта «Класс с захватом ресурса в виде текстового файла» представлена в Приложении в Главе 3.3.

Глава 4. Открытое одиночное наследование классов

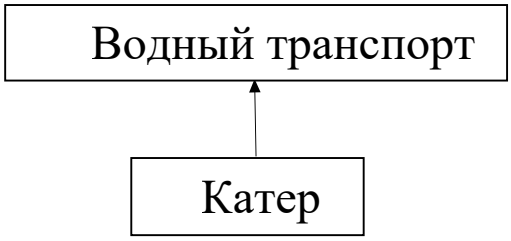
Наследование — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с инкапсуляцией, полиморфизмом и абстракцией), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом. Согласно языку C++, наследуемый класс называется базовым классом. Класс, который наследует от базового, называется производным классом. Таким образом, производный класс является специализированным вариантом базового. Производный класс наследует все члены, определенные в базовом классе, и добавляет к ним свои собственные специфические элементы. В C++ наследование реализуется путем включения одного класса (базового) в объявление другого (производного). Другими словами, в объявлении производного класса указывается, какой класс является для него базовым.

Члены класса часто объявляются закрытыми, чтобы предотвратить неавторизованный доступ или неразумное использование. Наследования от класса не нарушает ограничение доступа к закрытым членам. Таким образом, хотя производный класс включает все члены базового, он не может обратиться к закрытым членам базового класса. Решением этой проблемы является атрибут protected. Если член класса объявлен как protected, он доступен только членам своего класса и производных классов. Таким образом, атрибут protected позволяет члену наследоваться, но оставаться закрытым внутри иерархии классов. Данный метод используется и в моей работе. В иерархии классов различают два вида отношений наследования — одиночное или простое и множественное. Если в иерархии классов какой-либо производный класс связан отношением наследования непосредственно только с одним базовым классом, то

это пример одиночного наследования. Рассмотрим одиночное наследования для выбранного АТД «Катер».

4.1. Иерархия для двух классов

Для выбранного АТД «Катер» одиночное наследование выглядит следующим образом:



Для «Катера» «родителем» будет являться более широкая область «Водный транспорт».

4.2 Проект иерархии для двух классов

Базовый класс Water_transport содержит информацию о грузоподъемности катера и о пассажировместимости, которая рассчитывается как целочисленное деление грузоподъемности на 100 (средний вес 1 человека согласно принятым нормам). При чем элементы класса Water_ transport будут находиться под спецификатором доступа **protected**. Таким образом, используя описатель protected, можно создавать члены класса, которые закрыты в своем классе, но тем не менее наследуются и доступны из производного класса. Для базового класса создадим конструктор по умолчанию с заданными значениями аргументов и конструктор копирования. Для уничтожения объекта класса вызовем виртуальный деструктор. В языке программирования C++ деструктор полиморфного базового класса должен объявляться виртуальным. Только так обеспечивается корректное разрушение объекта производного класса через указатель на соответствующий базовый класс.

```
class Water_transport
{
protected:
    unsigned int tonnage;
    unsigned int passengers_capacity;
    /* default constructor */
    Water_transport(unsigned int tonnage = 530) : tonnage(tonnage)

    {
        passengers_capacity = tonnage / 100;
    }
    /* copy constructor */
    Water_transport(const Water_transport &water_transport) : tonn
age(water_transport.tonnage),
                                                                pass
engers_capacity(water_transport.passengers_capacity)
    {
    }
    /* destructor */
    virtual ~Water_transport()
    {
    }
};
```

Для производного класса создадим конструктор по умолчанию с заданными значениями аргументов, конструктор копирования, дружественную функцию для перегрузки операторов вывода и полезную функцию, которая определяет направление движения катера. В теле конструктора копирования используется ключевое слово this, которое представляет указатель на текущий объект данного класса. Соответственно через this мы можем обращаться внутри производного класса к членам базового класса, объявленных под спецификатором protected. Для уничтожения объекта класса вызовем деструктор.

```
class Motorboat : public Water_transport
{
    string name;           /* the name of the motorboat */
    unsigned int speed;    /* the speed of the motorboat(km per ho
ur) */
    unsigned int movement; /* the movement of the motorboat (0-
start moving, 1-
moving forward, 2 - moving back, 3 - moving right, 4 - moving left
, 5 - stop moving) */
```

```
    unsigned int time;     /* motorboat travel time (hour) */

public:
    /* default constructor */
    Motorboat(unsigned int tonnage = 530,
              string name = "Yamaha 12X",
              unsigned int speed = 120,
              unsigned int movement = 1,
              unsigned int time = 2) : Water_transport(tonnage),
                                       name(name),
                                       speed(speed),
                                       movement(movement),
                                       time(time)
    {
    }
    /* copy constructor */
    Motorboat(const Motorboat &motorboat) : name(motorboat.name),
                                            speed(motorboat.speed)
,
                                            movement(motorboat.mov
ement),
                                            time(motorboat.time)
    {
        this->tonnage=motorboat.tonnage;
        this->passengers_capacity=motorboat.passengers_capacity;
    }
    /* destructor */
    ~Motorboat()
    {
    }
    /* options of the motorboat traffic */
    void moving();
    /* visualization of the motorboat */
    friend ostream &operator<<(ostream &, Motorboat);
};
```

Программная реализация для проекта «Одиночное наследование» представлена в Приложении в Главе 4.

Заключение

В курсовой работе были созданы проекты и реализован программный код для структуры, переменной структуры, класса, класса с захватом ресурса в виде свободной памяти и текстового файла, а также одиночное наследование для двух классов в отношении «родитель - потомок». В ходе работы многие эскизы проектов были неоднократно изменены и усовершенствованы после проверки экспертом.

В заключении хочется отметить, что данная курсовая работа помогла подробно разобраться в принципах ООП. Написание программных кодов для созданных эскизов проектов также улучшила мои знания языка C++.

Приложение

Глава 2.1

```
/* Motorboat */
#include <cstdio>
#include <iostream>
using namespace std;

struct Motorboat
{
    char *name;            /* the name of the motorboat */
    unsigned int speed;    /* the speed of the motorboat(km per ho
ur) */
    unsigned int movement; /* the movement of the motorboat (0-
start moving, 1-
moving forward, 2 - moving back, 3 - moving right, 4 - moving left
, 5 - stop moving) */
    unsigned int time;     /*motorboat travel time (hour) */
};

/* initialization of the motorboat */
void init(Motorboat *motorboat,
          char *name = "Yamaha 12X",
          unsigned int speed = 60,
          unsigned int movement = 0,
          unsigned int time = 3)

{
    motorboat->name = name;
    motorboat->speed = speed;
    motorboat->movement = movement;
    motorboat->time = time;
}

/* visualization of the motorboat */
void view(Motorboat *motorboat)
{
    printf("the motorboat name = %s \n", motorboat->name);
    printf("the motorboat speed = %d (km per hour)\n", motorboat-
>speed);
    printf("the motorboat travel time = %d (hour)\n", motorboat-
>time);
    if (motorboat->movement > 5)
    {
        printf("The motorboat cant be moving\n");
    }
    else
    {
        printf("The motorboat can be moving\n");
```

```
    }
}

/* options of motorboat traffic */
void moving(Motorboat *motorboat)
{
    switch (motorboat->movement)
    {
    case 0:
        printf("motorboat start moving\n");
        break;
    case 1:
        printf("motorboat moving forward\n");
        break;
    case 2:
        printf("motorboat moving back\n");
        break;
    case 3:
        printf("motorboat moving right\n");
        break;
    case 4:
        printf("motorboat moving left\n");
        break;
    case 5:
        printf("motorboat stop moving\n");
        break;
    }
}

int main()
{
    Motorboat SportMotorboat;
    init(&SportMotorboat);
    view(&SportMotorboat);
    moving(&SportMotorboat);
    printf("%c", '\n');

    Motorboat UsualMotorboat;
    init(&UsualMotorboat, "Yamaha 12Y", 90, 10, 5);
    view(&UsualMotorboat);
    moving(&UsualMotorboat);
    return 0;
}
```

Глава 2.2

```
/* Motorboat */
#include <cstdio>
#include <iostream>
#include <cstring>
using namespace std;
struct Motorboat
{
    char *name;               /* the name of the motorboat */
    unsigned int speed;       /* the speed of the motorboat(km per
 hour) */
    unsigned int movement;    /* the movement of the motorboat (0-
start moving, 1-
moving forward, 2 - moving back, 3 - moving right, 4 - moving left
, 5 - stop moving) */
    unsigned int time;        /*motorboat travel time (hour) */
    int additional_equipment; /* active component mark */
    union
    {
        char name_equipment[256];  /* mark =0 */
        int radio_range;           /* mark =1 */
        bool navigation_equipment; /* mark =2 */
    };
};

/* initialization of the motorboat */
void init(Motorboat *motorboat,
          char *name = "Yamaha 12X",
          unsigned int speed = 120,
          unsigned int movement = 1,
          unsigned int time = 2,
          int additional_equipment = 0)

{
    motorboat->name = name;
    motorboat->speed = speed;
    motorboat->movement = movement;
    motorboat->time = time;
    motorboat->additional_equipment = additional_equipment;
    switch (additional_equipment)
    {
    case 0:
        strcpy(motorboat->name_equipment, "sounder");
        break;
    case 1:
        motorboat->radio_range = 400;
        break;
    case 2:
```

```
        motorboat->navigation_equipment = true;
        break;
    default:
        motorboat->additional_equipment = 0;
        strcpy(motorboat->name_equipment, "sounder");
        break;
    }
}

/* visualization of the motorboat */
void view(Motorboat *motorboat)
{
    printf("the motorboat name = %s \n", motorboat->name);
    printf("the motorboat speed = %d (km per hour)\n", motorboat-
>speed);
    printf("the motorboat travel time = %d (hour)\n", motorboat-
>time);
    if (motorboat->movement > 5)
    {
        printf("The motorboat cant be moving\n");
    }
    else
    {
        printf("The motorboat can be moving\n");
    }
    printf("The additional equipment includes:\n");
    switch (motorboat->additional_equipment)
    {
    case 0:
        printf("%s\n", motorboat->name_equipment);
        break;
    case 1:
        printf("- radio communication\n");
        printf("the radio communication range = %d\n", motorboat-
>radio_range);
        break;
    case 2:
        printf("- navigation equipment\n");
        break;
    }
}

/* options of motorboat traffic */
void moving(Motorboat *motorboat)
{
    switch (motorboat->movement)
    {
    case 0:
        printf("motorboat start moving\n");
```

```
        break;
    case 1:
        printf("motorboat moving forward\n");
        break;
    case 2:
        printf("motorboat moving back\n");
        break;
    case 3:
        printf("motorboat moving right\n");
        break;
    case 4:
        printf("motorboat moving left\n");
        break;
    case 5:
        printf("motorboat stop moving\n");
        break;
    }
}

int main()
{
    Motorboat SportMotorboat;
    init(&SportMotorboat);
    view(&SportMotorboat);
    moving(&SportMotorboat);
    printf("%c", '\n');

    Motorboat UsualMotorboat;
    init(&UsualMotorboat, "Yamaha 12Y", 60, 2, 4, 1);
    view(&UsualMotorboat);
    moving(&UsualMotorboat);

    init(&UsualMotorboat, "Yamaha 12Z", 60, 3, 4, 2);
    printf("%c", '\n');
    view(&UsualMotorboat);
    moving(&UsualMotorboat);
    return 0;
}
```

Глава 3.1

```
/* Motorboat */
#include <iostream>
#include <string>
using namespace std;
class Motorboat
{
    string name;            /* the name of the motorboat */
    unsigned int speed;    /* the speed of the motorboat(km per ho
ur) */
    unsigned int movement; /* the movement of the motorboat (0-
start moving, 1-
moving forward, 2 - moving back, 3 - moving right, 4 - moving left
, 5 - stop moving) */
    unsigned int time;     /* motorboat travel time (hour) */

public:
    /* default constructor */
    Motorboat(string name = "Yamaha 12X",
              unsigned int speed = 120,
              unsigned int movement = 1,
              unsigned int time = 2) : name(name),
                                       speed(speed),
                                       movement(movement),
                                       time(time)
    {
    }
    /* copy constructor */
    Motorboat(const Motorboat &motorboat) : name(motorboat.name),
                                            speed(motorboat.speed)
,
                                            movement(motorboat.mov
ement),
                                            time(motorboat.time)
    {
    }
    /* options of the motorboat traffic */
    void moving();
    /* visualization of the motorboat */
    friend ostream &operator<<(ostream &, Motorboat);
};
void Motorboat::moving()
{
    switch (movement)
    {
    case 0:
        cout << "motorboat start moving" << endl;
        break;
```

```
    case 1:
        cout << "motorboat moving forward" << endl;
        break;
    case 2:
        cout << "motorboat moving back" << endl;
        break;
    case 3:
        cout << "motorboat moving right" << endl;
        break;
    case 4:
        cout << "motorboat moving left" << endl;
        break;
    case 5:
        cout << "motorboat stop moving" << endl;
        break;
    }
}
/* visualization of the motorboat */
ostream &operator<<(ostream &stream, Motorboat motorboat)
{
    stream << "the motorboat name = " << motorboat.name << "\n";
    stream << "the motorboat speed = " << motorboat.speed << " (km
 per hour)\n";
    stream << "the motorboat travel time = " << motorboat.time <<
" (hour)\n";
    if (motorboat.movement > 5)
    {
        stream << "The motorboat cant be moving\n";
    }
    else
    {
        stream << "The motorboat can be moving\n";
    }
    return stream;
}
int main()
{
    Motorboat SportMotorboat;
    cout << SportMotorboat;
    SportMotorboat.moving();
    cout << endl;
    Motorboat UsualMotorboat("Yamaha 12Y", 60, 2, 4);
    cout << UsualMotorboat;
    UsualMotorboat.moving();
    return 0;
}
```

Глава 3.2.1

```
/* Motorboat */
#include <iostream>
#include <string>
using namespace std;
class Motorboat
{
    unsigned int movement; /* the movement of the motorboat (0-
start moving, 1-
moving forward, 2 - moving back, 3 - moving right, 4 - moving left
, 5 - stop moving) */
    unsigned int speed;    /* the speed of the motorboat(km per ho
ur) */
    unsigned int time;     /* motorboat travel time (hour) */
    int count_of_name;
    string *name; /* the name of the motorboat */
public:
    /* default constructor */
    Motorboat(unsigned int movement = 1,
              unsigned int speed = 120,
              unsigned int time = 2,
              int count_of_name = 2,
              string name_1 = "Yamaha 12X",
              string name_2 = "Yamaha 12Y") : movement(movement),
                                              speed(speed),
                                              time(time),
                                              count_of_name(count_
of_name)
    {
        name = new string[count_of_name];
        switch (count_of_name)
        {
        case 1:
        {
            name[0] = name_1;
            break;
        }
        case 2:
        {
            name[0] = name_1;
            name[1] = name_2;
            break;
        }
        }
    }
    /*copy constructor */
    Motorboat(const Motorboat &motorboat) : movement(motorboat.mov
ement),
```

```
                                                    speed(motorboat.speed)
,
                                                    time(motorboat.time),
                                                    count_of_name(motorboa
t.count_of_name)

    {
        name = new string[motorboat.count_of_name];
        switch (motorboat.count_of_name)
        {
        case 1:
        {
            name[0] = motorboat.name[0];
            break;
        }
        case 2:
        {
            name[0] = motorboat.name[0];
            name[1] = motorboat.name[1];
            break;
        }
        }
    }
    /* destructor */
    ~Motorboat()
    {
        delete[] name;
    }
    /* options of the motorboat traffic */
    void moving();
    /* visualization of the motorboat */
    friend ostream &operator<<(ostream &, Motorboat);
};
void Motorboat::moving()
{
    switch (movement)
    {
    case 0:
        cout << "motorboat start moving" << endl;
        break;
    case 1:
        cout << "motorboat moving forward" << endl;
        break;
    case 2:
        cout << "motorboat moving back" << endl;
        break;
    case 3:
        cout << "motorboat moving right" << endl;
        break;
```

```
    case 4:
        cout << "motorboat moving left" << endl;
        break;
    case 5:
        cout << "motorboat stop moving" << endl;
        break;
    }
}
/* visualization of the motorboat */
ostream &operator<<(ostream &stream, Motorboat motorboat)
{
    switch (motorboat.count_of_name)
    {
    case 1:
        stream << "the motorboat name = " << motorboat.name[0] <<
"\n";
        break;
    case 2:
        stream << "the motorboat name = " << motorboat.name[0] <<
" and " << motorboat.name[1] << "\n";
        break;
    }
    stream << "the motorboat speed = " << motorboat.speed << " (km
 per hour)\n";
    stream << "the motorboat travel time = " << motorboat.time <<
" (hour)\n";
    if (motorboat.movement > 5)
    {
        stream << "The motorboat cant be moving\n";
    }
    else
    {
        stream << "The motorboat can be moving\n";
    }
    return stream;
}
int main()
{
    Motorboat SportMotorboat;
    cout << SportMotorboat;
    SportMotorboat.moving();
    cout << endl;

    Motorboat UsualMotorboat_1(2, 60, 4, 1, "Yamaha 12Z");
    cout << UsualMotorboat_1;
    UsualMotorboat_1.moving();
    cout << endl;

    Motorboat UsualMotorboat_2(UsualMotorboat_1);
```

```
    cout << UsualMotorboat_2;
    UsualMotorboat_2.moving();
    cout << endl;
    return 0;
}
```

Глава 3.2.2

```
/* Motorboat */
#include <iostream>
#include <string>
using namespace std;
class Motorboat
{
    unsigned int movement; /* the movement of the motorboat (0-
start moving, 1-
moving forward, 2 - moving back, 3 - moving right, 4 - moving left
, 5 - stop moving) */
    unsigned int speed;    /* the speed of the motorboat(km per ho
ur) */
    unsigned int time;     /* motorboat travel time (hour) */
    int count_of_name;
    string *name; /* the name of the motorboat */
public:
    /* default constructor */
    Motorboat(unsigned int movement = 1,
              unsigned int speed = 120,
              unsigned int time = 2,
              int count_of_name = 2,
              string name_1 = "Yamaha 12X",
              string name_2 = "Yamaha 12Y") : movement(movement),
                                              speed(speed),
                                              time(time),
                                              count_of_name(count_
of_name)
    {
        name = new string[count_of_name];
        if (name ==nullptr)
        {
            throw "Memory is not allocated";
        }
        switch (count_of_name)
        {
        case 1:
        {
            name[0] = name_1;
            break;
        }
        case 2:
        {
            name[0] = name_1;
            name[1] = name_2;
            break;
        }
        }
```

```
    }
    /*copy constructor */
    Motorboat(const Motorboat &motorboat) : movement(motorboat.mov
ement),
                                            speed(motorboat.speed)
,
                                            time(motorboat.time),
                                            count_of_name(motorboa
t.count_of_name)

    {
        name = new string[motorboat.count_of_name];
        switch (motorboat.count_of_name)
        {
        case 1:
        {
            name[0] = motorboat.name[0];
            break;
        }
        case 2:
        {
            name[0] = motorboat.name[0];
            name[1] = motorboat.name[1];
            break;
        }
        }
    }
    /* destructor */
    ~Motorboat()
    {
        delete[] name;
    }
    /* options of the motorboat traffic */
    void moving();
    /* visualization of the motorboat */
    friend ostream &operator<<(ostream &, Motorboat);
};
void Motorboat::moving()
{
    switch (movement)
    {
    case 0:
        cout << "motorboat start moving" << endl;
        break;
    case 1:
        cout << "motorboat moving forward" << endl;
        break;
    case 2:
        cout << "motorboat moving back" << endl;
```

```
        break;
    case 3:
        cout << "motorboat moving right" << endl;
        break;
    case 4:
        cout << "motorboat moving left" << endl;
        break;
    case 5:
        cout << "motorboat stop moving" << endl;
        break;
    }
}
/* visualization of the motorboat */
ostream &operator<<(ostream &stream, Motorboat motorboat)
{
    switch (motorboat.count_of_name)
    {
    case 1:
        stream << "the motorboat name = " << motorboat.name[0] <<
"\n";
        break;
    case 2:
        stream << "the motorboat name = " << motorboat.name[0] <<
" and " << motorboat.name[1] << "\n";
        break;
    }
    stream << "the motorboat speed = " << motorboat.speed << " (km
 per hour)\n";
    stream << "the motorboat travel time = " << motorboat.time <<
" (hour)\n";
    if (motorboat.movement > 5)
    {
        stream << "The motorboat cant be moving\n";
    }
    else
    {
        stream << "The motorboat can be moving\n";
    }
    return stream;
}
int main()
{
    Motorboat SportMotorboat;
    cout << SportMotorboat;
    SportMotorboat.moving();
    cout << endl;

    Motorboat UsualMotorboat_1(2, 60, 4, 1, "Yamaha 12Z");
    cout << UsualMotorboat_1;
```

```
    UsualMotorboat_1.moving();
    cout << endl;

    Motorboat UsualMotorboat_2(UsualMotorboat_1);
    cout << UsualMotorboat_2;
    UsualMotorboat_2.moving();
    cout << endl;
    return 0;
}
```

Глава 3.3

```
/* Motorboat */
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
class Motorboat
{
    string name;        /* the name of the motorboat */
    unsigned int speed;    /* the speed of the motorboat(km per ho
ur) */
    unsigned int movement; /* the movement of the motorboat (0-
start moving, 1-
moving forward, 2 - moving back, 3 - moving right, 4 - moving left
, 5 - stop moving) */
    unsigned int time;     /* motorboat travel time (hour) */
    ifstream stream;       /* input file stream */

public:
    /* default constructor */
    Motorboat(const string file_name = "SportMotorboat.txt")
    {
        stream.open(file_name);
        if (stream.is_open())
        {
            getline(stream,name);
            stream >> speed;
            stream >> movement;
            stream >> time;
        }
        else
        {
            /* data member default initialization */
            name = "Yamaha 12Z";
            speed = 60;
            movement = 0;
            time = 3;
        }
    }
    /* copy constructor */
    Motorboat(const Motorboat &motorboat) : name(motorboat.name),
                                            speed(motorboat.speed)
,
                                            movement(motorboat.mov
ement),
                                            time(motorboat.time)
    {
```

```
    }
    /* destructor */
    ~Motorboat()
    {
        if (stream.is_open())
            stream.close();
    }
    /* options of the motorboat traffic */
    void moving();
    /* visualization of the motorboat */
    friend ostream &operator<<(ostream &, Motorboat);
};
void Motorboat::moving()
{
    switch (movement)
    {
    case 0:
        cout << "motorboat start moving" << endl;
        break;
    case 1:
        cout << "motorboat moving forward" << endl;
        break;
    case 2:
        cout << "motorboat moving back" << endl;
        break;
    case 3:
        cout << "motorboat moving right" << endl;
        break;
    case 4:
        cout << "motorboat moving left" << endl;
        break;
    case 5:
        cout << "motorboat stop moving" << endl;
        break;
    }
}
/* visualization of the motorboat */
ostream &operator<<(ostream &stream, Motorboat motorboat)
{
    stream << "the motorboat name = " << motorboat.name << "\n";
    stream << "the motorboat speed = " << motorboat.speed << " (km
 per hour)\n";
    stream << "the motorboat travel time = " << motorboat.time <<
" (hour)\n";
    if (motorboat.movement > 5)
    {
        stream << "The motorboat cant be moving\n";
    }
    else
```

```
    {
        stream << "The motorboat can be moving\n";
    }
    return stream;
}
int main()
{
    Motorboat SportMotorboat;
    cout << SportMotorboat;
    SportMotorboat.moving();
    cout << endl;
    Motorboat UsualMotorboat("UsualMotorboat.txt");
    cout << UsualMotorboat;
    UsualMotorboat.moving();
    return 0;
}
```

Глава 4

```
#include <iostream>
#include <string>
using namespace std;
/* Base class */
class Water_transport
{
protected:
    unsigned int tonnage;
    unsigned int passengers_capacity;
    /* default constructor */
    Water_transport(unsigned int tonnage = 530) : tonnage(tonnage)

    {
        passengers_capacity = tonnage / 100;
    }
    /* copy constructor */
    Water_transport(const Water_transport &water_transport) : tonn
age(water_transport.tonnage),
                                                              pass
engers_capacity(water_transport.passengers_capacity)
    {
    }
    /* destructor */
    virtual ~Water_transport()
    {
    }
};
/* Derived Class */
class Motorboat : public Water_transport
{
    string name;           /* the name of the motorboat */
    unsigned int speed;    /* the speed of the motorboat(km per ho
ur) */
    unsigned int movement; /* the movement of the motorboat (0-
start moving, 1-
moving forward, 2 - moving back, 3 - moving right, 4 - moving left
, 5 - stop moving) */
    unsigned int time;     /* motorboat travel time (hour) */

public:
    /* default constructor */
    Motorboat(unsigned int tonnage = 530,
              string name = "Yamaha 12X",
              unsigned int speed = 120,
              unsigned int movement = 1,
              unsigned int time = 2) : Water_transport(tonnage),
                                       name(name),
```

```
                                        speed(speed),
                                        movement(movement),
                                        time(time)
    {
    }
    /* copy constructor */
    Motorboat(const Motorboat &motorboat) : name(motorboat.name),
                                            speed(motorboat.speed)
,
                                            movement(motorboat.mov
ement),
                                            time(motorboat.time)
    {
        this->tonnage=motorboat.tonnage;
        this->passengers_capacity=motorboat.passengers_capacity;
    }
    /* destructor */
    ~Motorboat()
    {
    }
    /* options of the motorboat traffic */
    void moving();
    /* visualization of the motorboat */
    friend ostream &operator<<(ostream &, Motorboat);
};
void Motorboat::moving()
{
    switch (movement)
    {
    case 0:
        cout << "motorboat start moving" << endl;
        break;
    case 1:
        cout << "motorboat moving forward" << endl;
        break;
    case 2:
        cout << "motorboat moving back" << endl;
        break;
    case 3:
        cout << "motorboat moving right" << endl;
        break;
    case 4:
        cout << "motorboat moving left" << endl;
        break;
    case 5:
        cout << "motorboat stop moving" << endl;
        break;
    }
}
```

```
/* visualization of the motorboat */
ostream &operator<<(ostream &stream, Motorboat motorboat)
{
    stream << "the tonnage of the motorboat = " << motorboat.tonna
ge << " kg\n";
    stream << "the passengers capasity of the motorboat = " << mot
orboat.passengers_capacity << "\n";
    stream << "the motorboat name = " << motorboat.name << "\n";
    stream << "the motorboat speed = " << motorboat.speed << " (km
 per hour)\n";
    stream << "the motorboat travel time = " << motorboat.time <<
" (hour)\n";
    if (motorboat.movement > 5)
    {
        stream << "the motorboat cant be moving\n";
    }
    else
    {
        stream << "the motorboat can be moving\n";
    }
    return stream;
}
int main()
{
    Motorboat SportMotorboat;
    cout << SportMotorboat;
    SportMotorboat.moving();
    cout << endl;

    Motorboat UsualMotorboat_1(475,"Yamaha 12Y", 60, 2, 4);
    cout << UsualMotorboat_1;
    UsualMotorboat_1.moving();
    cout << endl;
    return 0;
}
```

Библиографический список

1. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд.: перевод с английского под редакцией И. Романовского и Ф. Андреева, 2008. – 720 с.
2. В.М.Дёмкин. Основы объектно-ориентированного программирования в примерах на C++: Учебное пособие. – Н.Новгород: НФ ГУ-ВШЭ, 2005. – 148 с.
3. С. Макконнелл. Совершенный код: Практическое руководство по разработке программного обеспечения. - Москва, 2010, 121-134 с.
4. Герберт Шилдт. C++ для начинающих: Учебное издание. – Москва, 2013, 364-400 с.
5. Ravesli. Программирование для начинающих: [Электронный ресурс]. URL: https://ravesli.com/. (Дата обращения: 12.12.2020-21.12.2020).
6. CyberForum. C++ для начинающих: [Электронный ресурс]. URL: https://www.cyberforum.ru/cpp-beginners/. (Дата обращения: 19.12.2020).