

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие для выполнения лабораторных работ студентов
IT-направлений очной и заочной форм обучения

Оглавление

1 Краткая теория и пример выполнения лабораторных работ.....	4
1.1 Классы. Модификаторы доступа. Поля и методы.....	4
1.2 Вывод на консоль	18
1.3 Работа с динамической памятью в C++ и конструкции инициализации	20
1.4 Модификатор const.....	22
1.5 Ссылка	25
1.6 Применение модификатора const и для повышения надежности класса и определения конструктора копирования	26
1.7 Обработка исключений.....	30
1.8 Перегрузка функций, методов и операторов	33
1.9 Параметры по умолчанию	40
1.10 Наследование	42
1.11 Интерфейсы, абстрактные классы, виртуальные методы.....	48
1.12 Шаблоны.....	55
2 Лабораторная работа № 1. Повторение языка программирования C.....	58
3 Общие требования к лабораторным работам № 2 – 6.....	60
4 Варианты заданий лабораторных работ	61
Вариант 1. «Расчет налога и налогового вычета»	61
Лабораторная работа № 2. Базовый синтаксис классов	61
Лабораторная работа № 3. Перегрузка методов и операторов	61
Лабораторная работа № 4. Наследование	61
Лабораторная работа № 5. Абстрактные классы и интерфейсы.....	62
Лабораторная работа № 6. Шаблоны.....	62
Вариант 2. «Календарный план-график ипотеки»	62
Лабораторная работа № 2. Базовый синтаксис классов	62
Лабораторная работа № 3. Перегрузка методов и операторов	63
Лабораторная работа № 4. Наследование	63
Лабораторная работа № 5. Абстрактные классы и интерфейсы.....	63
Лабораторная работа № 6. Шаблоны.....	63
Вариант 3. «Суточный рацион»	63
Лабораторная работа № 2. Базовый синтаксис классов	64
Лабораторная работа № 3. Перегрузка методов и операторов	64
Лабораторная работа № 4. Наследование	64
Лабораторная работа № 5. Абстрактные классы и интерфейсы.....	64
Лабораторная работа № 6. Шаблоны.....	65
Вариант 4. «Планирование уплаты счетов по электроэнергии»	65
Лабораторная работа № 2. Базовый синтаксис классов	65
Лабораторная работа № 3. Перегрузка методов и операторов	65
Лабораторная работа № 4. Наследование	66
Лабораторная работа № 5. Абстрактные классы и интерфейсы.....	66
Лабораторная работа № 6. Шаблоны.....	66
Вариант 5. «Учет дискового пространства сотрудников».....	66
Лабораторная работа № 2. Базовый синтаксис классов	66
Лабораторная работа № 3. Перегрузка методов и операторов	67
Лабораторная работа № 4. Наследование	67
Лабораторная работа № 5. Абстрактные классы и интерфейсы.....	67
Лабораторная работа № 6. Шаблоны.....	68
Вариант 6. «Поиск оптимальной стоимости товара».....	68
Лабораторная работа № 2. Базовый синтаксис классов	68

Лабораторная работа № 3. Перегрузка методов и операторов	68
Лабораторная работа № 4. Наследование	69
Лабораторная работа № 5. Абстрактные классы и интерфейсы	69
Лабораторная работа № 6. Шаблоны	69
Вариант 7. «Учет товаров обихода»	69
Лабораторная работа № 2. Базовый синтаксис классов	69
Лабораторная работа № 3. Перегрузка методов и операторов	70
Лабораторная работа № 4. Наследование	70
Лабораторная работа № 5. Абстрактные классы и интерфейсы	70
Лабораторная работа № 6. Шаблоны	70
Вариант 8. «Учет звонков»	70
Лабораторная работа № 2. Базовый синтаксис классов	71
Лабораторная работа № 3. Перегрузка методов и операторов	71
Лабораторная работа № 4. Наследование	71
Лабораторная работа № 5. Абстрактные классы и интерфейсы	72
Вариант 9. «Учет сдачи лабораторных работ»	72
Лабораторная работа № 2. Базовый синтаксис классов	72
Лабораторная работа № 3. Перегрузка методов и операторов	72
Лабораторная работа № 4. Наследование	73
Лабораторная работа № 5. Абстрактные классы и интерфейсы	73
Лабораторная работа № 6. Шаблоны	73
Вариант 10. «Статистика рабочего времени на выполнения задач»	73
Лабораторная работа № 2. Базовый синтаксис классов	73
Лабораторная работа № 3. Перегрузка методов и операторов	74
Лабораторная работа № 4. Наследование	74
Лабораторная работа № 5. Абстрактные классы и интерфейсы	74
Лабораторная работа № 6. Шаблоны	74
Вариант 11. «Учет приема лекарств»	74
Лабораторная работа № 2. Базовый синтаксис классов	74
Лабораторная работа № 3. Перегрузка методов и операторов	75
Лабораторная работа № 4. Наследование	75
Лабораторная работа № 5. Абстрактные классы и интерфейсы	75
Лабораторная работа № 6. Шаблоны	75
Вариант 12. «Результаты обучения модели»	75
Лабораторная работа № 2. Базовый синтаксис классов	76
Лабораторная работа № 3. Перегрузка методов и операторов	76
Лабораторная работа № 4. Наследование	76
Лабораторная работа № 5. Абстрактные классы и интерфейсы	77
Лабораторная работа № 6. Шаблоны	77
Вариант 13. «Учет результатов антиплагиата»	77
Лабораторная работа № 2. Базовый синтаксис классов	77
Лабораторная работа № 3. Перегрузка методов и операторов	78
Лабораторная работа № 4. Наследование	78
Лабораторная работа № 5. Абстрактные классы и интерфейсы	78
Лабораторная работа № 6. Шаблоны	78
Список литературы	79
ПРИЛОЖЕНИЕ А (справочное) Пример работы с датой и временем	80

1 КРАТКАЯ ТЕОРИЯ И ПРИМЕР ВЫПОЛНЕНИЯ ЛАБОРАТОРНЫХ РАБОТ

1.1 Классы. Модификаторы доступа. Поля и методы

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования. Или если еще более просто, парадигма программирования, основанная на использовании классов, их объектов и трех-четырех принципов: инкапсуляция, полиморфизм, наследование и абстракция (об каждом понятии позже).

Класс – это пользовательский тип данных, состоящий из методов и полей, схож с конструкцией структура (*struct*), но имеет свои особенности.

Поле класса (свойство, атрибут) – переменная или переменная-объект, объявленная внутри класса.

Метод – функция, объявленная внутри класса и используемая для манипуляции над значениями полей и реализации логики класса (функционала).

Класс содержит поля и методы. Примером класса может выступать окно графического интерфейса. Что схожее между окнами? У них есть размеры, заголовки, они имеют логику открытия, закрытия, сворачивания, масштабирования и так далее. Можно предположить, что размеры и заголовок окна могут быть полями некоторого обобщенного (абстрактного) представления окна, а вот функции открытия, закрытия и изменения размера окна – методами класса, с помощью которых можно задать выполнение какой-то фиксированной логики на эти события (например, при изменении размеров окна хотим перегруппировать элементы управления внутри него). Мы понимаем, что есть что-то общее у всех окон, и что для реализации приложения нам может понадобиться создать несколько окон условно по одному шаблону. Поэтому можем сформировать один класс «Окно», а потом создавать экземпляры этого класса – объекты.

Объект – это конкретный экземпляр заданного класса, или если сказать проще – это переменная сложного составного типа – класса. Объектов класса может быть много – а класс один.

Другим примером класса может быть описание сотрудника. У сотрудников есть ряд общих важных для нас параметров: фамилия, имя, отчество, должность, размер заработной платы, паспортные данные, адрес, номер счета, его доступность, необходимая к выдаче зарплата и другое, а также мы можем проводить с ним различные операции: начислять зарплату, отправлять в отпуск или командировку. Поэтому, представив эту сущность – «Сотрудник» – можно задать механизмы структурирования данных о сотруднике и их обработки в виде класса. Отчасти класс может переключаться в описании предметной области с сущностью (таблицей) в базах данных, где класс похож на таблицу: его поля – на колонки-атрибуты таблицы, а строки – на экземпляры класса.

Рассмотрим, как будет выглядеть класс для более простой сущности. Например, нам необходимо вести учет товара на складе. У товара есть название и его количество. Нам необходимо знать, сколько товара есть на данный момент на складе, а какой закончился. Определим класс «*Product*». В C++ класс обозначается ключевым словом **class**, далее следует имя, после чего внутри блоковых скобок определяются поля и методы. Объявление класса, как и структуры, должно заканчиваться точкой с запятой.

На рисунке 1.1 представлен пример, где:

- «*count*», «*name*» – поля класса;
- «*isEmpty*» – метод класса;
- «*Product*» – наименование класса;
- **class** – ключевое слово, объявляющее класс, после которого следует его наименование, а в блоковых скобках указывается структура класса.
- класс завершается точкой с запятой, как и *struct*.

```

class Product
{
public: // модификатор доступа

    int count = 0; // поле количество продукта
    char* name;    // поле наименования продукта

    bool isEmpty() { // метод класса
        return count == 0; // обращение к полю класса
    }
};

```

Рисунок 1.1 – Простейший класс

Каждый класс должен быть определен в отдельных файлах: заголовочном файле и файле исходного кода. Тело (реализация) методов класса должно находиться не в заголовочном файле («h»), а в файле исходного кода («сpp»). В заголовочном файле может быть только прототип метода.

В примерах пособия для удобства отображения кода деление на файл исходного кода и заголовочный файл проходить не будет. Хотя класс всегда определяется в отдельных «h» и «сpp» файлах, а название этих файлов соответствует наименованию класса, кроме определения шаблонных классов (будут рассмотрены в самом конце).

В отличие от логики *struct* в языке C классы имеют следующие особенности. Каждое поле класса имеет **модификатор доступа** – область видимости переменной или функции для внешних объектов:

- *public* – поле или метод видно всем – можно к ним обращаться вне класса;
- *private* – поле или метод видно только внутри класса;
- *protected* – поле или метод видно только внутри класса или его потомка (про наследование позже).

Доступ к полям *private* можно получить только внутри методов данного класса, как, например, в методе «*isEmpty*» мы обратились к полю «*count*» внутри класса. Доступ к **public**-полям и методам объекта класса из внешнего мира (в месте его использования) можно иметь через оператор «.» для переменной (статического объекта) или «->» – для указателя (динамического объекта). В функции *main* определим объекты класса «*Product*» двумя способами – статически и динамически (рисунок 1.2).

```

int main() { // точка входа в приложение

    Product staticMemoryProduct; // определили статически
    Product* dynamicMemoryProduct = new Product; // определили динамически

    staticMemoryProduct.count = 1; // изменили значение поля
    dynamicMemoryProduct->count = 1; // изменили значение поля

    delete dynamicMemoryProduct; // освободили память

    return 0;
}

```

Рисунок 1.2 – Обращение к полям объекта класса «Product»

Как видно из примера на рисунке 1.2, через оператор точка был получен доступ к полю объекта, определенного статически (память предоставлена в сегменте стека), а доступ через оператор стрелка – к динамическому объекту, адрес в памяти которого лежит в указателе (память захватили мы, память под объект лежит в heap-куче).

Если взглянуть на пример рисунка 1.1 после анализа рисунка 1.2, то возникает ощущение, что обращение к полю «*count*» в методе «*IsEmpty*» производилось как к локальной или глобальной переменной, а не через оператор доступа для определенного объекта («*имя_объекта.имя_поля*»). Но это не совсем так. Внутри кода класса указателем на текущий объект является *this*, то есть специализированное слово *this* – это указатель

на текущий объект класса, в котором мы находимся в данный момент и для которого метод был вызван. Данное слово может быть опущено внутри класса, так как мы предполагаем, что к полям и методам класса всегда имеем доступ внутри класса. Поэтому формально, реализация метода на рисунке 1.3а равносильна реализации на рисунке 1.3б.

```
bool Product::isEmpty() {
    return this->count == 0;
}
```

а)

```
bool Product::isEmpty() {
    return count == 0;
}
```

б)

Рисунок 1.3 – Использование слова *this*:

а) обращение к полю класса через *this*; б) обращение к полю класса без *this*;

В примере на рисунках «Product::» указывает, что этот метод является частью класса «Product», так можно определять методы в файле «сpp», то есть вне блоковых скобок класса, чтобы компилятор понимал, для какого класса определен метод.

Когда же может понадобиться использование ключевого слова *this*? Наиболее распространенным примером может быть уточнение, к какой именно переменной мы хотим обратиться, когда существует совпадение имен переменных, например, поля и формального параметра метода. В код на рисунке 1.4 добавим метод, который бы позволил изменить значение поля «count», где входным параметром будет являться формальный параметр с таким же именем «count». Тогда легко заметить, что непонятно будет компилятору, какая переменная к какой переменной присваивается, и возникнет ошибка компиляции.

```
class Product
{
public:
    int count = 0;
    char* Name;

    bool isEmpty() {
        return this->count == 0;
    }

    void setCount( int count ) {
        count = count; // warning: explicitly assigning value of variable of
                        // type 'int' to itself
    }
};
```

Рисунок 1.4 – Ошибка, связанная с одинаковым наименованием переменных

Для разрешения данного конфликта достаточно уточнить ключевым словом *this*, где именно необходимо использовать поле класса (рисунок 1.5).

```
void Product::setCount(int count) {
    this->count = count;
}
```

Рисунок 1.5 – Исправление ошибки, связанной с одинаковым наименованием поля и параметра

Еще одним примером использования слова *this* может быть возврат из метода указателя или ссылки на экземпляр класса. Пример такого использования будет приведен в подразделе про перегрузку функции, методов и операторов.

Стоит отметить, что доступ к полям «во внешнем коде» – например, в функции «*main*» – мы получили только к тем из них, которые помечены модификатором *public*. Поэтому, если бы мы написали следующим образом, как показано на рисунке 1.6, то мы бы увидели следующую ошибку в функции *main* (рисунок 1.7).

```
class Product
{
private:
    int count = 0;
public:
    char* Name;
    bool isEmpty() {
        return this->count == 0;
    }
    void setCount(int count) {
        this->count = count;
    }
};
```

Рисунок 1.6 – Изменение модификатора доступа у поля на *private*

```
int main() {
    Product staticMemoryProduct;
    Product* dynamicMemoryProduct = new Product;

    staticMemoryProduct.count = 1; // error: 'count' is a private member of 'Product'
    dynamicMemoryProduct->count = 1; // error: 'count' is a private member of 'Product'

    bool isEmpty = staticMemoryProduct.isEmpty(); // без ошибок
    isEmpty = dynamicMemoryProduct->isEmpty(); // без ошибок

    delete dynamicMemoryProduct;

    return 0;
}
```

Рисунок 1.7 – Ошибка обращения к *private*-полю

На рисунке 1.7 видно принципиальное отличие *private*-поля «*count*» и *public*-метода «*isEmpty*», а, именно, какой из модификаторов определяет методы и поля, к которым можно будет иметь доступ вне кода класса – во внешнем коде.

Может возникнуть вопрос, а что, если необходим метод или поле, которые были бы общими для всех объектов данного класса и была бы возможность вызывать методы без создания экземпляра данного класса? Для этого существует модификатор *static*. Его допустимо применять в методах, которые не обращаются внутри своего тела к методам и полям класса (то есть к членам экземпляра класса). Также статические поля и методы должны быть помечены модификатором *public* для доступа к ним из внешнего кода, где вместо оператора точка «.» для доступа к ним нужно будет использовать имя класса и оператор пространства имен (например, «*Product::*»), так как экземпляр класса в данном случае не создается.

Внесем еще изменение – уберем модификатор *private* из определения класса (рисунок 1.8). И окажется, что поле «*count*» – осталось приватным, а ошибка не исчезла. Почему же так произошло? А все потому, что по умолчанию все поля и методы класса – **приватные**, то есть если перед объявлением поля или метода не следует модификатор доступа – все поля и методы *private*. Как же насчет «*Name*», «*isEmpty*» и «*setCount*»? А вот перед этими методами и полями есть модификатор *public*, который определяет блок публичных методов и полей. Это означает, что внутри класса можно перемежать блоки публичных, приватных и защищенных полей, но по правилам правильного оформления кода лучше всего их группировать, а имена публичных полей начинать с большой буквы, при-

ватных и защищенных – с маленькой буквы (или выделять приватные поля приставкой «*m_*» в имени).

```
class Product
{
    int count = 0;
public:
    char* Name;
    bool isEmpty() {
        return this->count == 0;
    }
    void setCount( int count ) {
        this->count = count;
    }
};
```

Рисунок 1.8 – Другой способ задания *private*-области

Исправим ошибку в коде функции *main*, как показано на рисунке 1.9.

```
int main() {
    Product staticMemoryProduct;
    Product* dynamicMemoryProduct = new Product;

    staticMemoryProduct.setCount( 1 ); // без ошибок
    dynamicMemoryProduct->setCount( 1 ); // без ошибок

    bool isEmpty = staticMemoryProduct.isEmpty(); // без ошибок
    isEmpty = dynamicMemoryProduct->isEmpty(); // без ошибок

    delete dynamicMemoryProduct;

    return 0;
}
```

Рисунок 1.9 – Правильный подход в изменении поля класса

Для чего вообще нужны модификаторы доступа? Изначально ООП основывался на парадигме – данные сами управляют собою. И действительно – в ООП некоторая предметная область делится на сущности, а сущность, кроме некоторых своих свойств, имеет специализированные методы, которые меняют ее состояния. Поэтому любой внешний код может поменять состояние объекта без знания его внутренней логики, просто вызвав метод смены состояния. Но, с другой стороны, только сам класс знает о своей внутренней логике, и прямое вмешательство в изменение его полей может повлечь к плачевным последствиям. Поэтому было решено – ограничить доступ к части полей и методов класса для того, чтобы через специализированные методы проверять те значения параметров, которые передает внешний код (входные данные), и допускать только санкционированные и допустимые изменения внутри класса. Кроме этого, определять логику, как эти данные будут передаваться. Как правило, практически все поля класса определяются приватными. Эта идея называется **инкапсуляцией** или **сокрытием данных**. Такой подход позволяет:

- скрывать сложность, позволяя программистам забыть о ней при работе над остальными частями программы;
- скрывать источники изменений с целью локализации результатов возможных изменений [1].

Исходя из вышесказанного, можно сделать вывод о том, что поле «*Name*», которое определено строкой (массив последовательности символов *char*, которые завершаются нуль-терминатором, а это поле – это указатель на этот массив), внешним воздействием может быть инициализировано неверным значением. Например, внешнее воздействие может обнулить указатель, освободить память под указателем, присвоить пустую строку, или не соответствует правилу, что имя может начинаться только с буквы, которое мы хо-

тели бы определить. Тогда нам бы нужно (и даже требуется) организовать класс, как указано на рисунке 1.10.

```
#include <string.h>

using namespace std;

class Product
{
    int count = 0;
    char* name = NULL;
public:
    bool isEmpty() {
        return this->count == 0;
    }
    void setCount( int count ) {
        this->count = count;
    }
    void setName( char *value ) {

        if( value == NULL ) { // проверяем входной параметр на ноль
            // бросаем исключение
            throw "NULL указатель пришел в качестве аргумента";
        }
        unsigned int sizeStr = strlen( value ); //получаем размер строки

        if( !sizeStr ) { // если строка нулевого размера - пустая
            throw "Строка не может быть пустой";
        }

        if( ( value[0] >= 'a' && value[0] <= 'z' ) ||
            ( value[0] >= 'A' && value[0] <= 'Z' ) ) {
            // если первый символ строки буква

            if( name != NULL ) {
                delete name; // если уже есть строка, ее надо освободить
                name = NULL;
            }

            name = new char[ sizeStr + 1 ]; // захватываем память под строку
            memset( name , 0 , sizeof(char)*(sizeStr + 1) ); // обнуляем память
            strcpy( name , value ); // копируем строку
        }
        else {
            throw "Параметр должен начинаться с буквы";
        }
    }
};

int main() {

    Product staticMemoryProduct;
    Product* dynamicMemoryProduct = new Product;

    staticMemoryProduct.setCount( 1 );
    dynamicMemoryProduct->setCount( 1 );

    staticMemoryProduct.setName( "table N111" );
    dynamicMemoryProduct->setName( "sofa N112" );

    delete dynamicMemoryProduct;

    return 0;
}
```

Рисунок 1.10 – Правильный подход в изменении строкового поля класса

В коде на рисунке 1.10 ключевым словом *throw* установлены условия «выбрасывания исключения». Про эту конструкцию C++ можно узнать из соответствующего подраздела. Пока остановимся на том, что эта конструкция позволит известить внешний код

об возникновении ошибки. В методе «*setName*» были использованы операторы C++ *new-delete*, функции работы со строками и памятью. Их описание можно прочитать в справочной документации на ресурсе «*srppreference*» [2].

Если быть очень внимательным, то можно заметить, что в поле «*name*», захватывается динамическая память, но она нигде не освобождается. А также можно предположить, что для создания и заполнения полей одного объекта, необходимо как минимум две строки кода (для «*count*» и «*name*») в функции *main*. Поэтому хотелось бы объединить эту операцию в один метод, а лучше, чтобы создание объекта и инициализация полей проводилась с помощью метода, который бы содержал список формальных параметров, которые бы определяли начальное значение для этих полей объекта при его создании. То есть хотелось бы создание и подготовку объекта упростить. Для решения данных проблем в C++ существуют такие специализированные методы, как конструктор и деструктор.

Конструктор – это метод, который вызывается при выполнении программы (фактически не нами), когда создается экземпляр класса. Его наименование совпадает с именем класса.

Определим конструктор без формальных параметров, который заполняет поля параметрами со значениями по умолчанию (рисунок 1.11). Здесь представлен конструктор, который не имеет входных формальных параметров и проводит инициализацию полей класса константными значениями. Обычно в конструкторе могут происходить всевозможные инициализации, захват памяти, открытие файлов и т.д.

```
class Product
{
    int count = 0;
    char* name = NULL;
public:
    Product() {
        setCount( 1 );
        setName( "Default product" );
    }

    bool isEmpty() {

        return this->count==0;
    }

    ...
};
```

Рисунок 1.11 – Конструктор по умолчанию

Конструктор в примере будет вызван в указанных строках кода на рисунке 1.12.

```
int main() {

    // вызов конструктора для статического объекта
    Product staticMemoryProduct;

    // вызов конструктора для динамического объекта
    Product* dynamicMemoryProduct = new Product;

    delete dynamicMemoryProduct;

    return 0;
}
```

Рисунок 1.12 – Места вызова конструктора

Внимательный человек отметит, что у конструктора, как у функции или метода есть круглые скобки при определении формальных параметров, и мы привыкли, что при вызове метода всегда используются круглые скобки. Поэтому отметим, что никто не запрещает сделать так, как указано на рисунке 1.13.

```
int main() {
    Product staticMemoryProduct();
    Product* dynamicMemoryProduct = new Product();

    delete dynamicMemoryProduct;

    return 0;
}
```

Рисунок 1.13 – Использование круглых скобок при создании объекта

По той же логике, как слово *this* исключается в местах, где неявно определено, что внутри класса используются поля или методы класса, также можно опускать обозначение круглых скобок в конструкторе без параметров. Определенный конструктор без параметров называется **конструктором по умолчанию**. Почему «по умолчанию»? Это означает, что при пропуске входных параметров конструктора, или в местах, где явно невозможно определить параметры, будет происходить вызов конструктора по умолчанию. Например, при создании массива объектов, как показано на рисунке 1.14.

```
int main() {
    Product staticMemoryProduct;
    Product* dynamicMemoryProduct = new Product;

    Product arrayProduct[ 10 ]; // вызов конструкторов для каждого элемента массива

    delete dynamicMemoryProduct; // место остановки выполнения

    return 0;
}
```

Рисунок 1.14 – Вызов конструктора по умолчанию для массива объектов

Тогда в отладчике (в точке останова – красный шарик, который можно поставить около номера строки) можно посмотреть заполнение объектов, которые соответствуют настройке полей определенного конструктора по умолчанию (рисунок 1.15).

Имя	Значение	Тип
arrayProduct	@0x61fe74	Product[10]
[0]	@0x61fe74	Product
count	1	int
> name	"Default product"	char *
[1]	@0x61fe7c	Product
count	1	int
> name	"Default product"	char *
[2]	@0x61fe84	Product
count	1	int
> name	"Default product"	char *
[3]	@0x61fe8c	Product
count	1	int
> name	"Default product"	char *
[4]	@0x61fe94	Product
count	1	int
> name	"Default product"	char *

Рисунок 1.15 – Промежуточные значения объектов в точке останова

Стоит отметить, что при динамическом определении массива с использованием операторов *new-delete* конструктор по умолчанию также будет вызван автоматически. Такая же логика применима, когда свой класс используется в таких контейнерах стандартной библиотеки, как «*std::map*», «*std::vector*», «*std::dictionary*» и других. Более подробно про них можно почитать на ресурсе «*cpp-reference*» [2].

Отметим, что при определении конструктора опускается его возвращаемое значение, так как по определению ясно, что его никогда не будет, так как конструктор вызывается не мы, а компилятор, а цель конструктора, заполнить необходимые поля объекта значениями и выполнить предварительные подготовительные операции.

Создадим еще один **конструктор с параметрами**, как показано на рисунке 1.16.

```
class Product
{
    int count = 0;
    char* name = NULL;
public:
    Product() {
        setCount( 1 );
        setName("Default product");
    }

    Product( int count, char* name ) {
        setCount( count );
        setName( name );
    }

    bool isEmpty() {
        return this->count == 0;
    }
    // пропуск строчек для экономии места в демонстрации (но они должны быть)
};

int main() {

    Product staticMemoryProduct( 2 , "table N0001" );
    Product* dynamicMemoryProduct = new Product( 2 , "table N0002" );

    delete dynamicMemoryProduct;

    return 0;
}
```

Рисунок 1.16 – Конструктор с параметрами

Существуют различные типы конструкторов в зависимости от определенных формальных параметров – например, **конструктор копирования**, который в качестве формального параметра должен принимать экземпляр данного класса и производить полное копирование объекта в создаваемый объект (то есть скопировать один объект в другой новый объект). Для построения качественного конструктора копирования нам обходимо будет познакомиться с модификатором *const* и понятием ссылка. В соответствующих подразделах представим пример конструктора копирования для заданного класса.

Стоит отметить, что в данном случае у нас конструкторы публичные, то есть они находятся в блоке *public*. Но также для формирования некоторой сложной логики можно определять и приватные конструкторы – например, когда мы хотим создавать объект внутри класса и только там (например, шаблон «одиночка» – «singleton» [3]).

Также внимательный читатель мог заметить, что мы определили два конструктора, которые являются методами, которые, в свою очередь, являются функциями (функциями класса), а мы помним из курса С, что функции и переменные в своей области видимости должны иметь уникальные имена. Например, из курса С мы знаем, что эта проблема решается добавлением суффиксов к имени функции (то есть изменением-дополнением имени), добавлением директив препроцессора (исключаем ненужные функции) и т.д. Но мы уже знаем, что область видимости для всех методов внутри класса – это сам класс. Так как это получилось? Почему компилятор не ругается на нас за «multiple definition»? На этот вопрос ответим в подразделе про перегрузку функций, методов и операторов.

Если ни один конструктор не создан явно, будет создаваться конструктор по умолчанию компилятором. Если создан хоть бы один конструктор, то он будет использоваться

для создания объекта. То есть если будет определен конструктор с параметрами, то будет невозможно создать объект без указания параметров, так как конструктор по умолчанию не сформируется компилятором. В этом случае при необходимости нужно будет самостоятельно задать конструктор по умолчанию.

Мы решили проблему инициализации полей, но до сих пор не решили проблему с освобождением динамической памяти из-под указателя поля имени продукта: для выполнения этой операции можно использовать деструктор.

Деструктор — это метод, который вызывается (не нами), когда объект перестает существовать. Эти моменты наступают, когда:

- переступается блоковая скобка блока, внутри которого объявлен статический объект (и он соответственно уничтожается);
- освобождается память из-под динамического объекта оператором *delete*;
- завершается программа, что влечет уничтожение глобальных или статических объектов.

То есть это зависит от типа памяти и области жизни переменной, и соответствует правилам ее уничтожения. В деструкторе также можно освободить память, закрыть файлы и т.д. Мы настраиваем объект класса с помощью конструктора — создает и настраиваем ресурсы, а освобождаем все ресурсы в деструкторе, который вызывается в месте, когда нам объект уже больше не нужен, и он будет уничтожен.

Из этого следует, что деструктор всегда один, у него нет формальных параметров, он должен быть всегда *public*. Объявляется он, как и конструктор, с именем класса в качестве наименования, но впереди добавляется символ тильды «~». Определим деструктор, глобальную переменную типа класса и обозначим места вызова деструктора, что продемонстрировано на рисунке 1.17.

Стоит отметить, что мы добавили глобальный объект «*_globalProduct*». Местом вызова его конструктора является момент запуска приложения, а деструктора — момент времени завершения приложения. Также стоит отметить, что в коде на рисунке 1.17 мы явно видим, почему классы и структуры должны завершаться точкой с запятой — чтобы явно указать компилятору место завершения перечисления объявлений переменных данного типа-класса.

```
#include <string.h>

using namespace std;

class Product
{
    int count = 0;
    char* name = NULL;

public:
    Product() {
        setCount( 1 );
        setName( "Default product" );
    }

    Product(int count, char* name) {
        setCount(count);
        setName(name);
    }

    ~Product() {
        if(name != NULL) {
            delete[] name;
            name = NULL;
        }
    }

    bool isEmpty() {
```

```

        return this->count == 0;
    }
    // пропуск строчек для экономии места в демонстрации (но они должны быть)
} _globalProduct;

int main() {
    Product staticMemoryProduct( 2, "table N0001" );
    Product* dynamicMemoryProduct = new Product( 2, "table N0002" );

    _globalProduct.setName("table N003");

    delete dynamicMemoryProduct; // место вызова деструктора dynamicMemoryProduct

    return 0;
} // место вызова деструктора staticMemoryProduct

```

Рисунок 1.17 – Деструктор и его вызов

Ранее мы говорили о том, что для полей необходимо определять метод изменения их значения. Но если внимательно посмотреть на код, можно понять, что мы только изменяли поля «count» и «name», но совершенно не предусмотрели механизмы получения их значений из внешнего кода. Если начать говорить про методы изменения и получения значений полей, нужно понять идею модификаторов и селекторов.

Селекторы (get) и **модификаторы (set)** – методы класса, позволяющие получать значения полей класса и изменять поля класса. Это связано с инкапсуляцией – если мы закрыли поля, мы должны как-то «мочь» санкционированно получить их значения или их модифицировать во внешнем коде. Поэтому по общей договоренности создаются данные методы – они не являются общей конструкцией языка, а являются соглашением между программистами и входят в требования парадигмы (такая логика перенесена в Java, в C# для этого определена специальная конструкция). Причем средства рефакторинга в средах разработки предполагают механизмы по их быстрому созданию. Что же это дает: селекторы («select» – выбрать) обязаны предоставлять копию поля, чтобы приемник никак не смог его изменить через полученное значение. Также селекторы удобны для выдачи чуть-чуть подредактированного свойства. Селекторы обычно начинают со слова «get», а далее следует имя поля. Обычно у них нет формальных параметров.

Добавим селекторы для полей «count» и «name», как показано на рисунке 1.18.

```

#include <string.h>

using namespace std;

class Product
{
    int count = 0;
    char* name = NULL;

public:
    // пропуск строчек для экономии места в демонстрации (но они должны быть)
    int getCount() {
        return count;
    }

    char *getName() {
        if(name == NULL) { // если строка не заполнена
            return NULL; // возвращаем ноль
        }

        unsigned int sizeStr = strlen( name ); // получаем размер строки
        char* retName = new char[ sizeStr + 1 ]; // захватываем память под строку
        memset(retName, 0, sizeof(char)*(sizeStr + 1)); // обнуляем память
        strcpy(retName, name); // копируем строку

        return retName; // возвращаем копию строки
    }
}

```

```
};

int main() {

    Product staticMemoryProduct( 2, "table N0001" );
    Product* dynamicMemoryProduct = new Product( 3, "table N0002" );

    printf("There are %i of product \" %s \" \r\n",
        staticMemoryProduct.getCount(),
        staticMemoryProduct.getName() ); // ВЫВОД В КОНСОЛЬ

    printf("There are %i of product \" %s \" \r\n",
        dynamicMemoryProduct->getCount(),
        dynamicMemoryProduct->getName() ); // ВЫВОД В КОНСОЛЬ

    delete dynamicMemoryProduct;

    return 0;
}
```

Рисунок 1.18 – Добавление селекторов

Запустим приложение и получим вывод на рисунке 1.19.

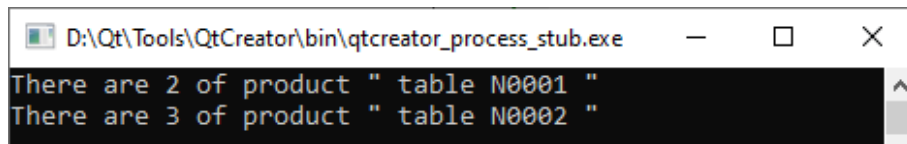


Рисунок 1.19 – Вывод приложения

Уже можно предположить проблему логики метода «*getName*», которая будет рассмотрена позже.

Модификаторы обеспечивают изменение свойства с валидацией (проверкой) входных данных, как, например, мы проверяли соответствие наименования продукта требованиям. Также изменение одного поля может повлиять на изменение другого поля – это должно быть обеспечено модификатором. Например, в классе, описывающий корзину интернет-магазина, при добавлении в список покупок одного продукта, кроме добавления в общий список продуктов, должен увеличиться общий счетчик покупок, потраченная сумма и другое. Модификаторы обычно начинают со слова «*set*», а далее следует имя поля. Обычно в качестве формальных параметров у них выступает новое значение поля.

Добавим поле забронированных продуктов, то есть тех продуктов, которые может выкупить только забронировавший их человек, поэтому они должны не отражаться как доступные к выдаче через склад. А также, обозначим поле, которое говорит о том, сколько фактически единиц товара есть с учетом зарезервированных продуктов на складе. То есть поле «*count*» показывает, сколько есть доступных к выдаче товаров, «*reserved*» – сколько забронировано, а метод «*getTotalCount*» покажет, сколько доступных к выдаче и забронированных есть на складе. Предположим, что бронь всегда выкупается. Также добавим методы вывода на консоль полей в стиле C и в стиле C++. Более подробно в следующем разделе. Финальный вариант к концу текущего подраздела представлен на рисунке 1.20.

```
#include <iostream>
#include <string.h>
using namespace std;

class Product
{
    int count = 0;
    char* name = NULL;
    int reserved = 0;

public:

    Product() {
```

```

        setCount( 1 );
        setName( "Default product" );
    }

    Product( int count, char* name ) {
        setCount( count );
        setName( name );
    }

    ~Product(){
        if( name != NULL ) {
            delete[] name;
            name = NULL;
        }
    }

    bool isEmpty() {
        return this->count == 0;
    }

    void setCount( int count ) {
        this->count = count;
    }

    void setName( char *value ) {

        if( value == NULL ) { // проверяем входной параметр на ноль

            throw "NULL указатель пришел в качестве аргумента"; // бросаем исключение
        }
        unsigned int sizeStr = strlen(value); // получаем размер строки

        if( !sizeStr ) { //если строка нулевого размера - пустая

            throw "Строка не может быть пустой"; // бросаем исключение
        }

        if( ( value[0] >= 'a' && value[0] <= 'z' )
            || ( value[0] >= 'A' && value[0] <= 'Z' ) ) {
            // если первый символ строки буква
            if( name != NULL ) {
                delete name;
                name = NULL;
            }

            Name = new char[ sizeStr + 1 ]; // захватываем память под строку
            memset( name , 0 , sizeof(char)* ( sizeStr + 1 ) ); // обнуляем память
            strcpy( name , value ); // копируем строку
        }
        else {
            throw "Параметр должен начинаться с буквы";
        }
    }

    int getCount() {
        return count;
    }

    char *getName() {
        if( name == NULL ) { // если строка не заполнена
            return NULL; // возвращаем ноль
        }

        unsigned int sizeStr = strlen( name ); // получаем размер строки
        char* retName = new char[sizeStr+1]; // захватываем память под строку
        memset(retName, 0, sizeof(char) * (sizeStr + 1) ); // обнуляем память
        strcpy(retName, name); // копируем строку

        return retName; // возвращаем строку
    }
}

```



```

int getReserved() {
    return reserved;
}

void setReserved( int value ) {
    if(value>0) { // если мы бронируем
        if(count - value <= 0) //сравниваем с доступными
        {
            throw "Нет столько на складе";
        }

        count -= value; // изымаем из доступных
        reserved += value; // добавляем в бронь
    }
    else { // если выдаем бронь
        if(value + reserved >= 0) { // если столько доступно в брони
            reserved += value; // выдаем из брони
        }
        else {
            throw "Нет столько в брони"; //хотим выдать больше
        }
    }
}

int getTotalCount() {
    return reserved + count; // общее число на складе
}

void showPrintf() {
    printf(
        "There are %i of product \" %s \" where %i available and %i reserved \r\n",
        // вывод в стиле СИ
        getTotalCount(),
        getName(),
        getCount(),
        getReserved()
    );
}

void showCout() {
    cout << "There are " // вывод с стиле C++
    << getTotalCount()
    << " of product \" "
    << getName() << " \", where "
    << getCount() <<" available and "
    << getReserved() << " reserved "<< endl;
}
};

int main() {
    Product staticMemoryProduct( 2, "table N0001" );
    Product* dynamicMemoryProduct = new Product( 3, "table N0002" );

    staticMemoryProduct.setReserved( 1 );
    dynamicMemoryProduct->setReserved( 2 );

    staticMemoryProduct.showPrintf();
    dynamicMemoryProduct->showCout();

    delete dynamicMemoryProduct;

    return 0;
}

```

Рисунок 1.20 – Промежуточный вариант кода класса

1.2 Вывод на консоль

В предыдущем разделе в коде на рисунке 1.20 было представлено два метода класса, содержащих вывод на консоль (рисунок 1.21). Стоит отметить, что внутри них доступ к полям «*name*», «*count*» и «*reserved*» можно было получить напрямую, а не через селекторы, так как к ним мы бы обращались в методе внутри класса.

```
void Product::showPrintf() {
    printf("There are %i of product \" %s \" where %i available and %i reserved \r\n",
        // вывод в стиле СИ
        getTotalCount(),
        getName(),
        getCount(),
        getReserved()
    );
}

void Product::showCout() {
    cout << "There are " // вывод с стиле С++
    << getTotalCount()
    << " of product \" "
    << getName() << " \", where "
    << getCount() << " available and "
    << getReserved() << " reserved "<< endl;
}
```

Рисунок 1.21 – Вывод на консоль

В чем принципиальные отличия двух способов вывода? Во-первых, «*printf*» («*print formatted*») – это функция в стиле С, а «*cout*» («*console output*») – это глобально определенный объект класса *ostream* – определение в С++-стиле. Во-вторых, функция «*printf*» определена в библиотеке *stdio* («*standart input-output*»), а «*cout*» объявлен в библиотеке *iostream* («*input-output stream*»). Функция «*printf*» имеет два формальных параметра: строку форматирования и ни одного или множество входных параметров разных типов (для работы с множеством параметров в своих функциях смотри библиотеку «*stdarg.h*» [4]). Если посмотреть на структуру вызова методов «*cout*», покажется, что творится какая-то «магия». Использование оператора «<<» регламентируется **перегрузкой операторов** – более подробно рассказано об этом в соответствующем подразделе. Давайте для наглядности заменим оператор «<<» на метод «*out*» с одним параметром любого типа. Тогда бы функция выглядела, так, как показано на рисунке 1.22.

```
void Product::showCout() {
    cout->out("There are ");
    cout->out(getTotalCount());
    cout->out(" of product \" " );
    cout->out(getName());
    cout->out(" \", where ");
    cout->out(getCount());
    cout->out(" available and ");
    cout->out(getCount());
    cout->out(getReserved());
    cout->out("\r\n");
}
```

Рисунок 1.22 – Замена оператора на вызов метода

При этом отметим про «любой тип»: это регламентируется **перегрузкой функций**, то есть можно создать методы под одним именем для разных сигнатур, что будет рассмотрено в последующем подразделе. Далее представим, что метод «*out*» возвращает указатель на текущий класс, то есть пусть метод «*out*» на псевдокоде выглядит так, как указано на рисунке 1.23.

```
ostream* ostream::out(...) {
    // логика вывода
    return this;
}
```

Рисунок 1.23 – Формирование метода «out» на псевдокоде

Тогда можно переформатировать предыдущий код следующим образом, как показано на рисунке 1.24, то есть каждый раз метод возвращает указатель на текущий объект, для которого будет вызываться снова этот метод.

```
void Product::showCout() {
    cout->out( "There are " )
    ->out( getTotalCount() )
    ->out( " of product \" " )
    ->out( getName() )
    ->out( " \", where " )
    ->out( getCount() )
    ->out( " available and " )
    ->out( getCount() )
    ->out( getReserved() )
    ->out( "\\r\\n" );
}
```

Рисунок 1.24 – Рекурсивный вызов метода «out» по возвращаемому указателю на объект

Теперь представим, что существует механизм замены вызова функции на применение оператора (а он существует), а также представим, что строку переноса каретки «\r\n» (перенос на новую строку) можно заменить каким-то объектом или константой «endl» (end line), как показано на рисунке 1.25.

Тогда мы получили исходный код как в примере на рисунке 1.21.

```
void Product::showCout() {
    cout << "There are "
    << getTotalCount()
    << " of product \" "
    << getName()
    << " \", where "
    << getCount()
    << " available and "
    << getCount()
    << getReserved()
    << endl;
}
```

Рисунок 1.25 – Замена метода «out» на оператор

Если в коде на рисунке 1.20 закомментировать строку с определением пространства имен: «using namespace std;», получим ошибку «use of undeclared identifier 'cout'» и «use of undeclared identifier 'endl'». Это означает, что глобальный объект класса «ostream» и объект «endl» определены в пространстве имен «std», и необходимо или определить его в начале заголовочного файла явно, или использовать оператор определения пространства имен для этих объектов, как показано на рисунке 1.26.

```
void Product::showCout() {
    std::cout << "There are " //вывод с стиле C++
    << getTotalCount()
    << " of product \" "
    << getName() << " \", where "
    << getCount() << " available and "
    << getReserved() << " reserved " << std::endl;
}
```

Рисунок 1.26 – Явное определение пространства имен объекта

Может возникнуть вопрос, какой из двух механизмов работы с консолью использовать. Ответ – любой, который удобен. В свою очередь, механизмы *iostream* могут быть

настроены с помощью своих параметров для конкретного вывода (например, определить сразу для вывода всех вещественных количество десятичных знаков). Также механизм перегрузки операторов может позволить определить правило ввода или вывода для собственного класса при работе с консолью (смотри перегрузку оператора «<<» для собственных классов на странице документации по адресу [5]). Также такой подход может быть использован для механизма сериализации – сохранения состояния экземпляра класса в файл и восстановление состояния из файла.

1.3 Работа с динамической памятью в C++ и конструкции инициализации

В самом начале раздела столкнулись с операторами языка C++ *new-delete*, отвечающими за захват и освобождение динамической памяти из *heap* (кучи). В отличие от C, в котором за захват динамической памяти отвечало семейство функций *malloc-free*, операторы *new-delete* являются, именно, конструкцией языка – операторами, а не функциями, и входят в его стандарт (спецификацию). Давайте вспомним, как работали функции *malloc-free*. Для захвата динамической памяти определяли указатель на один элемент (переменную) или на множество элементов (динамический массив) заданного типа данных. При этом, обычно использовали динамическую память для массивов, для которых заранее до запуска программы не известен их размер (число элементов).

Давайте пойдем последовательно. Например, нам нужен массив типа *int* из 10 элементов из кучи. Тогда применим функцию *malloc*, как показано на рисунке 1.27.

```
void *futureIntArray = malloc( sizeof( int ) * 10 );
```

Рисунок 1.27 – Захват динамической памяти в C

Функция *malloc* выдает нам кусок памяти из кучи определенного размера в байтах. Поэтому использован оператор *sizeof* для определения размера в байтах для конкретного типа, а его результат умножен на число элементов. Далее функция *malloc* предоставляет указатель *void* на кусок памяти. Почему «*void**»? Потому что функция *malloc* совершенно не знает, как будет использоваться эта память и элементы какого типа в ней будут. Это для функции не важно – она просто захватывает кусок памяти заданного размера. Потом уже мы сами приводим указатель *void** к указателю нужного типа (рисунок 1.28), чтобы потом пользоваться всей мощностью адресной арифметики и оператора разыменования и индексации.

```
int* intArray = (int*) malloc( sizeof(int) * 10 );
```

Рисунок 1.28 – Приведение указателя к нужному типу данных

Далее проводилась инициализация элементов значениями или обнулялись все элементы с помощью функции *memset*, так как по умолчанию то, что взято из кучи или сегмента стека (для локальных переменных) проинициализировано мусором. После того, как память была использована, ее необходимо было освободить: для этого использовалась функция *free* (рисунок 1.29).

```
free( intArray );
```

Рисунок 1.29 – Освобождение динамической памяти

Фактически в эту функцию поступает не *int**, а *void**, так как срабатывает неявное приведение типов, потому что для этой функции также не имеет значения, какого типа указатель. Ее задача освободить кусок памяти в байтах, которые до этого захватила функция *malloc*. Причем внутри этой логики для каждого выданного куска памяти хранится информация о том, сколько памяти в байтах было захвачено, поэтому в функции *free* не нужно указывать размер этого куска памяти.

В чем же особенность операторов *new-delete*? Оператор *new* выдает не *void**-указатель, а указатель заданного типа на кусок памяти, и самостоятельно рассчитывает его

размер в байтах, то есть разработчику необходимо указать количество элементов, как показано на рисунке 1.30.

```
int* intArray = new int[ 10 ];
```

Рисунок 1.30 – Захват динамической памяти оператором *new*

Динамический массив в этом случае также инициализирован мусором, так как он также взят из кучи. Также при использовании *delete* для массивов необходимо при их уничтожении указать, что это указатель на массив (рисунок 1.31).

```
delete[] intArray;
```

Рисунок 1.31 – Освобождение динамической памяти

Но в первом подразделе уже видели, что фактически массив объектов класса был заполнен значениями. Но сейчас мы используем массив элементов простого типа. Кто-то мог уже догадаться, что при использовании оператора *new* происходит вызов конструктора по умолчанию, а при использовании оператора *delete* происходит вызов деструктора. То есть в С-варианте функциям не было дела до того, для какого типа происходит захват памяти, поэтому при использовании классов нужно применять операторы *new-delete*, чтобы в программе было понятно, для какого класса вызывать конструктор и деструктор. Возникает закономерный вопрос, существует ли возможность сделать так, чтобы можно было бы провести инициализацию массива в месте захвата памяти. Да, возможно, с таким же подходом, как при статическом определении массивов, как показано на рисунке 1.32, с той разницей, что оператору *new* необходимо обязательно указывать количество элементов.

```
int intArrayStatic [] {1,2,3,4,5}; // инициализация статического массива  
int* intArrayDynamic = new int[ 5 ] {1,2,3,4,5}; // - динамического массива
```

Рисунок 1.32 – Инициализация элементов массива при объявлении

У любопытных может возникнуть вопрос, а собственно массив состоит из элементов, а класс из полей, нельзя ли проводить инициализацию некоторых полей таким же образом? Можно, варианты инициализации объекта представлены на рисунке 1.33.

```
Product staticMemoryProduct( 2, "table N0001" );  
Product* dynamicMemoryProduct = new Product( 3, "table N0002" );  
  
Product otherProductC11{ 1,"table N003" };  
Product otherProduct = { 1,"table N003" };  
  
Product* dynamicMemoryProductOther = new Product { 3, "table N0002" };
```

Рисунок 1.33 – Инициализация полей класса

Но при этом в данном случае есть особенность – все три варианта определяют вызов одного конструктора с двумя параметрами. Если мы захотим провести инициализацию только одного параметра, мы получим ошибку. Это связано с тем, что все поля в используемом классе определены с модификатором *private*. Поэтому компилятор попытается подобрать соответствующий конструктор при создании объекта. То есть для доступа к такой инициализации для *private*-полей, нужно определить конструктор. Если же определим класс только с публичными полями, то такая инициализация будет доступна без определения соответствующего конструктора, то есть компилятор сам сформирует такой конструктор. Такой подход будет полезен для задач, как в примере ниже – для определения класса двумерной координаты (рисунок 1.34): в такой задаче нет смысла «закрывать» поля класса, так как он будет использован как контейнер для данных.

```

class Point {
public:
    int x;
    int y;
};

int main() {
    Point point1 = { 7, 8 }; // список инициализаторов
    Point point2 { 9, 10 }; // uniform-инициализация (C++11)

    return 0;
}

```

Рисунок 1.34 – Агрегатная инициализация публичных полей класса

Стоит также отметить, что, если в чистом C для определения нулевого указателя рекомендовано было использовать константу *NULL*, то в C++ для этого определено ключевое слово *nullptr*. Рекомендуется использовать, именно, его.

1.4 Модификатор *const*

Мы говорили о том, что в ООП закладывалась некоторая идея, которую можно трансформировать в то, что манипуляция значениями полей объектов должна быть ограничена и сокрыта для того, чтобы обезопасить механизмы изменения состояний. Для этого же принципа, кроме модификаторов доступа, в C++ появился модификатор *const*, цель которого ограничить возможность изменения данных. Он может быть применен к переменным, формальным параметрам, полям, объектам и методам.

Начнем с переменных. Объявим переменную с модификатором *const*, попробуем ее изменить и получим ошибку, как показано на рисунке 1.35.

```

int main() {
    const int constIntValue = 0;

    constIntValue = 2; // error: cannot assign to variable 'constIntValue'
                       // with const-qualified type 'const int'

    return 0;
}

```

Рисунок 1.35 – Попытка изменения переменной, объявленной с модификатором *const*

То есть можно сделать выводы, что при обозначении переменной с модификатором *const*, ее изменение после инициализации будет запрещено.

Если мы захотим получить указатель на данную переменную (оператор «&» – получение адреса, где находится переменная), то компилятор сообщит нам, что данный указатель должен иметь тип *const int** (рисунок 1.36).

```

const int constIntValue = 0;

// error: cannot
// initialize a variable of type 'int *' with an rvalue of
// type 'const int *'
int *pconstIntValueFalse = &constIntValue;

const int *pconstIntValue = &constIntValue; // все хорошо

```

Рисунок 1.36 – Попытка получения неконстантного указателя на константную переменную

Рассмотрим использование модификатора *const* для массива или указателя, когда модификатор *const* следует перед именем типа или перед символом указателя (помним, что имя массива – это указатель на его первый элемент, а оператор «*» – разыменование указателя – получение значения по адресу). На рисунке 1.37 представлены варианты определения указателей на массив.

```

int simpleIntArray[ 2 ] = {0};

const int *      v1 = (const int*) &simpleIntArray; // нельзя менять значение
int const *      v2 = (const int*) &simpleIntArray; // равносильно const int*

int* const       v3 = (int* const) &simpleIntArray; //нельзя передвигать указатель

const int* const v4 = (const int* const)&simpleIntArray; // ничего нельзя

int a = v1[ 1 ]; // можно - const int *
*v1 = 1;         // error
v1[ 1 ] = 1;     // error
*(v1 + 1) = 1;   // error
v1++;           // можно

a = v2[ 1 ];     // можно - int const * (равносильно const int*)
*v2 = 1;         // error
v2[ 1 ] = 1;     // error
*(v2 + 1) = 1;   // error
v2++;           // можно

a = v3[ 1 ];     // можно - int* const
*v3 = 1;         // можно
v3[ 1 ] = 1;     // можно
*(v3 + 1) = 1;   // можно
v3++;           // error

a = v4[ 1 ];     // можно - const int* const
*v4 = 1;         // error
v4[ 1 ] = 1;     // error
*(v4 + 1) = 1;   // error
v4++;           // error

```

Рисунок 1.37 – Применение модификатора *const* для указателей

Можно сделать выводы, что установка модификатора *const* перед знаком «*» позволяет запретить изменять значения массива, но позволяет передвигать указатель и получать значения по адресу, а после – запрещает передвигать указатель, но позволяет получить значения элемента массива и их изменять. Общие правила применения модификатора к указателям указаны в таблице 1.1.

Таблица 1.1 – Правила применения модификатора *const* для указателей

Допустимость операции	<i>const type *</i>	<i>type * const</i>	<i>const type * const</i>
получить значение по индексу	да	да	да
передвинуть указатель	да	нет	нет
изменить значение элемента	нет	да	нет

Рассмотренные правила применения модификатора к локальным переменным и указателям используются и для переменных формальных параметров и типа возвращаемого значения функций.

Для методов модификатор *const* определяет их как константные и указывается после списка формальных параметров. **Константные методы** отличаются тем, что не могут менять значения полей своего класса и доступны для **константных объектов** – объектов, которые после вызова конструктора не меняют значения своих полей. Например, для описанного ниже класса на рисунке 1.38 образуется следующая ошибка, в которой константный метод пытается изменить значение поля.


```

class OneFieldAndFunc {
    int Field = 0;
public:
    void setField(const int value) {
        Field = value; // не-константный метод может менять поля
    }
    int getField() const {
        Field = 0; // error: cannot assign to non-static data member
                  // within const member function 'getField'
        return Field;
    }
};

```

Рисунок 1.38 – Попытка изменения поля в константном методе

Но при этом, стоит отметить, что с использованием механизма перегрузки функций никто не запрещает сделать два метода – константный и не-константный, первый из них будет использоваться для константных объектов, а второй для не-константных. Укажем два метода и создадим два объекта, определив то, в каком месте какой метод будет вызываться (рисунок 1.39).

```

class OneFieldAndFunc {
    int Field = 0;
public:
    void setField(const int value) {
        Field = value;
    }
    int getField() const { // константный метод
        return Field;
    }

    int getField() { // не-константный метод
        Field = 2;
        return Field;
    }
};

int main() {
    const OneFieldAndFunc constObj; // константный объект
    OneFieldAndFunc nonConstObj;    // не-константный объект
    constObj.getField();             // вызов константного метода
    nonConstObj.getField();          // вызов не-константного метода
    return 0;                        // место точки останова
}

```

Рисунок 1.39 – Создание и вызов константного и не-константного метода

Тогда в месте точки останова мы увидим следующие значения (рисунок 1.40).

Имя	Значение	Тип
▼ constObj	@0x61fecc	OneFieldAndFunc
Field	0	int
▼ nonConstObj	@0x61fec8	OneFieldAndFunc
Field	2	int

Рисунок 1.40 – Содержимое константного и не-константного объектов в точке останова

У первого объекта значение поля не изменилось, то есть для него была вызвана константная функция. А вот у второго объекта соответственно была вызвана не-константная функция.

Стоит отметить, что для ссылки, которая будет описана в следующем разделе, применение модификатора *const* запрещает изменять значение переменной, на которую ссылается ссылка. В том числе, константная ссылка на не-константный объект позволяет с ним взаимодействовать как с константным.

1.5 Ссылка

Указатель – это переменная, хранящая адрес. **Ссылкой** же можно называть новое имя переменной, или указателем, который всегда направлен на одну переменную и не требует разыменования для получения значения. Пример использования ссылки представлен на рисунке 1.41.

```
int main() {
    int intValue=0;

    int& rIntValue = intValue; // определили ссылку на переменную
    rIntValue = 1;             // в этом месте переменная intValue сменила значение на 1

    return 0;
}
```

Рисунок 1.41 – Формирование ссылки на локальную переменную

В рамках блока покажется, что ссылка не имеет смысла, так как зачем иметь два объекта, которые, по сути, определяют один объект внутри одного блока. Но механизм ссылок очень полезен для определения формальных параметров функций и методов.

В языке С мы использовали два механизма передачи параметров: **по значению** и **по адресу**. При передаче **по значению** в функцию поступает *копия значения* передаваемого параметра, поэтому изменение переменной формального параметра внутри функции не меняло фактический параметр. Для изменения переменной внутри функции параметр передавался в функцию по *указателю* и выполнялась **передача параметра по адресу**. Это ярко видно было при использовании функций *printf* и *scanf*. В *printf* передавались параметры по значению, так как эти значения передавались только для того, чтобы их вывести на консоль. В *scanf* же подавался адрес переменной (указатель), так как внутри *scanf* необходимо было заполнить переменную введенным с консоли значением. Такой же механизм используется активно в WinAPI: если в параметрах функции увидим передачу в нее не значения, а адреса, то можем предположить, что внутри функции будет происходить модификация этой переменной.

Использование ссылки же позволит передать исходную переменную без получения адреса (не как указатель), но с возможностью ее изменения внутри функции. То есть главные плюсы **передачи значения по ссылке** – нет копирования исходного объекта и его можно в функции изменять. Само появление ссылок связано с тем, что объекты сами по себе могут быть объемными в памяти и их копирование нежелательно, плюс сама процедура копирования и вставки потребует вызова конструкторов и деструкторов, которые необходимо будет продумывать. С учетом использования в полях класса файловых указателей, динамической памяти и других ресурсов, которые нужно очень аккуратно копировать и на которые нужно внимательно перенаправлять доступ, использование ссылок снижает большую долю проблем с разработчика. Рассмотрим пример передачи параметров разными способами, как показано на рисунке 1.42.

```
void N1(int a) { // передача по значению
    a += 1;      // получили копию, ее изменение не изменит фактический параметр
}
void N2(int* a) { // передача по указателю
    *a += 1;      // получили адрес, по нему поменяли значение
}
void N3(int& a) { // передача по ссылке
    a += 1;      // получили исходную переменную по ее новому имени - ссылке
}
int main() {
    int intValue = 0;
    int* ptr = &intValue;
    int& ref = intValue; // у intValue есть новое имя

    std::cout << intValue << " "; // intValue == 0
}
```

```

std::cout << *ptr << " "; // intValue == 0
std::cout << ref << "\r\n "; // intValue == 0

N1(intValue);
std::cout << intValue << " "; // intValue == 0

N2(&intValue);
std::cout << intValue << " "; // intValue == 1

N3(intValue);
std::cout << intValue << " "; // intValue == 2

return 0;
}

```

Рисунок 1.42 – Разные варианты передачи значений в функции

Вывод консоли представленного приложения указан на рисунке 1.43.

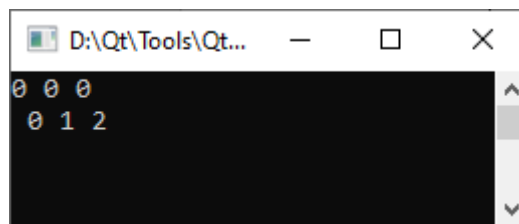


Рисунок 1.43 – Вывод консоли при выполнении кода на рисунке 1.42

1.6 Применение модификатора *const* и для повышения надежности класса и определения конструктора копирования

Появление модификатора *const* позволяет повысить надежность класса, так как его применение позволит контролировать возможность модификации входных параметров и возвращаемых значений методов, а также полей класса.

Например, все селекторы класса должны быть константными методами, чтобы не допускать внутри них модификацию значений полей, а также, чтобы их можно было вызвать для константных объектов. Входные параметры явно не должны быть изменены внутри класса, например, данные, передаваемые по указателям. Кроме этого, возвращаемые указатели и ссылки на поля, как правило, делают константными.

Произведем модификацию кода класса с учетом принятых соглашений.

В конструкторе (рисунок 1.44) с двумя параметрами сделаем первый параметр константной ссылкой, которая позволяет передавать переменные и константы без копирования значений, но без возможности модификации значения. Для второго параметра ограничим изменение элементов массива символов и движение указателя.

```

Product::Product( const int& count, const char* const name ) {
    setCount( count );
    setName( name );
}

```

Рисунок 1.44 – Изменение конструктора класса

Определим все селекторы и методы «*showPrintf*», «*showCout*», «*isEmpty*» и «*getTotalCount*» как константные методы, так как они позволяют получить константному объекту значения его состояния.

Если раньше мы боялись возвращать строку с именем, так как ее кто-то может изменить во внешнем коде, теперь мы можем ограничить изменения возвращаемого значения и спокойно ее возвращать (рисунок 1.45). Кроме этого, везде используем ключевое слово нулевого указателя *nullptr* вместо константы *NULL*.

```

const char * const Product::getName() const {
    if(name == nullptr) { // если строка не заполнена
        return nullptr;    // возвращаем нулевую ссылку
    }
    return name; // возвращаем строку по константному указателю
}

```

Рисунок 1.45 – Возвращение строки по константному указателю

С учетом рассмотренного в предыдущих подразделах можем начать рассматривать **конструктор копирования**, целью которого является создание нового объекта на основании другого путем копирования его полей (состояния). В этот конструктор в качестве параметра должен поступать объект такого же класса. Нам желательно передать этот объект в метод в исходном виде, а не его копию, что можно реализовать с помощью **ссылки**. При этом для защиты копируемого объекта от модификации внутри конструктора создаваемого объекта его нужно будет передать по константной ссылке. Тогда понимаем, что конструктор копирования – это конструктор класса, формальный параметр которого представляет константную ссылку на объект этого класса. Этот конструктор автоматически вызывается, когда объект создается путем инициализации его другим объектом (инициализация через оператор присвоения) или при передаче объекта по значению в функцию, где в формальный параметр происходит также копирование исходного объекта.

Определим конструктор копирования, запретив изменение объекта, который передаем по ссылке, как показано на рисунке 1.46. В данном случае в реализации конструктора копирования обеспечиваем не только копирование значения поля «*count*», но и захват памяти и копирование значения «*name*». Прямое копирование указателя в этом случае недопустимо. Стоит также отметить, что для передаваемого объекта доступ к его приватным полям присутствует, так как он бы происходил внутри класса, поэтому допустимо заменить вызов селекторов на прямой доступ к полям объекта.

```

Product::Product(const Product& obj) {
    setCount( obj.getCount() );
    setName( obj.getName() );
}

```

Рисунок 1.46 – Определение конструктора копирования

Поменяем входные параметры модификаторов на *const int&* и *const char * const*. Изменим логику метода «*setCount*» на логику, подобную «*setReserved*», чтобы учитывать особенности предметной области: появление нового товара или изъятие товара со склада увеличивает или уменьшает их количество (рисунок 1.47).

```

void Product::setCount( const int& count ) {
    if( count > 0 ) {
        this->count += count;
    }
    else {
        if( this->count + count >= 0 ) {
            this->count -= count;
        }
        else {
            throw "Столько товаров нет на складе"; //бросаем исключение
        }
    }
}

```

Рисунок 1.47 – Изменение бизнес-логики метода

Итоговый класс на конец данного подраздела представлен на рисунке 1.48.

```

#include <iostream>
#include <string.h>

```

```

using namespace std;

class Product
{
    int count = 0;
    char* name = nullptr;
    int reserved = 0;
public:
    Product() {
        setCount( 1 );
        setName( "Default product" );
    }

    Product( const int& count, const char* const name ) {
        setCount( count );
        setName( name );
    }

    Product( const Product& obj ) {
        setCount( obj.getCount() );
        setName( obj.getName() );
    }

    ~Product() {
        if(name != nullptr) {
            delete[] name;
            name = nullptr;
        }
    }

    bool isEmpty() const {
        return this->count == 0;
    }

    void setCount( const int& count ) {
        if( count > 0 ) {
            this->count += count;
        }
        else {
            if( this->count + count >= 0 ) {
                this->count += count;
            }
            else {
                throw "Столько товаров нет на складе"; //бросаем исключение
            }
        }
    }

    void setName( const char * const value ) {
        if( value == nullptr ) { // проверяем входной параметр на ноль
            throw "nullptr-указатель пришел в качестве аргумента";
            // бросаем исключение
        }
        unsigned int sizeStr = strlen( value ); // получаем размер строки

        if( !sizeStr ) { // если строка нулевого размера - пустая
            throw "Строка не может быть пустой";
        }

        if( ( value[0] >= 'a' && value[0] <= 'z' ) ||
            ( value[0] >= 'A' && value[0] <= 'Z' ) ) {
            // если первый символ строки буква
            if(name!= nullptr) {
                delete name;
                name = nullptr;
            }

            name = new char[ sizeStr + 1 ]; // захватываем память под строку
            memset( name, 0 , sizeof(char) * (sizeStr + 1) ); // обнуляем память
            strcpy( name , value); // копируем строку
        }
    }
}

```

```

        else {
            throw "Параметр должен начинаться с буквы";
        }
    }

void setReserved( const int& value ) {
    if( value > 0 ) { // если мы бронируем

        if( count - value <= 0 ) { // сравниваем с доступными
            throw "Нет столько на складе";
        }

        count -= value; // изымаем из доступных
        reserved += value; // добавляем в бронь
    }
    else { //если выдаем бронь
        if(value + reserved >= 0) { //если столько доступно в брони

            reserved += value; //выдаем из брони
        }
        else {
            throw "Нет столько в брони"; //хотим выдать больше
        }
    }
}

int getCount() const {
    return count;
}

const char * const getName() const {
    if( name == nullptr ) { // если строка не заполнена
        return nullptr; //возвращаем ноль
    }
    return name; //возвращаем строку
}

int getReserved() const {
    return reserved;
}

int getTotalCount() const {
    return reserved + count; // общее число на складе
}

void showPrintf() const {
    printf("There are %i of product \"%s\" where %i available and %i reserved \r\n",
        //вывод в стиле СИ
        getTotalCount(),
        getName(),
        getCount(),
        getReserved()
    );
}

void showCout() const {
    cout << "There are " //вывод в стиле C++
        << getTotalCount() <<
        << " of product \" "
        << getName() << " \", where "
        << getCount() << " available and "
        << getReserved() << " reserved " << endl;
}
};

```

```

int main() {
    Product staticMemoryProduct( 2 , "table N0001" );

    staticMemoryProduct.setReserved( 1 );
    staticMemoryProduct.setCount( -1 );

    staticMemoryProduct.showPrintf();

    return 0;
}

```

Рисунок 1.48 – Промежуточный вариант класса

1.7 Обработка исключений

В тексте класса вы видели оператор **throw**. Конструкция **throw-try-catch** используется для обработки ошибок – исключительных ситуаций или **исключений** (*Exception*). Вспомним, как проходила обработка ошибок в С. У функций, внутри которых могли появиться критические ошибки (файл не найден, неправильный формат введенных данных и т.д.), в качестве возвращаемого значения могли быть выданы коды ошибки.

Например, из справочной документации: «если функции *fopen()* удалось открыть указанный файл, возвращается указатель на *FILE*. Если же файл не может быть открыт, возвращается *NULL*» или «функция *scanf()* возвращает число, равное количеству полей, для которых успешно присвоены значения. При обнаружении ошибки до присвоения значения первого поля функция *scanf()* возвращает значение *EOF*».

Далее для того, чтобы обработать ошибку (вывести сообщения пользователю или самостоятельно разрешить проблему), необходимо было вызывать специализированные функции вывода типа ошибки или посмотреть код ошибки в документации и конструкцией *switch-case* обрабатывать варианты ошибок. При этом, если было необходимо определить функцию, которая могла вернуть ошибку, вложенный вызов функций мог привести к проблеме. Так как если функция «*func1*» вызывает «*func2*», а «*func2*» вызывает «*func3*» и ошибка возникает в функции «*func3*», то вывести код ошибки в месте вызова «*func1*», может быть затруднительно.

Если еще представить такой «снежный ком» при обработке ошибок в методах классов, можно предположить, что это может вызвать затруднение. В особенности у методов, у которых возвращаемых значений нет и вызываете их не вы – конструкторы (хотя использовать *throw* в конструкторе нежелательно, но не всегда). После того, как изучим механизм наследования, обработка ошибок С-стиле может вызвать у вас панику.

Поэтому в С++ был определен другой механизм обработки исключений. А именно, оператор **throw** – бросает исключение, блок **try** – ограничивает область кода, в которой может возникнуть исключение, блок **catch** – ловит исключение для его обработки. То есть условно, с помощью конструкции **throw** обозначается событие, что возникло исключение. В **try** оборачивается небезопасный участок кода. А с помощью **catch** можно поймать исключение на любом этапе вызова методов. Что же можно «бросать»? Бросать можно значения простых типов, строки, а так же объекты, наследуемые от класса *std::exception*. Для облегчения понимания процесса будем использовать строки в качестве исключений (то есть текст с описанием ошибки – исключительной ситуации).

Вспомним, в каких ситуациях в коде на рисунке 1.48 были определены исключительные ситуации:

- попытка выдачи или изъятие товаров больше, чем есть на складе, «*setCount*»;
- при изменении имени в модификаторе пришла null-строка;
- при изменении имени в модификаторе пришла пустая строка;
- при изменении имени в модификаторе пришла строка, которая начинается не с буквы;
- попытались забронировать товаров больше, чем есть на складе;
- попытались выдать из брони товаров больше, чем есть на брони.

В функции *main* обернем код работы с классами в блок `try-catch` и поменяем вызов «*setCount*» и «*setReserved*» местами (рисунок 1.49).

Как видно из кода на рисунке 1.49, в блоковые скобки блока **try** поместили работу с объектами класса, а в блок **catch** в качестве параметра попадает константная строка, которая будет выведена на консоль, то есть **обработка исключения** в этом случае заключается в выводе сообщения ошибки.

Отметим, что для отображения кириллического текста в консоли в Visual Studio можно в функции *main* добавить строчку «`setlocale(LC_ALL, "Russian");`», в Qt, который используется для демонстрации, в среде Window все несколько сложнее. Поэтому рекомендовано использовать латинский алфавит для отладочного кода, мы же будем смотреть значения переменных через отладчик.

```
int main() {
    try {
        Product staticMemoryProduct( 2 , "table N0001" );

        staticMemoryProduct.setCount( 1 );
        staticMemoryProduct.setReserved( 1 );

        staticMemoryProduct.showPrintf();
    }
    catch ( const char* exception ) {

        cout << exception; // точка останова
    }

    return 0;
}
```

Рисунок 1.49 – Генерация ошибки и вывод текста исключения

Начнем создавать исключительные ситуации и просматривать промежуточные значения переменной «*exception*», чтобы определить, попадаем ли мы в необходимую точку выполнения программы.

Подадим в конструктор класса `null`-строку (рисунок 1.50) и исследуем, что в строке с выставленной точкой останова (рисунок 1.51) исследуем значение «*exception*» в блоке **catch** (рисунок 1.52).

```
Product staticMemoryProduct( 2, nullptr );
```

Рисунок 1.50 – Передача некорректного параметра

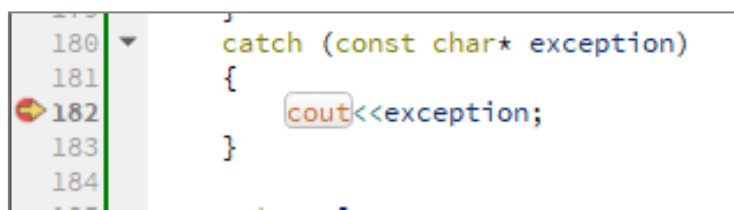


Рисунок 1.51 – Точка выполнения в блоке **catch**

Имя	Значение
> exception	"NULL указатель пришел в качестве аргумента"

Рисунок 1.52 – Текст исключения

Подадим пустую строку при создании объекта (рисунок 1.53).

```
Product staticMemoryProduct( 2, "" );
```

Имя	Значение	Тип
> exception	"Строка не может быть пустой"	char *

Рисунок 1.53 – Генерация ошибки и текст исключения

Подадим название, начинающееся с цифры (рисунок 1.54).

```
Product staticMemoryProduct( 2, "1table N0001" );
```

Имя	Значение	Тип
> exception	"Параметр должен начинаться с буквы"	char *

Рисунок 1.54 – Генерация ошибки и текст исключения

Введем корректное имя товара и попытаемся выдать три товара с пустого склада (рисунок 1.55).

```
Product staticMemoryProduct( 2, "table N0001" );  
staticMemoryProduct.setCount( -3 );
```

Имя	Значение	Тип
> exception	"Столько товаров нет на складе"	char *

Рисунок 1.55 – Генерация ошибки и текст исключения

Попытаемся забронировать три товара со склада при наличии только одного (рисунок 1.56).

```
Product staticMemoryProduct( 2, "table n100" );  
staticMemoryProduct.setCount( 1 );  
staticMemoryProduct.setReserved( 4 );
```

Имя	Значение	Тип
> exception	"Нет столько на складе"	char *

Рисунок 1.56 – Генерация ошибки и текст исключения

Попытаемся выдать из брони один товар при ее отсутствии (рисунок 1.57).

```
Product staticMemoryProduct( 2, "table n100" );  
staticMemoryProduct.setCount( 1 );  
staticMemoryProduct.setReserved( -1 );
```

Имя	Значение	Тип
> exception	"Нет столько в брони"	char *

Рисунок 1.57 – Генерация ошибки и текст исключения

Стоит отметить, что сообщения об исключениях, представленные в примере, являются примером сообщений, которые плохо уточняют ситуацию исключения. Например, полезным было бы вывести информацию о том, для какого, именно, товара (по имени) сформировалось исключение, при попытке какой операции возникла ошибка, уточнить, сколько есть на складе товаров, в том числе в брони, и сколько пытаются выдать или переместить из склада на бронь.

Изменим текст исключений (рисунок 1.58). С учетом того, что формирование строк с использованием функций *strcat* и *itoa* будет громоздким, используем класс «*std::string*» с конкатенацией строк через перегруженный оператор сложения и функцию «*to_string*».

Более подробно о том, как определяется перегрузка операторов в классе «*std::string*», будет рассмотрено в следующем подразделе.

```
...
throw string("setCount:: Столько товара по наименованию " )
        + name + " нет на складе для выдачи: есть "
        + to_string( this->count )
        + " шт, пытаемся выдать "
        + to_string( -count )+" шт.";
...
throw string( "setName:: Пришла нулевая строка в качестве параметра" );
...
throw string( "setName:: Пришла пустая строка в качестве параметра" );
...
throw string( "setName:: Параметр должен начинаться с буквы, пришла строка -
\"") + value + "\"";
...
throw string("setReserved:: Столько товара по наименованию ")
        + name + " нет на складе для брони: есть "
        + to_string( this->count )
        + " шт, пытаемся забронировать "
        + to_string( value ) + " шт.";
...
throw string( "setReserved:: Столько товара по наименованию " )
        + name + " нет на брони для выдачи: есть "
        + to_string(reserved)
        + " шт, пытаемся выдать "
        + to_string(-value)+" шт.";
...
catch (string exception)
...

```

Рисунок 1.58 – Модификация текста сообщений исключений

Новые тексты исключений представлены на рисунке 1.59.

```
setName:: Пришла нулевая строка в качестве параметра

setName:: Пришла пустая строка в качестве параметра

setName:: Параметр должен начинаться с буквы, пришла строка - "table n100"

setCount:: Столько товара по наименованию table n100 нет на складе для выдачи: есть
2 шт, пытаемся выдать 3 шт.

setReserved:: Столько товара по наименованию table n100 нет на складе для брони: есть 3
шт, пытаемся забронировать 4 шт.

setReserved:: Столько товара по наименованию table n100 нет на брони для выдачи: есть 0 шт,
пытаемся выдать 2 шт.
```

Рисунок 1.59 – Новый текст исключений

1.8 Перегрузка функций, методов и операторов

В прошлых подразделах очень часто упоминалась перегрузка функций, методов и операторов. Идея перегрузки функций заключается в том, что для одного имени можно определить множество функций, которые отличаются друг от друга списком формальных параметров (сигнатурой). Например, в коде на рисунке 1.48 с помощью этого механизма было сформировано три конструктора.

Приведем еще один пример перегрузки функций. Например, нам необходимо создать функцию, которая бы выводила на консоль переменные или константы следующих типов: *int*, *double*, *char** и *enum* (перечисление). Правила вывода: для *int* – как минимум 10 знаков, для *double* – 10 знаков до запятой и 3 знака после, для *enum* – вывод наименования его элемента в кавычках, для строки – вывод не более 10 символов в кавычках. Можно предположить, что нам нужно определить четыре функции с одним именем, но в качестве входного параметра должны поступать значения разных типов.

Тогда получим следующий код на рисунке 1.60.

```
enum EnumValue {EnumValue1 = 1, EnumValue2, EnumValue3}; //определение enum

void printToConsole() {                                // напечатать переноса каретки
    printf( "\r\n" );
}

void printToConsole( int number ) {                    // напечатать int на 10 знаков
    printf( " %10.i " , number );
}

void printToConsole( double number ) {                // напечатать double на 10+3 знака
    printf( " %13.3f " , number );
}

void printToConsole( const char* str ) {              // напечатать строки в кавычках на 10 знаков
    printf( " \"%0.10s\" " , str );
}

void printToConsole( EnumValue enumValue ) { // напечатать наименование enum в кавычках
    switch ( enumValue ) {
        case EnumValue1: printf( " \"EnumValue1\" " ); break;
        case EnumValue2: printf( " \"EnumValue2\" " ); break;
        case EnumValue3: printf( " \"EnumValue3\" " ); break;
    }
}
```

Рисунок 1.60 – Пример перегрузки функций

Как видно в коде на рисунке 1.60, были определены пять функций: одна без параметров для вывода переноса каретки и четыре штуки – для каждого типа данных. В функции *main* произведем вызов функций для переменных и констант различного типа (рисунок 1.61).

```
int main() {
    // переменные
    int intValue = 1000;
    double doubleValue = 34234.9304848392;
    const char* strValue = "Some string value\0";
    EnumValue enumValue = EnumValue1;

    // переменные приводимых типов
    float floatValue = 94849.44949f;
    short shortValue = 125;

    // определяет тип переменной, выбирает подходящую функцию
    printToConsole( intValue );
    printToConsole( doubleValue );
    printToConsole( enumValue );
    printToConsole( strValue );

    printToConsole();

    // определяет тип константы, выбирает подходящую функцию
    printToConsole( 54783 );
    printToConsole( 38498.0013324 );
    printToConsole( EnumValue2 );
    printToConsole( "const string" );
}
```

```

    printToConsole();

// определяет тип переменной, приводит ее к приводимому типу подходящей функции
    printToConsole( shortValue ); // short -> int
    printToConsole( floatValue ); // float -> double

    return 0;
}

```

Рисунок 1.61 – Вызов перегруженных функций

В результате чего мы получим следующий вывод, как показано на рисунке 1.62.

```

D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
1000 34234.930 EnumValue1 Some strin
54783 38498.001 EnumValue2 const stri
125 94849.453

```

Рисунок 1.62 – Вывод консоли при выполнении кода на рисунке 1.61

Из рассмотренного кода на рисунке 1.61 мы можем понять, что механизм перегрузки функций не только позволяет создавать функции для разных типов формальных параметров, но и определяет механизм, по которому для заданного типа фактических параметров происходит вызов функции, у которой формальные параметры совпадают по типу или же могут быть приводимы к их типу. Такое свойство определения конкретной функции в зависимости от типа обрабатываемых данных вносит свой вклад в понятие **полиморфизма времени компиляции**.

Стоит отметить, что для перегрузки функций необходимо, чтобы одна функция отличалась от другой количеством формальных параметров или сочетанием их типов (сигнатурой), но при этом функции с одинаковыми наименованиями и списком формальных параметров, но разными возвращаемыми значениями перегрузке не подлежат. Поэтому при следующем варианте реализации функций возникнет ошибка (рисунок 1.63).

```

void printToConsole() {
    printf( "\r\n" );
}
//error: functions that differ only in their return type cannot be overloaded
int printToConsole() {
    printf( "\r\n" );
}

```

Рисунок 1.63 – Невозможная перегрузка функции

Но в коде на рисунке 1.48 перегружали конструктор, который не является функцией, а является методом. Поэтому можно заключить, что существует перегрузка методов.

Для проверки обернем предыдущий код в класс «*OurConsoleOutput*», а в качестве возвращаемого значения для каждой функции обозначим ссылку на данный класс «*OurConsoleOutput&*» и вернем разыменованный указатель на текущий экземпляр класса *this* (для того, чтобы передать по ссылке, а не по указателю). Изменим способ вызова функций, как в подразделе «Вывод на консоль». Как будет происходить данный механизм: разыменовав *this*, мы получим ссылку на значение текущего экземпляра класса, далее компилятор определит тип возвращаемого значения как ссылку, и вернет в место вызова данную ссылку. Такой способ позволяет цепочкой «нанизывать» вызовы функций для одного класса. Полученный код представлен на рисунке 1.64.

```

using namespace std;

enum EnumValue{ EnumValue1 = 1,EnumValue2, EnumValue3};

class OurConsoleOutput {
public:

```

```

OurConsoleOutput& printToConsole() {
    printf( "\\r\\n" );
    return *this;
}

OurConsoleOutput& printToConsole( int number ) {
    printf( " %10.i " , number );
    return *this;
}

OurConsoleOutput& printToConsole( double number ) {
    printf( " %10.3f " , number );
    return *this;
}

OurConsoleOutput& printToConsole( const char* str ) {
    printf( " \\\"%0.10s\\\" " , str );
    return *this;
}

OurConsoleOutput& printToConsole( EnumValue enumValue ) {
    switch ( enumValue ) {
        case EnumValue1: printf( " \\\"EnumValue1\\\" " ); break;
        case EnumValue2: printf( " \\\"EnumValue2\\\" " ); break;
        case EnumValue3: printf( " \\\"EnumValue3\\\" " ); break;
    }
    return *this;
}
} outConsoleOutput;

int main() {
    int intValue = 1000;
    double doubleValue = 34234.9304848392;
    const char* strValue = "Some string value\\0";
    EnumValue enumValue = EnumValue1;

    float floatValue = 94849.44949f;
    short shortValue = 125;

    outConsoleOutput.printToConsole( intValue )
        .printToConsole( doubleValue )
        .printToConsole( enumValue )
        .printToConsole( strValue )
        .printToConsole()
        .printToConsole( 54783 )
        .printToConsole( 38498.0013324 )
        .printToConsole( EnumValue2 )
        .printToConsole( "const string" )
        .printToConsole()
        .printToConsole( shortValue )
        .printToConsole( floatValue );

    return 0;
}

```

Рисунок 1.64 – Перегрузка методов класса

Как видно из предыдущего кода, место цепочки вызовов «*printToConsole*» очень загружено из-за длинного имени методов, точки и круглых скобок. Можно предположить, что неплохо было бы заменить вызов функции на вызов оператора. В этом нам поможет механизм перегрузки операторов. Давайте предположим, что хотим перегрузить оператор битового сдвига «<<» для вывода на консоль одного значения. Приведем пример для перегрузки оператора для типа *int* и сравним объявление метода и объявление перегрузки оператора (рисунок 1.65).

```

OurConsoleOutput& printToConsole( int number ) {
    printf( " %10.i " , number );
    return *this;
}

OurConsoleOutput& operator <<( int number ) {
    printToConsole( number );
    return *this;
}

```

Рисунок 1.65 – Определение одной реализации для перегрузки метода и оператора

Можно заметить, что разница только в том, что наименование метода изменилось на сочетание ключевого слова **operator** и знака оператора. Но фактически механизм объявления оператора значительно отличается от объявления функции или метода тем, что имеет ряд ограничений, например:

- запрещено перегружать операторы «.» и «.*»;
- число параметров регламентируется тем, унарный или бинарный это оператор, например, «+» – бинарный оператор, поэтому левый операнд – это текущий экземпляр класса, а правый операнд – это входной параметр, всего может быть два параметра, а унарный оператор «++» может быть применен только с пустым списком параметров, то есть только один параметр – это текущий объект;
- нельзя забывать о приоритетах операторов и нужно продумывать их логику с их учетом;
- нельзя использовать параметры по умолчанию (смотри соответствующий подраздел).

Перегрузим оператор в соответствии с логикой четырех определенных ранее методов (рисунок 1.66).

```

...
OurConsoleOutput& operator << (int number ) {
    printToConsole( number );
    return *this;
}
OurConsoleOutput& operator << ( double number ) {
    printToConsole( number );
    return *this;
}
OurConsoleOutput& operator << ( const char* str ) {
    printToConsole( str );
    return *this;
}
OurConsoleOutput& operator << ( EnumValue enumValue ) {
    printToConsole( enumValue );
    return *this;
}

...
const char* endln = "\r\n";
outConsoleOutput
    << intValue
    << doubleValue
    << enumValue
    << strValue
    << EnumValue2
    << endln
    << 54783
    << 38498.0013324
    << "const string"
    << endln
    << shortValue
    << floatValue;
...

```

Рисунок 1.66 – Перегрузка оператора и его применение

Каким образом, можно внести логику перегрузки операторов в код на рисунке 1.48. Выдача и закупка товара производится через функцию «*setCount*». Более удобным способом было бы добавлять закупку через оператор «+=», а выдавать товар через оператор «-=». Также интересно было бы производить расчет полного количества единиц товаров на складе через оператор «+=». Для этого внутри класса перегрузим операторы «+=» и «-=», как показано на рисунке 1.67.

```
void operator += ( int count ) {
    setCount(count);
}
void operator -= ( int count ) {
    setCount(-count);
}
```

Рисунок 1.67 – Перегрузка операторов внутри класса

Стоит отметить, что практически все операторы возвращают значение, поэтому для того, чтобы не исказить смысл оператора, желательно в перегрузке продумать возвращаемое значение. Тогда для кода на рисунке 1.67 необходимо выполнить модификацию на рисунке 1.68. Мы же оставим первый вариант реализации, хотя второй более предпочтительный.

```
Product& operator += ( int count ) {
    setCount(count);
    return *this;
}
Product& operator -= ( int count ) {
    setCount(-count);
    return *this;
}
```

Рисунок 1.68 – Перегрузка операторов внутри класса

Как было сказано ранее, было бы хорошо производить расчет общего числа всех товаров на складе. Без перегрузки операторов данный код выглядел бы следующим образом, как показано на рисунке 1.69, а нам бы хотелось иметь следующую запись, как показано на рисунке 1.70.

```
Product* products = new Product[ countProduct ] {
    Product( 2 , "table n100"),
    Product( 10, "table n101"),
    Product( 5 , "table n102"),
    Product( 1 , "table n103"),
    Product( 15, "table n104")
};

int allAvailableCount = 0 ;

for( int i = 0 ; i < countProduct ; ++i ) {
    allAvailableCount += products[ i ].getTotalCount();
}

delete[] products;
```

Рисунок 1.69 – Расчет общего количества единиц товара на складе

```
for(int i = 0 ; i < countProduct ; ++i ) {
    allAvailableCount += products[ i ];
}
```

Рисунок 1.70 – Расчет общего количества единиц товара на складе с использованием перегруженного оператора

Если «+=» – используемый перегружаемый оператор, то левым операндом у него является переменная «*allAvailableCount*», которая определена типом *int*, а правый операнд

– элемент массива «*products*» класса «*Product*». Но ранее видели в коде на рисунке 1.67, что левым операндом является объект класса «*Product*».

Для решения таких ситуаций необходимо определить перегрузку оператора вне класса, то есть перегрузить оператор не в смысле **метода** класса, а в смысле **функции**. Когда оператор перегружается вне класса, для бинарных операторов должно быть определено ровно два формальных параметра, для унарных – один. А теперь продумаем перегрузку: нам необходимо прибавить к первому входному параметру второй входной параметр, а далее полученное значение присвоить к первому входному параметру и вернуть из функции данное значение (рисунок 1.71). Кажется, все просто, но в таких случаях важно продумать, как передавать параметры и как возвращать значение – по значению или по ссылке. По логике видно, что первый входной параметр точно должен быть передан по ссылке, так как он изменяется. Второй входной параметр можно также передать по ссылке, чтобы сэкономить память, но при этом внутри перегрузки его нужно запретить изменять. Возвратить необходимо первый параметр нужно по ссылке для соблюдения смысла оператора.

```
тип_возвращаемого_значения operator += ( входной_параметр1 , входной_параметр2 )
```

Рисунок 1.71 – Шаблон перегрузки оператора вне класса

Тогда код перегрузки оператора и его использование в функции `main` выглядит следующим образом, как показано на рисунке 1.72.

```
...

void operator += (int count) {
    setCount( count );
}
void operator -= ( int count ) {
    setCount( -count );
}
}; // конец класса

int& operator += ( int& leftArg , const Product& product ) {
    return leftArg += product.getTotalCount();
}

int main() {
    try {
        const int countProduct = 5;
        Product* products = new Product[ countProduct ] {
            Product( 2 , "table n100" ),
            Product( 10, "table n101" ),
            Product( 5 , "table n102" ),
            Product( 1 , "table n103" ),
            Product( 15, "table n104" )
        };

        cout << "First day\r\n";

        int allAvailableCount = 0;

        for( int i = 0; i < countProduct ; ++i ) {
            allAvailableCount += products[ i ];
            products[ i ].showCout();
        }
        cout << "All: " << allAvailableCount << "\r\n";

        products[2] -= 3; // изменения за день
        products[4] += 3;
        products[0] += 2;

        cout << "Second day\r\n";

        allAvailableCount = 0;
```

```

    for(int i = 0; i < countProduct ; ++i ) {
        allAvailableCount += products[i];
        products[ i ].showCout();
    }

    cout << "All: " << allAvailableCount << "\r\n";

    delete [] products;
}
catch (string exception) {
    cout << exception;
}
return 0;
}

```

Рисунок 1.72 – Промежуточный вариант класса к концу подраздела

Вывод на консоль предложенного варианта реализации приложения представлен на рисунке 1.73.

```

First day
There are 2 of product " table n100 ", where 2 available and 0 reserved
There are 10 of product " table n101 ", where 10 available and 0 reserved
There are 5 of product " table n102 ", where 5 available and 0 reserved
There are 1 of product " table n103 ", where 1 available and 0 reserved
There are 15 of product " table n104 ", where 15 available and 0 reserved
All: 33
Second day
There are 4 of product " table n100 ", where 4 available and 0 reserved
There are 10 of product " table n101 ", where 10 available and 0 reserved
There are 2 of product " table n102 ", where 2 available and 0 reserved
There are 1 of product " table n103 ", where 1 available and 0 reserved
There are 18 of product " table n104 ", where 18 available and 0 reserved
All: 35

```

Рисунок 1.73 – Вывод на консоль при работе приложения

Стоит учитывать, что перегрузка оператора вне класса делает его реализацию внешним кодом для класса, поэтому внутри реализации внешнего оператора приватные поля класса будут недоступными. Поэтому если в этом случае необходимо иметь доступ к приватным полям класса, необходимо либо создать методы доступа к этим полям, либо сделать оператор дружественным: объявить его внутри класса с модификатором *friend*. Стоит отметить, что последний способ не определит перегрузку оператора внутри класса, а только будет способом пометки его дружественности и не будет влиять на его объявление.

1.9 Параметры по умолчанию

Параметры по умолчанию – это механизм определения функции или метода, позволяющий при вызове функции или метода пропускать те фактические параметры, которые в большинстве случаев будут иметь одно и то же значение. Например, пусть для класса «*Product*» мы знаем, что при поступлении нового товара на склад он поставляется обычно в единичном объеме («на пробу»). Поэтому формально было бы удобно определить еще один конструктор не с двумя параметрами, а с одним – именем товара.

Тогда нам бы требовалось создать конструктор с одним параметром следующего вида, как показано на рисунке 1.74.

```

Product::Product(const char* const name) {
    setCount(1);
    setName(name);
}

```

Рисунок 1.74 – Конструктор с одним параметром, где одно поле класса заполняется константным значением

Но, с другой стороны, можно воспользоваться параметрами по умолчанию: в конструкторе «*Product*» с двумя параметрами поменять эти параметры местами в сигнатуре и сделать параметр «*count*» по умолчанию равным единице, как показано на рисунке 1.75. Смена параметров местами связана с тем, что необязательные параметры должны быть всегда в конце списка параметров, а обязательные параметры всегда должны предшествовать параметрам по умолчанию.

```
Product(const char* const name, const int& count = 1) {
    setCount(count);
    setName(name);
}
```

Рисунок 1.75 – Изменение конструктора с использованием параметров по умолчанию

Тогда можно было бы объявление массива объектов в функции *main* определить следующим образом, как показано на рисунке 1.76.

```
Product* products = new Product[ countProduct ]{
    Product( "table n100" ), // тут компилятор вторым параметром подставит единицу
    Product( "table n101" ,10),
    Product( "table n102" ,5 ),
    Product( "table n103" ), // тут компилятор вторым параметром подставит единицу
    Product( "table n104", 15)
};
```

Рисунок 1.76 – Создание объектов одним конструктором с параметром по умолчанию

Можно было бы пойти дальше и определить в конструкторе два параметра по умолчанию, как показано на рисунке 1.77. Но тогда компилятор выдаст нам ошибку «call to constructor of 'Product' is ambiguous», то есть у компилятора есть два варианта к вызову конструктора без параметров – конструктор по умолчанию или конструктор с двумя параметрами по умолчанию. Поэтому в таких случаях необходимо оставить только один из них.

```
Product( const char* const name = "New Product" , const int& count = 1 ) {
    setCount( count );
    setName( name );
}
```

Рисунок 1.77 – Конструктор со всеми необязательными полями, определение которого приводит к неоднозначности

В качестве другого примера применения параметров по умолчанию рассмотрим функцию библиотеки компьютерного зрения OpenCV (рисунок 1.78), используемую для распознавания узлов калибровочной сетки на изображении.

```
bool cv::findCirclesGrid (
    InputArray          image,
    Size                patternSize,
    OutputArray         centers,
    // параметр по умолчанию типа перечисления, приводимый к простому типу
    int                 flags = CALIB_CB_SYMMETRIC_GRID,
    // параметр по умолчанию – ссылка на объект класса
    const Ptr<FeatureDetector>& blobDetector = SimpleBlobDetector::create()
);
```

Рисунок 1.78 – Функция поиска сетки по ее круговым узлам в библиотеке OpenCV

Тогда можно понять, что параметрами по умолчанию могут быть формальные параметры простого или сложного типа, находящиеся в конце списка параметров. Невозможно пропустить параметр, который находится в середине или начале списка. Также параметры по умолчанию доступны при определении как методов, так и функций.

1.10 Наследование

В первом подразделе рассматривался пример класса, описывающий окно пользовательского интерфейса. Еще тогда у читателя мог возникнуть вопрос, что хотя все окна имеют что-то общее в визуальном представлении и в поведении, но они во многом разные: разное визуальное представление и поведение. Поэтому можно заключить, что был бы полезным механизм, позволяющий вносить в класс логику, которая совпадает с подобными ему сущностями, а далее расширять этот класс индивидуальными особенностями поведения и дополнительными данными. Для этого существует механизм наследования.

Наследование – это метод определения класса, при котором, кроме объявляемых методов и полей, происходит копирование методов и полей из другого класса. Класс, у которого «наследуют» методы и поля, называется родителем или **базовым классом**, а класс реципиент – потомком или ребенком, или **производным классом**. Именно, здесь и проявляются функциональные возможности применения модификатора **protected**. Если поля или методы базового класса помечены модификатором **protected**, то данные поля не будут «видимы» для внешнего кода, но класс, который наследуется от базового класса, будет иметь к ним доступ внутри себя (внутри класса). Этот механизм также инкапсулирует логику внутри класса и его потомков при наследовании. Но стоит отметить, что члены базового класса с модификатором доступа **private** будут добавляться в производный класс, но доступ к ним внутри производного класса не будет предоставлен.

Как было сказано ранее, можем при определении класса успешно наследовать поля и методы базового класса. Все бы было просто, если бы не **модификаторы наследования**: класс потомок может наследоваться от базового класса по разным правилам, которые определяют то, с каким модификатором члены базового класса войдут в класс потока. То есть, кроме того, что до этого нужно было следить за тем, как будут видны поля и методы класса во внешнем коде, то теперь необходимо продумывать, как в классе-потомке будут видны поля базового класса.

Начнем с того, как определить наследование. При определении производного класса после его имени указывается базовый класс с модификатором наследования, как показано на рисунке 1.79, где «*BaseClass*» – базовый класс, от которого будет проводиться наследование, «*ChildClass*» – производный класс. Если модификатор наследования не будет указан, будет применено *private*-наследование.

```
class ChildClass : <модификатор наследования> BaseClass {
```

Рисунок 1.79 – Определение наследования класса

Если при наследовании применен модификатор наследования **public**:

- *public*-члены базового класса доступны как *public*-члены производного класса;
- *protected*-члены базового класса доступны как *protected*-члены производного;
- *private*-члены недоступны производному классу.

Если при наследовании применен модификатор наследования **protected**:

- *public*- и *protected*-члены доступны как *protected*-члены производного класса;
- *private*-члены базового класса – недоступны производному классу.

Если при наследовании применен модификатор наследования **private**:

- *public*- и *protected*- члены доступны как *private*-члены производного класса.
- *private*-члены базового класса – недоступны производному классу.

То есть при всех типах наследования *private*-члены базового класса недоступны, при *public*-наследовании остальные члены сохраняют свой модификатор, при *private*- и *protected*-наследовании остальные члены класса приобретают соответствующий типу наследования модификатор. Если оформить эти правила в сводную таблицу, то получим следующие правила наследования методов и полей, указанные в **таблице 1.2**.

Таблица 1.2 – Правила установки модификатора доступа для наследуемых полей и методов при применении указанных модификаторов наследования

Модификатор наследования:	Модификатор доступа членов базового класса		
	<i>public</i>	<i>protected</i>	<i>private</i>
<i>public</i>	<i>public</i>	<i>protected</i>	-
<i>protected</i>	<i>protected</i>	<i>protected</i>	-
<i>private</i>	<i>private</i>	<i>private</i>	-

Стоит учесть, что такие методы как конструктор и деструктор не наследуются, но участвуют в создании и уничтожении объектов производного класса. Если не задать конструктор базового класса явно в конструкторе производного класса, будет вызван конструктор по умолчанию базового класса. Иерархия вызовов конструктора при создании объекта производного класса следующая: сначала вызывается конструктор базового класса, а далее конструктор потомка. При уничтожении наоборот – сначала деструктор потомка, потом деструктор родителя.

Определение конструктора базового класса в конструкторе потомка выглядит так, как показано на рисунке 1.80. При этом, повторим, что если не будет явно указываться конструктор базового класса, то будет вызван конструктор базового класса по умолчанию (рисунок 1.81).

```
ChildClass (формальные_параметры) : BaseClass (фактические_параметры) {
    // тело конструктора потомка
}
```

Рисунок 1.80 – Явное указание конструктора базового класса в конструкторе потомка

```
ChildClass (параметры) { // определен конструктор по умолчанию базового класса
    // тело конструктора потомка
}
```

Рисунок 1.81 – Неявное указание конструктора по умолчанию базового класса

Стоит отметить, что в фактические параметры конструктора базового класса можно передать значения формальных параметров конструктора потомка или константы. Также стоит отметить, что в теле конструктора потомка не должна происходить инициализация полей базового класса, которая нарушает логику его конструктора.

С конструктором все в принципе просто, но что делать, если в базовом и производном классах есть методы или поля, которые имеют одно имя?

Все зависит от того, что необходимо сделать:

- если необходимо вызвать метод объекта класса потомка, ничего не надо делать – он и так вызовется по умолчанию внутри и вне производного класса;
- если необходимо вызвать метод родителя, когда в потомке есть такой же метод, нужно явно указать через пространство имен –«*BaseClass::Method*», что необходимо вызвать метод именно базового класса для объекта потомка;
- если привести объект класса-потомка к базовому классу, будут вызываться методы базового класса.

Для класса продукта, определенного в предыдущих подразделах, представим, что для планирования доставки товаров нам необходимо знать следующие параметры: для стройматериалов знать вес в килограммах, а для мебели габариты ШВД (ширина, высота, длина). Таким образом, можно определить два производных класса: «*Furniture*» и «*ConstructionMaterials*». Можно предположить, что логика расчета доступных товаров на складе и на бронировании должна быть наследуема из базового класса «*Product*», а производные классы должны быть дополнены требуемыми полями и методами.

Определим перечисление *enum* для задания типа продуктов и поле, которое будет указывать, какого типа данный продукт, как показано на рисунке 1.82.

```

public:
enum ProductTypes{                               // типы продуктов
    FurnitureType,                               // тип мебель
    ConstructionMaterialsType, // тип строительный материал
    OtherType} ;                                // другой тип
protected:
    ProductTypes ProductType = OtherType; // тип продукта объекта

```

Рисунок 1.82 – Определение перечисления для типов товаров

Для того, чтобы можно было использовать перечисление во внешнем коде, оно объявлено в блоке *public*, а для того, чтобы производные классы имели доступ к полю типа продукта «*ProductType*», он определен в блоке *protected*-модификатора. В базовом классе определим селектор для получения значения типа продукта (рисунок 1.83).

```

ProductTypes Product :: getProductType() const {
    return ProductType;
}

```

Рисунок 1.83 – Метод получения значения типа товара

Определим класс «*Furniture*», публично наследуясь от «*Product*», чтобы потом можно было выполнять приведение ссылок и указателей. Определим параметры габаритов ШВД и константу для конвертирования типов в *private*-блок объявлений (рисунок 1.84).

```

class Furniture : public Product
{
    unsigned int width = 0; // ширина
    unsigned int length = 0; // длина
    unsigned int height = 0; // высота

    static const double SM_IN_METER = 100.0; // константа для перевода см в м
};

```

Рисунок 1.84 – Определение наследования и полей производного класса «*Furniture*»

Определим конструктор, который кроме параметров «*count*» и «*name*» принимает параметры габаритов, явно укажем конструктор базового класса с передачей двух первых параметров, а полю «*ProductType*» присвоим стандартное значение (рисунок 1.85). Помним, что конструктор должен быть определен в *public*-блоке.

```

Furniture::Furniture(const char* const name,
                    const int& count,
                    const unsigned int& width,
                    const unsigned int& length,
                    const unsigned int& height) : Product( name , count ) {
    this->width = width;
    this->length = length;
    this->height = height;
    ProductType = FurnitureType; // установка protected-поля базового класса
}

```

Рисунок 1.85 – Определение конструктора производного класса

Определим селекторы и модификаторы для всех полей производного класса. Кроме этого, добавим метод вычисления объема по значениям полей (рисунок 1.86).

```

double Furniture::getVolume() const {
    return ( width / SM_IN_METER )
        * ( height / SM_IN_METER )
        * ( length / SM_IN_METER );
}

```

Рисунок 1.86 – Метод вычисления объема

Определим новый метод вывода на консоль, который содержит, как вывод значений полей базового класса, так и производного (рисунок 1.87).

```

void Furniture::showCout() const {
    cout << "There are "
        << getTotalCount()
        << " of product \" "
        << getName() << " \", where "
        << getCount() << " available and "
        << getReserved() << " reserved .";

    cout << "Product type is Furniture "
        << width << "cm X "
        << length << "cm X "
        << height << "cm . Volume is "
        << getVolume() << " m3 \r\n";
}

```

Рисунок 1.87 – Метод вывода на консоль с учетом новых данных

Стоит отметить, что логика вывода значений полей базового класса практически не изменилась: только исключен перенос на новую строку. Поэтому, если бы нас устраивал вывод с переносом каретки, то можно было бы использовать метод базового класса : сначала применить стандартную логику вывода, а потом явно реализовать расширяющую логику (рисунок 1.88).

```

void Furniture::showCout() const {

    Product:: showCout(); // вызов метода базового класса - стандартный вывод

    cout << "Product type is Furniture " // расширение вывода
        << width << "cm X "
        << length << "cm X "
        << height << "cm . Volume is "
        << getVolume() << " m3 \r\n";
}

```

Рисунок 1.88 – Использование метода базового класса для работы с его полями

По той же логике создадим класс «*ConstructionMaterials*» (рисунок 1.89).

```

class ConstructionMaterials : public Product
{
    unsigned int weight = 0; // вес
public:
    ConstructionMaterials(const char* const name,
                          const int& count,
                          const unsigned int& weight): Product( name , count ) {
        this->weight = weight;
        ProductType = ConstructionMaterialsType;
    }

    unsigned int getWeight() const {
        return weight;
    }
    void setWeight(const unsigned int& value) {
        weight = value;
    }

    void showCout() const {
        cout << "There are "
            << getTotalCount()
            << " of product \" "
            << getName() << " \", where "
            << getCount() << " available and "
            << getReserved() << " reserved .";

        cout << "Product type is Furniture ConstructionMaterials "
            << weight << "kg. \r\n";
    }
};

```

Рисунок 1.89 – Реализация второго производного класса

Стоит отметить, что если внутри производных классов попробовать получить доступ к полям «*count*», «*name*» и «*reserved*» базового класса, то ничего не выйдет, так как компилятор выдаст ошибку «'name' is a *private* member of 'Product'». Если доступ к этим полям необходим внутри производных классов, их определение нужно перенести в *protected*-блок.

Рассмотрим и проанализируем различные сочетания вызова методов разных классов, как показано на рисунке 1.90.

```
int main() {
    try {
        // определение объектов производных от Product классов
        Furniture          furniture( "table n101", 10,30,40,50);
        ConstructionMaterials cMaterial( "paint n001", 2, 10    );

        // определение объекта базового класса Product
        Product product("box n100");

        cout << "Furniture output " << endl;

        // используем public-оператор Product в объекте производного класса
        Furniture += 1;

        cout << "As Child" << endl;

        // вызов метода производного класса
        furniture.showCout();

        cout << "As Base" << endl;

        // вызов метода базового класса из-под объекта производного
        furniture.Product::showCout();

        cout << "Product output " << endl;

        product -= 1;
        product.showCout();

        cout << "ConstructionMaterials output " << endl;

        // используем public-оператор Product в объекте производного класса
        cMaterial += 10;

        cout << "As Child" << endl;

        // вызов метода производного класса
        cMaterial.showCout();

        cout << "As Base" << endl;

        // вызов метода базового класса из-под объекта производного
        cMaterial.Product::showCout();

        const int productCount = 3;

        Product* products[ productCount ];

        // приведение указателя на объект производного класса
        // к указателю базового класса
        products[0] = &furniture;
        products[1] = &cMaterial;

        products[2] = &product;

        int allAvailableCount = 0;

        cout << "All output " << endl;

        for(int i = 0 ; i < productCount ; ++i ) {
```

```

        allAvailableCount += *products[ i ];

        // автоматический вызов метода базового класса для всех объектов
        cout << "As Base" << endl;
        products[i]->showCout();

        // если указатель на объект производного класса
        if( products[i]->getProductType() == Product::FurnitureType ) {
            cout << "As Child" << endl;
            // обратное приведение и вызов метода производного класса
            ( (Furniture*)products[i] )->showCout();
        }

        if( products[i]->getProductType()==
            Product::ConstructionMaterialsType ) {
            cout << "As Child" << endl;
            ( (ConstructionMaterials*)products[i] )->showCout();
        }
    }

    cout << "All: " << allAvailableCount<<"\r\n";
}
catch (string exception) {
    cout << exception;
}

return 0;
}

```

Рисунок 1.90 – Применение сформированных классов

Вывод сформированного приложения представлен на рисунке 1.91.

```

D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
Furniture output
As Child
There are 11 of product " table n101 ", where 11 available and 0 reserved .Product type is Furniture 30cm X 40cm X 50cm .Volume is 0.06 m3
As Base
There are 11 of product " table n101 ", where 11 available and 0 reserved .
Product output
There are 0 of product " box n100 ", where 0 available and 0 reserved .
ConstructionMaterials output
As Child
There are 12 of product " paint n001 ", where 12 available and 0 reserved .Product type is Furniture ConstructionMaterials 10kg.
As Base
There are 12 of product " paint n001 ", where 12 available and 0 reserved .
All outPut
As Base
There are 11 of product " table n101 ", where 11 available and 0 reserved .
As Child
There are 11 of product " table n101 ", where 11 available and 0 reserved .Product type is Furniture 30cm X 40cm X 50cm .Volume is 0.06 m3
As Base
There are 12 of product " paint n001 ", where 12 available and 0 reserved .
As Child
There are 12 of product " paint n001 ", where 12 available and 0 reserved .Product type is Furniture ConstructionMaterials 10kg.
As Base
There are 0 of product " box n100 ", where 0 available and 0 reserved .
All: 23

```

Рисунок 1.91 – Вывод приложения

Как видно в предыдущем коде на рисунке 1.90, можно сделать заключение, что то, что произойдет вызов метода производного или метода базового класса, имеющих одно имя и сигнатуру, зависит от текущего типа объекта, ссылки или указателя. При этом, при текущей реализации, если **получить ссылку или указатель типа базового класса** на объект производного класса, вызов такого метода приведет к **выполнению метода базового класса, а не производного**. Но такой прием абстракции (работы с набором объектов производных классов через указатель базового класса) очень упрощает взаимодействия с разными объектами через один интерфейс.

Поэтому возникает вопрос, что же необходимо сделать, чтобы в следующем коде на рисунке 1.92 происходил вызов производного класса, а не базового?


```

Product* products[ productCount ];
products[ 0 ] = &furniture;
products[ 1 ] = &cMaterial;
products[ 2 ] = &product;

for(int i = 0; i < productCount ; ++i ) {
    products[ i ]->showCout();
}

```

Рисунок 1.92 – Пример единообразного обращения к разным производным классам через приведение к указателю базового класса

Ответ на данный вопрос ждет нас в следующем подразделе.

1.11 Интерфейсы, абстрактные классы, виртуальные методы

В предыдущем подразделе были рассмотрены примеры наследования нескольких классов от другого класса. Можно заметить, что не всегда легко отследить особенности связи одного родителя и одного потомка. Можно предположить, что все будет более запутано, если у ребенка есть и «папа», и «мама», и «бабушка», и «дедушка» и т.д. Задумываться об этом не надо, потому что множественное наследование от полноценных классов **запрещено**. Также как и оператор *goto* в языке C, с той разницей, что если оператор *goto* рекомендовано не использовать в большинстве случаев, то множественное наследование чаще запрещено на уровне синтаксиса языка и правил компилятора (но не в языке C++). Фактически, множественное наследование от полноценных классов считается плохим тоном, и делает приложение запутанным и трудно поддерживаемым.

Но что же делать, когда нужно обеспечить реализацию множественного наследования? Например, существует объект, который должен содержать логику двух разных сущностей?

Кроме этого, с предыдущего подраздела осталась проблема, когда мы хотим взаимодействовать с объектами производных классов по ссылке или указателю базового класса, но, чтобы эти объекты вели себя по логике производных классов. Например, в коде информационной системы есть классы различных документов, с которыми необходимо взаимодействовать одинаково: каждый документ можно вывести на печать или прорисовать в пользовательском интерфейсе для их обзора, но структура разных типов документов будет разной. То есть у каждого документа должны быть методы «*print*» и «*show*», но при этом тело данных методов будет реализовывать печать и отображение структуры конкретного типа документа. Уже встречали пример такого обобщенного доступа к объектам разных классов через базовый класс в коде на рисунке 1.92, но пока не рассмотрели то, как добиться в этом случае вызова нужных методов, а не метода базового класса.

Таким образом, необходимо рассмотреть такой способ организации наследования классов, чтобы:

- общаться с более сложными объектами через более упрощенный базовый класс;
- обеспечить вызов методов производных классов при взаимодействии с их объектами через указатель или ссылку базового класса;
- обеспечить возможность правильного наследования от нескольких классов.

Для последовательного разбора рассмотрим сначала, что такое интерфейс. **Интерфейс** – это класс, содержащий в себе *только константы и пустые методы*, называемые чистыми функциями. «Пустые» означает, что **чистые функции** содержат в себе только объявление методов без их реализации. Пример объявления чистой функции представлен на рисунке 1.93, где для метода применяется модификатор *virtual*, а тело метода заменяется конструкцией приравнивания к нулю.

```

virtual void Method() = 0;

```

Рисунок 1.93 – Определение чистой функции

Можно заключить, что интерфейс – это некоторый «каркас» класса, которому должен соответствовать класс, реализующий данный интерфейс. Это можно представить в виде механизма заказа требуемого класса. Например, Вася хочет, чтобы Маша написала класс, который бы решал СЛАУ методом Гаусса. Тогда Вася скажет, что ему необходимо, чтобы класс Маши содержал три метода: ввода данных (матрицы), проверки матрицы на соответствие условиям сходимости и получения вектора решений. Вася через этот интерфейс будет взаимодействовать с объектами класса Маши. Тогда Вася может передать Маше интерфейс, представленный на рисунке 1.94.

```
class ILAESSGaussMethod
{
public:
    enum ErrorType {
        InvalidSize,
        MatrixRowsLinearDependence,
        ZerosOnDiagonal,
        OK
    };

    virtual bool InputMatrix( const double** matrix ) = 0;
    virtual ErrorType Check() = 0;
    virtual double* Solve() = 0;
    virtual ~ILAESSGaussMethod() {} ;
};
```

Рисунок 1.94 – Пример определения интерфейса в C++:
определение абстрактного класса, содержащего только чистые методы и константы

Маша, в свою очередь, должна будет реализовать данный интерфейс, например, как показано на рисунке 1.95.

```
class LAESSGaussMethod: public ILAESSGaussMethod
{
    double** matrix = nullptr; // конкретные данные
public:
    bool InputMatrix(const double** matrix) override {
        // код копирования матрицы в нужном формате – конкретная реализация
    }

    ErrorType Check() override {
        // код проверки матрицы на соответствие условиям сходимости
        return OK;
    }

    double* Solve() override {
        double* result = nullptr;
        // код решения СЛАУ методом Гаусса – конкретная реализация
        return result;
    }

    ~LAESSGaussMethod() override {
        if( matrix != nullptr ) {
            delete matrix;
        }
    }
};
```

Рисунок 1.95 – Пример реализации интерфейса в C++ –
наследование от абстрактного класса

Вася же может смело вносить в свой код использование класса Маши через определенный интерфейс, как показано на рисунке 1.96, где Васе будет не важно, какие данные, логику методов и дополнительные методы внесла Маша – ему важен лишь набор методов и констант, которые ему доступны через интерфейс.

```

    LAESGaussMethod solving; // объект класса Маши
    ILAESGaussMethod& isolving = solving; // ссылка интерфейса на объект Маши

    double *result = nullptr;

    isolving.InputMatrix( nullptr ); // используются только доступные методы

    if( isolving.Check() == ILAESGaussMethod::OK ) {
        result = isolving.Solve();
    }

```

Рисунок 1.96 – Доступ к объекту класса через интерфейс

Стоит заметить, что при определении «*ILAESGaussMethod*» было использовано ключевое слово *class*, а не *interface*, как это могло быть в языках Java или C#. Это связано с тем, что в C++ нет интерфейсов в синтаксисе языка, но есть возможность определять абстрактные классы, которые содержат только чистые функции и константы, а класс «*ILAESGaussMethod*» является примером такого класса. Почему же стоит рассматривать эту взаимосвязь понятий «абстрактный класс и интерфейс» в рамках изучения C++? Потому что интерфейсы можно будет встретить в других языках, поэтому нужно привыкать организовывать взаимодействие объектов по правильным практикам через интерфейсы, а также в нотации UML, часто используемой для объектно-ориентированного моделирования, кроме понятия *class*, ведено понятие *interface*, в качестве которого для C++ необходимо будет применять абстрактные классы.

Стоит отметить, что для интерфейсов принято говорить, что классы их реализуют, а не наследуют. Это связано с тем, что в данном случае класс для реализации интерфейса должен реализовать все методы, которые содержатся в интерфейсе. С такими вариантами связей между классами также встретитесь в UML-нотации, так как там между базовыми и производными классами можно организовать связи наследования и реализации.

Если реализация методов интерфейса в классе заменяет чистые функции с пустой реализацией, то этот же механизм применяется для решения проблемы кода на рисунке 1.92. В этом случае нужно переопределить (*override*) метод «*showout*» в производных классах, указав в базовом классе, что он является виртуальным (*virtual*).

Виртуальная функция (метод) – это метод, который объявляется с ключевым словом *virtual* в базовом классе и может быть переопределен в одном или в нескольких производных классах. Виртуальные функции являются особыми методами, потому что при вызове метода объекта производного класса компилятор определяет *во время выполнения программы*, что необходимо вызвать метод производного класса, основываясь на исходном типе объекта (производном), а не на типе указателя или ссылки (базовом). Для разных объектов, классы которых переопределили виртуальную функцию, вызываются свои методы, даже если с ними взаимодействуют через ссылку или указатель базового класса. При этом виртуальная функция в классе может содержать базовую реализацию, может ее не содержать (чистая виртуальная функция), а при наследовании этот метод быть перезаписан (**переопределен**) методом производного класса для его объектов.

Классы, содержащие виртуальные методы, имеют полную реализацию методов, но некоторые из них являются виртуальными функциями, которые могли быть уже переопределены или будут переопределены в производных классах.

Абстрактный класс – это класс, в котором имеется хотя бы одна чистая виртуальная функция. Тогда **интерфейс** В C++ – это абстрактный класс, который содержит только константы и чистые функции.

Абстрактные классы используются для наследования группой потомков, к которым потом необходимо обращаться как к объектам одного типа – например, абстрактный класс документ, который содержит поле имени, виртуальные методы печати, принятия и отклонения. А его потомки – например, заявление, объяснительная и другие – содержат индивидуальные особенности логики и данных. Можем создать по экземпляру производных классов и поместить их в один массив указателей типа базового класса, а далее печатать

тать, принимать и отклонять документы, как обобщенный абстрактный документ (через базовый класс) без детализации их типа. При этом вызовутся методы, определенные для каждого из производных классов – это и называется **полиморфизмом времени выполнения**, когда объект одного типа ведет себя как объект другого в соответствии с его исходным классом, и реализует обобщение всех типов документов в базовый класс.

Также стоит отметить, что по абстрактному классу нельзя создавать объекты, так как все или часть его методов не реализованы (они чистые). Но можно создавать ссылки или указатели абстрактного класса на объекты классов, которые реализуют данный интерфейс – наследуются от данного абстрактного класса.

Сначала внесем изменения в классы, чтобы добиться необходимой логики в коде на рисунке 1.92, а, именно, применим для методов «*showCout*» в каждом классе модификатор *virtual*, то есть сделаем их виртуальными, а в производных классах еще и добавим модификатор *override* (рисунки 1.97 – 1.99).

```
virtual void showCout() const {
    cout << "There are "
        << getTotalCount()
        << " of product \" "
        << getName() << " \", where "
        << getCount() << " available and "
        << getReserved() << " reserved ." << "\r\n";
}
```

Рисунок 1.97 – Определение виртуального метода в базовом классе

```
virtual void showCout() const override {
    cout << "There are "
        << getTotalCount()
        << " of product \" "
        << getName() << " \", where "
        << getCount() << " available and "
        << getReserved() << " reserved .";

    cout << "Product type is Furniture "
        << width << "cm X "
        << length << "cm X "
        << height << "cm . Volume is "
        << getVolume() << " m3 \r\n";
}
```

Рисунок 1.98 – Переопределение метода в производном классе «*Furniture*»

```
virtual void showCout() const override {
    cout << "There are "
        << getTotalCount()
        << " of product \" "
        << getName() << " \", where "
        << getCount() << " available and "
        << getReserved() << " reserved .";

    cout << "Product type is Furniture ConstructionMaterials "
        << weight << "kg. \r\n";
}
```

Рисунок 1.99 – Переопределение метода в производном классе «*ConstructionMaterials*»

Стоит отметить, что внесение модификаторов *virtual* и *override* в производных классах необязательно – требуется только пометить метод в базовом классе как виртуальный, что автоматически позволит его переопределить в производных классах (*override*) и сделать его также виртуальным в них. Но явное задание модификатора *virtual* позволит визуально отметить те методы, которые являются виртуальными, а модификатор *override* позволяет явно указать компилятору, что метод используется для переопределения. Если при явно указанном модификаторе *override* компилятор не найдет подходящий по сигнатуре виртуальный метод базового класса на переопределение, то он выдаст ошибку ком-

пияции, без *override* он создаст перегрузку метода (*overload*). Поэтому явное внесение модификаторов позволяет выполнить профилактику ошибок и повышать читаемость кода.

После внесенных изменений в классах рассмотрим их применение в коде на рисунке 1.100. В полученном выводе приложения увидим (рисунок 1.101), что теперь при приведении объекта производного класса к указателю базового класса будут вызываться методы «*showCout*» не базового, а производных классов.

```
int main() {
    try {
        Furniture furniture( "table n101", 10 , 30 , 40 , 50 );
        ConstructionMaterials cMaterial( "paint n001" , 2 , 10 );
        Product product( "box n100" );

        furniture += 1;
        product -= 1;
        cMaterial += 10;

        const int productCount = 3;

        Product* products[ productCount ];

        products[ 0 ] = &furniture;
        products[ 1 ] = &cMaterial;
        products[ 2 ] = &product;

        int allAvailableCount = 0;

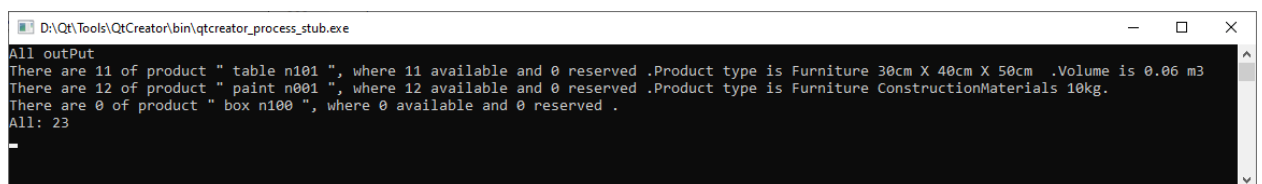
        cout << "All outPut " << endl;
        for( int i = 0 ; i < productCount ; ++i ) {
            allAvailableCount += *products[ i ];
            products[ i ]->showCout();
        }

        cout << "All: " << allAvailableCount << "\r\n";

    }
    catch ( string exception ) {
        cout<<exception;
    }

    return 0;
}
```

Рисунок 1.100 – Исследование поведения реализованной логики



```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
All outPut
There are 11 of product " table n101 ", where 11 available and 0 reserved .Product type is Furniture 30cm X 40cm X 50cm .Volume is 0.06 m3
There are 12 of product " paint n001 ", where 12 available and 0 reserved .Product type is Furniture ConstructionMaterials 10kg.
There are 0 of product " box n100 ", where 0 available and 0 reserved .
All: 23
```

Рисунок 1.101 – Вывод приложения на консоль

Пойдем далее и попробуем обеспечить слабую сопряженность. **Слабая сопряженность** – это слабое взаимодействие между объектами, то есть объекты знают друг о друге минимально возможное количество информации. Класс должен быть независим от других настолько, насколько это возможно, и другие должны о нем знать, насколько мало, сколько допустимо.

Поэтому в такие моменты, когда необходимо обеспечить слабую сопряженность кода, для различных объектов можно сформировать интерфейс, который ограничит доступный функционал только до тех функций, которые действительно нужны для взаимодействия. Предположим, что с нашим классом «*Product*» взаимодействует пользовательский интерфейс, которому нужно вывести данные об товаре, а на клавиатурный ввод вы-

носится функционал внесения или удаления товара со склада. Тогда нам необходимо определить два интерфейса: «*IShow*» и «*IChange*» (рисунок 1.102).

```
class IShowing { // интерфейс для вывода на консоль
public:
    virtual void showCout() const = 0;
    virtual ~IShowing(){};
};

class IChanging { // интерфейс для изменения количества товара на складе
public:
    virtual void operator += (int count) = 0;
    virtual void operator -= (int count) = 0;
    virtual ~IChanging(){};
};
```

Рисунок 1.102 – Определение интерфейсов

Добавим модификаторы *override* и *virtual* к соответствующему методу («*showCout*») и операторам («*+=*» и «*-=*»). Добавим наследование с модификатором *public* от обоих интерфейсов в базовый класс «*Product*», то есть класс «*Product*» будет наследоваться от двух интерфейсов. Также в функции «*main*» создадим цикл, который ожидает управляющие символы с консоли, которые позволят управлять объектами, и организуем доступ к объектам через интерфейс (рисунок 1.103).

```
Furniture furniture( "table n101",10 ,30,40,50 );
ConstructionMaterials cMaterial( "paint n001",2,10 );
Product product( "box n100" );

const int productCount = 3;

IShowing* ishowing[ productCount ]; // массив интерфейсов для вывода
ishowing[ 0 ] = &furniture;
ishowing[ 1 ] = &cMaterial;
ishowing[ 2 ] = &product;

IChanging* iedit[ productCount ]; // массив интерфейсов для изменений
iedit[ 0 ] = &furniture;
iedit[ 1 ] = &cMaterial;
iedit[ 2 ] = &product;

for(int i = 0 ; i < productCount ; ++i ) {
    cout << i << " ";
    ishowing[ i ]->showCout(); // обходим объекты и выводим через интерфейс
}

int currentIndex = 0;
bool isExit = false;

while(!isExit) {
    char currentChar = getchar(); // ожидаем символ команды
    cout << currentChar;

    switch (currentChar) {

        case '+': {
            iedit[ currentIndex ]->operator += ( 1 ); // оператор через интерфейс
        }
        break;

        case '-': {
            iedit[ currentIndex ]->operator -= ( 1 ); // оператор через интерфейс
        }
        break;

        case '0': currentIndex = 0; // вывод через интерфейс
            ishowing[ currentIndex ]->showCout();
            break;
```

```

        case '1':
            currentIndex = 1; ishowing[ currentIndex ]->showCout();
            break;

        case '2': currentIndex = 2; ishowing[ currentIndex ]->showCout();
            break;

        case 's': ishowing[ currentIndex ]->showCout(); break;

        case 'a':
            for(int i=0; i<productCount; ++i) {
                cout << i << "    ";
                ishowing[ i ]->showCout();
            }
            break;
        case 'e': isExit = true; break;
    }
}

```

Рисунок 1.103 – Изменение логики пользовательского интерфейса

Не смотря на описанные выше преимущества использования интерфейсов, в прошлом коде увидели страшную конструкцию «`iedit[currentIndex]->operator==(1);`». Ее использование является вынужденным в данном примере. Целью демонстрации было показать, проблемы, связанные «абстрактным» представлением класса «*IChange*», операторами и разыменованием указателя типа интерфейса, в том числе с обоснованием возможных ошибок. Мы работаем с указателем («*Product**»), проблема в том, что оператор у нас перегружен для класса «*Product*» (а не для указателя «*Product**»), соответственно, поэтому мы его в краткой форме для указателя применить не можем. Если же мы разыменуем указатель из-под «*IChange**», то мы не получим объект класса «*Product*», а если все-таки попробуем так сделать и после этого применить операторы «`+=`» или «`-=`», ничем хорошим это не закончится (так как у приведенного к «*IChange*» объекта абстрактного класса не будет реализации операторов). Поэтому по логике необходимо привести «*IChange**» к «*Product**», далее разыменить его, а потом применить оператор, но этот вариант без промежуточных переменных выглядит также не очень красиво: «`((Product*)iedit[currentIndex])-=1;`». Стоит предусматривать особенности приведения указателей к абстрактным классам и использования операторов из-под указателей на абстрактные классы. В случае же ссылки такой подход будет полностью оправдан.

Почему же еще необходимо организовывать взаимодействие объектов классов через интерфейсы? Это позволяет делать объекты взаимозаменяемыми. Если класс реализует интерфейс, он может быть передан другому коду, который взаимодействует по этому интерфейсу, в этом случае код не привязан к конкретному классу.

Стоит заострить внимание, что в данном случае для класса «*Product*» было выполнено наследование от двух абстрактных классов (интерфейсов), то есть было выполнено множественное наследование не от полных, а от абстрактных классов, что является более хорошим тоном по сравнению с классическим множественным наследованием.

Также стоит отметить, что в *switch* предложенного кода можно было сгруппировать несколько веток следующим образом, как показано на рисунке 1.104.

```

        case '0':
        case '1':
        case '2':
            currentIndex = currentChar - '0';
            ishowing[ currentIndex ]->showCout(); // вывод через интерфейс
            break;

```

Рисунок 1.104 – Объединение веток с одной логикой

Так как коды цифр идут по порядку в таблице кодировки, то можно получить требуемый индекс путем вычитания кода нулевого символа из кода текущего символа.

1.12 Шаблоны

При объявлении класса мы надеемся на универсальность его использования в данной предметной области. Поэтому иногда возникает необходимость указать некоторый универсальный тип данных полей класса или используемых параметров методов. Например, пусть известно, что некоторые товары класса «*ConstructionMaterials*», могут иметь вес в целых числах, а могут быть товары по 0,5 кг. В качестве решения можно было бы смириться с растратой памяти при целых значениях и изменить тип *int* на *double* или *float*, но есть и другой путь: воспользоваться шаблоном. **Шаблоны** позволяют заранее при определении не указывать тип данных поля, переменной или возвращаемого значения, а подставить его при объявлении объекта или при вызове метода или функции.

Так как структура шаблонов очень проста при inline-определении методов (по сравнению с реализацией методов вне шаблонного класса), ознакомьтесь самостоятельно с ней в коде шаблонного класса «*ConstructionMaterials*» на рисунке 1.105.

```
template<typename WeightType> // метка шаблона с универсальным типом
class ConstructionMaterials : public Product
{
    WeightType weight = 0;    // использование типа шаблона для поля
public:
    ConstructionMaterials(const char* const name,
                           const int& count,
                           const WeightType& weight): Product( name , count ) {
        this->weight = weight;
        ProductType = ConstructionMaterialsType;
    }

    WeightType getWeight() const {    // тип шаблона для возвращаемого параметра
        return weight;
    }
    void setWeight(const WeightType& value) { // тип шаблона для входной ссылки
        weight = value;
    }

    virtual void showCout() const override {
        cout << "There are " << // вывод с стиле C++
            << getTotalCount()
            << " of product \" "
            << getName() << " \", where "
            << getCount() << " available and "
            << getReserved() << " reserved .";

        cout << "Product type is ConstructionMaterials " << weight << "kg. \r\n";
    }
};

...
// конкретный тип подставляется при создании объекта,
// т.е. WeightType заменяется на double или int
ConstructionMaterials<double> cMaterial1("paint n001",2,10.1);
ConstructionMaterials<int> cMaterial2("paint n002",2,1);
...
```

Рисунок 1.105 – Определение шаблонного класса

Вывод полученного приложения представлен на рисунке 1.106.

Как было указано в коде на рисунке 1.105, в месте объявления класса указывается метка шаблона с ключевым словом *template*. Внутри треугольных скобок указывается ключевое слово *typename* после которого следует наименование для «подставляемого» типа, называемого **параметром шаблона**.


```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
0 There are 10 of product " table n101 ", where 10 available and 0 reserved .Product type is Furniture 30cm X 40cm X 50cm .Volume is 0.06 m3
1 There are 2 of product " paint n001 ", where 2 available and 0 reserved .Product type is ConstructionMaterials 10.1kg.
2 There are 2 of product " paint n002 ", where 2 available and 0 reserved .Product type is ConstructionMaterials 1kg.
s
sThere are 10 of product " table n101 ", where 10 available and 0 reserved .Product type is Furniture 30cm X 40cm X 50cm .Volume is 0.06 m3
1
1There are 2 of product " paint n001 ", where 2 available and 0 reserved .Product type is ConstructionMaterials 10.1kg.
s
sThere are 2 of product " paint n001 ", where 2 available and 0 reserved .Product type is ConstructionMaterials 10.1kg.
2
2There are 2 of product " paint n002 ", where 2 available and 0 reserved .Product type is ConstructionMaterials 1kg.
s
sThere are 2 of product " paint n002 ", where 2 available and 0 reserved .Product type is ConstructionMaterials 1kg.
+
+
sThere are 3 of product " paint n002 ", where 3 available and 0 reserved .Product type is ConstructionMaterials 1kg.
```

Рисунок 1.106 – Консольный вывод выполнения приложения

Тогда *typename* и *template* – это ключевые слова языка, а «*WeightType*» – это наименование, которое дает разработчика «подменяемому» типу, который может быть использован внутри класса вместо конкретного типа (рисунок 1.107).

```
template<typename WeightType>
class ConstructionMaterials : public Product
```

Рисунок 1.107 – Метка шаблонного класса

Далее в месте, где необходимо определять или использовать поле веса товара, применяется параметр шаблона «*WeightType*» вместо типа *int* (рисунок 1.108).

```
WeightType weight = 0;
```

Рисунок 1.108 – Использование параметра-типа шаблона

В месте создания объектов указываем явно, какой конкретный тип данных нужно использовать для данного объекта (рисунок 1.109).

```
ConstructionMaterials<double> cMaterial1("paint n001",2,10.1);
ConstructionMaterials<int> cMaterial2("paint n002",2,1);
```

Рисунок 1.109 – Подстановка типа при создании объектов шаблонного класса

Фактически, во время компиляции компилятор сформирует в данном случае две реализации класса: с типами *int* и *double*, и применит их для создания объектов. Поэтому шаблон создает не один класс, а **семейство классов**, состав которого определяется по тем конкретным типам, которые подставили при создании всех объектов шаблонного класса. Такой подход реализует **полиморфизм времени компиляции**, при котором откладывается время задания конкретных типов для универсальных методов, функций и классов с их определения до мест их применения.

Если необходимо сделать шаблонный класс, в котором в качестве параметра шаблона выступает не простой тип, а класс или структура, то рекомендуется заменить ключевое слово *typename* на *class*. Фактически, на данный момент принципиальной разницы между ними нет, поэтому можно использовать любое ключевое слово.

Стоит отметить, что для переноса реализации метода шаблонного класса в файл исходного кода, необходимо его определить следующим образом, как показано на рисунке 1.110.

```
template<typename WeightType>
WeightType ConstructionMaterials<WeightType>::getWeight() const {
    return weight;
}
```

Рисунок 1.110 – Определение реализации метода шаблонного класса вне класса

Кроме параметров-типов, в шаблоне можно задавать **параметры-константы**. Хороший пример представлен в Википедии, где демонстрируется замена кода не универсального (рисунок 1.111) на более обобщённый вариант с определением типа данных и размерности массива (рисунок 1.112).

```
class SomeClass {  
    int SomeValue;  
    int SomeArray[ 20 ];  
    ...  
};
```

Рисунок 1.111 – Обычное определение класса

```
template < int ArrayLength, typename SomeValueType >  
class SomeClass {  
    SomeValueType SomeValue;  
    SomeValueType SomeArray[ ArrayLength ];  
    ...  
};
```

Рисунок 1.112 – Определение шаблонного класса с параметрами типа и константы

Шаблоны также можно применять для функций или методов. Продемонстрируем пример из Википедии, в котором задается функция определения минимального из передаваемых параметров, где тип формальных параметров и возвращаемого значения определены параметром шаблона (рисунок 1.113).

```
template<typename T>  
T min( T a, T b ) {  
    return a < b ? a : b;  
}
```

Рисунок 1.113 – Определение шаблонной функции

Прекрасным примером применения шаблонов выступают такие контейнерные классы стандартной библиотеки шаблонов (STL), как «*std::vector*» (список), «*std::map*» (словарь), «*std::queue*» (очередь), «*std::stack*» (стек) и другие, представляющие дженерики – обобщённые коллекции элементов. Рекомендовано изучить данные классы STL и понятие контейнера и итератора в C++.

2 ЛАБОРАТОРНАЯ РАБОТА № 1. ПОВТОРЕНИЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ C

Необходимо изучить материал для повторения языка программирования C и написать консольное приложение. В приложении пользовательские данные должны приходиться и выводиться с помощью функций «printf» и «scanf». Вывод должен быть понятным и полным для пользователя, чтобы он понимал, что от него требуется и что он видит в выводе. Должна быть обеспечена проверка вводимых данных, при некорректно введенных данных приложение должно сообщать об ошибке, и или завершаться, или повторно просить ввод данных. Пользователь может ввести любые данные в консоль, в том числе текст или пустую строку. Пользователь должен успевать прочитать выводимые данные.

Материал третьей лекции может помочь с функциями «printf» и «scanf».

Необходимо защитить лабораторную работу устно на паре. После сдачи лабораторной работы сформировать отчет, содержащий титульный лист, цели и задачи лабораторной работы, ход работы (кратко в одном-трех абзацах), вставить исходный код приложения (Courier New 9-10 пт в рамке) и скриншоты выполнения, выводы по работе. Полученный отчет загрузить на «sdo.tusur».

Варианты для лабораторной работы приведены ниже:

1. Напишите программу, которая выводила бы на консоль с помощью функции «printf» название вуза, факультета, кафедры, номер группы, фамилию, имя и отчество свое и двух своих одноклассников в разных строках консоли. Данные для каждого студента должны храниться в структуре (struct), а в качестве строк использовать массив char.
2. Написать программу ввода и вывода значений переменных a , b и c (типа float) с шестью цифрами целой части и двумя – дробной. Строка форматирования должна выводить данные в виде: « a = значение b = значение c = значение». Использовать функции «printf» и «scanf».
3. Напишите программу возведения целого числа в N -степень: с консоли вводится число и значение N , это число умножается само на себя нужное количество раз и на экран выводится результат. N может быть не больше 100 (сделать проверку условия в программе). Использовать функции «printf» и «scanf».
4. Создать программу-калькулятор с четырьмя арифметическими операциями. Последовательность ввода данных с консоли: число, символ оператора, число. Вывести результат. Использовать функции «printf» и «scanf».
5. Вычислить факториал числа $n!$, где n вводится с консоли. Вставить условие на максимально допустимое число n . Использовать функции «printf» и «scanf».
6. Вычислить сумму натуральных четных чисел, не превышающих N , где N вводится с консоли. Для умножения или деления на 2 использовать битовый оператор, для проверки на четность mod – «%». Использовать функции «printf» и «scanf».
7. Дано натуральное число N , где N вводится с консоли. Определить, является ли оно простым. Натуральное число N называется простым, если оно делится без остатка только на единицу и на само себя. Число 13 – простое, так как делится только на 1 и 13, а число 12 таковым не является, так как делится на 1, 2, 3, 4, 6 и 12. Использовать функции «printf» и «scanf».
8. Поступает последовательность из N вещественных чисел. Определить наибольший элемент последовательности. Сначала вводится число N , далее последовательность чисел. Использовать функции «printf» и «scanf».
9. Поступает последовательность целых положительных чисел. 0 – конец последовательности. Определить количество простых чисел в последовательности. Натуральное число N называется простым, если оно делится без остатка только на единицу и на само себя. Число 13 – простое, так как делится только на 1 и 13, а число 12 таковым не является, так как делится на 1, 2, 3, 4, 6 и 12. Сначала вводится число N , далее последовательность чисел. Использовать функции «printf» и «scanf».

10. Вводится последовательность целых чисел, 0 – конец последовательности. Найти наименьшее число среди положительных, если таких значений несколько, определить, сколько их. Максимальная длина последовательности – 10 элементов, если пользователь ввел 10 элементов, известить его об ограничении и вывести результат. Использовать функции «printf» и «scanf».

11. Вычислить значение функции « $y(x)=2*x+3$ » для « $x=\{0;3\}$ » с шагом « $h=0.5$ ». Вычисление математической функции обернуть в программную функцию. Вывод должен для каждого шага выводить строку с форматированием вида «номер итерации = значение x = значение, y = значение». Выводить не более 3 знаков после запятой. Использовать функции «printf» и «scanf».

12. Вводится последовательность целых чисел, 0 – конец последовательности. Найти наибольшее четное число, если таких значений несколько, определить, сколько их. Максимальная длина последовательности – 10 элементов, если пользователь ввел 10 элементов, известить его об ограничении и вывести результат. Использовать функции «printf» и «scanf».

13. Вводится последовательность целых чисел, 0 – конец последовательности. Рассчитать сумму элементов, умноженных на весовой коэффициент « $N/10.0$ », где « $N=[1,10]$ » – номер элемента в последовательности. Максимальная длина последовательности – 10 элементов, если пользователь ввел 10 элементов, известить его об ограничении и вывести результат. Использовать функции «printf» и «scanf».

3 ОБЩИЕ ТРЕБОВАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ № 2 – 6

Перед выполнением лабораторных работ № 2 – 6 рекомендуется **изучить материал лекций** и краткую теорию для выполнения лабораторной работы, в которой отражаются основные понятия и конструкции, а также предложен пример решения задач.

Соотнесение подразделов краткой теории и содержания лабораторной работы представлено в таблице 3.1.

Таблица 3.1 – Соотнесение разделов краткой теории и лабораторных работ

Лабораторная работа	Подраздел
№ 2	1.1 Классы. Модификаторы доступа. Поля и методы
	1.2 Вывод на консоль
	1.3 Работа с динамической памятью в C++ и конструкции инициализации
	1.4 Модификатор const
	1.5 Ссылка
	1.6 Применение модификатора const и для повышения надежности класса и определения конструктора копирования
	1.7 Обработка исключений
№ 3	1.8 Перегрузка функций, методов и операторов
	1.9 Параметры по умолчанию
№ 4	1.10 Наследование
№ 5	1.11 Интерфейсы, абстрактные классы, виртуальные методы
№ 6	1.12 Шаблоны

Для оформления кода выбрать и использовать соглашения по форматированию кода и правилам именования и их использовать в проекте (например, «Google C++ Style», «Qt Coding Style», «Mozilla codebase C++ Coding style» и другие). Давать осознанные наименования классам, полям, методам, функциям и переменным. Каждый класс должен быть указан в отдельной паре «h» и «cpp», имя которых должно соответствовать имени файла. Файл с функцией «main» должен называться «main.cpp». Подключение заголовочных файлов библиотек проходит в заголовочном файле класса. Базовый класс должен содержать хотя бы одно поле с динамической памятью.

Цель работ – формирование рабочего класса. Поэтому данные при создании объектов класса в функции *main* можно передавать в конструктор в виде констант, чтобы облегчить отладку и проверку класса (не нужно их каждый раз вводить с консоли). Если хочется создать алгоритм взаимодействия с пользователем, нужно написать функции для тестирования объектов класса, чтобы они запускались перед основной логикой пользовательского интерфейса и проверяли объект класса на константных данных.

Каждую лабораторную работу делать в отдельном проекте (то есть в копиях проекта). Можно использовать любой компилятор C++ и среду разработки (не онлайн). При использовании системы контроля версий Git можно фиксировать коммитами состояния проектов, которые соответствуют лабораторным работам – тогда можно выполнять все работы в одном проекте. В случае создания проекта в репозитории нужно отправить его в приватный репозиторий GitHub (добавить преподавателя в коллабораторы).

Необходимо защитить лабораторные работы устно на парах. После сдачи лабораторной работы сформировать отчет, содержащий титульный лист, цели и задачи лабораторных, ход работы (кратко в одном-трех абзацах), вставить исходный код приложения (Courier New 9-10 пт в рамке) и скриншоты выполнения, выводы по работе. Полученный отчет загрузить на «sdo.tusur».

4 ВАРИАНТЫ ЗАДАНИЙ ЛАБОРАТОРНЫХ РАБОТ

В разделе приведены варианты заданий для лабораторных работ. Рекомендуется изучить общие требования к работам, представленные в предыдущем разделе. Учесть возможность подготовки работ с использованием системы контроля версий.

Вариант 1. «Расчет налога и налогового вычета»

Предметная область будущих классов. Вы хотите разработать приложение, которое бы позволяло проводить учет дохода пользователей и уплаченных налогов с данного дохода. Для этого Вам необходимо сформировать класс для расчета подоходного налога и доходов физического лица за один календарный год.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о налогоплательщике за один год. Данный класс может содержать следующие поля и методы:

- ИНН (строка – *char**);
- год расчета;
- налогооблагаемый доход (до вычета налога);
- неналогооблагаемый доход;
- итог суммы подоходного налога;
- итог суммы доходов;
- процент подоходного налога (значение по умолчанию – 13 %).

Конструктор: с параметрами (ИНН, год расчета).

Селекторы на все поля.

Метод вывода всех параметров на консоль с уточнением назначения полей.

Метод добавления статьи дохода налогооблагаемого (до вычета налога) и неналогооблагаемого: входные параметры – размер дохода и флаг того, является ли доход налогооблагаемым, реализовать механизм расчета суммы налогов и суммы доходов.

Деструктор на освобождение строки ИНН.

Определите, как минимум 6 *исключений* по вводу ИНН, ненулевого дохода и недопустимого значения года и др.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *конструктора* на создание уже частично рассчитанного дохода (то есть, кроме ИНН и года расчета, необходимо вносить налогооблагаемый и неналогооблагаемый доходы). Задумайтесь здесь о параметрах по умолчанию.

Выполнить перегрузку *метода добавления налогооблагаемого дохода* с логикой вноса дохода после вычета налога (без флага, один входной параметр – размер фактически полученного дохода после вычета налога, то есть нужно сделать обратное преобразование и рассчитать, сколько уплатили налога с этого дохода и какой был налогооблагаемый доход).

Выполнить перегрузку *оператора «>>»* с логикой *метода добавления налогооблагаемого дохода* после вычета налога.

Выполнить *перегрузку оператора «+=»* вне класса для расчета общего налога от всех внесенных налогоплательщиков: левый аргумент – переменная суммирования, входные параметры (при необходимости добавьте модификатор *const*):

```
<тип>& operator += (<тип>& sum, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте *производный класс*, который бы позволил учесть доступный налоговый имущественный вычет (поля стоимость жилья и сумма вычета). Установить модификатор и селектор для поля стоимости жилья и селектор для полученной суммы вычета. Добавить логику для расчета доступного вычета (налоговый вычет – это возможность вернуть налоги на сумму не более 13 % от стоимости жилья, но не более 13 % от 2 млн руб.). При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода налогов, которые не подлежат возврату (считаем, что всегда налоговый вычет получен) по всем объектам. Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, чтобы сумма налога могла быть с копейками и без копеек. Укажите в шаблоне константу на величину процента. Измените логику вывода на консоль в соответствии с шаблоном.

Вариант 2. «Календарный план-график ипотеки»

Предметная область будущих классов. Вам нужен класс, который бы позволял выводить информацию о текущих платежах по ипотеке за каждый месяц. Класс должен по вводимой информации (общая взятая сумма ипотеки, ключевая ставка и срок) рассчитывать состав платежа на каждый месяц, в том числе выплачиваемые проценты и сумму, уходящую на закрытие тела кредита.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о выплатах по ипотеке за каждый месяц. Данный класс может содержать следующие поля и методы:

- размер заемных средств (сколько взяли в кредит);
- размер ключевой ставки;
- срок кредита в годах;
- размер месячного платежа;
- размер уплачиваемых процентов по месяцам (массив);
- величина платежа, которая уйдет в погашение тела кредита, по месяцам (массив);
- тело кредита по месяцам (массив);
- общая переплата (величина выплаченных процентов).

Конструктор: с параметрами (размер заемных средств и размер ключевой ставки).

Селекторы на все поля (кроме массивов).

Метод расчета календарного плана-графика (формальный параметр – срок кредита в годах) должен позволять проводить пересчет ипотеки.

Про расчет месячного платежа см. статью «Аннуитет» в Википедии.

Метод вывода на консоль календарного плана-графика в виде таблицы, где строка – один месяц платежей с указанием номера месяца.

Деструктор на освобождение захваченной памяти под массивы.

Определите, как минимум 6 *исключений* по вводу срока платежа, допустимых значений ключевой ставки, а также по выходу за диапазон размера допустимых заемных средств (от 300 тыс. руб., но не более 100 млн, как правило, у ипотечных программ есть такое ограничение), попытка вывода не рассчитанного плана-графика и т.д.

Используйте *ссылки, константные методы и модификатор const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *конструктора*, где три входных параметра определяют размер заемных средств, размер ключевой ставки и срок кредита. Пусть сразу рассчитывает календарный план-график. Задумайтесь здесь о параметрах по умолчанию.

Выполнить перегрузку *оператора* «<<» для вывода параметров ипотеки без вывода плана-графика (при необходимости добавьте модификатор *const*):

```
ostream& operator << (ostream &out, ClassName& ourObject)
```

Выполнить перегрузку *оператора* «[]» для получения остатка (тела кредита) на конкретный месяц, который подается как индекс месяца в «[]».

Выполнить перегрузку *оператора* «>» вне класса для сравнения двух ипотек по уровню переплаты (при необходимости добавьте модификатор *const*):

```
bool operator > (ClassName& ourObject1, ClassName& ourObject2)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил вести учет выплат по страховке – то есть ипотека со страхованием (процент страховки, массив выплат по страховке за каждый год). Учитывать, что страховка начисляется на остаток (тело кредита) на месяц выплаты страховки и платится раз в год в месяц получения кредита. Установить селекторы для процента страховки и добавить этот параметр в входные данные конструктора. Внести выплаты по страховке в общую переплату. Выводить в методе вывода на консоль информацию о страховке. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода на консоль и оператора «<<». Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где в качестве констант определяются допустимые диапазоны заемных средств.

Вариант 3. «Суточный рацион»

Предметная область будущих классов. Необходимо вести учет потребления продуктов в суточном рационе (БЖУ) для оценки потребленных калорий. Для этого пользователь вносит продукт: состав белков, жиров, углеводов на 100 г и вес потребленного продукта в граммах.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о суточном рационе. Данный класс может содержать следующие *поля* и *методы*:

- количество приемов пищи;
- белки в г за один прием пищи (массив);
- жиры в г за один прием пищи (массив);
- углеводы в г за один прием пищи (массив);
- потребленная масса продуктов в г;
- потребленный объем калорий в ккал.

Общепринятые округлённые значения калоража: 1 г белка – 4 ккал; жира – 9 ккал; углеводов – 4 ккал.

Конструктор по умолчанию (с трехразовым питанием) и конструктор с параметром – количество приемов пищи.

Селекторы на все поля (кроме массивов).

Метод вывода на консоль суточного рациона со сводной информацией и выводом детализации по приемам пищи.

Метод вноса потреблённого продукта с параметрами: номер приема пищи, БЖУ на 100 г продукта, потреблённая масса продукта, обновляет информацию по текущему приему пищи и обновляет сводные параметры.

Деструктор на освобождение массивов.

Определите, как минимум 6 *исключений* по вводу ненулевых значений, допустимому номеру приема пищи в методе и т.д.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *конструктора копирования* (в этот день съели тоже самое, что в прошлый, т.е. тот же рацион).

Выполнить перегрузку *метода вывода* с вводом номера приема пищи и выводом информации только для него.

Выполнить перегрузку *оператора «[]»* для получения объема потребленных ккал для конкретного приема пищи.

Выполнить перегрузку *оператора «<<»* для вывода только сводной информации (при необходимости добавьте модификатор *const*):

```
ostream& operator << (ostream &out, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил устанавливать необходимое пороговое значение для БЖУ (отдельно для белков, жиров, углеводов и ккал), которые могли бы определить ограничение на прием пищи в этот день. Установить селекторы и модификаторы для этих полей. Создать методы, которые проверяют выход за этот порог, то есть они возвращают истину, если можно продолжать есть, и ложь – если нужно остановиться. При выводе на консоль указывать, если были превышены пороговые значения. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте

интерфейс для вывода на консоль. Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где можно будет вести учет в г и кг (для *double* будет расчет в кг, а для *int* – в г). Укажите в шаблоне константы на инициализационные данные (пороговые значения БЖУ) для производного класса. Измените логику вывода на консоль в соответствии с шаблоном (например, можно определять текущий тип условием «*sizeof(double) == 8, sizeof(int) == 4*»).

Вариант 4. «Планирование уплаты счетов по электроэнергии»

Предметная область будущих классов. Вы хотите анализировать платежи по электроэнергии и учет потраченных кВт/ч по годам. Для этого Вам нужно сформировать класс для этой задачи (хранения показаний за один год). Принимается, что тариф на электроэнергию меняется только раз в год. Год может начинаться с любого месяца.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для информации о годовых показаниях и платежах за электроэнергию (за год). Данный класс может содержать следующие *поля* и *методы*:

- показания за каждый месяц года (массив);
- год учета;
- начисленные платежи за каждый месяц года (массив);
- итоговая сумма платежей;
- тариф (стоимость кВт/ч);
- среднее потребление энергии в месяц.

Конструктор с параметром – тариф (стоимость кВт/ч).

Селекторы на все поля (кроме массивов).

Метод вывода на консоль информации за каждый месяц и сводной информации.

Метод ввода показаний с параметрами: номер месяца и показания, обновляет сводные параметры.

Деструктор на освобождение массивов.

Определите, как минимум 6 *исключений* по вводу параметров в методах: ввод нулевых значений тарифа, попытка получения среднего показания при отсутствии показаний хотя бы за один месяц, внос недопустимого номера месяца, внос показаний вне диапазона (сколько физически не могли за месяц потратить) и т.д.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *метода вывода* с вводом номера месяца и отображения информации по нему (задумайтесь здесь о параметре по умолчанию).

Выполнить перегрузку *оператора* «[]» для получения размера платежа за текущий месяц.

Выполнить перегрузку *оператора* «<<» для вывода только сводной информации (без информации по месяцам) (при необходимости добавьте модификатор *const*):

```
ostream& operator << (ostream& out, ClassName& ourObject)
```

Выполнить перегрузку *оператора* «+=» вне класса для получения суммированного значения средних показаний за все года: левый аргумент – переменная суммирования, входные параметры (при необходимости добавьте модификатор const):

```
<тип>& operator += (<тип>& sum, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил вносить дополнительные платежи (например, пеню) для каждого месяца. Поменяйте логику вычисления итоговой суммы платежей, вносу показаний и вывода на консоль. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода на консоль и вноса показаний. Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где можно будет вести учет для вещественных и целых показаний. Укажите в шаблоне константы на определение начала года (с какого месяца будет начинаться год, например, покупка квартиры была в марте, соответственно, год выплат – с марта по март).

Вариант 5. «Учет дискового пространства сотрудников»

Предметная область будущих классов. Вы системный администратор. В Вашей компании есть общее для всех сотрудников дисковое пространство с директориями для каждого сотрудника. В один день Вы обнаружили, что диск начинает заканчиваться. Вы предполагаете, что сотрудники Вашей организации хранят личную информацию на общем дисковом пространстве: личные фотографии, смешные картинки и видео, музыкальные библиотеки. Вы не хотите начинать репрессии и заставлять всех удалить личные данные с общего диска. Вы подумали, что можно вычислить тех сотрудников, которые сильно злоупотребляют дисковым пространством в личных целях. Для этого Вы написали скрипт, который проходит папку каждого сотрудника и производит поиск «запрещенных файлов». Вы хотите вывести отчет, сколько всего памяти затратил сотрудник для той или иной категории файлов. Также Вы предполагаете, что у Вас на общем диске есть «мертвые души» – папки сотрудников, которые были уже уволены, поэтому не пользуются диском. Вам нужен класс для хранения отчета по каждому сотруднику и заполнения его результатами скрипта.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения отчета об директории сотрудника. Возможные поля и методы:

- имя или путь до директории (строка – *char**),
- объем занятого дискового пространства (в ГБ),
- из него на личные нужды (объем дискового пространства под картинки, видео и аудиофайлы),
- количество личных файлов (картинки, видео, музыка),

- список расширений файлов, которые считаются личными (статический или динамический массив заполните внутри класса константными строками самостоятельно),
- дата последнего изменения файлов (дата, можно использовать `SYSTEMTIME`, `time` или `chrono`, смотри приложение А).

Определите *конструктор* с одним входным параметром – имя директории.

Определите *метод вывода* полной информации на консоль.

Определите *селекторы* для всех полей.

Создайте *метод добавления нового файла в статистику* – входные параметры: имя файла (*char**), его объем и дата его изменения, метод должен определить по расширению файла, относится ли файл к личным файлам (можно применить функцию `std::strstr`), заполнить соответствующие поля об количестве и объеме, также обновлять данные в поле об последних изменениях.

Определите *деструктор* для освобождения динамической памяти (строка под имя директории).

Определите, как минимум 6 *исключений* по вводу наименования директории, неверного ввода даты и др.

Используйте ссылки, константные методы и модификатор *const* в нужных местах при необходимости.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку конструктора с дополнительным параметром даты последнего изменения файлов (хотим, чтобы учет велся для пользователя, активно работающего в дисковом пространстве).

Перегрузите *метод добавления нового файла в статистику*, где на вход поступает не имя файла, а флаг того, является ли этот файл личным.

Перегрузите операторы для файлового для ввода-вывода. Выполните сохранение данных в файл и чтение данных из файла (при необходимости добавьте модификатор *const*):

```
friend std::fstream& operator << (std::fstream&, ClassName& ourObject)
friend std::fistream& operator >> (std::fistream&, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил учитывать количество потраченного пользователем Интернет-трафика (в ГБ). Установить селектор для этого поля и метод добавления статистики в него (прибавление значения, то есть предположить, что каждый час, например, анализируется трафик, и он суммируется с текущим значением путем вызова этого метода), метод обнуления этого поля. Дополнить вывод этой информацией. При необходимости измените модификаторы доступа.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода сводных данных, а также определите в нем (а потом реализуйте в потомке и базовом классе) метод валидации сотрудников на ответственных и недобросовестных (возвращает *true*, если потратил больше допустимого уровня дискового пространства), где для потомка будет учитываться, кроме объема памяти, еще и интернет-трафик. При необходимости измените модификаторы доступа у базового класса. Иерархия наследования: «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, указав константы на пороговые значения для определения добросовестности сотрудников. В шаблоне также определите шаблон на тип значения объемов памяти (чтобы можно было вести учет в *int* или *double*). Измените логику вывода на консоль в соответствии с шаблоном.

Вариант 6. «Поиск оптимальной стоимости товара»

Предметная область будущих классов. Вам очень понравились несколько телефонов в одном интернет-магазине. Но Вы продвинутый покупатель и решили не покупать не раздумывая, а посмотреть на динамику изменения цены, так как Вы подозреваете, что она меняется. Вы написали скрипт, который забирает текущую стоимость каждого телефона раз в день со страницы товара, теперь Вам нужен контейнер для хранения данной информации. Но при этом Вы боитесь, что товар закончится, поэтому Вы забираете с сайта еще количество доступных к покупке телефонов, чтобы при наличии малого количества товара в остатках экстренно его купить.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о цене и количестве конкретного товара. Возможные *поля* класса:

- количество дней мониторинга;
- URL до страницы с товаром (массив *char**);
- текущая стоимость;
- текущее количество товара;
- текущий день мониторинга;
- динамика изменения цены (массив) с частотой мониторингом раз в день;
- средняя, максимальная и минимальная стоимость.

Конструктор с параметрами – URL-ссылка на товар, общее предполагаемое количество дней мониторинга, текущее количество товара, текущая стоимость товара.

Селекторы на все поля (кроме массивов).

Метод триггера на небольшой остаток (возвращает *true*, если осталось менее 2 единиц товара).

Метод возврата текущей динамики изменения цены: возврат 0 – текущая цена стабильна, 1 – текущая цена растет, 2 – текущая цена падает, 3 – текущая цена в минимуме за весь мониторинг, 4 – текущая цена в максимуме за весь мониторинг. Оформить эти константы в *enum*.

Метод вывода на консоль динамики изменения цены и сводной информации по другим полям.

Метод вноса текущих данных по товару с параметрами: текущая стоимость и количество остатка товара, обновляет сводные параметры (в том числе увеличение дня мониторинга).

Деструктор на освобождение массивов.

Определите, как минимум 6 *исключений* по вводу параметров: на нулевые параметры, выход за время мониторинга, проданный товар и т.д.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить *перегрузку конструктора* с двумя параметрами (URL и текущий остаток, задумайтесь о параметрах по умолчанию).

Выполнить перегрузку метода вывода с вводом номера дня мониторинга и выводом информации только для него.

Выполнить перегрузку оператора «[]» для получения цены товара на определенный день мониторинга.

Выполнить перегрузку оператора «<<» для вывода только сводной информации (при необходимости добавьте модификатор const):

```
ostream& operator << (ostream &out, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил еще сохранять текущий рейтинг товара, количество положительных и отрицательных отзывов. Установить селекторы и модификаторы для этих полей. Реализовать метод вноса текущих данных по товару с учетом этих полей. Изменить *вывод на консоль* в производном классе с учетом новых полей. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода на консоль и оператора «<<». Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где можно будет вести учет цены товара с копейками и без. Укажите в шаблоне константу на пороговый уровень *триггера* на небольшой остаток.

Вариант 7. «Учет товаров обихода»

Предметная область будущих классов. Вы стали замечать, что Вас беспокоит, что у Вас часто заканчивается зубная паста, бумажные полотенца, шампунь и другие товары обихода в неподходящее время. Вы решили автоматизировать этот процесс и организовать закупки раз в три месяца всех товаров своего обихода, чтобы их хватало. Поэтому Вам необходимо посчитать, сколько в месяц уходит у Вас примерно на покупку таких товаров, и сможете ли Вы сэкономить, закупив их на складе оптом. Поэтому Ваша цель вести учет покупки товаров своего обихода в течение месяца, чтобы рассчитать, сколько Вам надо товаров.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс учета покупки товаров обихода. Возможные *поля* класса:

- наименование товара (тип *char**),
- текущая стоимость товара (последняя стоимость),
- максимальная стоимость и минимальная стоимость товара за весь период,
- количество заказанных товаров за весь период,
- сколько всего потрачено на данный товар за весь период,
- средняя стоимость товара (отношение двух предыдущих полей).

Конструкторы: с вводом имени, по умолчанию и копирования.

Селекторы на все поля.

Метод вывода всех параметров на консоль.

Метод покупки товара с указанием: количества и стоимости, учесть в этом методе заполнение полей: стоимость товара (последняя стоимость), максимальная стоимость и

минимальная стоимость товара за весь период (определить логику изменения максимума-минимума), число заказанных товаров, сколько всего потрачено на данный товар, средняя стоимость товара.

Деструктор на освобождение строки имени.

Определите, как минимум *6 исключений* по вводу наименования товара, ненулевой стоимости и количества товара.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *конструктора* на создание уже купленного товара (то есть кроме имени, формальные параметры: количество и стоимость). Задумайтесь здесь о параметрах по умолчанию.

Выполнить перегрузку *метода* покупки товара по текущей стоимости в классе (один входной параметр – количество, а стоимость товара остается прежней).

Выполнить перегрузку *оператора* «+=» с логикой метода покупки товара по текущей стоимости (то есть купили заданное количество по текущей цене).

Выполнить перегрузку *оператора* «+=» вне класса для расчета общей стоимости растраты на все товары: левый аргумент – переменная суммирования, входные параметры (при необходимости добавьте модификатор *const*):

```
<тип>& operator += (<тип>& sum, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил расширить базовый класс количеством использованных товаров, чтобы при выводе на консоль указывать, сколько товаров осталось неистраченными на текущий момент. Установить селектор для данного поля и метод регистрации траты заданного количества товаров. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса, в котором будут содержаться объекты производного и базового классов. Создайте интерфейс вывода данных на консоль. Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, чтобы количество товаров могло быть определено в вещественном или в целом типе, также определите в шаблоне константу, которая бы определяла единицу изменения товара (оформите константу в *enum*, например, «enum *ProductCount* {kg=1, count=2};»). Измените логику вывода на консоль в соответствии с шаблоном.

Вариант 8. «Учет звонков»

Предметная область будущих классов. Вы хотите проанализировать информацию о том, как часто Вы разговариваете с определённым абонентом и сколько времени Вы тратите на эти разговоры. Для этого Вам нужен класс, в который Вы будете заносить эту информацию.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о разговорах с абонентами. Данный класс может содержать следующие поля и методы:

- наименование абонента из телефонной книги (массив *char**);
- номер телефона (массив *char**);
- максимальная длительность разговора (мин);
- минимальная длительность разговора;
- длина разговоров за весь период;
- количество звонков за весь период.

Конструктор с параметрами – наименование, телефон.

Селекторы на все поля (кроме массивов).

Метод вывода на консоль информации о разговорах и абоненте.

Метод получения средней длины разговора.

Метод оценки частоты общения – часто или нечасто (если за период время разговоров превышает 60 минут, то выдает *true*).

Метод вноса информации о звонке с параметром: длительность разговора, обновлять значения сводных полей.

Деструктор на освобождение массивов.

Определите, как минимум 6 *исключений* по вводу данных: некорректная строка, нулевые значения и т.д.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку конструктора для ввода только номера телефона, который будет дублироваться в наименование абонента (подумайте тут о параметрах по умолчанию).

Перегрузите операторы для файлового для ввода-вывода. Выполните сохранение данных в файл и чтение данных из файла (при необходимости добавьте модификатор *const*):

```
friend std::ofstream & operator << (std::ofstream &, ClassName& ourObject)
friend std::ifstream & operator >> (std::ifstream &, ClassName& ourObject)
```

Выполнить перегрузку *оператора* «+=» вне класса для расчета суммы для расчета среднего времени на разговоры со всеми абонентами за весь период: левый аргумент – переменная суммирования, входные параметры (при необходимости добавьте модификатор *const*):

```
<тип>& operator += (<тип>& sum, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил указывать для каждого звонка рейтинг важности и выводить сводную информацию о том, какой средний рейтинг разговоров для абонента. В *методе вноса информации о звонке* добавить параметр рейтинга звонка, суммировать все рейтинги, рассчитывать среднее, как сумма рейтинга, деленная на число звонков. Установить селекторы для определенного поля или полей. Изменить вывод на консоль с учетом нового параметра. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода на консоль и оператора «+=». Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где можно определять константу для порогового значения *метода оценки частоты общения*.

Вариант 9. «Учет сдачи лабораторных работ»

Предметная область будущих классов. Вам нужен класс для учета сдачи лабораторных работ студентами с учетом темы, сложности лабораторных работ, итоговой оценки по дисциплине как среднее от полученных оценок.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о процессе сдаче лабораторных работ по заданной дисциплине студентом. Данный класс может содержать следующие *поля и методы*:

- дисциплина (определить *enum*);
- количество лабораторных работ;
- оценка за лабораторные работы (массив динамический);
- сложность лабораторных работ (массив динамический);
- текущая оценка по дисциплине;
- количество сданных лабораторных работ.

Конструктор с параметрами – дисциплина и количество лабораторных работ.

Селекторы на все поля (кроме массивов).

Метод вывода на консоль информации о дисциплине и текущему состоянию ее сдачи.

Метод вноса информации о сданной лабораторной работе (ее номер по порядку и оценка).

Деструктор на освобождение массивов.

Определите, как минимум 6 *исключений* по вводу данных: выход за диапазон, нулевые значения и т.д.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *конструктора* для ввода (кроме уже определенных параметров в прошлом конструкторе) сложности дисциплины (подумайте тут о параметрах по умолчанию).

Выполнить перегрузку *оператора* «<<» для вывода на консоль (при необходимости добавляйте модификатор *const*):

```
ostream& operator << (ostream& out, ClassName& ourObject)
```

Перегрузите *оператор* «[]» для вноса оценки, полученной при сдаче лабораторной работы.

Выполнить перегрузку *оператора* «+=» вне класса для расчета количества несданных работ по всем дисциплинам: левый аргумент – переменная суммирования, входные параметры:


```
<тип>& operator += (<тип>& sum, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил учитывать то, что был сдан ли отчет по конкретной лабораторной работе и сколько всего отчетов надо еще сдать. Отразить это при выводе на консоль. Создать метод установки сданного отчета, считать, что лабораторная работа сдана, когда она защищена и предоставлен отчет после ее сдачи. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода на консоль и оператора «[]». Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где итоговая оценка может быть вещественным или целым числом.

Вариант 10. «Статистика рабочего времени на выполнения задач»

Предметная область будущих классов. Вам нужен класс, который Вы будете использовать для учета задач, которые будут иметь описание, информацию о категории задачи, ее статусе, назначенном исполнителе, сроке исполнения и количестве часов, потраченных на задачу.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о задаче. Данный класс может содержать следующие *поля* и *методы*:

- наименование задачи (строка – *char**);
- срок сдачи (дата, можно использовать SYSTEMTIME, time или chrono, смотри приложение А);
- категория задачи (ошибка, новый функционал, поддержка – обернуть в *enum*);
- флаг того, закрыта или открыта задача;
- исполнитель (строка – *char**);
- оценка времени на выполнение задачи (час);
- фактическое количество часов, потраченных на задачу.

Конструктор с параметрами – наименование, категория, срок сдачи.

Селекторы на все поля (кроме массивов).

Метод вывода на консоль информации о задаче.

Методы (отдельные) установки исполнителя, установки оценки выполнения, закрытия задачи и проверки того, что просрочена ли задача (возвращает true, если на текущий день она просрочена).

Метод добавление времени, затраченного на промежуточные этапы выполнения задачи.

Деструктор на освобождение массивов.

Определите, как минимум 6 *исключений* по вводу данных: некорректная срока, нулевые значения и т.д.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *конструктора* для ввода дополнительно оценки времени выполнения и исполнителя (подумайте тут о параметрах по умолчанию).

Выполнить перегрузку *оператора* «<<» для вывода на консоль (при необходимости добавьте модификатор const):

```
ostream& operator << (ostream &out, ClassName& ourObject)
```

Перегрузите *оператор* «+=» для добавления времени выполнения.

Выполнить перегрузку *оператора* «+=» вне класса для возможности расчета общего затраченного времени на задачи: левый аргумент – переменная суммирования, входные параметры (при необходимости добавьте модификатор const):

```
<тип>& operator += (<тип>& sum, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил прикреплять к задаче ULR-ссылку на ветку репозитория (строка – *char**), в рамках которой выполняется задача, при создании объекта. Учесть вывод этого поля на консоль. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода на консоль и оператора «+=» для добавления времени выполнения. Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где можно задавать время выполнения в целых и вещественных часах.

Вариант 11. «Учет приема лекарств»

Предметная область будущих классов. Вам нужен класс, в котором Вы будете вести учет приема лекарства за один день. Планируется прием лекарств максимум 5 раз в день (например, утром, перед обедом, обед, ужин, на ночь). В классе должен учитывать план приема и отметку фактического приема в заданный час суток.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о приеме конкретного лекарства. Данный класс может содержать следующие *поля* и *методы*:

- наименование лекарства (массив – *char**);
- день приема (номер дня с начала года);
- количество приемов в день (равно 5);
- план приема в заданный час суток (массив из 5 элементов) – отмечает необходимость приема лекарства в заданное время суток;
- отметка приема (массив из 5 элементов) – отмечает прием лекарства в заданное время суток.

Определить *enum* для времени суток из 5 элементов.

Конструктор с параметрами – наименование лекарства и день.

Селекторы на все поля (кроме массивов).

Метод вывода на консоль информации о приеме лекарства (включая план и фактический прием).

Методы (отдельные) установки плана приема в определенное время суток и установки приема в определенное время суток, а также проверки необходимости принять лекарство в заданное время суток.

Метод проверки того, что фактический прием соответствует плану (лекарство в этот день приняли вовремя).

Деструктор на освобождение массивов (динамических).

Определите, как минимум 6 *исключений* по вводу данных: некорректная срока, недопустимое время суток и т.д.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *конструктора* с параметрами: копируемый объект и флаг того, нужно ли из него копировать план или нет. То есть можно в текущий день из предыдущего дня копировать наименование лекарства, план приема и номер дня, увеличенный на единицу, план приема.

Выполнить перегрузку *оператора* «<<» для вывода на консоль (при необходимости добавьте модификатор *const*):

```
ostream& operator << (ostream &out, ClassName& ourObject)
```

Перегрузите *оператор* «[]» для установки приема лекарства в заданное время суток.

Перегрузить *метод проверки* необходимости принять лекарство в заданное время суток без параметров – то есть, если параметр не будет определен, будет выводиться результат *метода проверки* того, что фактический прием за день соответствует плану.

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил прикреплять дополнительное пояснение к приему (строка – *char**, например, рекомендации по тому, что нужно принимать до еды, или нужно разжевывать или глотать капсулу). Создать метод для установки этого значения. Учесть вывод этого поля на консоль. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода на консоль и оператора «[]» для установки приема. Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где можно задавать константу для количества времени суток.

Вариант 12. «Результаты обучения модели»

Предметная область будущих классов. Вы занимаетесь машинным обучением. У Вас есть задача обучить несколько моделей с разными параметрами и оценить их точность. Для этого Вы определяете класс.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о результатах обучения модели. Данный класс может содержать следующие *поля* и *методы*:

- тип модели (нейронная сеть, машина опорных векторов, дерево решений, наивный байесовский классификатор, ансамблевый метод – оформить в *enum*);
- строка с заданными параметрами обучения модели – например, «-p 300 -n 1000» (массив – *char**);
- объем тестовой выборки (шт.);
- отдельные поля для результатов обучения модели при применении ее на тестовой выборке: True Positive (*TP*), False Positive (*FP*), False Negative (*FN*), True Negative (*TN*) (шт);
- доля правильных ответов алгоритма (*accuracy*):

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN};$$

- *precision* (точность):

$$precision = \frac{TP}{TP + FP};$$

- *recall* (полнота):

$$recall = \frac{TP}{TP + FN}.$$

Конструктор с параметрами – тип модели, строка с заданными параметрами обучения модели, объем тестовой выборки.

Селекторы на все поля (кроме массивов).

Метод вывода на консоль информации этой модели.

Метод установки результатов обучения (*TP*, *FP*, *FN*, *TN*).

Метод проверки того, что модель дает допустимую точность (возвращает *true*, если *accuracy*, *precision*, *recall* больше 70 %).

Деструктор на освобождение массивов.

Определите, как минимум 6 *исключений* по вводу данных: некорректная строка, недопустимое значение *TP*, *FP*, *FN*, *TN* – их сумма не равна объему тестовой выборки, деление на ноль и т.д.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *конструктора*, где вместо объема выборки задаются *TP*, *FP*, *FN*, *TN* (и по ним рассчитывается объем выборки).

Выполнить перегрузку *оператора* «<<» для вывода на консоль (при необходимости добавьте модификатор *const*):

```
ostream& operator << (ostream &out, ClassName& ourObject)
```

Перегрузите *оператор* «[]» для выдачи *accuracy*, *precision*, *recall* (то есть определить *enum* для 3 параметров и в «[]» подаем значения этих констант).

Выполнить перегрузку *оператора* «+=» вне класса для возможности расчета суммы *accuracy* от нескольких моделей: левый аргумент – переменная суммирования, входные параметры (при необходимости добавьте модификатор *const*):

```
<тип>& operator += (<тип>& sum, ClassName& ourObject)
```

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил добавить еще одну метрику – меру $F1$:

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}.$$

Учтите вывод этого поля на консоль и для оператора «[]». При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода на консоль и оператора «[]» для получения оценочных параметров. Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где можно было задавать параметры оценки в вещественных или в целых значениях.

Вариант 13. «Учет результатов антиплагиата»

Предметная область будущих классов. Вам нужен класс для учета прохождения процесса проверки на антиплагиат и соответствие объема работы требуемому количеству страниц для одной работы студента.

Лабораторная работа № 2. Базовый синтаксис классов

Сформируйте класс для хранения информации о результатах прохождения антиплагиата. Данный класс может содержать следующие *поля* и *методы*:

- имя студента (массив – *char**);
- наименование работы (массив – *char**);
- тип работы (ВКР, курсовая работа, отчет, отчет по практике и др., оформить в *enum*);
- количество страниц;
- процент последней проверки на антиплагиат;
- объем самоцитирования (%);
- константа границы прохождения проверки по объему оригинального текста (более 70 %);
- пройдена ли проверка.

Конструктор с параметрами – наименование работы, студент, тип работы, количество страниц.

Селекторы на все поля (кроме массивов).

Метод вывода на консоль информации о проверке.

Методы установки результатов прохождения антиплагиата для объема оригинального текста и самоцитирования в процентах.

Метод проверки соответствия объема работ по страницам (ВКР – более 60 стр., но до 100 стр., курсовая работа – от 15 до 25 стр., отчет по лабораторная или практическая работы – более 5 стр., но не более 50 стр., отчет по практике – более 20 стр.).

Метод проверки соответствия допустимому объему плагиата.

Деструктор на освобождение массивов.

Определите, как минимум 6 *исключений* по вводу данных: некорректная срока, недопустимый диапазон результата проверки и т.д.

Используйте ссылки, константные методы и модификатор *const* в нужных местах.

Лабораторная работа № 3. Перегрузка методов и операторов

Необходимо перегрузить методы и операторы.

Выполнить перегрузку *конструктора* без задания типа работы и количества страниц (подумайте тут о параметрах по умолчанию). Для задания типа работы и количества страниц определите отдельный метод.

Выполнить перегрузку *оператора* «<<» для вывода на консоль (при необходимости добавьте модификатор *const*):

```
ostream& operator << (ostream &out, ClassName& ourObject)
```

Перегрузите *операторы* «+=» и «-=» «для увеличения или уменьшения числа страниц (то есть студент может сказать, что после правок его работа увеличилась или уменьшилась на *N* страниц).

Перегрузить *метод проверки соответствия* допустимому объему плагиата, где входным параметром приходит порог (вместо 70 %).

Лабораторная работа № 4. Наследование

Создайте производный класс, который бы позволил добавлять флаги наличия дневника и отзыва работодателя для практик (учитывайте, что тут тип работ неизменен, поэтому в конструкторе можно не передавать этот параметр и устанавливать его по умолчанию). Создать методы для установки (модификаторы) и получения значений (селекторы). Учесть вывод этих полей на консоль. При необходимости измените модификаторы доступа у базового класса.

Лабораторная работа № 5. Абстрактные классы и интерфейсы

Обеспечьте возможность работы с массивом типа базового класса (массив указателей), в котором будут содержаться объекты производного и базового классов. Создайте интерфейс для вывода на консоль, *метода проверки соответствия* допустимому объему плагиата и объему страниц. Иерархия наследования «интерфейс → базовый класс → производный класс».

Лабораторная работа № 6. Шаблоны

Оформите шаблон, где можно задавать вещественные и целые проценты, оформить константой в шаблоне значение константы границы прохождения проверки по объему оригинального текста.

Список литературы

1. Макконнелл, С. Совершенный код / С. Макконнелл. – Москва : Русская редакция, 2010. – 896 с.
2. C++ reference. – Текст : электронный // cppreference.com: [сайт]. – 2024. – URL: <https://en.cppreference.com/w/> (дата обращения: 28.08.2024).
3. Паттерн Singleton (одиночка, синглет). – Текст : электронный // cppreference.com: [сайт]. – 2024. – URL: <https://web.archive.org/web/20221026083912/http://cppreference.ru/patterns/creational-patterns/singleton/> (дата обращения: 28.08.2024).
4. Standard library header <cstdarg>. – Текст : электронный // cppreference.com: [сайт]. – 2024. – URL: <https://en.cppreference.com/w/cpp/header/cstdarg> (дата обращения: 28.08.2024).
5. Перегрузка оператора << для собственных классов. – Текст : электронный // Microsoft Learn: [сайт]. – 2024. – URL: <https://docs.microsoft.com/ru-ru/cpp/standard-library/overloading-the-output-operator-for-your-own-classes?view=vs-2019> (дата обращения: 28.08.2024).

ПРИЛОЖЕНИЕ А
(справочное)
Пример работы с датой и временем

```
#include <iostream>
#include <time.h>
#include <sstream>
#include <iomanip>
#include <chrono>

// получение текущей даты-времени
tm getCurrentDayTime() {
    // хотим текущую дату время получить
    tm currentDayTime;
    // через chrono получаем текущее время-дату
    std::chrono::system_clock::time_point nowDateTime = std::chrono
                                                         ::system_clock::now();

    time_t intermediateDayTime = std::chrono
                                 ::system_clock::to_time_t(nowDateTime);

    // рассчитываем время по текущему часовому поясу
    localtime_s(&currentDayTime, &intermediateDayTime);

    return currentDayTime;
}

// перевод из строки в дату-время
tm getDayTimeFromString(std::string dateTimeString) {

    tm dayTimeFromString;
    //задаем дату как поток символов
    std::istringstream stringToTmConverter(dateTimeString);
    //преобразуем строку в структуру tm,
    // заполняются только те поля, которые указаны в строке
    stringToTmConverter >> std::get_time(&dayTimeFromString,
                                           "%d.%m.%Y %H:%M:%S" );

    //хотим, чтобы рассчитался день недели и прочие зависимые параметры
    mktime(&dayTimeFromString);

    return dayTimeFromString;
}

int main() {
    tm currentDayTime = getCurrentDayTime();
    tm dayTimeFromString = getDayTimeFromString( "13.10.2023 00:26:36" );
}
```