Оглавление

ВВЕДЕНИЕ

Данное пособие посвящено взаимодействию четырёх ключевых аспектов профессионального уровня создания программного обеспечения – грамотному, обоснованному выбору структур данных, масштабированию объектно-ориентированных решений, борьбе с уязвимостями и программной оптимизации уровня исходных текстов (source-level code optimization). К большому сожалению, именно этим вопросам уделяется мало внимания в традиционных курсах программирования, хотя выигрыш в качестве программного обеспечения здесь довольно велик и именно владение этой системой навыков в первую очередь отличает профессионального программиста от дилетанта.

Современный уровень развития языковых средств реализации программного обеспечения позволяет в значительной степени уйти от традиционного трудоёмкого и обычно неэффективного создания собственных структур данных, опираясь на идеально выверенные встроенные в языки программирования возможности, в первую очередь, так называемые контейнеры. Эти новые для языков абстракции требуют совершенно иного стиля разработки и реализации программ по сравнению с традиционным. Знакомству с устройством и функционалом этих важных программных инноваций, а также получению базовых навыков по поиску и исправлению программных уязвимостей посвящена первая часть пособия.

Вторая часть пособия нацелена на практическое освоение наиболее действенных методов оптимизации кода применительно к самым вычислительно трудоёмким моментам в программировании – циклам и массивам. Формирование систематических навыков создания высокоэффективных в отношении быстродействия и используемой памяти программных решений является неотъемлемым элементом профессиональной разработки приложений.

КЛАССЫ

Классы представляют собой новые типы данных, которые вводятся в код программистом. Обычно пользоваться стандартными типами данных (int, double и т.п.) для описания какого-либо явления или объекта не удобно, т.к. сложный объект характеризуется множеством полей (хранимых данных). Например, объект типа Студент должен содержать: имя, фамилию, отчество,

курс, группу и т.д. Полей в таком объекте много, они имеют разные типы. Создавать отдельную переменную для каждого поля и для каждого студента неудобно и нерационально. Необходимо объединить данные, относящиеся к одному отдельно взятому студенту (экземпляру или объекту типа Студент) в единую сущность. Для этого вводятся новые типы, которые зачастую обладают большим количеством полей (хранимых внутри себя переменных). В рамках языка Си можно было ввести подобную конструкцию в виде структуры. Но там структура могла содержать только данные, функции обработки структуры были отделены от объектов структуры. В рамках объектно-ориентированного подхода данные (поля) и функции (методы) объединяются внутри вводимого программистом типа данных (класса).

Например, для класса Дробь (Fraction) удобно наличие в рамках одной сущности и числителя, и знаменателя.

```
class Fraction
{
private:
 //поля: числитель и знаменатель

 int numerator;
 int denominator;
public:
…
}
```

При этом, при создании объекта типа (экземпляра класса) Fraction каждый экземпляр объекта будет иметь внутри себя два поля: числитель и знаменатель. Изменение числителя или знаменателя в одной дроби никак не связано с их значениями в другой дроби.

Инкапсуляция. Ключевые слова private, public

Внутри класса, кроме функций (методов) и переменных (полей), доступных извне, могут быть вспомогательные переменные и функции, которые используются при решении задач, но не должны быть доступны в интерфейсе. Такие поля и методы помещаются в область private. Те функции и переменные, которые могут быть доступны извне, помещаются в область public. В классе Fraction числитель и знаменатель должны быть закрыты

(private) от доступа извне. Иначе, в любой момент можно организовать следующую проблему:

```
Fraction F; F.denominator = 0;
```

Из-за того, что поля класса Дробь открыты, можно записать в любое поле некорректное значение без контроля. Обычно поля, которые хранят информацию, помечаются закрытыми (private), но при необходимости к ним предоставляется доступ извне: 1) функция get…(), «геттер», позволяет получить извне значение поля, 2) функция set…(), «сеттер», примет значение, которое нужно присвоить полю, проверит его на корректность и выполнит действие при положительном результате. Для класса Дробь это выглядит так:

```
class Fraction
{
private:
 //поля: числитель и знаменатель

 int numerator;
 int denominator;
public:

 //геттеры и сеттеры
 void setNumerator(int value)
 {
     numerator = value;
 }

 void setDenominator(int value)
 {
     if(value!=0) denominator = value;
 }

 int getNumerator()
 {
     return numerator;
 }
```

```
 int getDenominator()
 {
     return denominator;
 }
};
```

В функции main() можем создать объекты типа Дробь и присвоить числителю и знаменателю корректные значения.

```
Fraction F;
Fraction G;

F.setNumerator(1);
F.setDenominator(2);

G.setNumerator(1);
G.setDenominator(0);
```

В последней операции значение знаменателя не будет изменено, т.к. в функции setDenominator() включена проверка на присваемое значение. При этом невооруженным глазом видна необходимость дать начальные значения полям объекта класса (провести инициализацию) при его создании. Для этого надо реализовать конструкторы класса.

Конструкторы. Перегрузка функций

Конструктор – это функция, которая работает в момент создания объекта класса. Он обладает интересными отличительными особенностями: 1) конструктор имеет то же имя, что и сам класс, 2) конструктор не имеет типа (он и не должен никуда ничего возвращать, он настраивает значения полей и выделяет память под динамические массивы, которые используются в объекте), 3) чаще всего, конструкторы относятся к открытой зоне (public). При этом, зачастую конструкторов много (для того, чтобы можно было в разных ситуациях давать разные наборы значений для полей класса). Для того, чтобы этого добиться, применяется перегрузка функций (см. пункт Перегрузка функций далее). Например, для класса Дробь реализовано четыре конструктора:

```
class Fraction
```

```
{
protected:
 //поля: числитель и знаменатель

 int numerator;
 int denominator;
public:

 //конструкторы
 Fraction()
{
cout << "\nBase constructor0 is working";
numerator = 0;
denominator = 1;
}

 Fraction(int Numerator)
{
cout << "\nBase constructor1 is working";
numerator = Numerator;
denominator = 1;
}

 Fraction(int Numerator, int Denominator)
{
cout << "\nBase constructor2 is working";
numerator = Numerator;
denominator = Denominator;
}


 Fraction(Fraction& F)
{
numerator = F.numerator;
denominator = F.denominator;
}
…
};
```

В этом примере первый конструктор не имеет параметров – это конструктор по умолчанию. Он работает при создании объекта класса следующим образом:

```
Fraction F;
```

Т.к. в конструктор ничего не передано, используются значения по умолчанию (0 для числителя и 1 для знаменателя).

Второй конструктор принимает один параметр. Считаем, что речь идёт о целом числе, поэтому сохраняем это значение в числителе, а в знаменателе ставим значение 1. Для того, чтобы сработал этот конструктор, надо создать переменную следующим образом:

```
Fraction F1(3);
```

Если при создании дроби используются два параметра, работает третий конструктор – с двумя аргументами:

```
Fraction F(3, 4);
```

Конструктор копий (четвёртый, указанный в примере) срабатывает, если нужно скопировать дробь. Такая необходимость возникает в трёх случаях: 1) дробь передаётся в функцию (не по указателю, не по ссылке) и копируется, 2) дробь возвращается как результат работы функции с помощью оператора return (обычно объект создаётся внутри заканчивающей работу функции и «погибает» в ней, а его копия возвращается в точку вызова), 3) при создании объекта он инициализируется существующим объектом:

```
Fraction H = F;
```

Важное действие, которое производится в конструкторе, - это выделение памяти под динамические массивы, которые присутствуют в объекте класса. Например, для класса строка BaseString (p – динамический массив, len – количество заполненных элементов, capacity – объём выделенной памяти) необхо-

димо выделить память под массив p. В примере представлено два конструктора (один принимает динамический массив, откуда надо скопировать информацию, второй – объём памяти, который надо выделить):

```
class BaseString
{
private:
 char* p;
 int len;
 int capacity;
public:
 BaseString(char* ptr)
 {
      len = strlen(ptr) + 1;
      capacity = 256;
      p = new char[capacity];
      for(int i=0;i<len;i++)
      {
          p[i] = ptr[i];
      }
      p[len] = '\0';
 }

 BaseString(int Capacity = 256)
 {
      capacity = Capacity;
      p = new char[capacity];
      len = 0;
 }
…
}
```

В таких ситуациях важно, что указатели ptr и p не должны быть связаны с одним и тем же участком памяти. Если их просто присвоить, то в том месте, где произошёл вызов конструктора, может сработать очистка памяти, выделенной под массив ptr. При этом, указатель p будет связан с той же самой памятью, объектом типа BaseString будет нельзя воспользоваться (это сразу приведёт к

ошибке). Так же к ошибке приведёт и работа деструктора (см. пункт Деструктор), если там будет произведена попытка очистки памяти (уже очищенной).

Конструктор по умолчанию

Конструктор без параметров – это конструктор по умолчанию. Предполагается, что все поля приобретают значения по умолчанию (некоторые стандартные значения, которые подразумеваются, если никаких данных нет). В частности: для указателей зачастую используют значение NULL, для переменных типа строка – пустые строки, для дробей – 0 для числителя и 1 для знаменателя и т.п. Он работает при создании объекта некоторого класса MyClass:

```
MyClass M;
```

и имеет вид:

```
MyClass()
{
…
}
```

Если в классе не задан конструктор по умолчанию, он создаётся автоматически пустым.

Конструктор копий (копирования)

Конструктор копий срабатывает, если нужно скопировать объект класса. Такая необходимость возникает в трёх случаях: 1) объект передаётся в функцию (не по указателю, не по ссылке) и копируется, 2) объект класса возвращается как результат работы функции с помощью оператора return (обычно объект создаётся внутри заканчивающей работу функции и «погибает» в ней, а его копия возвращается в точку вызова), 3) при создании объекта он инициализируется существующим объектом, например, для дробей:

```
Fraction H = F;
```

Конструктор копий для произвольного класса MyClass имеет вид:

```
MyClass(MyClass& M)
{
```

```
}
```

В некоторых компиляторах требуется наличие ключевого слова const:

```
MyClass(const MyClass& M)
{

}
```

Здесь можно вспомнить: если объект передаётся в функцию не по ссылке, не по указателю, а в «обычном» режиме, он копируется. Т.к. здесь программируется операция копирования, такое действие допустить нельзя. Поэтому объект в конструктор копий надо передавать по ссылке. Но передача по ссылке обозначает не только то, что объект не копируется при передаче в функцию, но и то, что внутри неё он может измениться. Тот объект, который копируется, не должен меняться при операции копирования, и некоторые IDE проверяют это (требуют наличие слова const).

Если в классе не задан конструктор копий, он создаётся автоматически: в нём значения всех полей оригинала копируются в поля копии.

Конструктор копий приобретает особую важность при копировании объектов, содержащих динамические массивы. Если использовать стандартный конструктор копий, указатели будут присваиваться, поэтому, когда для одного объекта (оригинала или копии) будет вызван деструктор (освобождающий ту же память, на которую «смотрит» указатель во втором объекте), работа второго объекта будет парализована. Попытка работы его деструктора приведёт к ошибке (повторная очистка памяти), попытка чтения уже очищенного динамического массива тоже приведёт к ошибке. Поэтому в конструкторе копий надо для динамических массивов выделять новый участок памяти и копировать данные из оригинала. Например, для класса BaseString,отвечающего за работу над строкой:

```
BaseString(BaseString& s)
 {
      len = s.Length()+1;
      p = new char[s.capacity];
      capacity = s.capacity;
```

```
        for(int i=0;i<s.Length();i++)
        {
            p[i] = s[i];
        }
        p[len] = '\0';
    }
```

Деструктор

Выше, в пунктах, посвящённых конструкторам, было указано, что важно грамотно выделять память и инициализировать переменные в момент создания объекта класса. При исчезновении объекта (при выходе из блока, где он объявлен, или при исчезновении копии объекта) срабатывает деструктор. Эта функция обладает особенностями:

1. Имеет то же имя, что и класс.
2. Чтобы отличать деструктор от конструкторов, в названии используют тильду ~.
3. Деструктор может быть только один – он не перегружается и не принимает параметров. Так делается потому, что обычно он вызывается автоматически.

В деструкторе обычно идёт освобождение памяти, выделенной для динамических массивов в конструкторе. В случае, если деструктор не описан, работает функция с пустым телом.

Например, деструктор класса BaseString, приведённого как пример в предыдущих пунктах, работает так (в классе присутствует указатель p):

```
class BaseString
{
protected:
 char* p;
 int len;
 int capacity;
public:
…
 ~BaseString()
 {
      cout<<"\nBase Destructor\n";
      if(p!=NULL)
          delete[] p;
```

```
        len = 0;
    }
    };
```

Важно понимать, что деструктор вызывается не только для объектов, которые напрямую создаются программистом в коде, но и для копий объектов, которые являются параметрами функций или результатом, возвращаемым с помощью return (за исключением случаев передачи параметров или результата с помощью ссылок или указателей). Подробно основные подходы к работе с классами и инкапсуляции изложены в [1 – 6].

НАСЛЕДОВАНИЕ

Наследование позволяет связывать несколько классов. Класс-наследник может иметь доступ к некоторым функциям и полям базового класса (родителя). Таким образом, нет необходимости несколько раз дублировать один и тот же код. Особенно удачно этот подход себя проявляет, когда класс, созданный в рамках одного проекта, может быть использован в другом. При наследовании есть три основные возможности:

1. В наследнике можно пользоваться некоторыми методами и полями базового класса.
2. В наследнике можно переопределять методы базового класса (интерфейс остаётся одинаковым, но в каждом классе работает своя версия функции).
3. В наследнике можно создавать новые методы и поля.

Наследование используется в ситуациях, когда в наличии некоторый класс (родитель), реализующий часть нужной Вам функциональности. Но если его функций для решения задачи недостаточно, можно добавить дополнительные поля и функции в его наследник. Объект класса-наследника неявно содержит в себе объект класса-родителя, поэтому имеет доступ к его компонентам.

Доступ к полям и методам базового класса. Ключевые слова private, public, protected

Основная идея наследования – дать доступ из функций потомка к функциям родителя. Таким образом достигается возможность повторного использования уже созданного кода вместо ручного переписывания раз за разом одинаковой функциональности.

При создании класса-наследника указывается специальное ключевое слово, которое задаёт уровень доступа кода в наследнике по отношению к

родителю (какие функции наследнику разрешено вызвать в родительском классе, какие переменные ему доступны).

Чаще всего, наследование производится с ключевым словом public. В такой ситуации наследнику: 1) не доступны методы и поля, отмеченные как private, 2) доступны открытые (public) методы и поля. Здесь мы видим две крайних противоположности: закрытые переменные и функции не видны так же, как и извне класса, а открытые – доступны везде (и наследнику, и извне класса). Часто нужен промежуточный вариант: доступ извне закрыт (аналог private), но доступ из наследника есть (аналог public). Такими свойствами обладает ключевое слово protected (защищённый уровень). При наследовании с ключевым словом public защищённые методы и поля остаются доступны в методах наследника.

Рассмотрим наследование «оберток» для класса массива (урезанный аналог класса vector из библиотеки STL). Используем наследование с ключевым словом public (класс MyArrayParent – родитель, MyArrayChild – потомок / сын / дочерний класс). Обратите внимание на вызов в методе класса MyArrayChild команды push(), принадлежащей родителю MyArrayParent. Если функции Insert() требуется выполнить вставку в конец массива, вместо того, чтобы дублировать код функции родителя, производится вызов функции push().

```
class MyArrayParent
{
protected:
 //сколько памяти выделено?
 int capacity;
 //количество элементов - сколько памяти используем
 int count;
 //массив
 double* ptr;
public:
 …

 //добавление в конец нового значения
 void push(double value)
 {
      if (count < capacity)
      {
           ptr[count] = value;
```

```
            count++;
        }

    }

    double& operator[](int index)
    {
        //перегрузка оператора []
        if (index < 0 || index >= count)
            ;
        return ptr[index];
    }
};

class MyArrayChild : public MyArrayParent
{
public:
    MyArrayChild(int Dimension = 100) :
MyArrayParent(Dimension) { cout << "\nMyArrayChild con-
structor"; }
    MyArrayChild(double* arr, int len) :
MyArrayParent(arr, len) { cout << "\nMyArrayChild con-
structor"; }

    ~MyArrayChild() { cout << "\nMyArrayChild destruc-
tor\n"; }

    //вставка элемента
    void InsertAt(double value, int index = -1)
    {
    if (index < 0)
    {
        push(value); //функция родительского класса
        return;
    }
    for (int i = count; i > index; i--)
        ptr[i] = ptr[i - 1];
```

```
 ptr[index] = value;
 count++;
}

};
```

В табл. 1 указан уровень доступа к полю / методу класса-родителя из методов класса-наследника в зависимости от того: 1) с каким ключевым словом объявлено поле / метод, к которому осуществляется доступ, 2) с каким ключевым словом объявлен наследник (слово после двоеточия в определении класса-наследника).

Таблица 1. Уровень доступа к полю / методу родителя из метода дочернего класса в зависимости от ключевого слова, с которым объявлен метод / поле в родителе (первый столбец), и ключевого слова, с которым объявлен наследник (второй, третий и четвёртый столбцы)

Спецификатор доступа в родительском классе	Уровень доступа в дочернем классе			
	private	protected	public	Ключевое слово, с которым объявлен класс-наследник
private	не доступен	Не доступен	Не доступен	
protected	private	protected	protected	
public	private	protected	public	

Конструкторы и деструкторы при наследовании

Рассмотрим наследование «оберток» для класса массива (урезанный аналог класса vector из библиотеки STL). Используем наследование с ключевым словом public (класс MyArrayParent – родитель, MyArrayChild – потомок / сын / дочерний класс):

```
class MyArrayParent
{
protected:
 //сколько памяти выделено?
```

```
    int capacity;
    //количество элементов - сколько памяти используем
    int count;
    //массив
    double* ptr;
public:
…
};

class MyArrayChild : public MyArrayParent
{…};
```

Класс MyArrayChild наследуется с ключевым словом public от класса MyParent. Следовательно, он имеет доступ к открытым (public) и защищенным (protected) переменным и функциям. Доступа к private-области нет. Поэтому вводится ещё один спецификатор доступа: protected. Такие переменные ведут себя как закрытые при обращении извне класса (например, код:

```
MyArrayChild c; c.count = 10;
```

не сработает, т.к. переменная count закрыта для доступа извне). При этом, для наследников эта переменная доступна, как будто они находятся внутри родителя.

Первое, что следует сделать в наследнике, – описать конструкторы и деструкторы. При создании объекта производного класса неявно создаётся объект родительского класса, что и позволяет обращаться к его полям и методам. Аналогично матрёшке в объекте класса MyArrayChild находится объект MyArrayParent (условное сравнение, т.к. базовый класс может быть абстрактным).

Более того, в работе конструктора производного класса можно напрямую сделать вызов конструктора базового класса. На примере класса MyArrayChild:

```
MyArrayChild(int Dimension = 100) : MyArrayParent(Dimension) { cout << "\nMyArrayChild constructor"; }
```

Аналогично можно ввести и другие конструкторы, указывая, какие конструкторы базового класса следует вызывать.

Итак, при использовании такой схемы наследования конструкторы вызываются в следующем порядке:

1. MyArrayParent (родитель)
2. MyArrayChild (потомок)

При работе деструктора объекта производного класса всё происходит в обратном порядке:

1. MyArrayChild (потомок)
2. MyArrayParent (родитель)

Таким образом, ещё раз можно заметить: как в матрёшке, внутри объекта производного класса присутствует объект базового класса. У него тоже срабатывает деструктор при освобождении памяти, выделенной под объект производного класса. При этом, конечно, работа деструктора родительского класса не связана с деструкторами производных классов (в ситуации, когда создаётся объект базового класса).

Особые ситуации, связанные с освобождением памяти, выделенный под объект, на который есть в наличии указатель, рассматриваются в пункте Указатель на базовый класс. Виртуальный деструктор.

Подмена функций базового класса в наследнике

Ещё одна полезная возможность, которую надо иметь ввиду при наследовании, – это подмена функций родителей. Если Вы считаете, что функция в унаследованном классе должна работать по-другому (например, в классе стало больше параметров, нужно по-другому делать вывод), Вы можете переопределить её, - для класса-наследника она будет работать по-своему, имея то же имя. При этом, интерфейс у обоих классов остаётся одинаковым, введения новых функций, имена которых нужно запоминать, не требуется. Например, заменим функцию print():

```
class MyArrayParent
{
protected:
 //сколько памяти выделено?
 int capacity;
 //количество элементов - сколько памяти используем
 int count;
 //массив
 double* ptr;
public:
```

```
    …

    virtual void print()
    {
        cout << "\nMyArrParent, size: " << count<<",
values: {";
        int i = 0;
        for (i = 0; i < count; i++)
        {
            cout << ptr[i];
            if (i != count - 1)
                cout << ", ";
        }
        cout << "}";
    }
   };
   class MyArrayChild : public MyArrayParent
   {
   public:
    MyArrayChild(int Dimension = 100) :
MyArrayParent(Dimension) { cout << "\nMyArrayChild
constructor"; }

    ~MyArrayChild() { cout << "\nMyArrayChild
destructor\n";  }



    void print()
    {
        cout << "\nPrint method in the MyArrayChild
object";
        MyArrayParent::print();
   }
   };
```

В производном классе добавим в вывод одну строку и вызовем функцию print() из базового класса. Чтобы различать функции базового класса и

наследника (имеющие одинаковый интерфейс), используется операция области видимости «::». Слева от двойного двоеточия указывается, функция какого класса используется.

Альтернативный подход – можно вообще заменить функцию базового класса в наследнике:

```
    class MyArrayChild : public MyArrayParent
    {
    public:

     MyArrayChild(int Dimension = 100) :
MyArrayParent(Dimension) { cout << "\nMyArrayChild
constructor"; }

     ~MyArrayChild() { cout << "\nMyArrayChild
destructor\n";  }

     void print()
     {
          cout << "\nMyArrChild, size: " << count << ",
values: {";
          int i = 0;
          for (i = 0; i < count; i++)
          {
               cout << ptr[i];
               if (i != count - 1)
                    cout << ", ";
          }
          cout << "}";
     }
    };
```

Эта версия функции аналогична варианту функции из базового класса. Меняется только сообщение о названии класса, которое выводится в консоль.

В итоге, получим две версии функции для разных классов, которые имеют одно имя. В представленном коде будут вызываться разные функции print():

```
     MyArrayChild p1;
```

```
MyArrayChild p2;
…

p1.print();
p2.print();
```

При работе с указателем на базовый класс требуется виртуальность подменяемых функций. Подробнее см. в пункте «Указатель на базовый класс. Виртуальные функции».

Множественное наследование. Ключевое слово virtual при наследовании

В рамках C++ разрешено наследовать новый класс от нескольких классов. В целом, подход тот же:

```
class Derived : public Base1, public Base2
{…};
```

В методах производного класса есть доступ к методам и полям всех базовых классов в зависимости от уровня доступа к функции / переменной в самом базовом классе и ключевого слова, с которым организовано наследование.

В целом, наследование организовано таким образом, что внутри объекта производного класса присутствует неявно объект базового класса. Как раз к нему и происходит обращение, когда наследник использует методы или поля родителя. При этом можно организовать такую схему наследования, что один и тот же класс будет родителем какого-либо класса несколько раз (будет несколько раз встречаться в графе наследования). Пример такого графа наследования представлен на рис. 1 (стрелки указывают направление от наследника к родителю, поэтому A1 – родитель для классов B1, B2).

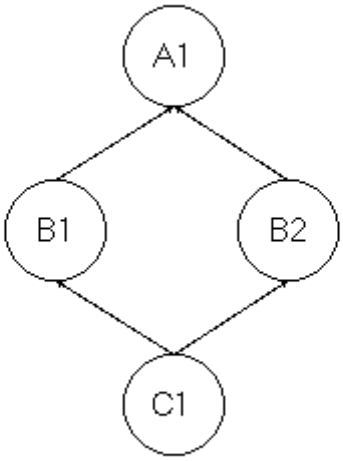

Рисунок 1. Граф множественного наследования (A1 – базовый класс для B1 и B2, класс C1 – наследник классов B1 и B2)

Учтём, что объект базового класса должен храниться в объекте наследника. Тогда объект класса A1 должен был бы в виде двух экземпляров храниться в объекте класса C1 (в роли «дедушки по линии B1» и «дедушки по линии другого родителя B2»). В таком случае возникает неоднозначность: при вызове полей и методов родителя A1, к какому именно объекту идёт обращение, а также нужно ли вообще поддерживать два объекта этого класса в потомке?

Решение этой ситуации выглядит следующим образом. Вводится виртуальное наследование (не путать с виртуальной функцией). При этом, гарантируется, что в любом классе потомка будет содержаться только по одному объекту любого родительского класса. Технически при описании класса-наследника указывается ключевое слово virtual.

Рассмотрим участок кода, соответствующий схеме наследования, изображённой на рис. 1.

```
class A1
{
  protected:
    int a1;
  public:
    A1(int arg) {a1 = arg; cout<<"\nA1 Constructor";}
    ~A1() {cout<<"\nA1 Destructor";}
```

```
        A1() {a1 = -1; cout<<"\nA1 Constructor0";}

        int GetB1() {return b1;}
    };

    class B1: virtual public A1
    {
       protected:
         int b1;
       public:
         B1(int arg, int arg1): A1(arg1) {b1 = arg;
cout<<"\nB1 Constructor";}
         ~B1() {cout<<"\nDerived Destructor";}
    };

    class B2: virtual public A1
    {
       protected:
         int b2;
       public:
         B2(int arg, int arg1): A1(arg1) {b2 = arg;
cout<<"\nB2 Constructor";}
         ~B2() {cout<<"\nDerived Destructor";}
    };

    class C1: public B1, public B2
    {
       protected:
         int c1;
       public:
         C1(int arg, int arg1, int arg2, int arg3, int
arg4): B1(arg1, arg2), B2(arg3, arg4) {c1 = arg;
cout<<"\nC1 Constructor";}
         ~C1() {cout<<"\nC1 Destructor";}
    };
```

При создании объекта типа C1 есть возможность передать разные значения для конструктора класса A1 через родителя B1 и через родителя B2 (что и

делается – обратите внимание на параметры arg1 – arg4 в конструкторе класса C1). Если бы такая возможность присутствовала, в этом коде удалось бы передать для одной и той же переменной родителя (а мы уверены, что в объекте класса C1 есть только один объект класса A1 при использовании виртуального наследования) разные значения. ***Чтобы избежать подобной неоднозначности, при работе такой схемы наследования (рис. 1) во время создания объекта типа C1 будет вызываться конструктор по умолчанию класса A1, какой бы вызов Вы ни написали в конструкторах классов C1, B1, B2.***

Следует отметить, что от использования множественного наследования в новых объектно-ориентированных языках (например, .NET, Java) отказываются, заменяя его на наследование только от одного родителя и применение интерфейсов (под интерфейсом в такой ситуации понимают набор функций, которые должны быть определены в наследнике).

Указатель на базовый класс. Виртуальные функции

При использовании указателя на базовый класс может возникать двойственность. Указатель на базовый класс может быть направлен на любой объект производного класса. Но какая функция будет вызываться по такому указателю: из базового класса (ему соответствует тип указателя) или из производного класса (указатель направлен на объект этого типа). Возможны оба варианта, но, конечно, больше всего нас интересует случай, когда есть возможность по указателю на базовый класс вызвать функцию из производного класса.

Рассмотрим пример наследования классов-оберток для массива. Введём функцию print(), варианты которой присутствуют и в базовом классе, и в производном классе.

```
class MyArrayParent
{
protected:
 //сколько памяти выделено?
 int capacity;
 //количество элементов - сколько памяти используем
 int count;
 //массив
 double* ptr;
public:
```

```
    …

    virtual void print()
    {
        cout << "\nMyArrParent, size: " << count<<",
values: {";
        int i = 0;
        for (i = 0; i < count; i++)
        {
            cout << ptr[i];
            if (i != count - 1)
                cout << ", ";
        }
        cout << "}";
    }
};

class MyArrayChild : public MyArrayParent
{
public:
    MyArrayChild(int Dimension = 100) :
MyArrayParent(Dimension) { cout << "\nMyArrayChild
constructor"; }

    ~MyArrayChild() { cout << "\nMyArrayChild
destructor\n";  }

    void print()
    {
        cout << "\nPrint method in the MyArrayChild
object";
        MyArrayParent::print();
    }
};
```

Если используются объекты классов MyArrayParent, MyArrayChild, то никаких разночтений нет. Вызываются функции, соответствующие типу объекта:

```
    MyArrayChild c; c.print() //вызов метода класса MyArrayChild
    MyArrayParent par; p.print() //вызов метода класса MyArrayParent
```

Указатель на базовый класс может быть направлен на объект производного класса:

```
    MyArrayParent* p = &c;
```

Т.к. функция print() ***не отмечена как виртуальная***, при её вызове выбор нужной версии целиком основывается на типе указателя. Он имеет тип родителя MyArrayParent, поэтому вызывается ***родительская версия***:

```
    p->print(); //работает функция родителя MyArrayParent (функция print() не виртуальная)
```

Чтобы заработала «магия» наследования, сделаем функцию print() виртуальной.

```
    class MyArrayChild : public MyArrayParent
    {
    public:
     MyArrayChild(int Dimension = 100) : MyArrayParent(Dimension) { cout << "\nMyArrayChild constructor"; }

     ~MyArrayChild() { cout << "\nMyArrayChild destructor\n";  }

     virtual void print()
     {
          cout << "\nPrint method in the MyArrayChild object";
          MyArrayParent::print();
    }
    };
```

Теперь при применении указателя типа базовый класс:

```
MyArrayChild c;
MyArrayParent* p = &c;
p->print();  //работает функция потомка MyArrayChild (функция print() виртуальная)
```

функция print() виртуальная, переопределяется в потомке MyArrayChild, по указателю типа базовый класс MyArrayParent, который направлен на объект производного типа MyArrayChild, вызывается версия функции из потомка.

Указатель на базовый класс. Виртуальный деструктор

Как видно из предыдущего пункта, есть некоторая неоднозначность в работе указателя на базовый класс (как именно она обрабатывается, указано выше). Рассмотрим ситуацию, когда в наличии указатель на базовый класс, который направлен на объект производного класса (тип объекта может быть не известен в этой точке кода). Что будет происходить при попытке очистки памяти?

```
MyArrayChild c;
MyArrayParent* p = &c;
 //другая функция, объект c не доступен, но есть указатель p
delete p;
```

В рамках кода, представленного выше, ***будет вызываться только деструктор класса MyArrayParent***. В общем случае, вызываются только деструкторы класса, соответствующего типу указателя, и деструкторы его родителей.

Чтобы вызывались все деструкторы по схеме наследования (вдобавок к деструкторам класса, соответствующего типу указателя, и деструкторам его родителей), ***нужно объявить деструктор класса MyArrayChild*** (класса, на объект которого направлен указатель) ***виртуальным. В целом, рекомендуется деструкторы всех классов в схеме наследования делать виртуальными.***

```
class MyArrayParent
```

```
    {
    protected:
     //сколько памяти выделено?
     int capacity;
     //количество элементов - сколько памяти используем
     int count;
     //массив
     double* ptr;
    public:
    …
     virtual ~MyArrayParent() { cout << "\nMyArrayParent
destructor\n";  }

    };


    class MyArrayChild : public MyArrayParent
    {
    public:
     …

     virtual ~MyArrayChild() { cout << "\nMyArrayChild
destructor\n";  }

    };
```

Указатель на базовый класс. Чисто виртуальные функции. Абстрактный класс

При реализации некоторой схемы наследования вполне может оказаться, что классы, которые находятся на вершине схемы наследования, являются слишком общими, и не ясно, как реализовать некоторые функции. Понимание, как работают определённые действия, появляется только при реализации наследников. Можно было бы оставить на уровне наследников возможность добавлять новые функции, но это привело бы к разнообразию интерфейсов у потомков таких классов. Чтобы интерфейс был единообразным, нужно объявить такие функции, но не давать их реализацию.

Для реализации такой цели применяются чисто виртуальные функции. В отличие от функций обычного класса-родителя, они не обладают телом. Тело

должно быть задано в производном классе. Рассмотрим класс Фигура – базовый класс для геометрических фигур. Введём массив для хранения параметров (для простоты положим, что это длины сторон и количество сторон фигуры). Но пока вид фигуры не определён точно, посчитать её площадь не возможно. Функция подсчёта площади getArea() чисто виртуальная – её нужно переопределить в потомке.

```
class Figure
{
      protected:
          double* sides;
          int Number;
      public:
          Figure(int NumberOfSides)
          {
               Number =  NumberOfSides;
               sides = new double[Number];
          }
          ~Figure() {delete[] sides;}

          virtual void SetSide(int index, double val-
ue) {if(index>=0 && index<Number) sides[index] = value;}
          virtual double GetSide(int index) {return
sides[index];}
          virtual double getArea() = 0;
};
```

Класс, обладающий хотя бы одной чисто виртуальной функцией, – абстрактный. Объект абстрактного класса создать нельзя. Указатель типа абстрактного класса создать можно, т.к. он может быть направлен на объект производного класса, где чисто виртуальные функции переопределены.

Например, унаследуем класс для квадрата от класса Фигуры. Формула подсчёта площади определена, поэтому переопределяем чисто виртуальную функцию getArea(). Класс Square перестаёт быть абстрактным – объект этого типа создать можно. Ещё раз обратим внимание: при переопределении функций базового класса соблюдается единый интерфейс (функции производного класса называются так же, как и в родительском классе).

```
class Square: public Figure
{
	public:
		Square():Figure(1)
		{
			int i=0;
			for(i=0;i<Figure::Number;i++)
			{
				Figure::sides[i] = 0;
			}
		}
		~Square() {}
		double getArea()
		{
			return sides[0]*sides[0];
		}
};

int main()
{
 Figure *F; //объект типа Figure создать нельзя, но
указатель Figure* объявить можно

 Square S; //Square уже не абстрактный класс
 F = &S;
 F->SetSide(0, 2);
 double a = F->getArea();
 cout<<"\nSquare area: "<<a<<"\n";…
}
```

Для более подробного исследования наследования рекомендуем обратиться к источникам [1 – 6].

ПОЛИМОРФИЗМ

Перегрузка функций

Под перегрузкой функций подразумевается возможность создавать несколько методов класса (функций, принадлежащих классу), которые имеют одинаковое имя, но отличаются по списку параметров.

При вызове такой функции нужная версия определяется по списку параметров, поэтому при таком выборе не должна возникать неоднозначность по типу передаваемых параметров. По команде вызова должна точно определяться нужная версия функции. Иначе, будет возникать ошибка.

Например, в «обертке» строки – классе BaseString – введём несколько функций поиска подстроки IndexOf(). Одна версия будет принимать объект искомой подстроки BaseString, другая – динамический массив char*, третья – один символ типа char.

```
class BaseString
{
protected:
 char* p; int len; int capacity;
public:
 BaseString(char* ptr)
 int IndexOf(char* s);
int IndexOf(char s);
int IndexOf(BaseString s);};
```

В следующем участке кода вызываются три версии этой функции:

```
BaseString s1("test");
int n1 = s1.IndexOf('t');
int n2 = s1.IndexOf("te");
int n3 = s1.IndexOf(s2);
```

Перегрузка операций

Перегрузка операций – это возможность ввести для своих классов операции, которые присутствуют в языке программирования. В некоторых объектно-ориентированных языка позволительно не только переопределять существующие операции, но и вводить свои. В C++ можно переопределять только такие операции, которые уже есть в языке программирования. При этом, их арность (число операндов), приоритет, ассоциативность не меняются. Кроме удобства (когда объекты классов Дробь или Комплексное число приобретают реализацию арифметических операций), перегрузка нужна, например, для того, чтобы объект пользовательского класса можно было вводить/выводить в поток/файл (перегрузка операций >>, <<), чтобы правильно работала операция присваивания = (см. конструктор копий) и т.д.

В C++ не перегружаются операции:

1. Тернарный оператор «?:»
2. Доступ к полям класса по операции «точка» . , доступ к полям класса по указателю «*.»;
3. Операции sizeof, typeid и операторы преобразования типа.

Чаще всего, логически перегрузку операций делят на несколько типов по: 1) числу операндов (унарные, бинарные), 2) по типу возвращаемого значения (логические, присваивания и другие), 3) по особенностям оператор-функции (перегрузка с помощью метода или дружественной функции).

Перегрузка бинарных операций с помощью методов

Функция, которая должна сработать в определённый момент (обработать арифметическую операцию) должна иметь особенную структуру, так же, как конструкторы и деструкторы. Функции, которые переопределяют операции, имеют особое название: operator&, где & - переопределяемая (перегружаемая) операция. Например, operator+ для переопределения бинарного или унарного плюса, operator= для переопределения операции присваивания, operator== для перегрузки операции проверки на равенство и т.д. В остальном это обычные функции: в их прототипе есть тип возвращаемого значения и передаваемые аргументы.

Следует иметь ввиду: при переопределении операции происходит вызов метода по следующей схеме (для класса Дробь Fraction):

```
Fraction H = G * F;     =>      G.operator*(F);
```

В объектно-ориентированном программировании чаще всего функции – это методы внутри класса. Здесь у объекта G типа Fraction происходит вызов метода operator*. Первый аргумент уже фактически передан в функцию – это объект, внутри которого работает метод. Второй операнд надо передать в функцию в виде параметра. Внутри класса Fraction создаём открытый метод:

```
Fraction operator*(Fraction F)
{
    Fraction Res;
    Res.numerator = numerator* F.numerator;
    Res.denominator = denominator* F.denominator;
    return Res;
}
```

Здесь переопределена операция умножения дробей стандартным образом. Для обращения к полям первого операнда (в примере выше – G) не требуется никаких дополнительных инструментов: просто numerator, denominator, т.к. фактически операция выполняется внутри объекта G, первого операнда. Для обращения к полям второго операнда используется его имя в функции и операция точка: F.numerator, F.denominator. Обратим внимание, что доступ к ним из этой функции есть, даже учитывая их принадлежность к области private. При передаче параметра в функцию можно использовать ссылку, чтобы аргумент не копировался.

Разрешено перегружать оператор-функции. В нашем примере можно добавить функцию умножения дроби на число (вдобавок к указанной выше функции умножения на дробь):

```
Fraction operator*(Fraction F)
 {
      Fraction Res;
      Res.numerator = numerator* F.numerator;
      Res.denominator = denominator* F.denominator;
      return Res;
 }
Fraction operator*(int value)
 {
      Fraction Res;
      Res.numerator = numerator* value;
      Res.denominator = denominator;
      return Res;
 }
```

Аналогично можно переопределить любую другую бинарную операцию в другом классе. Особым случаем является перегрузка постфиксного оператора ++ (его перегрузка получает параметр, но сам оператор остаётся унарным, см. Перегрузка унарных операций с помощью методов).

Также бинарными операциями, о которых речь пойдёт ниже, являются операции ввода/вывода в поток, логические операции, присваивание.

Обратим внимание на перегрузку операций скобки. Операция [ ] принимает один параметр, поэтому в «обертке массива» MyArray (содержит динамический массив ptr, количество заполненных элементов count) она имеет смысл доступа к i-му элементу массива ptr и вид:

```
class MyArray
{
protected:
 //сколько памяти выделено?
 int capacity;
 //количество элементов - сколько памяти используем
 int count;
 //массив
 int* ptr;
public:
int& operator[](int index)
 {
      //перегрузка оператора []
      if (index < 0 || index >= count)
          ;
      return ptr[index];
 }
…
}
```

Возвращать элемент массива по ссылке требуется для того, чтобы работал не только запрос значения в массиве (например, при выводе):

```
MyArray arr;
…
cout << arr[5];
```

но и операция, в которой значения массива меняется, например, в присваивании:

```
arr[3] = 5;
```

Оператор [ ] принимает только один параметр – обычно индекс массива. Если ведётся работа с многомерными массивами, индексов несколько, а перегрузка [ ] принимает только один параметр, это может быть не удобно. В таком случае, можно воспользоваться операцией ( ), которая чаще

воспринимается как подмена вызова функции, но здесь может приобрести и смысл альтернативного оператора индекса. Например, для класса Матрица:

```
class Matrix
{
protected:
 double** ptr;
 int height;
 int width;
public:
 double& operator()(int row, int column)
 {
      return ptr[row][column];
 }
…
}
…
BaseMatrix M1;
M1[1, 1] = 5;
```

Перегрузка унарных операций с помощью методов

Хотя чаще всего работа идёт с бинарными операциями, иногда приходится работать и с унарными операциями. Например, унарные версии операций + и – в коде выглядят так:

```
-M1;
+M1;
```

При перегрузке с помощью методов оператор-функция, соответствующая унарной операции, не будет принимать операндов. Например, в случае перегрузки унарного минуса код

```
-M1;
```

преобразуется к виду:

```
M1.operator-();
```

Например, перегрузка унарного минуса для класса Дробь выглядит так:

```
Fraction operator-()
 {
      Fraction Res;
      Res.numerator = -numerator;
      Res.denominator = denominator;
      return Res;
 }
```

Особым образом переопределяются операции инкремента ++ и декремента -- в связи с тем, что у них есть две версии: префиксная (++p) и постфиксная (p++). Префиксная версия (++p) переопределяется так же, как и обычная унарная операция:

```
Fraction operator++() {…}
```

Чтобы отличать от неё реализацию постфиксного оператора (p++), в его оператор-функцию добавляют один фиктивный параметр, не участвующий в действиях:

```
Fraction operator++(int v) {…}
```

Перегрузка бинарных операций с помощью дружественных функций

У подхода, описанного выше, есть один существенный недостаток. Бинарные операции, в которых первый операнд – объект пользовательского класса, а второй – это объект стандартного типа или пользовательского класса, перегружаются просто (см. Перегрузку бинарных операций с помощью методов). Проблемы возникают в случае, когда первым операндом является объект стандартного типа. Например, для дроби Fraction операция:

```
Fraction H = 2 * F;
```

- предполагает наличие у объекта 2 типа int операции умножения, «понимающей», что такое дробь Fraction. Такое было возможно, только пока программист сам создавал новый тип данных и задавал его методы – оператор-функции. Здесь потребовалось бы «отредактировать тип int», что не возможно:

```
H = 2 * F;    => 2.operator*(F);
```

Для случая произвольной бинарной операции & вида a&b, где a – объект стандартного типа, b – объект пользовательского класса приходится создавать функции, которые не принадлежат пользовательскому классу. При этом, стандартный подход состоит в том, что: 1) им нужно иметь доступ к закрытым переменным класса, не будучи его методами, 2) открытого интерфейса (геттеров и сеттеров, которые позволят менять значения закрытых переменных) может не быть в наличии. Поэтому такие функции объявляются дружественными (функция, которая нарушает инкапсуляцию, имея доступ к закрытым private полям и методам класса, не являясь его методом), хотя при наличии открытого интерфейса доступа к элементам класса, можно обойтись «без дружественности». Такие функции оформляются следующим образом.

Шаг 1. Внутри класса задаётся прототип функции, и она отмечается дружественной для класса (в прототипе функции нет тела, но есть вся информация, необходимая для её вызова: список параметров (достаточно указать их типы и порядок), название, тип возвращаемого значения.

Шаг 2. Вне класса задаётся описание функции.

```
class Fraction
{
private:
 //поля: числитель и знаменатель
 int numerator;
 int denominator;
public:
…
friend Fraction operator*(int n1, Fraction F2);
…
};
Fraction operator*(int n1, Fraction F2)
{
 Fraction Res;
 Res.numerator = F2.numerator * n1;
 Res.denominator = F2.denominator;
 return Res;
}
```

Сама реализация находится вне класса. Основная часть кода, конечно, не имеет особенностей. Отметим: «дружественность» функции нужна для доступа к полям числителя и знаменателя извне класса – из функции operator*. Если для класса реализован открытый интерфейс: например, геттеры и сеттеры для числителя и знаменателя (или геттеры и конструктор, принимающий значения числителя и знаменателя), то дружественность не требуется, и функция объявляется вне класса Дробь без специальных указаний внутри.

```
Fraction operator*(int n1, Fraction F2)
{
 Fraction Res(F2.getNumerator() * n1,
F2.getDenominator());
 return Res;
}
```

Перегрузка операций с помощью дружественных функций особенно важна для операций, где первым операндом является объект, который встроен в C++ и не меняется. Например, при перегрузке операций ввода/вывода в поток >>, << первым операндом всегда является объект потока, внутрь которого внести новую оператор-функцию нельзя. Поэтому для переопределения операций >>, << применяются дружественные функции (см. Перегрузку операций ввода/вывода в поток).

Перегрузка логических операций

В ходе выполнения логических операций операнды проверяются на справедливость какого-либо утверждения. Например: верно ли, что первый операнд больше второго, либо равен второму и т.п. Таким образом, результатом операции является не объект пользовательского класса или один из параметров стандартного типа, а логическое значение типа bool. Иногда для обратной совместимости тип этих оператор-функций – int. Логические операции ==, !=, <, >, <=, >= можно переопределять по одному принципу. Например, для комплексного числа Complex проверка на равенство == будет перегружаться следующим образом:

```
class Complex
{
private:
 double real;
```

```
 double image;
public:
 …

 bool operator==(Complex& C2)
 {
      return real == C2.real && image == C2.image;
 }
};
```

Также можно возвращать результат типа int:

```
int operator==(Complex& C2)
 {
      return real == C2.real && image == C2.image;
 }
```

В рамках C++ нельзя переопределять те операции, которых нет в языке. Но можно вводить операции, которых «нет в обычной математике». Если в этом есть необходимость, можно переопределить операцию сравнения для комплексного числа:

```
int operator<(Complex& C2)
 {
      return …;
 }
```

Перегрузка операции присваивания

Для классов, не содержащих указатели, в такой оператор-функции всё просто: надо просто присвоить всем полям копии значения полей оригинала. Например, для класса Комплексное число (поля real, image) производится просто присваивание полей:

```
 Complex& operator=(Complex& C2)
 {
      real = C2.real;
      image = C2.image;
      return *this;
 }
```

Обратим внимание: поля того объекта, что стоит слева от =, принимают значения полей того объекта, что стоит справа: `a = b => a.operator=(b)`. Если Вы не реализуете эту функцию, то она будет сгенерирована автоматически: поля буду присваиваться друг другу так же, как сделано в примере функции выше.

Возвращать один из операндов требуется, чтобы правильно обрабатывались выражения, в которых операция = встречается много раз. Например, в операции:

```
a = b = c+5;
```

сначала сработает вычисление выражения `c+5` (поэтому операция = работает справа налево), затем производится присваивание `b = c+5`, затем результат этой операции присваивается a: `a = b`. Если бы тип возвращаемого значения у функции operator= был бы void, то последнее действие `a = b` не сработало бы. Принято возвращать тот объект, который находится слева от =. Т.к. это тот объект, внутри которого работает метод operator=, используется ключевое слово this.

В случае реализации операции для классов, в которых есть указатели, всё сложнее. Ситуация та же, что и при работе над конструктором копий. Пусть в классе MyClass есть указатель p (динамический массив). При выполнении присваивания:

```
MyClass a;
MyClass b;
…
a = b;
```

будет произведено присваивание `a.p = b.p`. Если в массиве *p* объекта *a* содержалась информация, то она не просто потеряна (что соответствует логике выполнения операции), но происходит утечка памяти (участок памяти, на который был направлен указатель *a.p* до выполнения операции = потерян и не освобождён). Но последствия ещё хуже. Если один из объектов исчезает, а второй продолжает существовать, то для первого сработает деструктор, который очистит ту же самую память, с которой связан указатель *p* во втором объекте (например, объект *a* передан в функцию, *b* – объект, существующий только внутри функции, для него сработает деструктор при окончании работы

функции, а объект *a* вернётся из функции с уже ошибочным указателем *p*). Когда произойдёт попытка доступа к памяти в оставшемся объекте или сработает его деструктор, произойдёт ошибка, которую очень тяжело «отловить» (доступ к освобождённой памяти). В такой ситуации в этой функции надо выделять новый участок памяти и копировать данные туда. Для класса Строка (*p* – динамический массив, содержащий символы) перегрузка операции = почти совпадает с работой конструктора копий (кроме наличия возвращаемого значения) и выглядит так:

```
class BaseString
{
private:
 char* p;
 int len;
 int capacity;
public:
 …
 char& operator[](int i) {return p[i];}

 BaseString& operator=(BaseString& s)
 {
      cout<<"\nBase Operator = \n";
      len = s.Length()+1;
      p = new char[s.capacity];
      capacity = s.capacity;
      for(int i=0;i<s.Length();i++)
      {
          p[i] = s[i];
      }
      p[len] = '\0';
      return *this;
 }

 BaseString(BaseString& s)
 {
      cout<<"\nBase Copy Constructor\n";

      len = s.Length();
```

```
        p = new char[s.capacity];
        capacity = s.capacity;
        for(int i=0;i<s.Length() - 1;i++)
        {
            p[i] = s[i];
        }
        p[len-1] = '\0';
    }
};
```

Перегрузка операций ввода/вывода в поток

Операцию ввода/вывода в поток надо перегружать при помощи дружественной функции. Слева находится объект типа ostream/istream (или их наследники). Внутри этого объекта не известно о существовании пользовательских типов данных. Поэтому по той же схеме, которая имела место для бинарных операций, где первым операндом является объект стандартного типа (здесь – объект потока), а вторым – объект пользовательского типа данных, нужно использовать дружественную функцию, принимающую оба операнда (см. пункт Перегрузка бинарных операций с помощью дружественных функций).

Для класса Комплексное число введём две дружественные функции, перегружающие операции << и >>.

```
class Complex
{
private:
 double real;
 double image;
public:

 …

 friend ostream& operator << (ostream& ustream, Com-
plex obj);
 friend istream& operator >> (istream& ustream, Com-
plex& obj);
};
```

Используется ссылка на поток, чтобы не создавать копии потоков ввода/вывода. Так же, как и при реализации операции =, нужно возвращать ссылку на поток, чтобы работали операции с вводом/выводом множества переменных. Например, для команды:

```
cout<<a<<b;          => 1) operator<<(cout, a), 2) operator<<(cout, b)
```

сначала выводится переменная *a*, и результат этой операции должен быть ссылкой на поток, которая может использоваться для вывода объекта *b*. Отдельно от класса пишем реализации функций:

```
ostream& operator << (ostream& ustream, Complex obj)
{
 if (obj.Real() == 0.0)
 {
      if (obj.Image() != 0.0)
          ustream << obj.Image() << "i";
      else
          ustream << "0.0";
 }
 else
 {
      if (obj.Image() > 0.0)
          ustream << obj.Real() << " + " <<
obj.Image() << "i";
      if (obj.Image() < 0.0)
          ustream << obj.Real() << " - " << -
obj.Image() << "i";
      if (obj.Image() == 0.0)
          ustream << obj.Real();
 }

 return ustream;
}

istream& operator >> (istream& ustream, Complex& obj)
{
```

```
    double a, b;
    ustream >> a;
    obj.SetReal(a);
    ustream >> b;
    obj.SetImage(b);
    return ustream;
}
```

При выполнении операции >> объект класса будет меняться, поэтому он подаётся по ссылке.

Те же операции будут работать и в случае ввода/вывода в файл в связи с наследованием (см. рис. 2, см. пункт «Наследование»).

Классы файловых потоков ofstream (запись в файл), ifstream (чтение из файла) являются наследниками потоков ostream и istream соответственно (см. рис. 2). Согласно правилам наследования, чтение и запись комплексного числа в файл уже реализованы: представленные в примере 1 функции работают и на ввод/вывод в консоль, и на ввод/вывод в файл.

Но всё же есть тонкость. Операция вывода в консоль для числа, например, (1, -3) должна дать:

1-3i,

а то же действие при записи в файл должно туда записать два значения: 1 -3.

Если в файл будет записываться такая же строка, что и выводится в консоль (вида 1-3i), то при чтении файла придется производить разбор этой строки (которая может иметь множество разных видов в зависимости от комбинации значений действительной и мнимой части), а проще прочитать два числа.

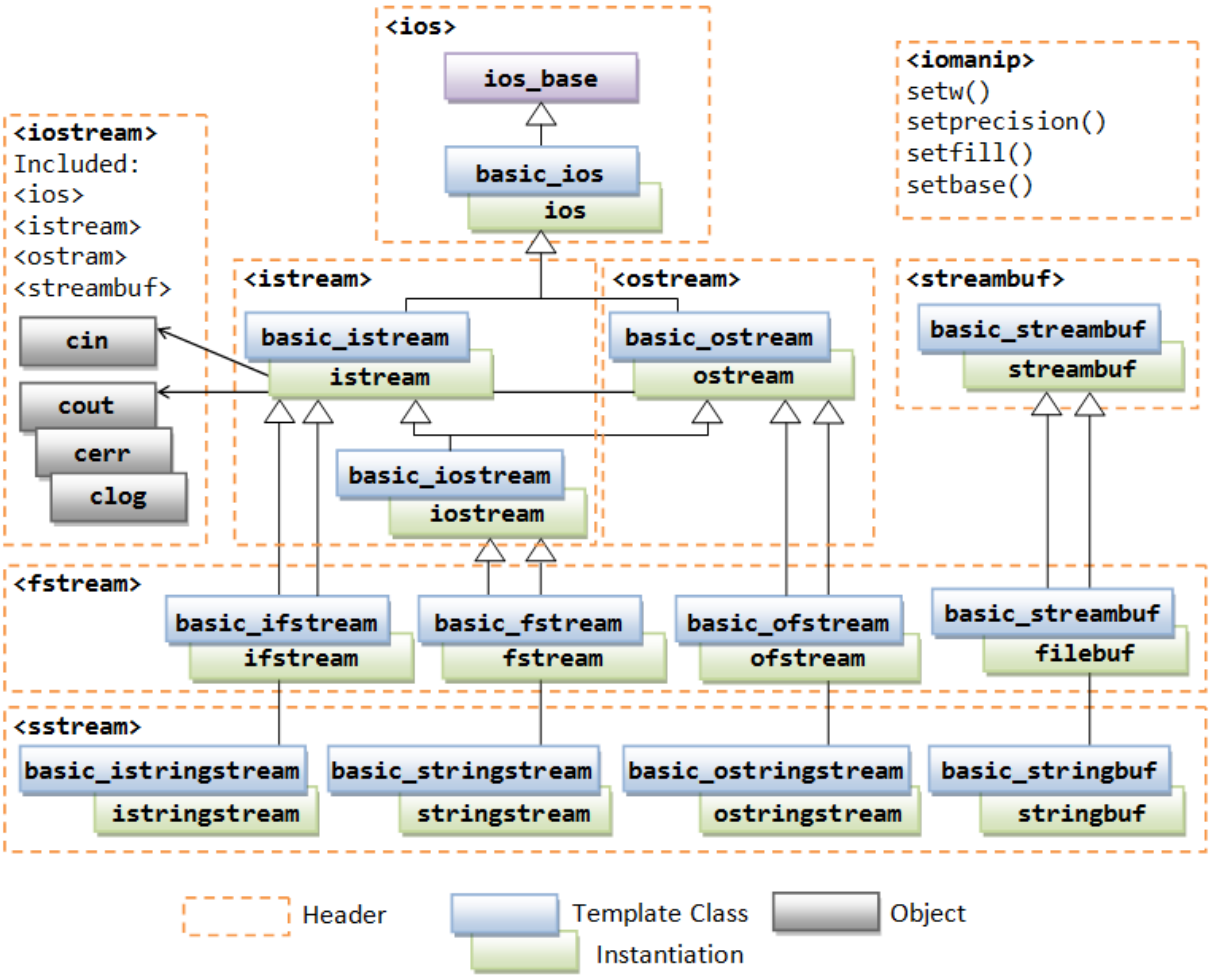

Рисунок 2. Иерархия классов в системе ввода/вывода в C++

Теоретически достаточно было бы дописать ещё один оператор вывода в файл (обратите внимание на тип потока ofstream, здесь f означает «файловый»):

```
    friend ofstream &operator << (ofstream &ustream, Complex obj);
    …
    ostream& operator << (ofstream& ustream, Complex obj)
    {
          ustream << obj.real << " " << obj.image;
          return ustream;
    }
```

Тогда для файлового вывода работает одна версия (для класса-наследника ofstream), а для вывода в консоль своя (из примера 1).

К сожалению, на практике этот подход не работает, т.к. «за кулисами» производится преобразование между типами ostream/ofstream.

Для различия между выводом в консоль и файл придётся использовать другой подход. Как было показано в предыдущем занятии, указатель типа базового класса может быть направлен на объект производного класса и может вызывать виртуальные функции из производного класса. Поэтому имея такой указатель в коде, мы не можем точно знать, на объект какого типа он направлен: на объект базового класса или на объект одного из наследников. Здесь ofstream (файловый поток) – наследник ostream.

Чтобы определить тип объекта ustream, используем команду typeid() (см. пункт Наследование). В её арсенале, в частности, есть метод name(), возвращающий имя класса, объектом которого является параметр.

Результат команды `typeid(X).name()` в случае, если X – объект потока ostream (cout):

```
    class                    std::basic_ostream<char,struct
std::char_traits<char> >
    В случае ofstream:
    class                  std::basic_ofstream<char,struct
std::char_traits<char> >
```

Проще не работать с этой строкой целиком, а использовать проверку:

```
    if              (typeid(ustream).name()              ==
typeid(ofstream).name())
```

Если условие выполнено, эта функция вызвана для записи в файл, иначе – для вывода в консоль. Такое разделение и реализуем:

```
    ostream& operator << (ostream& ustream, Complex obj)
    {

     if              (typeid(ustream).name()              ==
typeid(ofstream).name())
     {
    //файловый вывод
         ustream << obj.real << " " << obj.image;
         return ustream;
     }
     //консольный вывод
```

```
 …
 return ustream;
}
```

Что же касается самих операций работы с файлами в самом простом виде, повторяется интерфейс работы с консолью.

```
ofstream fout; fout.open("test.txt", ios::out);
 if (fout)
 {
      fout << z;
      fout.close();
 }
 ifstream fin("test.txt");
 if (fin)
 {
      try
      {
           fin >> z;
           fin.close();
           cout << "\n" << z;
      }
      catch (...)
      {
           cout << "\nException: failed to read file";
      }
}
```

Как видно из этих примеров, путь к файлу, который нужно открыть, можно передать прямо в конструкторе или затем использовать команду open(). Кроме пути к файлу, при необходимости настроим режим работы с ним (например, ios::out – только запись в файл; ios::app – записывать информацию не в начало файла, а переместиться в конец (append); ios::bin – работа в двоичном режиме; ios::truncate – очистить содержимое файла перед записью и т.д.).

Проверка, удалось ли открыть файл: `if(fout)` или `if(fout!=NULL)`.

Не забываем в конце работы с файлом закрыть его: `fout.close()`.

Запись в файл числа z: `fout<<z;`

Чтение из файла информации и его запись в z: `fin>>z;`

Если требуется перемещаться по файлу при чтении или записи, можно использовать команды seekp() и родственные ей.

Шаблоны функций

Зачастую операции, которые необходимо запрограммировать, действуют одинаково для различных типов входных данных. Например, при обмене значений элементов местами их тип не влияет на логику:

```
void swapargs(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Код функции, которая будет работать с объектами типа double или Fraction не будет отличаться нигде, кроме типа параметров a, b и временной переменной temp. В рамках C можно было бы использовать макрос, введённый с помощью директивны #define, где типа не было бы вовсе. В C++ появляется ещё одна возможность – создать шаблон функции. Параметром шаблона как раз становится тип входных параметров. Для того, чтобы указать, что речь идёт не об обычной функции, а о шаблоне функции, используется ключевое слово template и добавляется информация о параметризуемом типе. Например, та же функция для обмена значений параметров местами в виде шаблона выглядит так:

```
template <class X> void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

При вызове по типу параметров функции становится ясен тип X. При этом, никаких дополнительных сведений при вызове шаблона функций не требуется. Ниже приведён код, в котором используется функция swapargs( ):

```
int i = 10, j = 20;
float x = 10.1, y = 23.3;
swapargs (i, j); // обмен целых
swapargs (x, y) ; // обмен действительных
```

Если типов-параметров несколько, они передаются через запятую. Например:

```
template <class X, class Y> void swapargs(X &a, Y
&b);
```

Большинство базовых операций, которые часто используются в программировании, могут быть реализованы с помощью шаблонов функций (для их логики не важен тип обрабатываемых параметров): поиск значения в массиве, обмен элементов местами, замена одного значения другим, большинство сортировок и т.д.

При необходимости определить, какой именно тип скрывается за X или Y, можно воспользоваться оператором typeid (см. пункт Динамическое определение типа. Оператор typeid).

Шаблоны классов

Аналогично при создании классов часть полей может иметь произвольный тип. Особенно это важно для классов-контейнеров (для хранения данных). Обычно тип данных, которые хранятся в контейнере, не важен для логики методов контейнера. Например, для класса, подготовленного для реализации операций на массиве (поиск значения, обмен значений и т.д.), не важно, какого типа данные хранятся в нём. Эти действия работают одинаково и для массивов целых чисел, и в случае вещественных чисел, и в случае объектов пользовательских классов. Аналогично ситуации с шаблонами функций можно ввести шаблон классов. Например, шаблон «обертки» для массива:

```
template<class T>
class MyArrayTemplate
{
protected:
 //сколько памяти выделено?
 int capacity;
 //количество элементов - сколько памяти используем
```

```
 int count;
 //массив
 T* ptr;
public:
…
};
```

При создании объекта шаблона класса приходится указывать тип-параметр (в отличие от ситуации с шаблоном функции, где тип можно было точно определить по вызову функции):

```
MyArrayTemplate<double> arr;
int i = 0;
for (i = 10; i >= 0; i--)
arr.push(i + 1);
```

Отметим важную особенность: все методы шаблона класса являются шаблонами функций (хотя для большинства IDE не требуется это указывать явно).

Динамическое определение типа. Оператор typeid

В C++ может иметь место несколько ситуаций, когда в определённом месте кода не известен тип некоторой переменной. В рамках этого пособия такое может произойти: 1) в шаблоне функции (которая может быть методом в шаблоне класса), 2) при использовании указателя на базовый класс (он может быть направлен на объект производного класса, но какого именно?).

Для определения типа объекта можно воспользоваться оператором typeid. Результат этой операции – объект типа typeinfo, обладающего операцией сравнения. Например, введём указатели типа базового типа Base, которые будут направлены на объекты производного типа Derived:

```
 Base* p1;
 Base* p2;
 Base* p3;
 Base b;
 Derived1 d1;
 Derived2 d2;
 Derived2 d3;
```

```
 p->print();
 p1 = &d1;
 p2 = &d2;
 p3 = &d3;
if(typeid(p3)==typeid(p2))
    cout<<"\nTypes of p2 and p3 are the same";
 if(typeid(p1)==typeid(p2))
    cout<<"\nTypes of p1 and p2 are the same";
```

Кроме этого, присутствует операция name(), позволяющая получить тип переменной в виде строки:

```
cout<<"\np1: "<< typeid(p1).name();
cout<<"\np2: "<< typeid(p2).name();
```

Приведение типов

При приведении типов можно выделить две основные ситуации: 1) требуется провести приведение между типами, не связанными схемой наследования; 2) требуется провести преобразование типа от типа родителя к типу потомка (например, мы убедились, что указатель на базовый класс направлен на объект производного класса, и хотим сделать преобразование указателей, чтобы пользоваться функциями из класса-наследника).

Первый случай можно реализовать в стиле C по схеме:

```
type2 res = (type2)param;
```

Здесь param – это исходная переменная типа type, res – переменная другого типа (к которому делается приведение). Например:

```
int res = (int)param;
```

Аналогичное преобразование с помощью оператора static_cast выглядит так:

```
int res;
double dres = 0;
dres = static_cast<double>(res);
cout<<"\nStatic_cast<double>: "<<dres;
```

Если классы, между которыми нужно сделать преобразование, встроены в единую схему наследования, то следует использовать оператор dynamic_cast. На практике, если указатель типа базовый класс (Base) направлен на объект производного класса (Derived), может потребоваться сделать преобразование типов *Base => *Derived. Тогда появится возможность пользоваться функциями, введёнными на уровне класса Derived. Это выглядит так:

```
Base* p1;
Derived1 *p_d;
p_d = dynamic_cast<Derived1*>(p1);
if(p_d) cout<<"\nCast was successful";
else cout<<"\nCast was unsuccessful";
```

Если преобразование проведено успешно, указатель p_d примет ненулевой результат.

Также в литературе к курсу можно найти описания работы операторов const_cast, reinterpret_cast.

Подробное изложение различных элементов полиморфизма в C++ дано в [1 – 6].

СИСТЕМА ОБРАБОТКИ ИСКЛЮЧЕНИЙ

Ошибки могут возникать в коде, который Вы не контролируете, – в библиотечных функциях. Но то же может происходить и в написанном Вами коде. Причём, речь идёт не о синтаксических ошибках (неправильное употребление конструкций языка, орфографические ошибки), но и логических ошибках. Код может оказаться не рассчитан на некоторые комбинации входных параметров, о которых разработчики не задумались, рассматривая основные комбинации.

Код, который может сработать с ошибкой (сгенерировать исключение) помещается в блок try {…} – мы пробуем выполнить некоторое действие.

Также можно сгенерировать исключение самостоятельно с помощью команды throw – бросаем исключение, кто-то должен его «поймать». Иначе «приложение упадёт».

Обработка исключений происходит в блоке catch {}. В связи с тем, что исключения бывают разные: не удалось открыть файл для чтения или записи, выход за границы массива, не удалось разделить на 0 и т.д., - на них нужно реагировать по-разному. Команды throw и блоки catch должны различать

разные исключения, чтобы по-разному на них реагировать. Исключения для решения этой задачи различаются по типу.

Начнём с простейшего кода для класса «Дробь»:

```
class Fraction
{
private:
 //числитель
 int numerator;
 //знаменатль
 int denominator;
public:
 //get-теры - получение значений числителя и
знаменателя извне
 int getNumerator() { return numerator; }
 int getDenominator() { return denominator; }

 //установка числителя
 void setNumerator(int value) { numerator = value; }

 //умножение
 Fraction operator*(Fraction F)
 {
     Fraction Res;
     Res.numerator = numerator * F.numerator;
     Res.denominator = denominator * F.denominator;
     return Res;
 }

 //вывод
 void print()
 {
     cout << numerator<<"/"<<denominator;
 }
};
```

Введём сеттер для знаменателя, а вслед за ним и конструктор.

Если пользователь попытается установить нулевое значение для знаменателя, может оказаться недостаточным просто отменить это действие. Мы сгенерируем ошибку в этом случае. Создадим класс ошибки «Деление на ноль»: сохраним в нём числитель дроби, над которой шла работа во время возникновения ошибки, а также добавим сообщение, которое содержит информацию об ошибке и может быть выведено.

```
class DivisionByZero
{
private:
 //числитель
 int numerator;

 //сообщение об ошибке
 char c[26];
public:
 //конструктор
 DivisionByZero(int Numerator)
 {
     numerator = Numerator;
     strcpy(c, "attempt to divide by zero");
 }

 //get-тер - получение числителя извне
 int getNumerator() { return numerator; }

 //вывод сообщения
 void print()
 {
     cout<<"Exception: " << c << ", numerator = " <<
numerator;
 }
};
```

Будем генерировать это исключение, когда пользователь передаёт значение 0 для знаменателя в конструктор класса «Дробь» или использует это значение в сеттере. Изменения в классе «Дробь»:

```
//конструктор
 Fraction(int Numerator = 0, int Denominator = 1)
 {
      if (Denominator == 0)
           throw DivisionByZero(Numerator);

      //здесь всё хорошо
      numerator = Numerator;
      denominator = Denominator;
 }

 //set-тер для знаменателя
 void setDenominator(int value)
 {
      if (value == 0)
           throw DivisionByZero(numerator);

      //здесь всё хорошо
      denominator = value;
}
```

Для класса «Дробь» всё готово: когда пользователь передаёт некорректные значения, мы генерируем исключение. Но в коде, где дробь будет использоваться, также надо включить обработку исключений. Сейчас они «бросаются», но никто их не «ловит», поэтому приложение «падает». В блоках кода, где записывается новое значение в знаменатель, будем пользоваться блоком try {…}. Он окружает потенциально опасный участок кода. За ним следует набор команд catch, которые «ловят» исключения и обрабатывают их. Причём, каждый catch ловит исключение своего типа и обрабатывает именно его. Получим такой код:

```
int main()
{
 //здесь всё хорошо
 Fraction F1(1, 2);
 Fraction F2(1, 3);
 Fraction F3 = F1 * F2;
 F3.print();
```

```
    //а здесь сгенерируем исключение
    try
    {
        Fraction G(1, 0);
        //сюда мы уже не попадём
        Fraction H = G * F1;
        H.print();
    }
    catch (DivisionByZero d)
    {
        //здесь обрабатываем исключение
        cout << "\nDivision by zero exception has been
caught:\n";
        d.print();
    }

    //вторая попытка
    try
    {
        F1.setDenominator(0);
        //сюда мы уже не попадём
        F3 = F1 * F2;
        F3.print();
    }
    catch (DivisionByZero d)
    {
        //здесь обрабатываем исключение
        cout << "\nDivision by zero exception has been
caught:\n";
        d.print();
    }
    return 0;
}
```

Обратите внимание: ошибка сгенерирована внутри функций конструктора или установки значения знаменателя. Внутри них нет блока catch, поэтому выполнение передаётся в функцию, вызвавшую их, – main. Далее выполнение

кода не передаётся обратно в то место, где произошло исключение. После выполнения кода в блоке catch выполняются команды, следующие за ним (после обработки первой ошибки выполняется код, следующий за комментарием «вторая попытка»).

В следующем примере введем новое исключение: не будем позволять пользователю делать знаменатель отрицательным. Пусть знак хранит только числитель. При попытке установить в знаменателе отрицательное значение, будем генерировать исключение типа NegativeDenominator. Фрагмент класса «Дробь»:

```
    //конструктор
    Fraction(int Numerator = 0, int Denominator = 1)
    {
        if (Denominator == 0)
            throw DivisionByZero(Numerator);
        if (Denominator < 0)
            throw NegativeDenominator(0, 1,
Denominator);

        //здесь всё хорошо
        numerator = Numerator;
        denominator = Denominator;
    }
    //set-тер для знаменателя
    void setDenominator(int value)
    {
        if (value == 0)
            throw DivisionByZero(numerator);
        if (value < 0)
            throw NegativeDenominator(numerator,
denominator, value);

        //здесь всё хорошо
        denominator = value;
    }
```

Теперь в функции main набор catch’ей будет состоять из двух блоков:

```
int main()
{
 //здесь всё хорошо
 Fraction F1(1, 2);
 Fraction F2(1, 3);
 Fraction F3 = F1 * F2;
 F3.print();

 //а здесь сгенерируем исключение
 try
 {
     Fraction G(1, 0);
     //сюда мы уже не попадём
     Fraction H = G * F1;
     H.print();
 }
 catch (DivisionByZero d)
 {
     //здесь обрабатываем исключение
     cout << "\nDivision by zero exception has been
caught:\n";
     d.print();
 }
 catch (NegativeDenominator n)
 {
     //здесь обрабатываем исключение
     cout << "\nNegative denominator exception has
been caught:\n";
     n.print();
 }

 //вторая попытка
 try
 {
     F1.setDenominator(-1);
     //сюда мы уже не попадём
     F3 = F1 * F2;
     F3.print();
```

```
    }
    catch (DivisionByZero d)
    {
        //здесь обрабатываем исключение
        cout << "\nDivision by zero exception has been
caught:\n";
        d.print();
    }
    catch (NegativeDenominator n)
    {
        //здесь обрабатываем исключение
        cout << "\nNegative denominator exception has
been caught:\n";
        n.print();
    }
    return 0;
}
```

В случае возникновения ошибки внутри блока try {…} управление передаётся в блоки catch и они просматриваются последовательно. Возможно, мы не учтём всех проблем, поэтому нам нужен catch, который будет «ловить всё» - исключения всех типов. Он имеет вид catch(…). У него есть трудность определения, какая же ошибка случилась. Она решается методами, выходящими за рамки курса.

Обратите внимание: при обработке исключения ***срабатывает не более одного блока*** catch: они просматриваются сверху вниз, пока какой-то один не подойдёт по типу. Выполняются команды внутри блока catch, а затем выполнение передаётся за последний блок catch (после обработки первой ошибки выполняется код, следующий за комментарием «вторая попытка»). Поэтому, если Вы поставите какой-либо catch за блок catch(…), он не сработает никогда, т.к. перед ним всё будет «ловить» и обрабатывать catch(…).

Использование наследования в системе обработки исключений

Нетрудно заметить, что наши исключения похожи. У них есть одна основа: числитель, знаменатель, сообщение об ошибке. Неудобно оперировать исключениями, которые имеют различную структуру. Если общую основу поместить в базовый класс для всех исключений, то 1) у всех исключений будет точно определённый минимальный набор полей, который, конечно, можно

расширять, 2) если нужно отреагировать на все исключения, которые описаны в коде, можно использовать catch с исключением родительского класса.

В родительский класс включим сообщение об ошибке, функцию её вывода, конструктор копий.

В дочерних классах нужно переопределить функцию вывода (в базовом классе для этого она объявлена виртуальной), добавить конкретику об ошибке в виде полей класса.

```
class Exception
{
protected:
 //сообщение об ошибке
 char *str;
public:
 Exception(const char* s)
 {
      str = new char[strlen(s)+1];
      strcpy_s(str, strlen(s) + 1, s);
 }
 Exception(const Exception& e)
 {
      str = new char[strlen(e.str) + 1];
      strcpy_s(str, strlen(e.str) + 1, e.str);
 }
 ~Exception()
 {
      delete[] str;
 }

 //функцию вывода можно будет переопределить в
производных классах, когда будет ясна конкретика
 virtual void print()
 {
      cout << "Exception: " << str;
 }
};

//теперь наследуем ошибки от Exception
```

```
    class DivisionByZero: public Exception
    {
    //деление на 0
    private:
     //числитель
     int numerator;

    public:
     //конструктор
     DivisionByZero(int Numerator): Exception("attempt to
divide by zero")
     {
          numerator = Numerator;
     }

     //get-тер - получение числителя извне
     int getNumerator() { return numerator; }

     //вывод сообщения - переопределение виртуальной
функции базового класса
     void print()
     {
          cout<<"Exception: " << str << ", numerator = "
<< numerator;
          //также можно воспользоваться функцией вывода
базового класса и потом дописать числитель
          //из-за того, что она называется так же, как эта
функция, указываем класс, откуда её берём
          //Exception::print();
          //cout << ", numerator = " << numerator;
     }
    };

    class NegativeDenominator: public Exception
    {
     //отрицательный знаменатель
    private:
     //исходная дробь
```

```
    //числитель
    int numerator;
    //знаменатель
    int denominator;
    //значение, которое пытались установить
    int value;
   public:
    //конструктор
    NegativeDenominator(int Numerator, int Denominator,
int Value): Exception("attempt to set negative
denominator")
    {
        numerator = Numerator;
        denominator = Denominator;
        value = Value;
    }

    //get-теры - получение дроби извне
    int getNumerator() { return numerator; }
    int getDenominator() { return denominator; }
    //получение значения, которое пытались установить в
знаменатель
    int getValue() { return value; }

    //вывод сообщения
    void print()
    {
        cout << "Exception: " << str << ", value = " <<
value << ", Fraction = "<<numerator<<"/"<<denominator;
    }
   };
```

В функции main теперь можно подходить к «отлову» исключений двояко: если Вы знаете, что «ловить», в конце набора блоков catch ставите catch(Exception e) – он будет ловить все ошибки, спроектированные Вами. Перед ним должны стоять блоки catch для ошибок классов-потомков. Так же, как и во втором примере, если Вы поставите первым catch(Exception e), то любую ошибку дочернего класса можно привести к типу родительского класса

и сработает именно этот catch. Это похоже на ситуацию с catch(…) в предыдущем примере. Если не сработал catch(Exception e), то произошла ошибка, не спроектированная Вами в приложении. Для неё есть блок catch(…).

В последнем примере из функции f_i вызывается функция f_{i+1}. i = 1,2,3: f1() -> f2() -> f3() -> f4(). В функции f4 генерируется исключение целого типа.

```
    double f4(double a, double b)
    {
     //выполним здесь какую-либо недопустимую операцию и
посмотрим, где она будет обработана
     //сгенерируем исключение сами
     throw 1;
     return a * b;
    }

    double f3(double a, double b)
    {
     return f4(a, b);
    }

    double f2(double a, double b)
    {
     try
     {
          return f3(a, b);
     }
     catch (Exception e)
     {
          cout << "\nChildren of Exception are caught
here";
     }
     catch (...)
     {
          cout << "All exceptions are caught here";
     }
     return 0;
    }
```

```
double f1(double a, double b)
{
 return f2(a, b);
}

int main()
{
 //вложенные вызовы f1 -> f2 -> f3 -> f4
 //где будет обработано исключение, брошенное в функции f4?
 cout << "\nNested f() functions test\n";
 cout<<"\n"<<f1(1, 0);

 return 0;
}
```

Внутри этой функции нет блока try-catch, поэтому её выполнение прекращается. Выполнение передаётся в вызвавшую функцию f3. Там такого блока тоже нет, поэтому исполнение передаётся в функцию f2. Там переходим к блокам catch. Причём созданное нами исключение Exception не совпадает по типу с 1, поэтому этот блок «не поймал» исключение. Работает блок catch(…).

Обратите внимание: catch(Exception e) обработает все исключения, которые мы сами создали в своём коде и унаследовали от этого класса. Если появляется исключение, не описанное нами, его может обработать catch(…).

Рекомендуется исключения наследовать от класса std::exception. Это базовый класс для исключений в стандартной библиотеке.

Подробное изложение системы обработки исключений в C++ можно найти в работах [1 – 6].

СИСТЕМА ВВОДА/ВЫВОДА

Подробное описание перегрузки операций ввода/вывода из/в поток/файл см. в пункте «Перегрузка операций ввода/вывода в поток».

Флаги, регулирующие вывод

Для того, чтобы настраивать вывод в консоль или файл, устанавливаются флаги вывода. Значения флагов заданы в классе ios, а значит, доступны во всех его потомках. Установка флага (значение 1) производится командой setf(), а снятие (значение 0) – командой unsetf(). В начале примера указано, как

сбрасывать и устанавливать флаги. Значение флагов в большинстве примеров ясно по названию. Отсылаем читателя к подробным учебникам по C++ для разбора каждого флага по отдельности.

```
ios::fmtflags f;
 f = cout.flags();

cout.unsetf(ios::dec);
 cout.setf(ios::hex);
 cout.setf(ios::scientific);


 if (f & ios::skipws) cout << "\nskipws is set";
 else cout << "\nskipws is unset";

 if (f & ios::left) cout << "\nleft is set";
 else cout << "\nleft is unset";

 if (f & ios::right) cout << "\nright is set";
 else cout << "\nright is unset";

 if (f & ios::internal) cout << "\nskipws is set";
 else cout << "\ninternal is unset";

 if (f & ios::dec) cout << "\ndec is set";
 else cout << "\ndec is unset";

 if (f & ios::oct) cout << "\noct is set";
 else cout << "\noct is unset";

 if (f & ios::hex) cout << "\nhex is set";
 else cout << "\nhex is unset";

 if (f & ios::showbase) cout << "\nshowbase is set";
 else cout << "\nshowbase is unset";

 if (f & ios::showpoint) cout << "\nshowpoint is set";
 else cout << "\nshowpoint is unset";
```

```
    if (f & ios::showpos) cout << "\nshowpos is set";
    else cout << "\nshowpos is unset";

    if (f & ios::uppercase) cout << "\nuppercase is set";
    else cout << "\nuppercase is unset";

    if (f & ios::scientific) cout << "\nscientific is
set";
    else cout << "\nscientific is unset";

    if (f & ios::fixed) cout << "\nfixed is set";
    else cout << "\nfixed is unset";

    if (f & ios::unitbuf) cout << "\nunitbuf is set";
    else cout << "\nunitbuf is unset";

    if (f & ios::boolalpha) cout << "\nboolalpha is set";
    else cout << "\nboolalpha is unset";
```

Поэкспериментируйте с различными флагами. Попробуйте установить флаг, выполнить вывод.

Полезными командами, настраивающими формат вывода, являются: precision(), width(), fill(). В отличие от флагов, они принимают параметры на вход: в первом случае – количество разрядов после запятой, во втором – какого размера поле отводится под выводимое число (сколько позиций), в третьем: в случае, если запись числа содержит меньше разрядов, чем определено в width(), каким символом заполнить оставшееся место.

Манипуляторы вывода

Манипуляторы – это объекты, редактирующие формат вывода (фактически они повторяют флаги), которые вставляются прямо в команду вывода. Например, вывод числа в научном виде с помощью флагов будет выглядеть следующим образом:

```
    cout.setf(ios::scientific);
    cout<<x;
```

Манипулятор scientific будет использоваться прямо в команде вывода:

```
cout << scientific <<x;
```

Создание собственного манипулятора вывода без параметра – это создание функции, которая принимает ссылку на поток и возвращает её же (похоже на перегрузку функции вывода в консоль, но здесь нет информации о выводимом объекте, т.к. настраиваем вывод в целом).

```
ostream& my_manip(ostream& ustream)
{
 ustream.fill('%');
 ustream.width(10);
 ustream.precision(3);
 ustream.setf(ios::scientific);
 return ustream;
}
```

Здесь ustream – редактируемый поток. Использование своего манипулятора не отличается от применения стандартного манипулятора вывода:

```
cout << "\nz = " << my_manip << z;
```

Подробнее с работой манипуляторов можно ознакомиться в литературе к пособию. Создание манипуляторов с параметрами выходит за рамки курса.

Более углублённый подход к системе ввода/вывода в C++ можно найти в работах [1 – 6].

СТРУКТУРЫ ДАННЫХ

Массив

Массив – непрерывный участок памяти, выделенный под элементы одинакового размера. Благодаря этому, операция индексации [i] работает быстро: известен адрес начала массива в памяти, размер одного элемента, поэтому расположение i-го элемента получить легко. Например, для массива type *arr:

$$\&arr[i] = \&arr[0] + sizeof(type) * i,$$

где операция sizeof(type) показывает, сколько байт отведено под один элемент массива. Эта операция выполняется за O(1).

Такая организация элементов в памяти позволяет быстро добавлять элементы в конец массива (при условии, что выделено достаточно памяти и есть свободное место в конце массива). Если *len* – длина участка в памяти (в количестве элементов), выделенного под массив, а *count* – количество заполненных элементов, запись нового элемента *elem* в конец массива *arr* выглядит так (см. рис. 3):

```
arr[count] = elem;
count++;
```

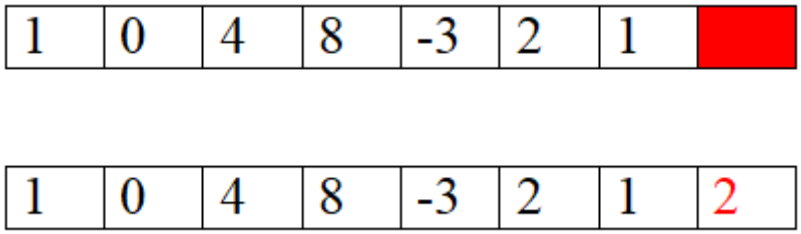

Рисунок 3. Добавление элемента 2 в конец массива

Эта операция выполняется за константное время O(1).

Добавление элемента в середину выполняется медленнее, т.к. все элементы, которые должны оказаться после вставленного элемента нужно сдвинуть вправо. Выполнять это приходится, работая индивидуально с каждым элементом. Существуют структуры данных, которые позволяют выполнять эту операцию быстрее, чем за O(n) (например, см. операции над массивом на основе декартова дерева, treap). При использовании концепции массива операция вставки / удаления элемента из середины массива оценивается за O(n). Здесь и далее n – это оценка количества элементов в массиве. В целом, та же ситуация с добавлением элемента в начало массива. С точки зрения производительности, лучше всего, если удаётся переместить начало массива вперёд к новому элементу с индексом 0, но это возможно не всегда. В общем случае надо сдвинуть все оставшиеся элементы массива на одну позицию (аналогично вставке / удалению из середины массива, см. рис. 4).

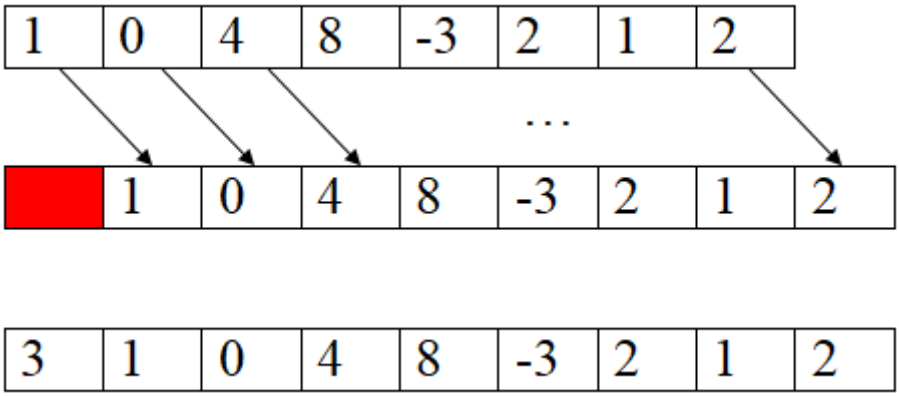

Рисунок 4. Добавление элемента 3 в начало массива

При этом за константное время можно выполнять операции, которые не требуют перемещения или сравнения большого количества элементов. Например, обмен местами двух элементов (swap(i, j), где i, j – индексы переставляемых элементов) не задействует никаких других элементов в массиве, поэтому выполняется за O(1).

Если массив не упорядочен, то во время поиска элемента *elem_to_find* по значению необходимо просматривать все элементы от начала массива к концу или в противоположном направлении. Такая операция оценивается в O(n):

```
for(i=0;i<len;i++)
    if(arr[i]==elem_to_find)
        …
```

Если массив упорядочен, то поиск по нему можно выполнить за O(log n). Чаще всего применяется алгоритм бинарного поиска, на каждой итерации которого рассматриваемая часть массива делится на две части. При просмотре элемента, который расположен в середине, возможны четыре ситуации: 1) искомый элемент найден, 2) искомый элемент слева, 3) искомый элемент справа, 4) размер рассматриваемой части массива стал равен 0, элемент не найден. На рис. 5 представлен пример бинарного поиска элемента 12, рассматриваемая часть массива выделена красным, середина массива выделена жирным шрифтом.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

1	2	3	4	5	6	7	8	9	10	**11**	12	13	14	15	16	17	18	19	20

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**16**	17	18	19	20

1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15	16	17	18	19	20

1	2	3	4	5	6	7	8	9	10	11	**12**	13	14	15	16	17	18	19	20

Рисунок 5. Бинарный поиск элемента 12 в упорядоченном массиве

Итак, массив – это простая структура данных. Из-за простоты она по умолчанию используется как основная структура данных практически везде.

Операции, которые выполняются эффективно (за O(1)): индексация, добавление элемента в конец, обмен элементов местами, поиск в отсортированном массиве.

Связный список

Связный список отличается от массива тем, что под него не обязательно выделяется непрерывный участок памяти. При отсутствии этого требования невозможно просчитать, где находится i-й элемент в памяти. Для перемещения между элементами вместе с данными хранят указатель на следующий элемент *Element* next* (см. рис. 6). Здесь *Element* – это тип звеньев в списке.

Рисунок 6. Схема односвязного списка

В такой конфигурации речь идёт об односвязном списке. Указатель на первый элемент списка здесь обозначен как *Element* begin*. Если в каждом элементе списка, вдобавок, есть указатель на предыдущий элемент *Element* prev*, то речь идёт о двусвязном списке (см. рис. 7).

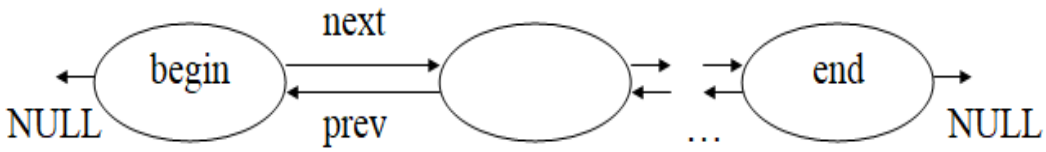

Рисунок 7. Схема двусвязного списка

Указатель на последний элемент списка здесь обозначен как *Element* end*.

Чтобы получить доступ к i-му элементу, необходимо установить указатель на начало списка и перечислить i элементов в цикле. Эта операция оценивается в O(n), где n – число элементов в списке. Из-за этого у контейнера list нет операции [i], а перебирать элементы надо при помощи итераторов. Более подробно о перечислении элементов контейнеров можно прочесть в разделе об итераторах.

В отличие от массива, быстрее выполняется вставка и удаление элементов. Предполагается, что задан указатель на элемент, за или перед которым нужно выполнить вставку, или указатель на удаляемый элемент. При этом перемещать множество элементов в памяти не требуется. Достаточно настроить указатели элемента перед вставляемым / удаляемым элементом и после него. Вставка элемента *Element* newElem* в односвязный список вслед на элементом *Element* current* схематично представлена на рис. 8. Красным цветом выделены редактируемые указатели. Кроме этого, может потребоваться изменить указатели на начало и конец списка. Количество этих действий ограничено и не зависит от длины списка. Эта операция оценивается в O(1).

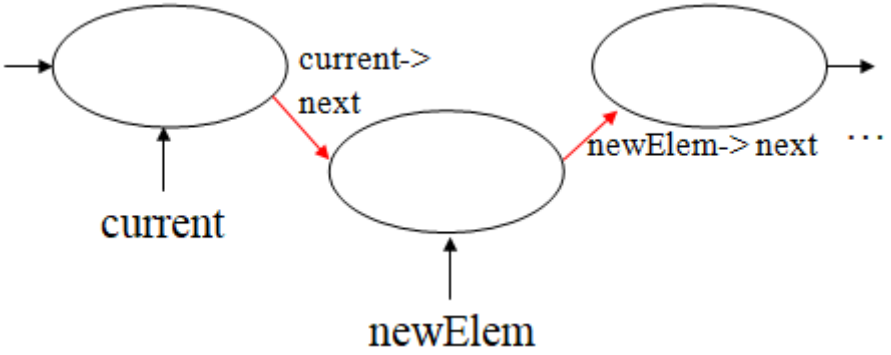

Рисунок 8. Вставка элемента в односвязный список

Вставка элемента в двусвязный список, в целом, аналогична. Нужно отредактировать больше указателей: кроме указателей на следующий элемент, ещё и указатели на предыдущий (см. рис. 9).

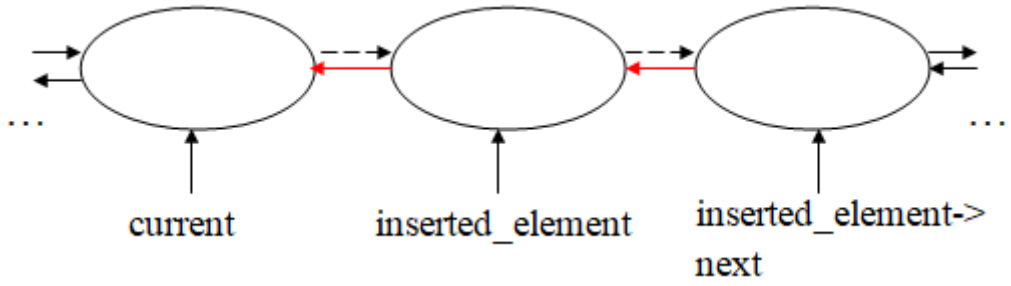

Рисунок 9. Вставка элемента в двусвязный список

Эффективными операциями (выполняются за O(1)) также остаются добавление и удаление элемента с начала или конца списка. Применять бинарный поиск в списке неэффективно из-за неэффективной операции индексации (выполняется за O(n)), поэтому, даже при наличии упорядоченности элементов в списке, следует пользоваться схемой линейного поиска.

На основе массива или связного списка могут вводиться структуры данных стек и очередь. В стек элементы добавляются в конец и извлекаются с конца (LIFO – Last In First Out). В очередь элементы также добавляются в конец, но извлекаются с начала (FIFO – First In First Out).

Подробное исследование операций на различных структурах данных можно найти в [7 – 11].

МЕТОДЫ СОЗДАНИЯ БЕЗОПАСНОГО КОДА

Указатели

В рамках этой части пособия разделим указатели на функции и указатели на данные. Указатели на функции могут быть перезаписаны таким образом, чтобы передавать управление коду злоумышленника. Указатель на объект тоже может быть изменён для выполнения произвольного кода, например, если он используется для присваивания.

Переполнение буфера – это общее название уязвимостей, связанных с возможностью чтения / записи памяти, не принадлежащей переменной или массиву, созданному в коде. Часто возникает из-за некорректно ограниченных циклов следующих типов [12]:

1. Цикл, ограниченный сверху или снизу, – цикл, выполняющий перебор n значений массива, n должно быть меньше объёма выделенной под массив памяти (при некорректной работе происходит выход за границы массива).
2. Цикл, ограниченный адресом первого или последнего элемента: во время выполнения цикла адрес указателя увеличивается или снижается,

пока он не станет равен / меньше / больше граничного значения (при некорректной проверке ограничения происходит выход за границы массива).

3. Цикл, ограниченный нулевым ограничителем: в цикле увеличивается адрес указателя, пока он не указывает на нулевой элемент (для некорректно ограниченных строк в конце массива может отсутствовать ноль).

Для того, чтобы переполнение можно было использовать для перезаписи указателя на функцию или данные, нужно выполнение всех условий, следующих далее.

1. Буфер должен быть выделен в том же сегменте памяти, что и указатель на объект или целевая функция.
2. В случае цикла, ограниченного сверху, нулевым символом или последним элементом массива, буфер должен находиться по адресам, меньшим, чем адрес целевой функции или указателя на данные. Для цикла, ограниченного снизу или первым элементом массива, – наоборот: буфер должен находиться по адресам, большим, чем целевой адрес.
3. Буфер не должен быть корректно ограничен (перебирая элементы в цикле, можно выйти за его реальные пределы).

Указатель на функцию используется так, чтобы в рамках неизменного кода можно было вызвать функцию, заданную пользователем. Такие функции должны иметь определённый интерфейс (тип возвращаемого значения, типы и последовательность переменных зафиксированы), но вместо корректной функции по указателю можно передать вредоносный код. Аналогичная уязвимость может возникнуть при перезаписи таблиц виртуальных функций и обработки исключений. Фактически и в том, и в другом случае можно получить доступ к адресам функций, которые должны вызываться при обработке перегрузки функций или при организации системы обработки исключительных ситуаций (более подробно см. в главе 3 [12]).

При переполнении буфера в стеке эффективной стратегией считается создание «стековой канарейки».

Одна из эффективных контрмер против действий злоумышленников, использующих указанные выше уязвимости, – стратегия W^X (запись или исполнение, но не одновременно). Она позволяет делать участок памяти либо записываемым, либо исполняемым, но не одновременно (что делает бесполезной перезапись адресов указателей на функции).

В некоторых работах, цитируемых в [12], предлагается стратегия кодирования и декодирования указателей на функции: вместо адреса в чистом виде нужно хранить его шифр), но она не реализована широко.

Ошибки управления памятью

В рамках C для выделения участка динамической памяти использовались команды malloc() / calloc(), а для освобождения памяти – команде free().

Одна из часто встречающихся проблем – ожидание от команды malloc() не только выделения участка памяти, но и его инициализации (что присуще оператору new из C++). Таким образом, *отсутствие проверки инициализации памяти, выделенной с помощью malloc(), может привести к доступу к неинициализированной памяти*. Например, в следующем примере функции, умножающий матрицу на вектор, это происходит из-за отсутствия инициализации памяти, выделенной для указателя p (в коде предполагается, что начальное значение p[i] – это ноль):

```
int* product(int** matr, int* v, int n)
{
 int* p = malloc(n * sizeof(int));
 int i, j;
 for (i = 0; i < n; i++)
     for (j = 0; j < n; j++)
         p[i] += matr[i][j] * v[j];
 return p;
}
```

Память может быть проинициализирована функцией memset() или при выделении памяти с помощью calloc(). Об уязвимостях, связанных с отсутствием инициализации (в корректном в остальном коде), можно прочесть в [12]. Неинициализированная память может быть в дальнейшем распределена в других операциях выделения памяти, что потенциально может привести к утечке важной информации.

В C++ при использовании оператора new для выделения памяти есть две ситуации:

1) если инициализатор в new опущен, объект инициализируется по умолчанию; если инициализации не происходит, объект имеет неопределенное значение;

2) если инициализирующее значение есть, применяются правила инициализации.

Пример кода, в котором инициализация запускается:

```
int* p1 = new int();
int* p2 = new int(5);
```

В следующей команде инициализация памяти, выделенной под указатель p3, не производится:

```
int* p3 = new int;
```

Другая важная проблема при выделении памяти – проверка значения, обозначающего количество байт или количество ячеек памяти (которые могут занимать более одного байта), которое необходимо выделить. *Оно может оказаться отрицательным, нулевым или слишком большим*.

Если выделение памяти невозможно, функции распределения памяти в C возвращают нулевые указатели. Следует иметь план действий на случай, если выделить память, необходимую для работы приложения, не удалось. В некоторых приложениях известны уязвимости, когда даже после неудачной попытки выделения памяти, работала операция чтения / записи значения в памяти на некотором смещении от нулевого указателя.

В случае отсутствия проверки неудачи выделения памяти или при обращении по указателю, который не инициализирован, может произойти чтение / запись в области памяти, которая расположена вблизи нулевого адреса. На большинстве машин это приводит к ошибке сегментации, но в некоторых системах подобный указатель может считаться вполне «законным» и тогда происходит обращение к специальной области памяти, не предназначенной для пользовательского доступа.

Оператор new имеет две основные формы. В случае, если запрошенная память не выделена, генерируется исключение `std::bad_alloc`. Если оператор new используется с параметром `std::nothrow`, исключение не генерируется, оператор new возвращает NULL:

```
int* p4 = new(std::nothrow) int;
```

Следующий участок кода не корректен из-за того, что работа оператора new либо успешно завершается, либо генерирует исключение (в код внутри else нельзя попасть никогда – вместо этого будет сгенерировано исключение):

```
int* p = new int;
if (p)
{
    //условие всегда истинно
}
else
{
    //недостижимый код
}
```

Более подробно о переопределении операций выделения памяти в C и C++ можно прочесть в [1 – 3, 12, 13].

Если память уже освобождена с помощью free() или оператора delete, но происходит доступ к ней, обычно это приводит к аварийному завершению без дополнительных ошибок. Пример некорректного освобождения памяти, выделенной под связный список, приведён ниже (head – указатель на первый элемент списка, next – указатель на следующий элемент):

```
for (p = head; p != NULL; p = p->next)
    free(p);
```

К моменту обращения `p = p->next` на нулевой итерации цикла память по указателю p уже освобождена. Подумайте, как грамотно освобождать память, выделенную под связный список.

Запись в освобождённую память может привести к перезаписи и повреждению структур, используемых диспетчером памяти, или другими объектами, используемыми в коде в том случае, если освобождённый участок памяти к моменту обращения уже используется другими объектами.

Утечка памяти происходит, когда динамическая память, которая уже использована в программе и далее не нужна, не освобождается. Утечки – существенная проблема для процессов, которые должны работать в системе долго (и требуют памяти в течение длительного времени). Они являются основой атак на истощение ресурсов. Если злоумышленник сможет определить внешнее воздействие, которое приводит к выделению памяти, и она в

дальнейшем не освобождается, доступная в системе память может быть постепенно исчерпана.

Поведение при *выделении памяти нулевого размера* зависит от реализации: либо выполняются действия по выделению памяти такие же, как при выделении памяти ненулевого размера, либо возвращается нулевой указатель. В первом случае указатель не должен использоваться для доступа к объекту, т.к. чтение или запись приведут к неопределённому поведению.

При использовании операторов выделения памяти new / new[ ] и освобождения памяти delete / delete[ ] *не следует забывать об их соответствии друг другу*. При выделении памяти под один объект используется new, а для освобождения такой памяти – delete. Для выделения памяти под массив используется new[ ], при этом освобождает такую память оператор delete[ ]. Эти операторы могут быть переопределены для пользовательских классов.

Сложные структуры данных и многоступенчатые операции освобождения памяти (например, хранение объектов сложных пользовательских классов, обладающих деструкторами, в структурах данных, классы которых имеют свои деструкторы) требуют особого внимания. Кроме утечки памяти (в ситуации, когда память, выделенная под элементы списка, освобождается не целиком), возможна проблема *многократного освобождения памяти* [12].

Наиболее эффективный способ предотвращения проблем с памятью – это соблюдение следующих соглашений в написании кода, управляющем памятью.

1. Следует использовать одинаковый шаблон для выделения и освобождения памяти (например, выделение памяти – только в конструкторах, освобождение – только в деструкторах пользовательских классов).
2. Следует выделять и освобождать память в одном и том же модуле.
3. Выделение и освобождение памяти должно соответствовать друг другу (см. проблему несоответствия операторов new/new[] и delete/delete[] выше).

Кроме этого, некорректный код может быть найден в процессе анализа с помощью инструментов времени выполнения (см. главу 4 в [12]).

Работа со строками

Строки в C/C++ – это массивы или «обертки» для массивов, поэтому многие ошибки из указанных в предыдущем пункте актуальны и для строк. Укажем некоторые особые исключительные ситуации.

Копирование неограниченных строк происходит при копировании строк из источника в массив символов фиксированной длины. При получении пользовательского ввода или чтении данных из файла заранее не ясно, сколько символов содержится во входных данных (нельзя выделить заранее массив достаточной длины), поэтому требуется либо определить длину (при чтении строки из файла), либо ввести ограничение по количеству получаемых символов (усечение строки). Подобный ввод чреват проблемой переполнения буфера – выхода за границы выделенного для работы участка памяти. Следует использовать безопасные функции с окончанием _s для работы со строками: gets_s(), printf_s(), scanf_s() и т.п., а также их аналоги для работы с файловыми потоками – и усечение строк при копировании, вводе и конкатенации. Кроме этой ошибки, выделяют проблему *некорректно ограниченных строк*, например: строка, которую нужно скопировать, может иметь некорректное завершение (например, отсутствие символа конца строки ‘\0’). Обработка такой строки может привести к проблеме переполнения буфера (цикл, перебирающий символы строки, не закончится на её реальном окончании в памяти, а будет перебирать элементы, которые находятся за границами памяти, выделенной под строку). Эти проблемы решаются ограничением по числу итераций в циклах, перебирающих элементы строк (в случае ввода из файла или ввода информации пользователем этот подход превращается в усечение строк). Например, в операции копирования применяется ограничение по числу символов MAX_LEN (если символ конца строки не будет встречен при переборе элементов исходной строки с индексами менее MAX_LEN, произойдёт усечение входной строки):

```
#define MAX_LEN 1000
char* copy(char* str)
{
 char* res;

 int len = 0;
 while(str[len]!='\0' && len<MAX_LEN) len++;

 res = (char*)malloc(sizeof(char)*(len+1));
 int i=0;
 for(i=0;i<len;i++)
 {
     res[i] = str[i];
```

```
    }
    res[len] = '\0';

    return res;
}
```

Ошибка сдвига на единицу происходят при неправильном перечислении символов в строке. Первый символ в строке имеет индекс 0 (как и первый элемент в любом массиве), но в некоторых языках программирования реализован «более дружелюбный» подход, при котором первый символ в строке имеет индекс 1. Эта ошибка не каждый раз ведёт к катастрофическим последствиям (приложение может не завершится аварийно), но в цикле перебора символов в строке самый первый будет игнорироваться, что может повлечь за собой неверную логику в работе программы (а т.к. синтаксической ошибки нет, то «отловить» такую ошибку тяжело).

При работе со строками в C/C++ особенно часто встречается проблема переполнения буфера, т.к. строки определены как массивы символов с завершающим нулевым символом, не производится неявной проверки выхода за границы массива, функции стандартной библиотеки (устаревшие варианты) могут не выполнять такой проверки (пользуйтесь функциями с окончанием _s). Рекомендуется пользоваться классами std::string / std::basic_string для представления строк, т.к. там реализованы соответствующие проверки входных для объекта класса данных. Правда, и в этом случае возможны ошибки: 1) использования недействительных или неинициализированных итераторов для перебора элементов строки, 2) передачи некорректного индекса в строке, 3) передачи некорректной позиции итератора.

Подробное изложение проблем, связанных с работой со строками, уязвимостей, которые могут быть использованы для атак, стратегий противодействия им представлено в главе 2 источника [12], а также в работах [13, 14].

ТРЕБОВАНИЯ И РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Работы выполняются в среде разработки по выбору студента, например: Visual Studio, CodeBlocks, GCC + Eclipse, Clang.

Каждая практическая работа выполняется в виде отдельного проекта (в систему СДО или на почту преподавателя отправляется файл *cpp или ссылка в облако на него).

Решение каждой задачи или подзадачи оформляется в виде отдельной функции (кроме функции main()). Входные параметры задаются непосредственно в коде или запрашиваются у пользователя с консоли, а затем передаются в эту функцию. Функция возвращает результат с помощью оператора return, или он может быть доступен по указателю. Вывод результата в консоль осуществляется вне этой функции.

Необходимо тщательно протестировать работу решения на разнообразных входных данных.

ПРАКТИЧЕСКИЕ РАБОТЫ

Практическая работа №1. Перегрузка операций

Задание 1.1

Постройте класс С, указанный в варианте (табл. 1.1). Поля должны быть закрытыми. Введите необходимый набор конструкторов и деструктор (при необходимости). Введите для него перегрузку операций, указанных в варианте, с помощью компонентных функций. Продемонстрируйте работу построенных операций в функции main.

Код 1.1. Пример класса Дробь с перегруженной операцией сложения

```
#include <iostream>

using namespace std;

class Fraction
{
private:
 //поля: числитель и знаменатель
 int denominator;
 int numerator;
public:
 //конструкторы
 Fraction() {numerator = 0; denominator = 1;}
 Fraction(int Numerator) {numerator = Numerator; denominator
= 1;}
 Fraction(int Numerator, int Denominator) {numerator = Numer-
ator; denominator = Denominator;}
 Fraction(const Fraction& F) {numerator = F.numerator; denom-
inator = F.denominator;}

 Fraction operator*(Fraction F)
 {
      Fraction Res;
      Res.numerator = numerator* F.numerator;
      Res.denominator = denominator* F.denominator;
      //включить сокращение дроби
      return Res;
 }
```

```
 void print()
 {
      std::cout<<numerator<<"/"<<denominator;
 }
};

int main()
{
 Fraction F1(2, 3);
 Fraction F2(-1, 3);
 Fraction F = F1*F2;
 F.print();

 char c; std::cin>>c;
 return 0;
}
```

Таблица 1.1. Варианты типов хранимых в контейнере значений и условие предиката в функции фильтрации

Вариант	Класс С	Реализуемая операция
1.	Комплексное число	Унарный плюс – комплексное сопряжение, бинарный оператор * – стандартное умножение на действительное и комплексное число
2.	Дробь	Сложение с целым числом, вычитание целого числа
3.	Точка (3 компоненты)	Сложение с вектором (из 3 компонент), вычитание двух точек – вектор
4.	Вектор (3 компоненты)	Сложение / вычитание вектора
5.	Вектор (3 компоненты)	Сложение с точкой (3 компоненты), скалярное произведение с вектором
6.	Буква (один элемент char)	Операция сравнения <, независимая от регистра
7.	Комплексное число	Унарный плюс – применение модуля к действительной и мнимой части, бинарный оператор / – стандартное деление на действительное и комплексное число
8.	Точка (2 компоненты)	Унарный плюс – отражение относительно оси $y = x$, вычитание двух точек – вектор

Продолжение таблицы 1.1

9.	Матрица (2x2)	Матрица представлена 4 числами. Ввести операцию бинарного умножения на число или матрицу
10.	Матрица (2x2)	Матрица представлена 4 числами. Перегрузить унарную операцию транспонирования, бинарную операцию сложения с числом (число умножить на единичную матрицу) и матрицей
11.	Матрица (2x2)	Матрица представлена 4 числами. Перегрузить унарную операцию определения детерминанта, бинарную операцию вычитания числа (число умножить на единичную матрицу) или матрицы
12.	Комплексное число	Операция сравнения < (сначала сравнить по действительной части, в случае равенства – по мнимой), унарная операция проекции на действительную ось
13.	Комплексное число	Операция сравнения < (сначала сравнить по модулю, в случае равенства – по действительной части, в случае их равенства – по мнимой), унарная операция проекции на мнимую ось
14.	Дробь	Унарная операция смены местами числителя и знаменателя, операция сравнения < с дробью или вещественным числом
15.	Точка (3 компоненты)	Унарные операции проекции на ось x, плоскость OXY, сложения точки и вектора (из 3 компонент)
16.	Многочлен 2-й степени	Многочлен представлен тремя коэффициентами. Перегрузить унарную операцию определения корня, наибольшего по модулю. Определить бинарную операцию сложения многочленов
17.	Вектор (2 компоненты)	Бинарная операция определения угла между двумя векторами, скалярное произведение
18.	Вектор (2 компоненты)	Унарная операция проекции на ось абсцисс, унарная операция проекции на ось ординат, бинарная операция проекции на произвольную ось
19.	Окружность (радиус и 2 координаты)	Операция сравнения < (сравнение по радиусу), унарная операция – находится ли окружность полностью в одной из четвертей (bool)

Продолжение таблицы 1.1

20.	Окружность (радиус и 2 координаты)	Операция сложения с вещественным числом – увеличение радиуса на это значение, унарная операция – изменение знаков координат
21.	Квадрат (длина стороны и 2 координаты центра), стороны параллельны осям кординат	Операция умножения на положительное вещественное число – умножение стороны на это значение, унарная операция – вычисление периметра
22.	Квадрат (длина стороны и 2 координаты центра), стороны параллельны осям кординат	Операция умножения на положительное вещественное число – умножение координат на это значение, унарная операция – вычисление площади
23.	Отрезок из начала координат (2 компоненты)	Операция сложения (вычитания) с вещественным числом – сложение этого числа с координатами. Унарная операция – поворот на 90 градусов против часовой стрелки
24.	Окружность (радиус и 2 координаты)	Операция сравнения < (сравнение по расстоянию центра окружности от центра координат), унарная операция – вычисление длины окружности
25.	Отрезок на плоскости (4 компоненты)	Операция сложения (вычитания) с вещественным числом – смещение отрезка вверх (вниз) без вращения. Унарная операция – вычисление длины отрезка
26.	Матрица (2x2)	Матрица представлена 4 числами. Операция сравнения < (сравнение по величине следа), унарная операция – вычисление следа матрицы.

Окончание таблицы 1.1

27.	Матрица (2x2)	Матрица представлена 4 числами. Операция сравнения < (сравнение по величине определителя), унарная операция – вычисление определителя.
28.	Точка (2 компоненты)	Операция сложения с вещественным числом – смещение по оси y. Унарная операция – отражение относительно оси OX
29.	Отрезок с серединой в начале координат (2 компоненты — координаты одного конца)	Операция умножения на вещественное число – увеличение длины всего отрезка в это количество раз. Унарная операция – отражение отрезка относительно оси OY
30.	Матрица (2x2)	Матрица представлена 4 числами. Операция сравнения < (сравнение по сумме модулей всех элементов), унарная операция – вычисление суммы модулей всех элементов матрицы.

Задание 1.2

Дополните класс C из задания 1.1. Введите в него дружественные функции, в которых работает перегрузка операций для недостающих комбинаций расположения операндов. Для определенности в табл. 1.2 представлены операции, которые необходимо реализовать с помощью дружественных функций.

Примечание: *например, в задании 1.1 требовалось перегрузить операцию умножения комплексного числа на действительное: z = w * x: здесь z, w – комплексные числа, x – действительное число. В задании 1.2 нужно ввести функции для перегрузки операций так, чтобы заработала операция z = x * w.*

Таблица 1.2. Варианты типов хранимых в контейнере значений и условие предиката в функции фильтрации

Вариант	Класс C	Реализуемая операция
1.	Комплексное число	Бинарный оператор * – стандартное произведение действительного и комплексного числа
2.	Дробь	Сложение целого числа и дроби
3.	Точка (3 компоненты)	Сложение действительного числа a и точки: все компоненты точки увеличиваются на a
4.	Вектор (3 компоненты)	Сложение вектора с действительным числом a: вектора: все компоненты вектора увеличиваются на a
5.	Вектор (3 компоненты)	Умножение на действительное число a
6.	Буква (один элемент char)	Операция сравнения < с целым числом (ему ставится в соответствие символ из таблицы ASCII)
7.	Комплексное число	Бинарный оператор / – стандартное деление действительного числа на комплексное число
8.	Точка (2 компоненты)	Вычитание действительного числа a и точки P, в результате получается новая точка с компонентами: $\{a - P_x, a - P_y\}$
9.	Матрица (2x2)	Матрица представлена 4 числами. Ввести операцию бинарного умножения действительно числа на матрицу
10.	Матрица (2x2)	Матрица представлена 4 числами. Ввести бинарную операцию сложения числа и матрицы (число умножить на единичную матрицу) и матрицы
11.	Матрица (2x2)	Матрица представлена 4 числами. Перегрузить бинарную операцию вычитания матрицы из числа (число представить как единичную матрицу, умноженную на него)
12.	Комплексное число	Операция сравнения < действительного и комплексного числа (сравнить по действительной части; если она совпадает, то сравнить по мнимой части)

Продолжение таблицы 1.2

13.	Комплексное число	Операция сравнения < действительного и комплексного числа (сначала сравнить по модулю, в случае равенства – по действительной части, в случае их равенства – по мнимой)
14.	Дробь	Операция сравнения < целого или вещественного числа с дробью
15.	Точка (3 компоненты)	Сложение числа и точки: к первой компоненте точки добавить число
16.	Многочлен 2-й степени	Многочлен представлен тремя коэффициентами. Определить бинарную операцию сложения и умножения действительного числа на многочлен
17.	Вектор (2 компоненты)	Бинарная операция умножения вещественного числа на вектор
18.	Вектор (2 компоненты)	Бинарная операция деления вещественного числа на вектор: каждая компонента вектора – это частное от деление числа на её первоначальное значение: $a / V = \{a/V_x, a/V_y\}$.
19.	Окружность (радиус и 2 координаты)	Унарная операция – находится ли окружность полностью в одной из четвертей (bool) Унарная операция – в какой четверти полностью находится окружность. Если однозначного ответа нет, вернуть 0 (int)
20.	Окружность (радиус и 2 координаты)	Операция сложения вещественного числа и окружности – увеличение радиуса на это значение, унарная операция – изменение знака координаты по оси абсцисс (x)
21.	Квадрат (длина стороны и 2 координаты центра), стороны параллельны осям кординат	Операция умножения вещественного положительного числа на объект класса фигуры – умножение длины стороны на это значение, унарная операция – вычисление площади

Продолжение таблицы 1.2

22.	Квадрат (длина стороны и 2 координаты центра), стороны параллельны осям кординат	Операция умножения вещественного положительного числа на объект класса фигуры – умножение координат на это значение, унарная операция – вычисление периметра
23.	Отрезок из начала координат (2 компоненты)	Операция сложения (вычитания) вещественного числа и отрезка – сложение этого числа с координатами. Унарная операция – поворот на 180 градусов по часовой стрелке
24.	Окружность (радиус и 2 координаты)	Операция сравнения < вещественного числа и окружности (сравнение по расстоянию центра окружности от центра координат), унарная операция – вычисление площади круга
25.	Отрезок на плоскости (4 компоненты)	Операция сложения (вычитания) вещественного числа и отрезка – смещение отрезка вверх (вниз) без вращения. Унарная операция – вычисление расстояния середины отрезка до начала координат
26.	Матрица (2x2)	Матрица представлена 4 числами. Операция сравнения < числа с матрицей (сравнение по величине следа), унарная операция – вычисление суммы элементов на побочной диагонали матрицы.
27.	Матрица (2x2)	Матрица представлена 4 числами. Операция сравнения < числа с матрицей (сравнение по величине определителя), унарная операция – определение существования обратной матрицы (bool).
28.	Точка (2 компоненты)	Операция сложения вещественного числа с точкой – смещение по оси y. Унарная операция – отражение относительно оси OY

Окончание таблицы 1.2

29.	Отрезок с серединой в начале координат (2 компоненты — координаты одного конца)	Операция умножения вещественного числа на отрезок – увеличение длины всего отрезка в это количество раз. Унарная операция – отражение отрезка относительно оси OX
30.	Матрица (2x2)	Матрица представлена 4 числами. Операция сравнения < числа с матрицей (сравнение по сумме модулей всех элементов), унарная операция – вычисление среднего значения всех элементов матрицы.

Практическая работа №2. Операции над массивом

Задание 2.1

В заготовке к работе предоставлен класс MyArrayParent, частично реализующий функции массива. Дополните класс необходимыми функциями:

1. Конструктором копий MyArrayParent (const MyArrayParent& v).
2. Оператором =.
3. Оператором [ ] для обращения к элементу по индексу.
4. Конструктором, принимающим и копирующим информацию из существующего массива.
5. Функцией поиска элемента в массиве по значению IndexOf().

Задание 2.2

В классе MyArrayChild, унаследованном от класса MyArrayParent, необходимо добавить функции (обратите внимание на то, что можно пользоваться функциями, уже реализованными в базовом классе):

1. Удаления элемента из произвольного места в массиве RemoveAt().
2. Вставки элемента в некоторое место в массиве InsertAt().
3. Добавьте функцию согласно Вашему варианту (см. табл. 2.1). Результатом обработки массива должен быть новый объект класса. Исходный массив не должен меняться.

Задание 2.3

Унаследуйте новый класс MySortedArray от MyArrayChild. Мы предполагаем, что этот класс содержит упорядоченный массив. Переопределите функции из пункта 3 задания 2.2, имея ввиду, что массив отсортирован. Переопределите операцию добавления элемента (push) таким образом, чтобы массив оставался отсортированным при добавлении нового элемента. Переопределите функцию поиска (IndexOf()), имея ввиду, что массив отсортирован.

Таблица 2.1. Функции для пункта 5 задания 2.1 и задания 2.3

Вариант	Функция
1.	Получить индексы всех вхождений максимума в массив
2.	Во входном массиве – целые числа. Поместить в массив на выходе только простые числа из него
3.	Получить все индексы вхождения некоторой переменной element
4.	Входной массив содержит 4-значные целые числа вида $abcd$. Поместить в массив-результат только те, у которых $ab = cd$

Продолжение таблицы 2.1

5.	Выделить в массиве элементы z, подчиняющиеся выражению: $x^2 + y^2 = z^2, x, y$ - целые
6.	Во входном массиве – целые числа. Произвести арифметический сдвиг на n позиций влево (вправо) каждого элемента
7.	Удалить все нечётные числа в массиве
8.	Входной массив содержит 4-значные целые числа *abcd.* Поместить в выходной массив только те, у которых $a + b = c + d$
9.	Во входном массиве находятся коэффициенты поверхности второго порядка. Определить тип поверхности (используйте enum)
10.	Результат должен содержать все элементы массива, удовлетворяющие выражению *3k+1,* где *k* – целое.
11.	Выделить все элементы, отличающиеся от среднего арифметического в массиве не более, чем на параметр функции *diff*
12.	Входной массив содержит 4-значные целые числа *abcd.* Поместить в выходной массив только те, у которых $a + $ c делится на 3, $b + d$делится на 7
13.	Входной массив содержит 5-значные целые числа *abcde*. Поместить в выходной массив только те, у которых $a + b = d + e = c$
14.	Поместить в выходной массив элементы, большие, чем среднее геометрическое в массиве
15.	Входной массив содержит 5-значные целые числа *abcde*. Поместить в выходной массив только те, у которых $a * b = d * e = c$
16.	Получить сумму элементов, расположенных между первым и последним нулевыми элементами (если, конечно, массив содержит не менее 2-х нулей), и значение -1 (минус 1), если в массиве нулей меньше 2
17.	Получить индексы всех максимальных элементов массива
18.	Поместить в выходной массив индексы всех элементов, которые встречаются больше двух раз
19.	Все повторяющиеся элементы (дубли) заменить средним арифметическим исходного массива
20.	Заменить все элементы массива средним арифметическим их соседей, крайние элементы оставить нетронутыми

Окончание таблицы 2.1

21.	Переставить элементы массива, которые меньше входного параметра n в начало массива, остальные разместить в конце (порядок не важен)
22.	Отразить зеркально элементы массива относительно середины, но только для тех симметричных элементов, у которых совпадает четность
23.	Дан массив, где каждая пара элементов – координаты точки (x,y) на плоскости. Найти пары точек, которые симметричны относительно оси OY.
24.	Дан массив целых чисел. Получить сумму индексов элементов, кратных трем.
25.	Сдвинуть элементы массива вправо на величину, равную количеству отрицательных элементов массива. "Пустые" элементы слева заполнить нулями
26.	Нормализовать элементы массива по следующей формуле: new=(old – min) / (max – min) max - значение максимального элемента min – значение минимального элемента
27.	Отразить зеркально участки, находящиеся между отрицательными элементами
28.	Введите функцию сравнения двух массивов operator<. Первый массив считаем меньшим второго, если максимум в первом массиве меньше максимума во втором массиве
29.	Удалить из массива все элементы, равные параметру функции
30.	Верните из функции новый массив (типа MySortedArray/MyArrayChild), в котором удалены элементы, отклоняющиеся от среднего арифметического в массиве «очень далеко» (не более, чем величину p * mean(arr), где p – параметр функции)
31.	Написать функцию concat(), объединяющую два массива в один. В случае отсортированного массива MySortedArray предполагаем, что оба операнда-массива отсортированы, результат операции тоже должен оказаться отсортирован (при этом, пользоваться функцией сортировки после слияния массивов не рационально)

Код 2.1. Основные операции родительского класса MyArrayParent и наследование MyArrayChild

```
#include <iostream>

using namespace std;

class MyArrayParent
{
protected:
 //сколько памяти выделено?
 int capacity;
 //количество элементов - сколько памяти используем
 int count;
 //массив
 double* ptr;
public:
 //конструкторы и деструктор
 MyArrayParent(int Dimension = 100)
 {
      cout << "\nMyArray constructor";
      ptr = new double[Dimension];
      capacity = Dimension;
      count = 0;
 }
 //конструктор принимает существующий массив
 MyArrayParent(double* arr, int len)
 {
      cout << "\nMyArray constructor";
      //заполнить массив ptr, заполнить поля
 }
 //деструктор
 ~MyArrayParent()
 {
      cout << "\nMyArray destructor";
      if (ptr != NULL)
      {
           delete[] ptr;
           ptr = NULL;
      }
 }

 //обращение к полям
 int Capacity() { return capacity; }
```

```
int Size() { return count; }
double GetComponent(int index)
{
     if (index >= 0 && index < count)
          return ptr[index];
     //сгенерировать исключение, если индекс неправильный
     return -1;
}
void SetComponent(int index, double value)
{
     if (index >= 0 && index < count)
          ptr[index] = value;
     //сгенерировать исключение, если индекс неправильный
}

//добавление в конец нового значения
void push(double value)
{
     if (count < capacity)
     {
          ptr[count] = value;
          count++;
     }
     //что делаем, если массив заполнен?
}

//удаление элемента с конца
void RemoveLastValue()
{
     if (count >= 0)
          count--;
     //что делаем, если пуст?
}

double& operator[](int index)
{
     //перегрузка оператора []
}

MyArrayParent& operator=(const MyArrayParent& V)
{
     cout << "\noperator = ";
     //оператор =
     //arr1 = arr2 = arr3; где arr_i - объекты нашего класса
```

```
    }

    MyArray(const MyArrayParent& V)
    {
        cout << "\nCopy constructor";
        //создание копии объекта - в основном, при возвращении
результата из функции / передаче параметров в функцию
    }

    void print()
    {
        cout << "\nMyArr, size: " << count << ", values: {";
        int i = 0;
        for (i = 0; i < count; i++)
        {
            cout << ptr[i];
            if (i != count - 1)
                cout << ", ";
        }
        cout << "}";
    }



};

class MyArrayChild : public MyArrayParent
{
public:
    //используем конструктор родителя. Нужно ли что-то ещё?
    MyArrayChild(int Dimension = 100) : MyArrayParent(Dimension)
{ cout << "\nMyArrayChild constructor"; }

    ~MyArrayChild() { cout << "\nMyArrayChild destructor\n";  }

    //удаление элемента
    //void RemoveAt(int index = -1);

    //поиск элемента
    //int IndexOf(double value, bool bFindFromStart = true);

    //вставка элемента
    //void InsertAt(double value, int index = -1);
```

```
    //выделение подпоследовательности
    //MyArrayChild SubSequence(int StartIndex = 0, int Length =
-1)

    //добавление элемента в конец
    //operator + ?

};

int main()
{
 if (true)
 {
      MyArrayParent arr;
      int i = 0;
      for (i = 0; i < 10; i++)
      {
           arr.push(i + 1);
      }
      MyArrayParent p1;
      p1 = arr;
      p1.print();
      cout << "\n";
 }
 char c; cin >> c;
 return 0;
}
```

Практическая работа №3. Множественное наследование

Необходимо реализовать указанную схему наследования. В схеме «0» A1 – родитель для классов B1, B2.

1. Каждый класс содержит защищённую переменную. Например: A1 содержит переменную a1, B1 – b1, C2 – c2 и т.п. Каждый класс имеет конструктор, присваивающий этой переменной некое значение. Конструкторы производных классов должны вызывать конструкторы базовых классов и передавать туда параметры.

2. В базовых классах реализовать функцию print(), которая выводит на консоль имя класса, из которого она вызвана. В производных классах переопределять её, чтобы она выводила корректный результат. Например:

```
A1 a;
a.print();  => “A1”
B1 b;
b.print();  => “B1”
```

3. Написать функцию show(), которая выводит все доступные для объекта этого класса переменные. Например, в классе B1 доступны локальные переменные класса B1 и некоторые переменные из родительского класса A1. Переопределить её в производных классах.

```
A1 a;
a.show();  => только переменные в A1
B1 b;
b.show(); => переменные из B1 и A1
```

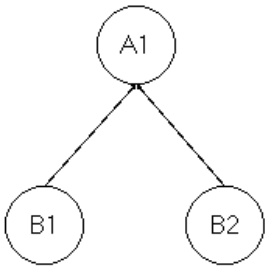

Рисунок 3.1. Схема наследования, состоящая из трех классов. A1 – родитель для классов B1, B2

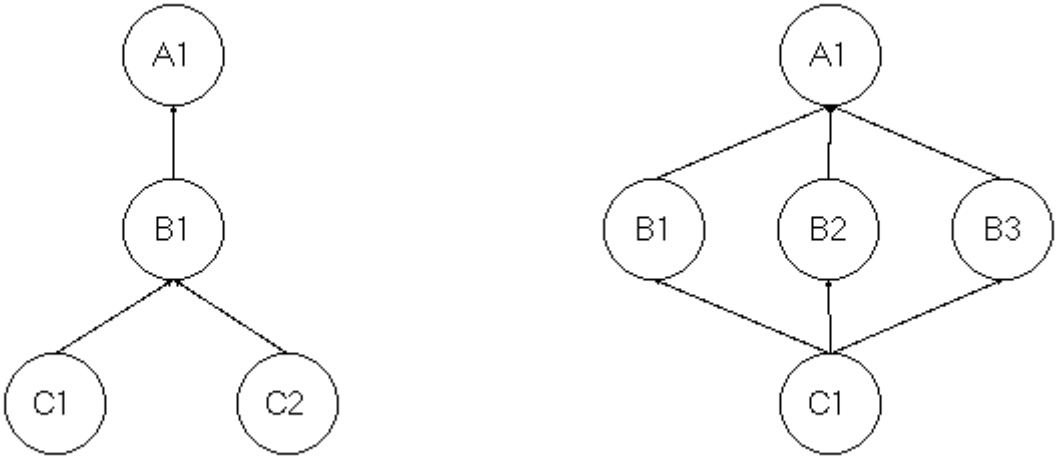

Рисунок 3.2. Вариант 1 (слева) и вариант 2(справа)

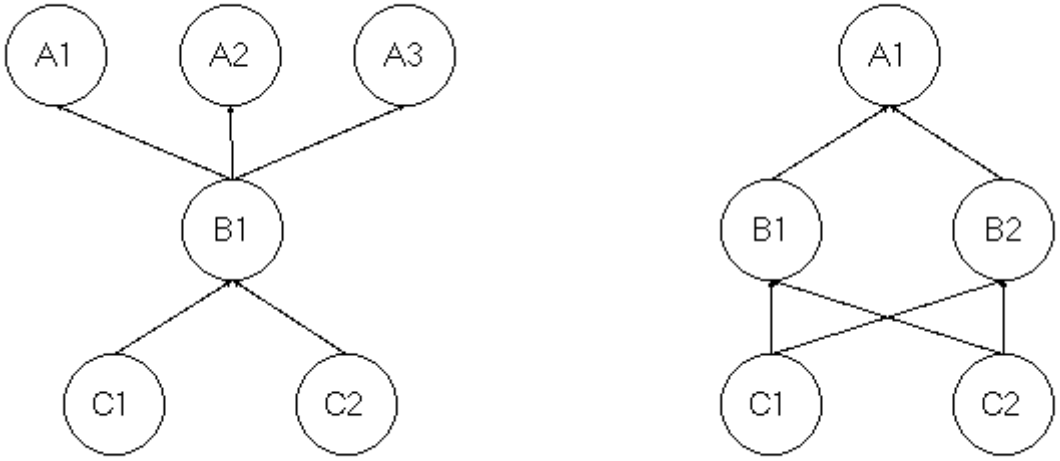

Рисунок 3.3. Вариант 3 (слева) и вариант 4(справа)

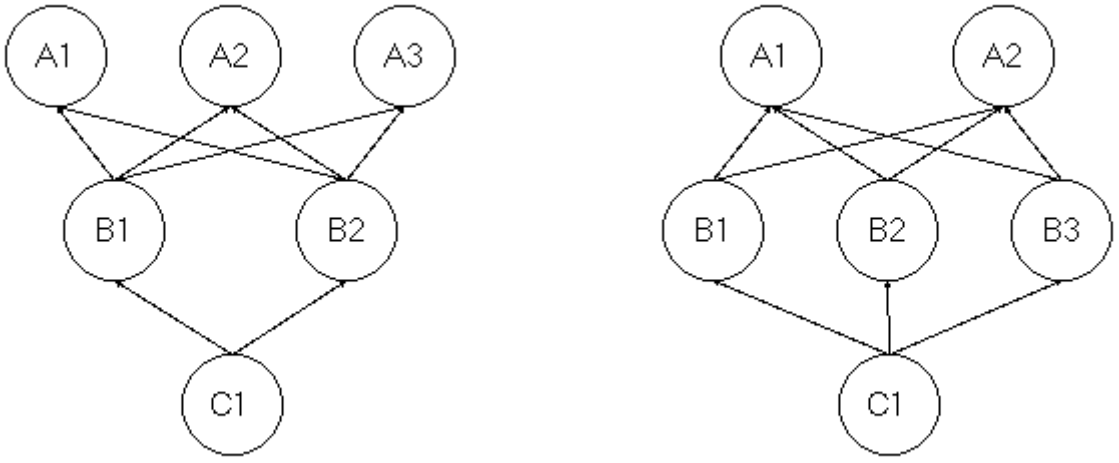

Рисунок 3.4. Вариант 5 (слева) и вариант 6(справа)

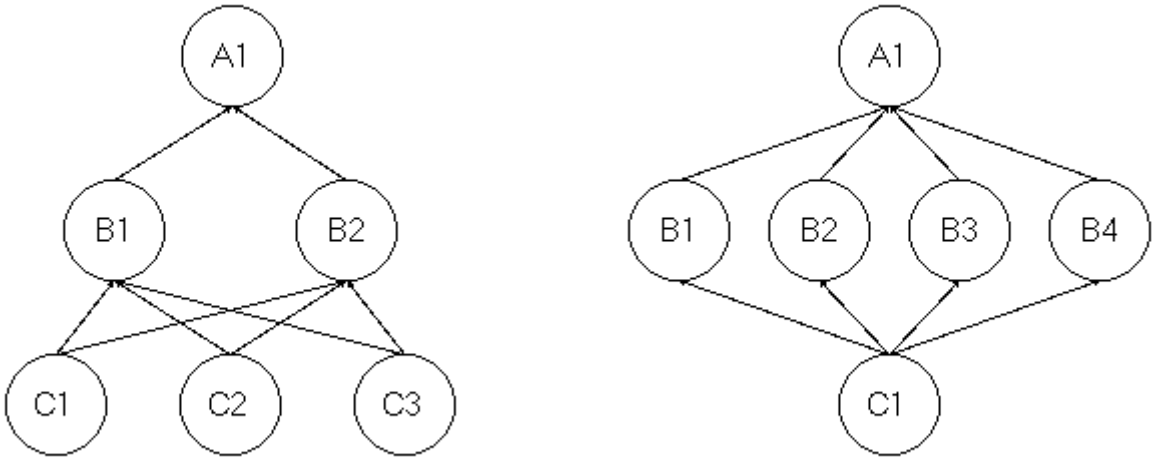

Рисунок 3.5. Вариант 7 (слева) и вариант 8(справа)

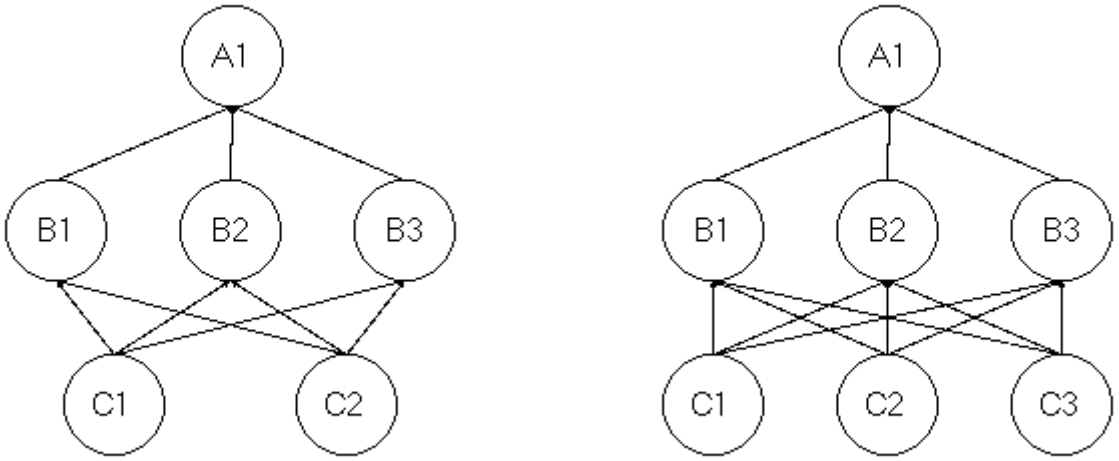

Рисунок 3.6. Вариант 9 (слева) и вариант 10(справа)

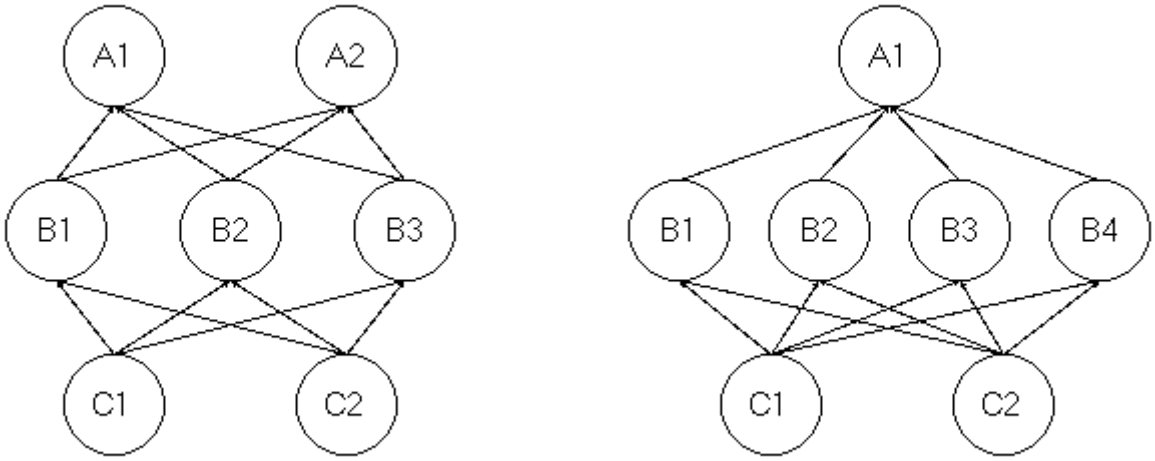

Рисунок 3.7. Вариант 11 (слева) и вариант 12(справа)

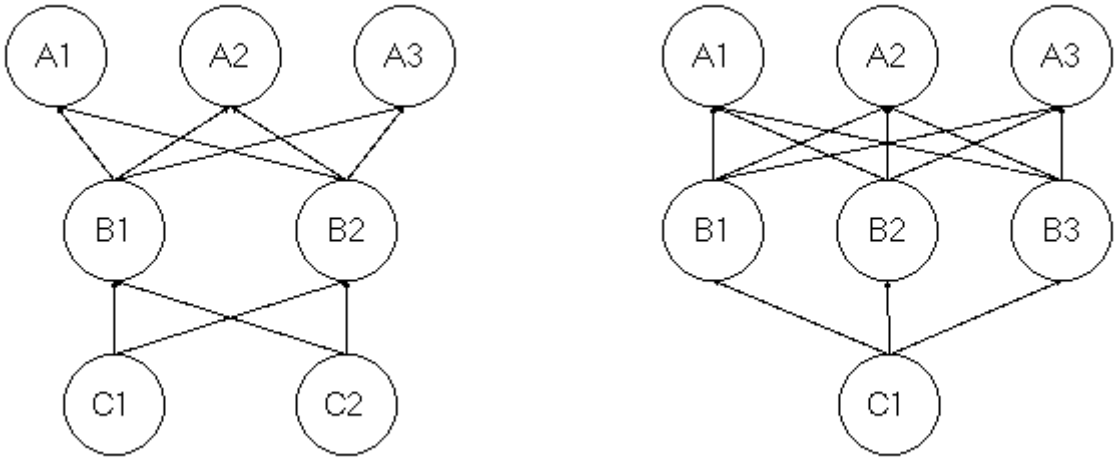

Рисунок 3.8. Вариант 13 (слева) и вариант 14(справа)

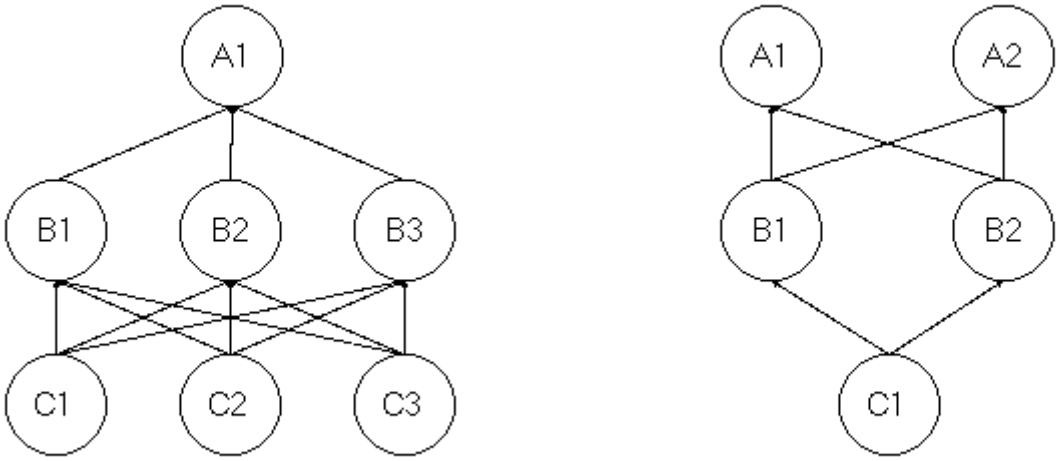

Рисунок 3.9. Вариант 15 (слева) и вариант 16(справа)

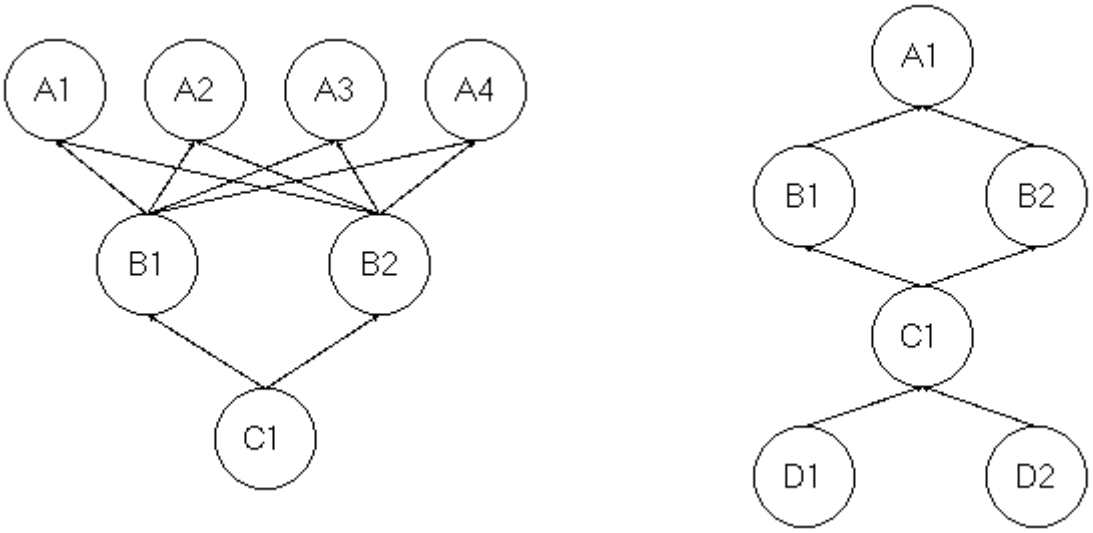

Рисунок 3.10. Вариант 17 (слева) и вариант 18(справа)

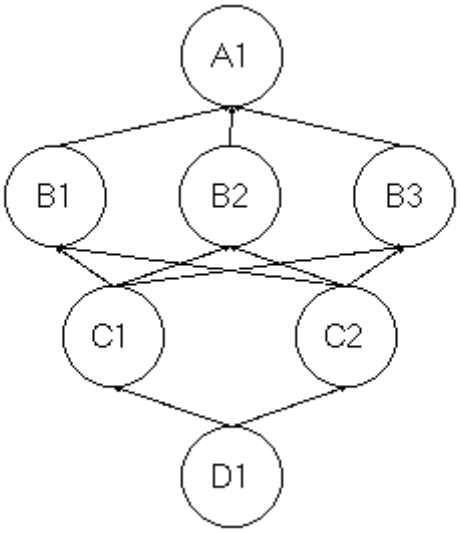

Рисунок 3.11. Вариант 19

Код 3.1. Пример схемы наследования, указанной на рис. 3.1.

```
#include<iostream>

using namespace std;

class A1
{
protected:
 int a1;
public:
```

```
     A1(int V1 = 0) { a1 = V1; cout << "\nClass A1 constructor";}
     virtual void print() { cout << "\nVariable of A1 class"; }
     virtual void show() { cout << "\na1 = " << a1; }
    };

    class B1: virtual public A1
    {
    protected:
     int b1;
    public:
     B1(int V1 = 0, int V2 = 0) : A1(V2) { b1 = V1; cout <<
"\nClass B1 constructor";  }
     virtual void print() { cout << "\nVariable of B1 class"; }
     virtual void show() { cout << "\nb1 = " << b1<<", a1 =
"<<a1; }
    };

    class B2 : virtual public A1
    {
    protected:
     int b2;
    public:
     B2(int V1 = 0, int V2 = 0) : A1(V2) { b2 = V1; cout <<
"\nClass B2 constructor"; }
     virtual void print() { cout << "\nVariable of B2 class"; }
     virtual void show() { cout << "\nb2 = " << b2 << ", a1 = "
<< a1; }
    };
    int main()
    {
     B1 test(1, 2);
     test.show();
     test.print();
     A1* ptr = &test;ptr->show();ptr->print();
     char c; std::cin>>c;
     return 0;
    }
```

Практическая работа №4. Операции над строками

Унаследуйте свой класс от базового класса «Строка». Реализуйте работу функций: конструкторов, конструктора копий, деструктора, operator=, как в базовом классе, так и в наследнике. Добавьте в наследник работу функции, указанной в варианте (табл. 4.1).

Таблица 4.1. Варианты методов класса Строка.

Вариант	Функция
1.	Написать функцию поиска LastIndexOf() последнего вхождения произвольной подстроки в строке.
2.	Написать функцию поиска IndexOf() первого вхождения произвольной подстроки в строке.
3.	Написать функцию, вводящую аналог операции “<” для сравнения двух строк (сравнение в алфавитном порядке).
4.	Написать функцию string_to_number(), проверяющую, является ли содержимое строки числом, и возвращающую это число в случае утвердительного ответа.
5.	Написать функцию, определяющую количество вхождений некоторой подстроки
6.	Написать функцию compare(), проверяющую совпадают ли две переданные ей строки
7.	Получить символ, который повторяется наибольшее количества раз. Если таких несколько, то вернуть массив из этих символов
8.	Написать функции, реализующие код Цезаря (и кодирование, и декодирование): буква заменяется на следующую за ней в алфавите (“test”->”uftu”)
9.	Написать функцию, проверяющую, является ли строка палиндромом
10.	Написать функцию, подсчитывающую число гласных
11.	Написать функцию endswith(), проверяющую, заканчивается ли строка переданным массивом символов
12.	Получить все цифры в строке в виде массива целых чисел
13.	Подсчитать количество знаков препинания (.,:;-) в исходной строке
14.	Написать функцию toupper(), преобразующую строку к верхнему регистру

Продолжение таблицы 4.1

15.	Подсчитать количество слов в строке. Словом будем считать последовательность, не содержащую цифр, имеющую длину не менее 3-х символов. Слова отделены друг от друга знаками препинания и пробелами.
16.	Подсчитать количество пар символов, в которых за согласной следует гласная.
17.	Написать функцию, подсчитывающую количество заглавных букв.
18.	Строку мысленно разбить на последовательности по 5 символов. Подсчитать количество пятерок символов, для которых среднее количество гласных больше среднего для всей строки.
19.	Написать функцию trim(), удаляющую все пробелы в начале и конце строки
20.	Написать функцию reverse(), заменяющую порядок символов в строке на обратный
21.	Написать функцию, удаляющую лишние пробелы в тексте
22.	Написать функцию insert(), вставляющую подстроку в строку в место, обозначенное индексом index
23.	Написать функцию startswith(), проверяющую, начинается ли строка переданным массивом символов
24.	Определить число символов в строке, которые больше заданного символа ("больше" в смысле алфавитного порядка). Учесть, что регистр не важен.
25.	Подсчитать количество слов, которые начинаются и заканчиваются гласной.
26.	Определить, является ли строка допустимым идентификатором в C
27.	Получить индекс самого длинного слова в строке.
28.	В строке присутствуют скобки (). Проверить, что они корректно расставлены - для каждой открывающей есть закрывающая, и они расположены в правильном порядке. Пример неверной записи: ")()(", "(())())"
29.	Для всех слов в строке посчитать сумму значений кодов ASCII для символов, входящих в слово. Вернуть наибольшее значение.
30.	Проверить, идет ли в строке пробел после каждого знака препинания.
31.	Написать функцию tolower(), преобразующую строку к нижнему регистру

Окончание таблицы 4.1

32.	Получить индекс последовательности с максимальным числом одинаковых символов. Если их несколько, вернуть индекс начала самой первой
33.	Подсчитать количество слов, в которых равное число гласных и согласных
34.	Написать функцию, которая принимает строку и возвращает новую, в которой все гласные переставлены в начало массива
35.	Получить индекс максимального по длине участка идущих подряд согласных

Код 4.1. Пример некоторых функций базового класса BaseString

```
#include <iostream>

using namespace std;

class BaseString
{
protected:
 char* p;  int len; int capacity;
public:
 BaseString(char* ptr)
 {
      cout<<"\nBase Constructor 1\n";
      len = strlen(ptr) + 1;
      capacity = 256;
      p = new char[capacity];
      for(int i=0;i<len;i++)
      {
           p[i] = ptr[i];
      }
      p[len] = '\0';
 }
 BaseString(int Capacity = 256)
 {
      cout<<"\nBase Constructor 0\n";
      capacity = Capacity;
      p = new char[capacity];
      len = 0;
 }
```

```
~BaseString()
{
	cout<<"\nBase Destructor\n";
	if(p!=NULL)
		delete[] p;
	len = 0;
}

int Length() {return len;}
int Capacity() { return capacity; }
//char* get() {return p;}
char& operator[](int i) {return p[i];}

BaseString& operator=(BaseString& s)
{
	cout<<"\nBase Operator = \n";
	len = s.Length();
	p = new char[s.capacity];
	capacity = s.capacity;
	for(int i=0;i<s.Length();i++)
	{
		p[i] = s[i];
	}
	//strcpy(p, s.get());
	p[len-1] = '\0';
	return *this;
}
BaseString(BaseString& s)
{
	cout<<"\nBase Copy Constructor\n";
	len = s.Length();
	p = new char[s.capacity];
	capacity = s.capacity;
	for(int i=0;i<s.Length() - 1;i++)
	{
		p[i] = s[i];
	}
	p[len-1] = '\0';
}
virtual void print()
{
	int i=0;
	while(p[i]!='\0'){cout<<p[i];i++;}}};
```

Практическая работа №5. Многомерные массивы, система обработки ошибок, операции с файлами

Задание 5.1

Унаследуйте класс от базового класса Многомерный массив. Убедитесь, что в родительском классе верно выделяется и освобождается память под многомерный массив. В производном классе напишите метод заполнения массива произвольным нетривиальным образом. Введите метод, который решает задачу, указанную в варианте (таблица 5.1), и возвращает из неё результат (входная матрица остаётся неизменной, на выходе должен быть новый массив или число).

Таблица 5.1. Варианты методов класса Многомерный массив

Вариант	Массив, функция
1.	В наличии прямоугольная плоская решётка с шагом 1. В узлах решётки находятся точечные массы, хранящиеся в заданной матрице *M*. Вычислить координаты центра масс решётки.
2.	Получить вектор, в котором содержатся максимальные элементы из каждой строки матрицы.
3.	Считая каждый столбец матрицы вектором, получить новую матрицу, в которой столбцы отсортированы по евклидовой норме.
4.	Построить из двух одинаковых треугольников Паскаля произвольного размера «ромб», симметричный относительно основания треугольника.
5.	Удалить из матрицы все нулевые строки и столбцы.
6.	Построить вектор, в котором компонента равна 1, если в соответствующей строке матрицы нули чередуются с ненулевыми элементами (без учёта кратности чередования), и 0 в противном случае.
7.	Получить вектор, каждая компонента которого содержит сумму элементов из соответствующей строки матрицы.
8.	Проверить, является ли матрица нильпотентной ($A^N = 0$, можно ограничиться некоторой наперёд заданной степенью).
9.	Дана матрица неотрицательных элементов. Получить вектор, каждая компонента которого содержит среднее геометрическое элементов соответствующего столбца матрицы.

Продолжение таблицы 5.1

10.	Проверить, является ли матрица идемпотентной ($A^N = A$, можно ограничиться некоторой наперёд заданной степенью).
11.	Получить вектор, каждая компонента которого содержит среднее арифметическое элементов из соответствующей строки матрицы.
12.	Привести матрицу к верхнетреугольному виду по методу Гаусса.
13.	В наличии решётка в виде параллелепипеда. В узлах решётки находятся точечные массы (с шагом 1 по *x, y, z*), значения которых собраны в заданный массив *M*. Вычислить координаты центра масс решётки.
14.	Даны матрица Грама и два вектора. Найти скалярное произведение векторов.
15.	Удалить все строки матрицы, в которых есть нулевые элементы.
16.	Применить ко всем элементам матрицы некоторую скалярную функцию, переданную по указателю.
17.	Отсортировать элементы во всех строках матрицы по убыванию.
18.	Удалить из матрицы все строки и столбцы, среднее арифметическое которых большие некоторого параметра border.
19.	Элементы входной матрицы считать высотами некоторого участка поверхности. Получить на выходе матрицу, в которой отмечены локальные минимумы карты.
20.	Найти все вхождения некоторого элемента element в матрице. Вернуть вектор, состоящий из их индексов.
21.	Получить вектор, в котором компонента нулевая, если соответствующая строка матрицы не отсортирована, и ненулевая, если все элементы строки расположены в порядке возрастания или убывания.
22.	Удалить из матрицы все строки и столбцы, содержащие элементы, большие некоторого параметра border.
23.	Удалить все строки матрицы, в которых есть некоторый наперёд заданный элемент.
24.	Сгладить все элементы матрицы - заменить их средним двух соседних. Для краевых элементов вторым соседом считать сам элемент.

Окончание таблицы 5.1

25.	Написать функцию, возвращающую новую матрицу, которая включает столбцы исходной матрицы, не содержащие ни одного нуля.
26.	Написать функцию, переворачивающую матрицу слева направо (вокруг воображаемой вертикальной оси).
27.	В строках матрицы, в которых есть отрицательные элементы, произвести циклический сдвиг вправо на число этих элементов (для каждой строки оно своё).
28.	Удалить все строки матрицы, в которых нулевых элементов более половины.
29.	Матрица представляет собой один из каналов изображения. На основе зигзаг-сканирования получить вектор (шаг 5: http://www.compression.ru/book/part2/part2__3.htm)
30.	Дана матрица неотрицательных элементов. Получить вектор, каждая компонента которого содержит среднее гармоническое элементов соответствующей строки матрицы.

Задание 5.2

Введите систему исключений, указанную на рис. 5.1. Встройте собственный тип исключения, которое может возникнуть при решении задачи из Вашего варианта, в общую структуру наследования. Продемонстрируйте обработку исключений с помощью операторов try – throw – catch (исключения генерируются в методах класса Матрица, обрабатываются – в main).

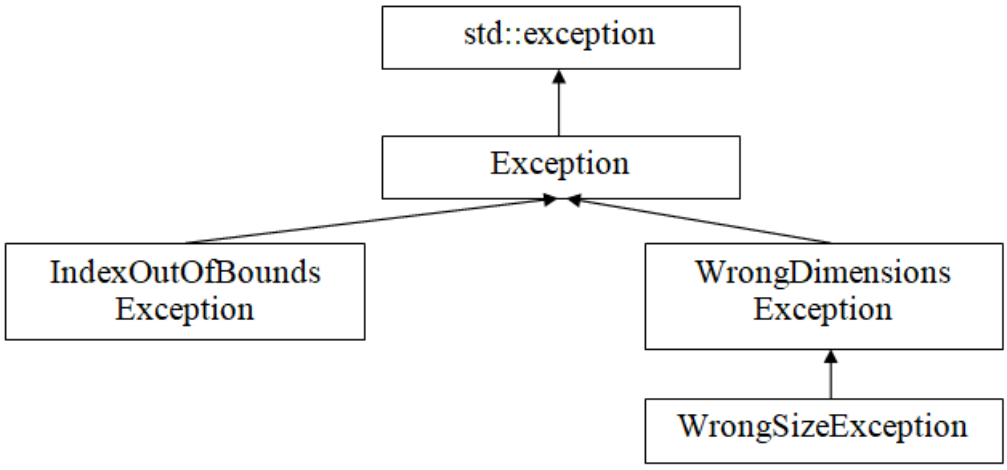

Рисунок 5.1. Схема наследования классов-исключений

Здесь Exception – базовый класс для исключений, которые могут произойти в классе Матрица.

IndexOutOfBounds – выход за границы массива.

WrongDimensionsException – участие в одной операции операндов (в основном, матриц) неправильных размеров (например, сложение матриц, размеры которых отличаются).

Задание 5.3

Сохраните результат обработки нескольких матриц в файл, извлеките результаты из файла, выведите их в консоль. Продемонстрируйте, что результаты, которые записывались в файл, и прочитанные из файла данные не отличаются.

Введите конструкторы и методы для чтения матрицы произвольного размера из файла.

Преобразуйте классы многомерных массивов к шаблонам классов.

Код 5.1. Пример базового класса для исключения Exception, базового класса для многомерного массива BaseMatrix

```
#include <iostream>
#include <fstream>
#include <typeinfo>

using namespace std;


class Exception: public std::exception
{
protected:
 //сообщение об ошибке
 char* str;
public:
 Exception(const char* s)
 {
      str = new char[strlen(s) + 1];
      strcpy_s(str, strlen(s) + 1, s);
 }
 Exception(const Exception& e)
 {
      str = new char[strlen(e.str) + 1];
      strcpy_s(str, strlen(e.str) + 1, e.str);
 }
 ~Exception()
 {
```

```
            delete[] str;
        }

        //функцию вывода можно будет переопределить в производных
классах, когда будет ясна конкретика
        virtual void print()
        {
            cout << "Exception: " << str;
        }
    };

    class BaseMatrix
    {
    protected:
        double** ptr;
        int height;
        int width;
    public:
        BaseMatrix(int Height = 2, int Width = 2)
        {
            if (Height <= 0 || Width <= 0)
                throw Exception("Non-positive size of matrix");
            height = Height;
            width = Width;
            ptr = new double* [height];
            for (int i = 0; i < height; i++)
                ptr[i] = new double[width];
        }

        BaseMatrix(const BaseMatrix& M)
        {
            height = M.height;
            width = M.width;
            ptr = new double* [height];
            for (int i = 0; i < height; i++)
            {
                ptr[i] = new double[width];
                for (int j = 0; j < width; j++)
                    ptr[i][j] = M.ptr[i][j];
            }
        }

        ~BaseMatrix()
        {
```

```
        if (ptr != NULL)
        {
            for (int i = 0; i < height; i++)
                delete[] ptr[i];
            delete[] ptr;
            ptr = NULL;
        }
    }

    void print()
    {
        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
                cout << ptr[i][j] << " ";
            cout << "\n";
        }
    }

    double& operator()(int row, int column)
    {
        if(row<0 || column<0 || row>=height || column>=width)
            throw Exception("Index is out of bounds");
        return ptr[row][column];
    }

    friend ostream& operator << (ostream& ustream, BaseMatrix
obj);
    friend istream& operator >> (istream& ustream, BaseMatrix&
obj);

};

ostream& operator << (ostream& ustream, BaseMatrix obj)
{
    //ustream<<my_manip;
    if (typeid(ustream).name() == typeid(ofstream).name())
    {
        ustream << obj.height << " " << obj.width << "\n";
        for (int i = 0; i < obj.height; i++)
        {
            for (int j = 0; j < obj.width; j++)
                ustream << obj.ptr[i][j] << "\n";
        }
```

```
        return ustream;
    }

    for (int i = 0; i < obj.height; i++)
    {
        for (int j = 0; j < obj.width; j++)
            ustream << obj.ptr[i][j] << " ";
        ustream << "\n";
    }

    return ustream;
}

istream& operator >> (istream& ustream, BaseMatrix& obj)
{
    if (typeid(ustream) == typeid(ifstream))
        ustream >> obj.height >> obj.width;

    for (int i = 0; i < obj.height; i++)
        for (int j = 0; j < obj.width; j++)
            ustream >> obj.ptr[i][j];

    return ustream;
}

ostream& my_manip(ostream& s)
{
    s.precision(4);
    s.fill('%');
    s.width(10);
    return s;
}

int main()
{
    try
    {
        BaseMatrix Wrong(-2, 0);
    }
    catch (Exception e)
    {
        cout << "\nException has been caught: "; e.print();
    }
    cout << "\n";
```

```
 BaseMatrix M;
 cin >> M;
 ofstream fout("out.txt");
 if (fout.is_open())
 {
      fout << M;
      fout.close();
 }
 ifstream fin("out.txt");
 BaseMatrix M1;
 if (fin)
 {
      fin>> M1;
      fin.close();
 }
 cout << M1; char c1; cin >> c1;

 return 0;
}
```

Практическая работа №6. Связные списки

Задание 6.1

1. Реализовать структуру для хранения представленных в задании данных (класс, представляющий Связный список, является шаблоном класса, в main его работа демонстрируется на примере хранения объекта структуры/класса из Вашего варианта, см. табл. 6.1 и код 6.1). Информация в элемент структуры вводится со стандартного потока ввода (предусмотреть автоматическое заполнение).

2. В классе Связный Список ввести деструктор. Поля класса должны быть защищёнными (protected). Ввести открытый интерфейс (public) для работы с ними. Унаследовать класс списка от базового класса Связный список с ключевым словом public. Переопределить функции push() и pop() – для добавления и извлечения элементов из списка. Написать функцию вставки нового элемента в произвольное место списка insert(), удаления remove(), поиска элемента find() по значению. Реализовать указанный вид поиска: под функцией фильтра предполагается поиск элементов, которые удовлетворяют условию element.field >, < value, под поиском – только element.info==value (element.info – поле info в элементе списка element, value – искомое значение). Фильтр возвращает новый список элементов, удовлетворяющих условию. Функцию фильтра и поиска реализовать в итеративной и рекурсивной формах. Класс, объекты которого хранятся в списке, указан в таблице 1.

Таблица 6.1. Классы, экземпляры которых необходимо хранить в списке и тип списка

Вариант	Структура, тип списка
1	Структура «Государство». Минимальный набор полей: название, столица, язык, численность населения, площадь. Добавление: в начало Удаление: из начала Поиск по названию страны, фильтр по площади.
2	Структура «Человек». Минимальный набор полей: фамилия, имя, пол, рост, вес, дата рождения, телефон, адрес. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по фамилии, фильтр по дате рождения.

Продолжение таблицы 6.1

3	Структура «Школьник». Минимальный набор полей: фамилия, имя, пол, класс, дата рождения, адрес. Добавление: в начало Удаление: с конца Поиск по фамилии, фильтр по классу.
4	Структура «Покупатель». Минимальный набор полей: фамилия, имя, город, улица, номера дома и квартиры, номер счёта, средний сумма чека. Добавление: в начало Удаление: из начала Поиск по фамилии, фильтр по средней сумме чека.
5	Структура «Пациент». Минимальный набор полей: фамилия, имя, дата рождения, телефон, адрес, номер карты, группа крови. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по фамилии, фильтр по группе крови.
6	Структура «Команда». Минимальный набор полей: название, город, число побед, поражений, ничьих, количество очков. Добавление: в начало Удаление: с конца Поиск по названию, фильтр по числу побед.
7	Структура «Стадион». Минимальный набор полей: название, виды спорта, год постройки, вместимость, количество арен. Добавление: в начало Удаление: из начала Поиск по названию, фильтр по году постройки.

Продолжение таблицы 6.1

8	Структура «Автовладелец». Минимальный набор полей: фамилия, имя, номер автомобиля, дата рождения, номер техпаспорта. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по номеру техпаспорта, фильтр по фамилии.
9	Структура «Автомобиль». Минимальный набор полей: марка, цвет, серийный номер, количество дверей, год выпуска, цена. Добавление: в начало Удаление: с конца Поиск по серийному номеру, фильтр по цене.
10	Структура «Фильм». Минимальный набор полей: фамилия, имя режиссёра, название, страна, год выпуска, стоимость, доход. Добавление: в начало Удаление: из начала Поиск по названию, фильтр по доходу.
11	Структура «Альбом». Минимальный набор полей: имя или псевдоним исполнителя, название альбома, количество композиций, год выпуска. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по названию альбома, фильтр по имени.
12	Структуры «Комплексное число», «Уравнение с комплексными коэффициентами второй степени», Список из уравнений с комплексными коэффициентами Минимальный набор полей: коэффициенты в уравнении, набор корней уравнения. Добавление: в начало Удаление: с конца Поиск по коэффициентам. Дополнительная функция: поиск корней уравнения.

Продолжение таблицы 6.1

13	Структура «Матрица». Минимальный набор полей: размерности, коэффициенты матрицы. Добавление: в начало Удаление: из начала Поиск по следу, фильтр по размерности. Дополнительная функция: вычисление определителя и следа, произведения.
14	Структура «Вектор». Минимальный набор полей: размерность, коэффициенты вектора. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по длине, фильтр по размерности. Дополнительная функция: вычисление длины вектора, скалярного произведения двух векторов.
15	Структура «Определённый интеграл». Минимальный набор полей: указатель на подынтегральную функцию, шаг вычисления (точность), пределы интегрирования. Добавление: в начало Удаление: с конца Фильтр по значению интеграла. Дополнительная функция: приближённое вычисление значения интеграла.
16	Структура «Уравнение». Минимальный набор полей: указатель на функцию f(x) в уравнении, шаг вычисления (точность), левая и правая границы области, в которой производится поиск решения уравнения. Добавление: в начало Удаление: из начала Поиск по решениям уравнения. Дополнительная функция: приближённое вычисление решения уравнения.

Продолжение таблицы 6.1

17	Структура «Государство». Минимальный набор полей: название, столица, язык, численность населения, площадь. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по численности населения, фильтр по языку.
18	Структура «Человек». Минимальный набор полей: фамилия, имя, пол, рост, вес, дата рождения, телефон, адрес. Добавление: в начало Удаление: с конца Поиск по фамилии, фильтр по адресу.
19	Структура «Школьник». Минимальный набор полей: фамилия, имя, пол, класс, дата рождения, адрес. Добавление: в начало Удаление: из начала Поиск по фамилии, фильтр по дате рождения.
20	Структура «Покупатель». Минимальный набор полей: фамилия, имя, город, улица, номера дома и квартиры, номер счёта. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по номеру счета, фильтр по городу.
21	Структура «Пациент». Минимальный набор полей: фамилия, имя, дата рождения, телефон, адрес, номер карты, группа крови. Добавление: в начало Удаление: с конца Поиск номеру карты, фильтр по группе крови..

Продолжение таблицы 6.1

22	Структура «Команда». Минимальный набор полей: название, город, число побед, поражений, ничьих, количество очков. Добавление: в начало Удаление: из начала Поиск по названию, фильтр по числу побед.
23	Структура «Автовладелец». Минимальный набор полей: фамилия, имя, номер автомобиля, дата рождения, номер техпаспорта. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по номеру автомобиля, фильтр по имени.
24	Структура «Государство». Минимальный набор полей: название, столица, язык, численность населения, площадь. Добавление: в начало Удаление: с конца Поиск по столице, фильтр по численности населения.
25	Структура «Программист». Минимальный набор полей: фамилия, имя, email, skype, telegram, основной язык программирования, текущее место работы, уровень (число от 1 до 10). Добавление: в начало Удаление: из начала Поиск по email, фильтр по уровню.
26	Структура «Спортсмен». Минимальный набор полей: фамилия, имя, возраст, гражданство, вид спорта, количество медалей. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по фамилии, фильтр по числу медалей.

Продолжение таблицы 6.1

27	Структура «Полином». Минимальный набор полей: степень полинома, коэффициенты Добавление: в начало Удаление: с конца Поиск по коэффициентам, фильтр по степени полинома. Дополнительная функция: поиск корня полинома на указанном отрезке.
28	Структура «Профиль в соц.сети». Минимальный набор полей: псевдоним, адрес страницы, возраст, количество друзей, интересы, любимая цитата. Добавление: в начало Удаление: из начала Поиск по псевдониму, фильтр по количеству друзей.
29	Структура «Велосипед». Минимальный набор полей: марка, тип, тип тормозов, количество колес, диаметр колеса, наличие амортизаторов, детский или взрослый. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по марке, фильтр по диаметру колеса.
30	Структура «Ноутбук». Минимальный набор полей: производитель, модель, размер экрана, процессор, количество ядер, объем оперативной памяти, объем диска, тип диска, цена. Добавление: в начало Удаление: с конца Поиск по марке, фильтр по цене.
31	Структура «Смартфон». Минимальный набор полей: марка, размер экрана, количество камер, объем аккумулятора, максимальное количество часов без подзарядки, цена. Добавление: в начало Удаление: из начала Поиск по марке, фильтр по размеру экрана.

Окончание таблицы 6.1

32	Структура «Фотоаппарат». Минимальный набор полей: производитель, модель, тип, размер матрицы, количество мегапикселей, вес, тип карты памяти, цена. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по модели, фильтр по размеру матрицы.
33	Структура «Супергерой». Минимальный набор полей: псевдоним, настоящее имя, дата рождения, пол, суперсила, слабости, количество побед, рейтинг силы. Добавление: в начало Удаление: с конца Поиск по псевдониму, фильтр по рейтингу силы.
34	Структура «Сериал». Минимальный набор полей: название, режиссер, количество сезонов, популярность, рейтинг, дата запуска, страна. Добавление: в начало Удаление: из начала Поиск по режиссеру, фильтр по популярности.
35	Структура «Программа». Минимальный набор полей: название, версия, лицензия, есть ли версия для android, iOS, платная ли, стоимость, разработчик, открытость кода, язык кода. Добавление: в конец (стек) Удаление: с конца (стек) Поиск по названию, фильтр по платности.

Задание 6.2

1. Кроме фильтра по указанным полям, необходимо реализовать универсальный фильтр по произвольному полю, который принимает указатель на функцию. Эта функция возвращает 1, если элемент удовлетворяет условию и 0 – в противном случае.

2. Выделите память для списка динамически (используя указатель на базовый класс), а затем, в конце работы кода, освободите её. Проверьте правильную работу деструкторов базового и производного классов (используйте виртуальный деструктор). Преобразуйте указатель на базовый класс в указатель

типа производного класса с помощью оператора dynamic_cast. Проверьте работу деструкторов в случае выполнения операции освобождения памяти для преобразованного указателя.

3. Напишите функции сохранения списка файл* save() и извлечения списка из файла* load(). Для ввода/вывода элемента списка в/из файла используйте переопределённые функции ввода/вывода в поток (сделайте различие между вводом/выводом в консоль и вводом/выводом в файл).

Код 6.1. Пример базового класса для связного списка LinkedList и наследования класса Очередь (Queue)

```
template<class T>
class Element
{
 //protected:
public:
 //переместить в protected
 Element* next;
 Element* prev;
 T info;

 Element(T data)
 {
      next = prev = NULL;
      info = data;
 }

 Element(Element* Next, Element* Prev, T data)
 {
      next = Next;
      prev = Prev;
      info = data;
 }

 Element(const Element& el)
 {
      next = el.next;
      prev = el.prev;
      info = el.info;
 }

 template<class T1>
 friend ostream& operator<<(ostream& s, Element<T1>& el);
```

```
};

template<class T1>
ostream& operator<<(ostream& s, Element<T1>& el)
{
 s << el.info;
 return s;
}

template<class T>
class LinkedList
{
 //protected:
public:
 //переместить в protected
 Element<T>* head;
 Element<T>* tail;
 int count;

 LinkedList()
 {
      head = tail = NULL;
      count = 0;
 }

 virtual Element<T>* pop() = 0;
 virtual Element<T>* push(T value) = 0;

 Element<T>& operator[](int index)
 {
      Element<T>* current = head;

      for (int i = 0;
           current != NULL && i < index;
           current = current->next, i++);

      return *current;
 }

 virtual bool isEmpty() { return (LinkedList<T>::count == 0);
}

 template<class T1>
 friend ostream& operator<<(ostream& s, LinkedList<T1>& el);
```

```
     virtual ~LinkedList()
     {
          cout << "\nBase class destructor";
          //дописать деструктор базового класса
          head = NULL; tail = NULL;
     }
    };

    template<class T1>
    ostream& operator<<(ostream& s, LinkedList<T1>& el)
    {
     Element<T1>* current;
     for (current = el.head; current != NULL; current = current-
>next)
          s << *current << "; ";
     return s;
    }

    template<class T>
    class Queue : public LinkedList<T>
    {
    public:
     Queue<T>() : LinkedList<T>() {}
     Queue<T>(T* arr, int len) : LinkedList<T>(arr, len) {}

     virtual Element<T>* push(T value)
     {
          if (LinkedList<T>::head == NULL) //if(count==0)
          {
               //пустой список
               LinkedList<T>::head = new Element<T>(value);
               LinkedList<T>::tail = LinkedList<T>::head;
          }
          else
          {
               //элементы уже есть
               LinkedList<T>::tail->next = new Element<T>(value);
               LinkedList<T>::tail->next->prev =
LinkedList<T>::tail;
               LinkedList<T>::tail = LinkedList<T>::tail->next;
          }
          LinkedList<T>::count++;
          return LinkedList<T>::tail;
```

```
 }

 virtual Element<T>* pop()
 {
      if (LinkedList<T>::head == NULL)
           return NULL;
      Element<T>* res = LinkedList<T>::head;
      if (LinkedList<T>::head == LinkedList<T>::tail)
           LinkedList<T>::head = LinkedList<T>::tail = NULL;
      else
           LinkedList<T>::head = LinkedList<T>::head->next;
      LinkedList<T>::count--;
      return res;
 }

 virtual ~Queue() { cout << "\nQueue class destructor"; }
};

class my_class
{
public:
 int data;
 my_class(int v = 0) { data = v; }

 ~my_class() { cout << "\nmy_class destructor"; }

 friend ostream& operator<<(ostream& s, my_class& value);

};

ostream& operator<<(ostream& s, my_class& value)
{
 s << value.data;
 return s;
}

int main()
{
 if (true)
 {
      cout << "Only object\n";
      Queue<int> Q;
      for (int i = 0; i < 100; i += 5)
           Q.push(i);
```

```
        for (; !Q.isEmpty(); cout << Q.pop()->info << ", ");
        cout << "\n";
    }

    if (true)
    {
        cout << "\nBase pointer 1\n";
        LinkedList<int>* ptr = new Queue<int>;
        for (int i = 1; i < 100; i += 5)
            ptr->push(i);
        cout << *ptr<< "\n";
        delete ptr;
        //добавить виртуальные деструкторы
    }
    if (true)
    {
        cout << "\nBase pointer 2\n";
        LinkedList<int>* ptr; Queue<int>* ptr2 = new
Queue<int>; ptr = ptr2;
        for (int i = 1; i < 100; i += 5)
            ptr->push(i);
        cout << *ptr << "\n";
        delete ptr;
        //добавить виртуальные деструкторы
    }
    if (true)
    {
        cout << "\nQueue pointer\n";
        Queue<int>* ptr = new Queue<int>;
        for (int i = 2; i < 100; i += 5)
            ptr->push(i);
        cout << *ptr << "\n";
        delete ptr;
    }
    if (true)
    {
        cout << "\nmy_class Queue test\n";
        Queue<my_class> Q;
        for (int i = 3; i < 100; i += 5)
            Q.push(my_class(i));
        cout << Q;
    }return 0;}
```

Практическая работа №7. Бинарные деревья

1. Постройте шаблон дерева поиска. Используйте его для хранения объектов класса C по ключам K в соответствии с таблицей (7.1) (ключи считаются уникальными).
2. Переопределите функцию вывода содержимого дерева с помощью функций successor() / predecessor() класса Node (переход к предыдущему и следующему элементу). Создайте интерфейс для этих функций на основе перегрузки операций ++ и --. Продемонстрируйте работу операций с помощью функции, которая находит некоторый элемент (значение для поиска задано с помощью параметра функции) и выводит все узлы, меньшие / большие, чем найденный элемент.
3. Создайте интерфейс для функции поиска в виде перегрузки операции [].
4. Введите функцию удаления узла Remove(), которая принимает указатель на удаляемый узел.
5. Унаследуйте класс АВЛ-дерево от класса Дерево поиска. Переопределите функции добавления и удаления узла.

Код 7.1. Класс бинарного дерева поиска

```
    #include <iostream>

    using namespace std;
    //узел
    template<class T>
    class Node
    {
    protected:
     //закрытые переменные Node N; N.data = 10 вызовет
ошибку
     T data;

     //не можем хранить Node, но имеем право хранить
указатель
     Node* left;
     Node* right;
    Node* parent;

     //переменная, необходимая для поддержания баланса
```

```
дерева
    int height;
   public:
    //доступные извне переменные и функции
   virtual void setData(T d) { data = d; }
    virtual T getData() { return data; }
    int getHeight() { return height; }

    virtual Node* getLeft() { return left; }
    virtual Node* getRight() { return right; }
    virtual Node* getParent() { return parent; }
    virtual void setLeft(Node* N) { left = N; }
    virtual void setRight(Node* N) { right = N; }
    virtual void setParent(Node* N) { parent = N; }

    //Конструктор. Устанавливаем стартовые значения для
указателей
    Node<T>(T n)
    {
        data = n;
         left = right = parent = NULL;
        height = 1;
    }

    Node<T>()
    {
        left = NULL;
        right = NULL;
        parent = NULL;
        data = 0;
        height = 1;
    }
    virtual void print()
    {
        cout << "\n" << data;
    }

    virtual void setHeight(int h)
```

```
    {
        height = h;
    }

    template<class T> friend ostream& operator<<
(ostream& stream, Node<T>& N);};
   template<class T>
   ostream& operator<< (ostream& stream, Node<T>& N)
   {
stream << "\nNode data: " << N.data << ", height: " <<
N.height;
    return stream;
   }
   template<class T>
   void print(Node<T>* N) { cout << "\n" << N-
>getData(); }
   template<class T>
   class Tree
   {
   protected:
    //корень - его достаточно для хранения всего дерева
    Node<T>* root;
   public:
    //доступ к корневому элементу
    virtual Node<T>* getRoot() { return root; }

    //конструктор дерева: в момент создания дерева ни
одного узла нет, корень смотрит в никуда
    Tree<T>() { root = NULL; }
   //рекуррентная функция добавления узла. Устроена
аналогично, но вызывает сама себя - добавление в левое
или правое поддерево
    virtual Node<T>* Add_R(Node<T>* N)
    {
        return Add_R(N, root);
    }

    virtual Node<T>* Add_R(Node<T>* N, Node<T>* Cur-
```

```
rent)
     {
          if (N == NULL) return NULL;
          if (root == NULL)
          {
               root = N;
               return N;
          }
    if (Current->getData() > N->getData())
          {
               //идем влево
               if (Current->getLeft() != NULL)
                    Current->setLeft(Add_R(N, Current-
>getLeft()));
               else
                    Current->setLeft(N);
               Current->getLeft()->setParent(Current);
          }

    if (Current->getData() < N->getData())
          {
               //идем вправо
               if (Current->getRight() != NULL)
    Current->setRight(Add_R(N, Current->getRight()));
                 else
                    Current->setRight(N);
               Current->getRight()->setParent(Current);
    }
          if (Current->getData() == N->getData())
               //нашли совпадение
               ;
          //для несбалансированного дерева поиска
          return Current;
     }

     //функция для добавления числа. Делаем новый узел с
этими данными и вызываем нужную функцию добавления в
дерево
```

```
    virtual void Add(T n)
    {
        Node<T>* N = new Node<T>;
        N->setData(n);
        Add_R(N);
    }
   virtual Node<T>* Min(Node<T>* Current=NULL)
    {
        //минимум - это самый "левый" узел. Идём по де-
реву всегда влево
        if (root == NULL) return NULL;
         if(Current==NULL)
            Current = root;
        while (Current->getLeft() != NULL)
            Current = Current->getLeft();

        return Current;
    }

    virtual Node<T>* Max(Node<T>* Current = NULL)
    {
        //минимум - это самый "правый" узел. Идём по
дереву всегда вправо
        if (root == NULL) return NULL;
        if (Current == NULL)
   Current = root;
        while (Current->getRight() != NULL)
            Current = Current->getRight();

        return Current;
    }

    //поиск узла в дереве. Второй параметр - в каком
поддереве искать, первый - что искать
    virtual Node<T>* Find(int data, Node<T>* Current)
    {
        //база рекурсии
        if (Current == NULL) return NULL;
```

```
        if (Current->getData() == data) return Current;
        //рекурсивный вызов

        if (Current->getData() > data) return
Find(data, Current->getLeft());
    if (Current->getData() < data) return Find(data,
Current->getRight());
     }
    //три обхода дерева
     virtual void PreOrder(Node<T>* N, void
(*f)(Node<T>*))
     {
        if (N != NULL)
            f(N);
        if (N != NULL && N->getLeft() != NULL)
            PreOrder(N->getLeft(), f);
        if (N != NULL && N->getRight() != NULL)
            PreOrder(N->getRight(), f);
     }
    //InOrder-обход даст отсортированную последователь-
ность
     virtual void InOrder(Node<T>* N, void
(*f)(Node<T>*))
     {
        if (N != NULL && N->getLeft() != NULL)
            InOrder(N->getLeft(), f);
        if (N != NULL)
            f(N);
    if (N != NULL && N->getRight() != NULL)
            InOrder(N->getRight(), f);
     }

     virtual void PostOrder(Node<T>* N, void
(*f)(Node<T>*))
     {
        if (N != NULL && N->getLeft() != NULL)
            PostOrder(N->getLeft(), f);
        if (N != NULL && N->getRight() != NULL)
```

```
            PostOrder(N->getRight(), f);
        if (N != NULL)
            f(N);
    }};
   int main()
   {
    Tree<double> T;
    int arr[15];
    int i = 0;
    for (i = 0; i < 15; i++) arr[i] = (int)(100 *
cos(15 * double(i+1)));
    for (i = 0; i < 15; i++)
        T.Add(arr[i]);
   Node<double>* M = T.Min();
    cout << "\nMin = " << M->getData() << "\tFind " <<
arr[3] << ": " << T.Find(arr[3], T.getRoot());

    void (*f_ptr)(Node<double>*); f_ptr = print;
    cout << "\n-----\nInorder:";
    T.InOrder(T.getRoot(), f_ptr);
    char c; cin >> c;
    return 0;
   }
```

Таблица 7.1. Ключ и тип объекта, хранимого в классе АВЛ-дерево

Вариант	**Ключ**	**Класс С**
1.	**Адрес**	«Объект жилой недвижимости». Минимальный набор полей: адрес, тип (перечислимый тип: городской дом, загородный дом, квартира, дача), общая площадь, жилая площадь, цена.
2.	**Название**	«Сериал». Минимальный набор полей: название, продюсер, количество сезонов, популярность, рейтинг, дата запуска, страна.
3.	**Назва-**	«Смартфон». Минимальный набор полей: название, размер экрана, коли-

	ние	чество камер, объем аккумулятора, максимальное количество часов без подзарядки, цена.
4.	**Фамилия и имя**	«Спортсмен». **Минимальный набор полей: фамилия, имя, возраст, гражданство, вид спорта, количество медалей.**
5.	**Фамилия и имя**	«Врач». Минимальный набор полей: фамилия, имя, специальность, должность, стаж, рейтинг (вещественное число от 0 до 100).
6.	**Международный код**	«Авиакомпания». Минимальный набор полей: название, международный код, количество обслуживаемых линий, страна, интернет-адрес сайта, рейтинг надёжности (целое число от -10 до 10).
7.	**Название**	«Книга». Минимальный набор полей: фамилия (первого) автора, имя (первого) автора, название, год издания, название издательства, число страниц, вид издания (перечислимый тип: электронное, бумажное или аудио), тираж.
8.	**Номер в каталоге**	«Небесное тело». Минимальный набор полей: тип (перечислимый тип: астероид, естественный спутник, планета, звезда, квазар), имя (может отсутствовать), номер в небесном каталоге, удаление от Земли, расчётная масса в миллиардах тонн (для сверхбольших объектов допускается значение Inf, которое должно корректно обрабатываться).
9.	**Название**	«Населённый пункт». Минимальный набор полей: название, тип (перечислимый тип: город, посёлок, село, деревня), числовой код региона, численность населения, площадь.
10.	**Имя или псевдоним исполни-**	«Музыкальный альбом». Минимальный набор полей: имя или псевдоним исполнителя, название альбома, количество композиций, год выпуска, количество проданных экземпляров.

	теля, название альбома	
11.	**Серийный номер**	«Автомобиль». Минимальный набор полей: имя модели, название производителя, цвет, серийный номер, количество дверей, год выпуска, цена.
12.	**Регистрационный номер автомобиля**	«Автовладелец». Минимальный набор полей: фамилия, имя, регистрационный номер автомобиля, дата рождения, номер техпаспорта.
13.	**Название фильма**	«Фильм». Минимальный набор полей: фамилия, имя режиссёра, название, страна, год выпуска, стоимость, доход.
14.	**Название, год постройки**	«Стадион». Минимальный набор полей: название, виды спорта, год постройки, вместимость, количество арен.
15.	**Название, город**	«Спортивная Команда». Минимальный набор полей: название, город, число побед, поражений, ничьих, количество очков.
16.	**Фамилия и имя**	«Пациент». Минимальный набор полей: фамилия, имя, дата рождения, телефон, адрес, номер карты, группа крови.
17.	**Фамилия и**	«Покупатель». Минимальный набор полей: фамилия, имя, город, улица,

	имя	номера дома и квартиры, номер счёта, средняя сумма чека.
18.	**Фамилия и имя**	«Школьник». Минимальный набор полей: фамилия, имя, пол, класс, дата рождения, адрес.
19.	**Название**	«Государство». Минимальный набор полей: название, столица, язык, численность населения, площадь.
20.	**Адрес**	«Сайт». Минимальный набор полей: название, адрес, дата запуска, язык, тип (блог, интернет-магазин и т.п.), cms, дата последнего обновления, количество посетителей в сутки.
21.	**Фамилия и имя**	«Человек». Минимальный набор полей: фамилия, имя, пол, рост, возраст, вес, дата рождения, телефон, адрес.
22.	**Название**	«Программа». Минимальный набор полей: название, версия, лицензия, есть ли версия для android, iOS, платная ли, стоимость, разработчик, открытость кода, язык кода.
23.	**Производитель, модель**	«Ноутбук». Минимальный набор полей: производитель, модель, размер экрана, процессор, количество ядер, объем оперативной памяти, объем диска, тип диска, цена.
24.	**Марка, диаметр колеса**	«Велосипед». Минимальный набор полей: марка, тип, тип тормозов, количество колес, диаметр колеса, наличие амортизаторов, детский или взрослый.
25.	**Фамилия и**	«Программист». Минимальный набор полей: фамилия, имя, email, skype, telegram, основной язык программирования, текущее место

	имя	работы, уровень (число от 1 до 10).
26.	**Псевдоним**	«Профиль в соц.сети». Минимальный набор полей: псевдоним, адрес страницы, возраст, количество друзей, интересы, любимая цитата.
27.	**Псевдоним**	«Супергерой». Минимальный набор полей: псевдоним, настоящее имя, дата рождения, пол, суперсила, слабости, количество побед, рейтинг силы.
28.	**Производитель, модель**	«Фотоаппарат». Минимальный набор полей: производитель, модель, тип, размер матрицы, количество мегапикселей, вес, тип карты памяти, цена.
29.	**Полный адрес**	«Файл». Минимальный набор полей: полный адрес, краткое имя, дата последнего изменения, дата последнего чтения, дата создания.
30.	**Производитель, название**	«Самолет». Минимальный набор полей: название, производитель, вместимость, дальность полета, максимальная скорость.

Задание 7.2

Используйте шаблон класса Heap (куча, пирамида) для хранения объектов в соответствии с таблицей 7.2 (используется упорядоченность по приоритету, в корне дерева – максимум). Реализуйте функции просеивания элементов вверх SiftUp() и вниз SiftDown() = Heapify(), на их основе реализуйте функции добавления узла push() и удаления корня дерева ExtractMax(). Выведите элементы Heap в порядке убывания приоритета с её помощью. Добавьте операцию удаления произвольного элемента Remove() (требуется просеивание вверх или вниз?) и деструктор.

Таблица 7.2. Варианты типов хранимых в контейнере значений и порядок сравнения полей для упорядочения объектов

Вариант	Класс C	Приоритет
1.	«Объект жилой недвижимости». Минимальный набор полей: адрес, тип (перечислимый тип: городской дом, загородный дом, квартира, дача), общая площадь, жилая площадь, цена.	Цена; адрес (по возрастанию)
2.	«Сериал». Минимальный набор полей: название, продюсер, количество сезонов, популярность, рейтинг, дата запуска, страна.	Рейтинг; название (по возрастанию)
3.	«Смартфон». Минимальный набор полей: название, размер экрана, количество камер, объем аккумулятора, максимальное количество часов без подзарядки, цена.	Цена, количество камер, размер экрана; название марки (по возрастанию)
4.	«Спортсмен». Минимальный набор полей: фамилия, имя, возраст, гражданство, вид спорта, количество медалей.	Количество медалей; возраст (по возрастанию); фамилия и имя (по возрастанию)
5.	«Врач». Минимальный набор полей: фамилия, имя, специальность, должность, стаж, рейтинг (вещественное число от 0 до 100).	Рейтинг, стаж; фамилия и имя (по возрастанию)
6.	«Авиакомпания». Минимальный набор полей: название, международный код, количество обслуживаемых линий, страна, интернет-адрес сайта, рейтинг надёжности (целое число от -10 до 10).	Надёжность, количество обслуживаемых линий; название (по возрастанию)
7.	«Книга».	Тираж; год издания

	Минимальный набор полей: фамилия (первого) автора, имя (первого) автора, название, год издания, название издательства, число страниц, вид издания (перечислимый тип: электронное, бумажное или аудио), тираж.	(по возрастанию); название (по возрастанию)
8.	«Небесное тело». Минимальный набор полей: тип (перечислимый тип: астероид, естественный спутник, планета, звезда, квазар), имя (может отсутствовать), номер в небесном каталоге, удаление от Земли, расчётная масса в миллиардах тонн (для сверхбольших объектов допускается значение Inf, которое должно корректно обрабатываться).	Масса; номер в каталоге (по возрастанию)
9.	«Населённый пункт». Минимальный набор полей: название, тип (перечислимый тип: город, посёлок, село, деревня), числовой код региона, численность населения, площадь.	Площадь, численность населения; числовой код региона (по возрастанию)
10.	«Музыкальный альбом». Минимальный набор полей: имя или псевдоним исполнителя, название альбома, количество композиций, год выпуска, количество проданных экземпляров.	Количество проданных экземпляров; количество композиций; год выпуска (по возрастанию); имя или псевдоним исполнителя (по возрастанию)
11.	«Фильм». Минимальный набор полей: фамилия, имя режиссёра, название, страна, год выпуска, стоимость, доход.	Доход, стоимость; год выпуска (по возрастанию); фамилия и имя режиссера (по возрастанию); название фильма (по возрастанию)

12.	«Автомобиль». Минимальный набор полей: имя модели, цвет, серийный номер, количество дверей, год выпуска, цена.	Цена; год выпуска; марка (по возрастанию); серийный номер (по возрастанию)
13.	«Автовладелец». Минимальный набор полей: фамилия, имя, регистрационный номер автомобиля, дата рождения, номер техпаспорта.	Регистрационный номер автомобиля; номер техпаспорта; фамилия и имя автовладельца (по возрастанию)
14.	«Стадион». Минимальный набор полей: название, виды спорта, год постройки, вместимость, количество арен.	Вместимость, количество арен, год постройки; название (по возрастанию)
15.	«Спортивная Команда». Минимальный набор полей: название, город, число побед, поражений, ничьих, количество очков.	Число побед, число ничьих; число поражений (по возрастанию); название (по возрастанию)
16.	«Пациент». Минимальный набор полей: фамилия, имя, дата рождения, телефон, адрес, номер карты, группа крови.	Номер карты; группа крови; фамилия и имя (по возрастанию)
17.	«Покупатель». Минимальный набор полей: фамилия, имя, город, улица, номера дома и квартиры, номер счёта, средняя сумма чека.	Средняя сумма чека; номер счёта; фамилия и имя (по возрастанию)
18.	«Школьник». Минимальный набор полей: фамилия, имя, пол, класс, дата рождения, адрес.	Класс; дата рождения (по возрастанию); фамилия и имя (по возрастанию)

19.	«Человек». Минимальный набор полей: фамилия, имя, пол, рост, возраст, вес, дата рождения, телефон, адрес.	Возраст, рост; вес (по возрастанию); фамилия и имя (по возрастанию)
20.	«Государство». Минимальный набор полей: название, столица, язык, численность населения, площадь.	Численность населения; площадь; название (по возрастанию)
21.	«Сайт». Минимальный набор полей: название, адрес, дата запуска, язык, тип (блог, интернет-магазин и т.п.), cms, дата последнего обновления, количество посетителей в сутки.	Количество посетителей в сутки, дата последнего обновления; адрес (по возрастанию)
22.	«Программа». Минимальный набор полей: название, версия, лицензия, есть ли версия для android, iOS, платная ли, стоимость, разработчик, открытость кода, язык кода.	Стоимость, версия; название (по возрастанию)
23.	«Ноутбук». Минимальный набор полей: производитель, модель, размер экрана, процессор, количество ядер, объем оперативной памяти, объем диска, тип диска, цена.	Цена, количество ядер, объем оперативной памяти, размер экрана; модель (по возрастанию)
24.	«Велосипед». Минимальный набор полей: марка, тип, тип тормозов, количество колес, диаметр колеса, наличие амортизаторов, детский или взрослый.	Диаметр колеса, количество колес; марка (по возрастанию)
25.	«Программист». Минимальный набор полей: фамилия, имя, email, skype, telegram, основной язык программирования, текущее место работы,	Уровень; основной язык программирования (по возрастанию); фамилия и имя (по

	уровень (число от 1 до 10).	возрастанию)
26.	«Профиль в соц.сети». Минимальный набор полей: псевдоним, адрес страницы, возраст, количество друзей, интересы, любимая цитата.	Количество друзей; псевдоним (по возрастанию)
27.	«Супергерой». Минимальный набор полей: псевдоним, настоящее имя, дата рождения, пол, суперсила, слабости, количество побед, рейтинг силы.	Рейтинг силы, количество побед; псевдоним (по возрастанию)
28.	«Фотоаппарат». Минимальный набор полей: производитель, модель, тип, размер матрицы, количество мегапикселей, вес, тип карты памяти, цена.	Цена, вес, размер матрицы; модель (по возрастанию)
29.	«Файл». Минимальный набор полей: полный адрес, краткое имя, дата последнего изменения, дата последнего чтения, дата создания.	Дата последнего изменения, дата последнего чтения; полный адрес (по возрастанию)
30.	«Самолет». Минимальный набор полей: название, производитель, вместимость, дальность полета, максимальная скорость.	Вместимость, дальность полета; производитель (по возрастанию), название (по возрастанию)

Код 7.2. Класс Heap (куча, пирамида)

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

 //узел дерева
template <class T>
class Node
```

```
{
private:
 T value;
public:
 //установить данные в узле
 T getValue() { return value; }
 void setValue(T v) { value = v; }

 //сравнение узлов
 int operator<(Node N)
 {
     return (value < N.getValue());
 }
 int operator>(Node N)
 {
     return (value > N.getValue());
 }
 //вывод содержимого одного узла
 void print()
 {
     cout << value;
 }
};
template <class T>
void print(Node<T>* N)
{
 cout << N->getValue() << "\n";
}

//куча (heap)
template <class T>
class Heap
{
private:
//массив
 Node<T>* arr;
 //сколько элементов добавлено
 int len;
```

```
    //сколько памяти выделено
    int size;
   public:
    //доступ к вспомогательным полям кучи и оператор
индекса
    int getCapacity() { return size; }
    int getCount() { return len; }

    Node<T>& operator[](int index)
    {
        if (index < 0 || index >= len)
            ;//?
            return arr[index];}
   //конструктор
    Heap<T> (int MemorySize = 100)
    {
        arr = new Node<T>[MemorySize];
        len = 0;
        size = MemorySize;
    }
    //поменять местами элементы arr[index1],
arr[index2]
    void Swap(int index1, int index2)
    {
        if (index1 <= 0 || index1 >= len)
            ;
        if (index2 <= 0 || index2 >= len)
            ;
        //здесь нужна защита от дурака

        Node<T> temp;
        temp = arr[index1];
        arr[index1] = arr[index2];
    arr[index2] = temp;
    }

    //скопировать данные между двумя узлами
    void Copy(Node<T>* dest, Node<T>* source)
```

```
    {
        dest->setValue(source->getValue());
    }

    //функции получения левого, правого дочернего эле-
мента, родителя или их индексов в массиве
    Node<T>* GetLeftChild(int index)
    {
        if (index < 0 || index * 2 >= len)
            ;
        //здесь нужна защита от дурака
        return &arr[index * 2 + 1];
    }
    Node<T>* GetRightChild(int index)
    {
        if (index < 0 || index * 2 >= len);
        //здесь нужна защита от дурака
         return &arr[index * 2 + 2];
    }
    Node<T>* GetParent(int index)
    {
        if (index <= 0 || index >= len)
            ;
        //здесь нужна защита от дурака

        if (index % 2 == 0)
            return &arr[index / 2 - 1];
        return &arr[index / 2];
    }
   int GetLeftChildIndex(int index)
    {
        if (index < 0 || index * 2 >= len)
            ;
        //здесь нужна защита от дурака
        return index * 2 + 1;
    }

    int GetRightChildIndex(int index)
```

```
    {
        if (index < 0 || index * 2 >= len)
            ;
        //здесь нужна защита от дурака

        return index * 2 + 2;
    }
    int GetParentIndex(int index)
    {
        if (index <= 0 || index >= len)
            ;
        //здесь нужна защита от дурака
    if (index % 2 == 0)
            return index / 2 - 1;
        return index / 2;
    }

    //просеять элемент вверх
    void SiftUp(int index = -1)
    {
        if (index == -1) index = len - 1;
        int parent = GetParentIndex(index);
        int index2 = GetLeftChildIndex(parent);
        if (index2 == index) index2 =
GetRightChildIndex(parent);
        int max_index = index;

        if (index < len && index2 < len && parent >= 0)
   {
            if (arr[index] > arr[index2])
                max_index = index;
            if (arr[index] < arr[index2])
                max_index = index2;
        }
        if (parent < len && parent >= 0 &&
arr[max_index]>arr[parent])
        {
            //нужно просеивание вверх
```

```
                Swap(parent, max_index);
                SiftUp(parent);
            }
        }
        //добавление элемента - вставляем его в конец мас-
сива и просеиваем вверх
        template <class T>
        void push(T v)
        {
            Node<T>* N = new Node<T>;
            N->setValue(v);
            push(N);
        }
       template <class T>
        void push(Node<T>* N)
        {
            if (len < size)
            {
                Copy(&arr[len], N);
                len++;SiftUp();}}
        //перечислить элементы кучи и применить к ним функ-
цию
        void Straight(void(*f)(Node<T>*))
        {
            int i;
            for (i = 0; i < len; i++)
            {
            f(&arr[i]);
            }
        }

       //перебор элементов, аналогичный проходам бинарного
дерева
        void InOrder(void(*f)(Node<T>*), int index = 0)
        {
            if (GetLeftChildIndex(index) < len)
                PreOrder(f, GetLeftChildIndex(index));
            if (index >= 0 && index < len)
```

```
            f(&arr[index]);
        if (GetRightChildIndex(index) < len)
            PreOrder(f, GetRightChildIndex(index));
    }
};
int main()
{
 Heap<int> Tree;

 Tree.push(1);
 Tree.push(-1);
 Tree.push(-2);
 Tree.push(2);
 Tree.push(5);
 Tree.push(6);
 Tree.push(-3);
 Tree.push(-4);
 cout << "\n-----\nStraight:";
void(*f_ptr)(Node<int>*); f_ptr = print;
 Tree.Straight(f_ptr);
return 0;
}
```

ВОПРОСЫ К ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ

Инкапсуляция, классы

1. Чем отличается класс от структуры в смысле Си? Как выбрать между классом и структурой в C++?
2. Что следует оставлять в основном теле определения класса и что разумно выносить из него? Приведите убедительный пример.
3. Могут ли разные части определения класса находиться в разных файлах?
4. Пространство имён класса. Как обращаться к членам класса: а) изнутри него, б) извне класса, в) из производных классов?
5. Можно ли внутри определения класса полноценно пользоваться объектами этого класса?
6. Доступ к членам класса: спецификаторы private, protected, public, дружественные функции.
7. Понятие интерфейса класса: что нежелательно туда выносить? Почему?
8. Устройство записи определения класса в самом общем виде: возможности.
9. Может ли один класс включен внутрь другого несколько раз?
10. Статические и нестатические атрибуты и методы класса: различия и возможности.
11. Чем отличается статический метод от статического атрибута класса?
12. Есть ли особенности наследования у классов со статическими членами?
13. Константные и неконстантные параметры метода. Константный метод: назначение, оформление.
14. Могут ли одноимённые методы с одинаковыми параметрами различаться лишь константностью по одному из параметров?
15. Предопределённый указатель this. Назначение, использование и предосторожности.
16. Можно ли указатель this передать не методу класса, а во внешнюю функцию?
17. Что делать, если часть начинки двух классов должна совпадать, а классы "не должны догадываться" друг о друге?

18. Дружественные функции: назначение и отличия от методов класса. Понятие дружественного класса.

19. Может ли класс A быть другом класса B, а обратное не справедливо?

20. Дружественные базовому классу функции и их использование из порождённых классов.

21. Может ли одна функция быть дружественной нескольким классам одновременно?

22. Может ли метод одного класса быть другом другого класса?

Перегрузка функций и операций

23. Перегрузка функций: назначение и правила. Проиллюстрируйте простыми примерами допустимых и недопустимых вариантов перегрузки.

24. Что будет, если в классе перегружена функция, внешняя по отношению к классу?

25. Перегрузка операций: основные правила и ограничения. Выбор реализации между чисто внешними, дружественными функциями и методами класса.

26. Может ли одна и та же операция в одном и том же классе быть перегруженной разными способами?

27. Общий случай выбора функции при наличии обычных перегруженных функций, функций с переменным количеством параметров, базового шаблона функции и его уточнений (спецификаций, перегрузок).

28. Перегрузка операции () и понятие функтора.

29. Перегрузка операций -> и ->* и концепция "умного указателя".

30. Перегрузка операции индексирования [], её константный и неконстантный варианты. Обобщение индекса и многомерные массивы.

31. Перегрузка операций << и >> и введение в класс собственных инструментов ввода/вывода.

32. Что и как нужно сделать, чтобы мочь выводить содержимое объекта некоторого класса на экран?

33. Можно ли так перегрузить вывод содержимого объекта некоторого класса на экран несколькими разными способами и их сочетать?

34. Перегрузка одноместных операций -, * и &. Зачем она нужна?

35. Как автоматически обнаруживать потери динамической памяти?

36. Перегрузка new и delete. Различие скалярной и векторной форм.

37. В чём различия обработки поступившего указателя в delete и delete[]?

38. Перегрузка двуместных операций +, -, *, /, %, &, |, ^.

39. Перегрузка двуместных операций с присваиванием =, +=, -=, *=, /=, %=, &=, |=, ^=.

40. Перегрузка операций сравнения ==, !=, <, <=, >, >= и её особенности.

41. Перегрузка одноместных операций ++ и --. Различия префиксной и суффиксной форм.

42. В каких случаях при перегрузке операций создаётся временный объект?

43. Конструктор и деструктор: назначение, решаемые задачи, взаимодействие с средствами выделения памяти.

44. Какими функциями могут быть конструктор и деструктор (статическим и нестатическим методом класса, дружественной функцией, виртуальной функцией)? Почему? Аргументируйте ответ.

45. Можно ли перегружать конструктор? Почему?

46. Можно ли перегружать деструктор? Почему?

47. Передача параметров в конструктор и перегрузка конструктора. Что можно сказать о перегрузке деструктора? Конструктор по умолчанию.

48. Перегрузка конструкторов и наследование: как эффективно программировать конструктор порождённого класса?

49. Потоковый ввод/вывод в C++. Перегрузка операций для этого и манипуляторы ввода/вывода.

Наследование

50. Способы наследования private, protected, public и спецификаторы в родительском классе private, protected, public. Что доступно в порождённом классе?

51. Способы наследования private, protected, public и спецификаторы в родительском классе private, protected, public.

52. Что доступно в классе, порождённом от порождённого от базового (2 ступень наследования) при первичном наследовании public?

53. Что доступно в классе, порождённом от порождённого от базового (2 ступень наследования) при первичном наследовании protected?

54. Наследование классов и пространства имён родительского и порождённого класса: как обращаться к члену родительского класса а) извне, б) из производных классов.

55. Конфликты имён в классах: причина и разрешение. Всегда ли оно возможно?

56. Сравните вставку (встраивание, погружение) класса в другой класс с наследованием: каковы возможности и ограничения в каждом случае?

57. В каком случае доступ к члену класса работает быстрее – при вставке (встраивании) класса или при наследовании? Почему?

58. Что такое множественное наследование, какие могут возникнуть проблемы? Как они решаются?

59. Виртуальные функции: способы задания, отличия от невиртуальных. Приведите сравнение на простом примере.

60. Что будет, если в родительском классе функция объявлена как виртуальная, в унаследованном от него – нет?

61. Имеется иерархия порождения классов A->B->C. Что будет в классе C, если в A функция была объявлена как обыкновенная, в B - как чисто виртуальная, а в C - снова как обыкновенная?

62. Виртуальные методы и абстрактные классы: назначение, отличия от обычных.

63. Зачем нужны абстрактные классы?

64. Статическое и динамическое связывание в контексте указателей на классы в иерархии порождения и преобразования типов C++.

65. Имеются ли хоть какие-то проверки допустимости преобразований при динамическом связывании?

66. Сложное наследование (несколько предков). Последовательность вызовов конструкторов и деструкторов для объектов производного класса в этом случае.

Обработка исключений

67. Генерация и обработка исключительных ситуаций: является ли необходимой, чем обусловлено её введение, как осуществляется (правила перехватывания).

68. Что будет, если для некоторого класса исключительных ситуаций нет обработчика?

69. Что будет, если один класс исключительных ситуаций порождён от другого?

70. Бывают ли такие функции, в которых принципиально нельзя генерировать исключительные ситуации?

71. Классы памяти в C++: автоматический, динамический, временный, статический, внешний, регистровый: различия в поведении и использовании.

72. Когда вызываются конструкторы статических атрибутов класса?

73. Что дальше будет происходить с временным объектом после завершения его создавшей функции?

Полиморфизм

74. Шаблоны функций: назначение, определение, применение.

75. Что позволяют шаблоны и не позволяет препроцессор Си?

76. Что позволяет препроцессор Си и не позволяют шаблоны?

77. Правила перегрузки шаблонов: приведите короткие примеры.

78. Типизированные, нетипизированные и шаблонные параметры шаблонов - назначение и примеры использования.

79. Чем может быть нетипизированный параметр шаблона?

80. Всякое ли имя класса может быть передано в типизированный параметр шаблона?

81. Что может получиться из шаблона при подстановке в него всех типизированных и нетипизированных параметров?

82. Параметры по умолчанию в шаблонах и перегрузка шаблонов: правила задания и выбора используемого шаблона.

83. Шаблоны классов: назначение, определение, применение и правила порождения объектов.

84. Способы порождения функций и классов из шаблонов. Приведите короткие примеры.

85. Уточнение (спецификация, перегрузка) шаблонов: назначение, выяснение, какой шаблон будет в каком случае использован.

86. Как меняются правила перегрузки функции при появлении шаблона функции?

Пространства имен

87. Средства динамического выделения/освобождения памяти и их отличие от конструктора и деструктора.

88. Стандартные средства динамического выделения / освобождения памяти и наследование: что будет использовано, если имеется перегрузка new / delete в базовом / порождённом классе?

89. Пространства имён: определение, использование, наименование, вложенность.

90. Как сделать, чтобы в некотором блоке использовать несколько пространств имён сразу? Каким требованиям они должны отвечать?

91. Могут ли в одном классе быть несколько пространств имён?

92. Могут ли в одном пространстве имён быть несколько классов?

93. Могут ли два пространства имён пересекаться (но не одно не содержится в другом)?

Преобразования типов

94. Неявные преобразования типа в C++: перечислите, приведите короткие примеры.

95. Явные преобразования типа в C++: встроенные формы const_cast, static_cast, dynamic_cast, reinterpret_cast. Правила выбора типа преобразования.

96. Функциональная форма преобразования типа в C++. Её реализация.

97. Что требуется сделать, чтобы можно было использовать функциональную запись преобразования из класса A в класс B?

98. Какие преобразования типа принимаются в расчёт только во время компиляции?

99. Преобразования указателей и ссылок в C++: особенности с учётом наследования.

Инициализация, ссылки, указатели

100. Инициализация объектов классов, перечислите все её способы и проиллюстрируйте примерами.

101. Что нужно сделать, чтобы имелась возможность инициализировать значение объекта в месте его определения?

102. Можно ли проинициализировать значение объекта в месте его определения без вызова конструктора?

103. Инициализирующий конструктор: назначение, способ оформления, способы применения, отличие от перегрузки операции присваивания.

104. Понятие ссылки: назначение, способы применения, побочные эффекты.

105. Всегда ли передача по ссылке лучше передачи по значению? Аргументируйте ответ.

106. Всегда ли передача по ссылке лучше передачи по указателю? Аргументируйте ответ.

107. Всегда ли можно обойтись без ссылки?

108. Отличия в синтаксисе C++ от Си, связанные с перегрузкой функций, преобразованием типов и доступом к полям структур.

Элементы безопасности кода

109. Безопасная работа со строками. Некорректно ограниченные строки. Усечение строк.

110. Безопасная работа со строками. Некорректно ограниченные строки. Переполнение буфера. Разрушение стека.

111. Безопасная работа со строками. Некорректно ограниченные строки. Переполнение буфера. Внедрение кода.

112. Стратегия снижения привилегий в ядре. W^X. Запись или исполнение, но не оба вместе.

113. Безопасность кода при применении указателя на функцию. W^X. Запись или исполнение, но не оба вместе.

114. Безопасность кода при применении указателя на данные. Обращение к освобождённой памяти. Многократное освобождение памяти

115. Безопасность кода при применении указателя на данные. Выделение памяти нулевого размера.

116. Безопасность кода при применении указателя на данные. Выделение и освобождение памяти с помощью new / delete, new[] / delete[].

117. Безопасность кода при применении указателя на данные. Исключения, генерируемые оператором new.

118. Безопасность кода при применении указателя на данные. Выделение памяти и ошибки при инициализации памяти

119. Безопасность кода при применении указателя на данные. Утечка памяти

СПИСОК ЛИТЕРАТУРЫ

1. Шилдт, Г. Самоучитель C++, 3-е издание / Г. Шилдт; пер. с англ. – Санкт-Петербург: БХВ-Петербург, 2003 – 688 с.
2. Шилдт, Г. Полный справочник по C++, 4-е издание / Г. Шилдт; пер. с англ. – Москва: Вильямс, 2006 – 800 с.
3. Шилдт, Г. Искусство программирования на C++ / Г. Шилдт; пер. с англ. – Санкт-Петербург: БХВ-Петербург, 2005 – 496 с.
4. Страуструп Б. Программирование. Принципы и практика с использованием C++ / Б.Страуструп; пер. с англ. – Москва: Вильямс, 2016 – 1328 с.
5. Страуструп Б. Язык программирования C++, 2-е издание / Б. Страуструп; пер. с англ. – Москва: Вильямс, 2019 – 320 с.
6. Страуструп Б. Язык программирования C++ для профессионалов, 2-е издание / Б.Страуструп; пер. с англ. – Москва: Вильямс, 2019 – 320 с.
7. Кнут, Д. Искусство программирования, том 1. Основные алгоритмы / Д. Кнут – Москва: «Вильямс», 2015. – 720 с.
8. Кнут, Д. Искусство программирования, том 3. Сортировка и поиск / Д. Кнут – Москва: «Вильямс», 2013. – 824 с.
9. Топп, У. Структуры данных в C++ / У. Топп, У. Форд; пер. с англ. – Москва: ЗАО «Издательство Бином», 1999. – 816 с.
10. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт; пер. с англ. – Санкт-Петербург, «Невский диалект», 2001. – 352 с.
11. Кормен, Т.Х. Алгоритмы: построение и анализ, 3-е издание / Т. Х. Кормен, Ч. И. Лейзерстон, Р.Л. Ривест, К. Штайн; пер. с англ. – Москва: «Вильямс», 2013 – 1328 с.
12. Сикорд Р.С. Безопасное программирование на C и C++, 2-е издание / Р.С. Сикорд; пер. с англ. – Москва: Вильямс, 2015 – 496 с. ил
13. Парфенов, Д.В. Язык Си: кратко и ясно: учебное пособие / Д.В. Парфенов – Москва: «Альфа-М», 2014 – 320 с.
14. Керниган, Б. Язык программирования Си, 3-е издание / Б. Керниган, Д. Ритчи; пер. с англ. – Москва: Вильямс, 2017 – 288 с.