Задание 7

Темы: динамические структуры данных, классы, перегрузка операторов.

Классы

Класс – пользовательская структура данных, описывающая некоторое понятие из заданной предметной области. Класс включает в себя как поля, так и методы для их обработки.

Также каждый класс имеет набор методов, служащих для инициализации нового экземпляра класса (конструкторы), присвоения одного экземпляра класса другому (оператор присваивания) и удаления (деструктор). Если реализации этих методов не предоставляет программист, они могут быть автоматически созданы компилятором.

Если в классе имеются указатели на выделенные ресурсы (динамически выделенную память, файловые дескрипторы, сокеты и т.д.), то **нельзя полагаться на созданные компилятором реализации**.

Поля и методы классов могут иметь спецификаторы доступа **public**, **private** и **protected**.

```
class MyClass
{
    int A;
    float B;
public:
    MyClass();
    int getA();
    float getB();
private:
    void PrintData();
    double CalculateSum();
protected:
```

```
    void DoSomething();
};
```

Поля и методы, для которых спецификатор доступа не задан в явном виде (поля A и B в примере класса MyClass) будут иметь спецификатор доступа **private**.

Поля и методы, объявленные со спецификатором **public** доступны как из внешнего по отношению к классу кода, так и из методов данного класса. Такие методы часто называют **открытым или публичным интерфейсом** класса. При этом объявлять поля класса со спецификатором **public** **не рекомендуется**, поскольку это может привести к нарушению принципов абстракции и инкапсуляции.

Поля и методы, объявленные со спецификатором **private** доступны только внутри методов того класса, в котором они объявлены и не доступны из вне. В закрытую часть обычно выносятся поля, а также вспомогательные методы и методы, выполняющие действия, которые могут измениться в следующих версиях, зависят от ОС или аппаратного обеспечения и т.д.

Поля и методы, объявленные со спецификатором доступа **protected** доступны как в методах класса, в котором они объявлены, так и в методах классов-наследников. Во внешнем коде такие поля и методы не доступны, т.е. аналогичны полям и методам, объявленным со спецификатором **private**.

Делать поля класса открытыми не рекомендуется. Изменение полей класса должно осуществляться посредством вызова методов интерфейса.

Это упрощает внесение изменений в код впоследствии. Так как, например, чтение или запись полей класса может потребовать выполнения дополнительных операций.

Указатель this

Методы класса, в отличие от полей, существуют в единственном экземпляре и являются общими для всех экземпляров класса.

Каждому нестатическому методу класса неявно передается указатель на текущий экземпляр класса (адрес объекта, к которому буде применен метод).

Этот адрес доступен посредством ключевого слова this.

Указатель this **не может быть изменен**.

Пример использования указателя this:

```
class A {
    int a;
public:
    void SetA(int A) {
        this->a = A;
    }
};
```

Конструкторы

Конструкторы предназначены для создания нового экземпляра класса (объекта). Конструкторы не имеют возвращаемых значений.

Конструктор по умолчанию

Объявление класса с конструктором по умолчанию:

```
class MyClass
{
    ...
public:
    MyClass();
};
```

Реализация конструктора по умолчанию:

```
MyClass::MyClass()
{
    ...
}
```

Конструктор по умолчанию используется при создании объекта без параметров. Создание возможно как в стэковой, так и в динамической памяти.

В следующих случаях будет вызван конструктор по умолчанию:

```
MyClass mc;
MyClass mc2();
MyClass mc3 = MyClass();
MyClass* pmc = new MyClass;
```

Если конструктор по умолчанию не создан программистом, то он **будет создан компилятором автоматически в том случае, если в классе не определены конструкторы с параметрами**. При этом созданный автоматически конструктор по умолчанию не выполняет никакой полезной работы. Он может только вызвать конструкторы по умолчанию для полей, имеющих тип какого-либо класса; поля, являющиеся стандартными встроенными типами данных (`int`, `double`, `char` и т.д.), а также указателями **не инициализируются** созданным автоматически конструктором по умолчанию.

Конструктор с параметрами

Конструктор с параметрами должен иметь не менее одного обязательного параметра. Тип параметров задается программистом исходя из решаемой задачи. Количество конструкторов с параметрами может быть любым. При это такие конструкторы, как и обычные функции, должны отличаться типом и/или количеством параметров.

Объявление класса с несколькими конструкторами с параметрами:

```
class MyClass
{
    ...
public:
    MyClass(int A);
    MyClass(double B);
    MyClass(int A, int C);
    MyClass(double B, double D);
};
```

Реализация конструкторов с параметрами:

```
MyClass::MyClass(int A)
{   ...  }

MyClass::MyClass(double B)
{   ...  }

MyClass::MyClass(int A, int C)
{   ...  }

MyClass::MyClass(double B, double D)
{   ...  }
```

Создание объектов конструкторами с параметрами возможно как в стэковой, так и в динамической памяти.

В следующих случаях будут вызваны конструкторы с параметарми:

```
MyClass mc(1);
MyClass mc2(2.0);
```

```
MyClass mc3 = MyClass(1, 3);
MyClass* pmc = new MyClass(2.0, 4.0);
```

Конструктор копирования

Конструктор копирования вызывается при создании нового объекта на основе уже существующего. При этом, обычно новый объект является «копией». То есть осуществляется копирование информации из существующего объекта во вновь созданный. Параметром конструктора копирования является **константная ссылка на объект того же класса**.

Если конструктор копирования не реализован программистом, то он может быть автоматически создан компилятором. Однако, такой конструктор копирования **выполняет лишь побайтовое копирование полей** существующего объекта во вновь создаваемый, что может быть **неприемлемо, если поля класса содержат указатели на выделенную память, файловые дескрипторы, сокеты и т.д.** В таком случае необходимо предоставить собственную реализацию конструктора копирования.

Рассмотрим класс MyString, реализующий строку. Пусть данный класс имеет два поля: ***buffer*** – указатель на массив символов, хранящихся в строке и ***length*** – длина текущей строки.

Объявление класса MyString с конструктором копирования:

```
class MyString
{
    char* buffer;
    int length;
public:
    MyString(const MyString& Other);
};
```

При создании нового объекта типа MyString на основе существующего необходимо, чтобы каждый объект имел свой собственный регион памяти, в котором хранятся символы строки (рис. 1).

```
MyString A = MyString(B);
```

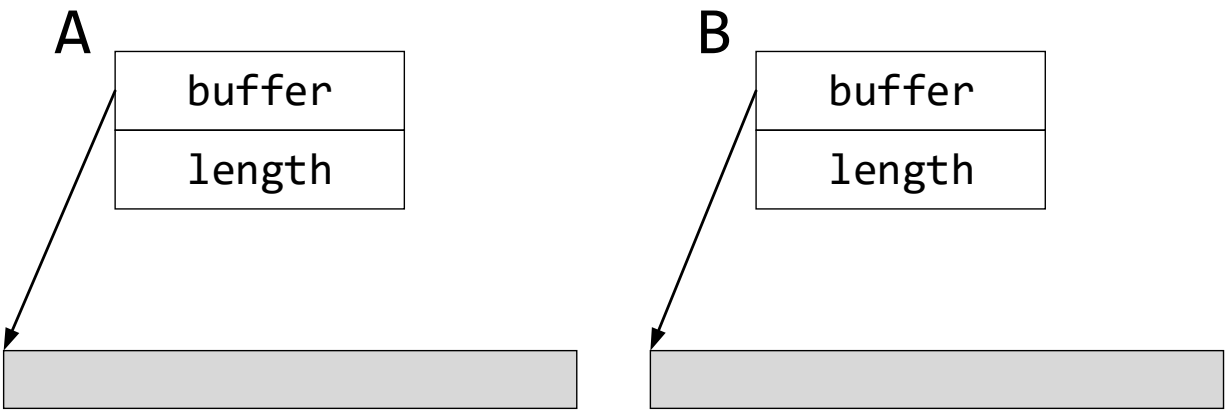

Рис. 1

Если программист не предоставит свою реализацию конструктора копирования, то автоматически созданная компилятором реализация будет выполнять лишь побайтовое копирование полей объекта. В таком случае поля ***buffer*** создаваемого объекта (A) и уже существующего (B) будут указывать на один и тот же регион памяти (рис. 2). В таком случае, изменение, например, символов в строке, описываемой объектом A, приведет к изменению символов другой строки, описываемой объектом B. Кроме того, если в деструкторе реализовано освобождение выделенных регионов памяти, то при удалении одной из этих строк, оставшиеся будут содержать адрес освобожденного региона памяти. Действия с данным регионом памяти могут привести к ошибке времени выполнения.

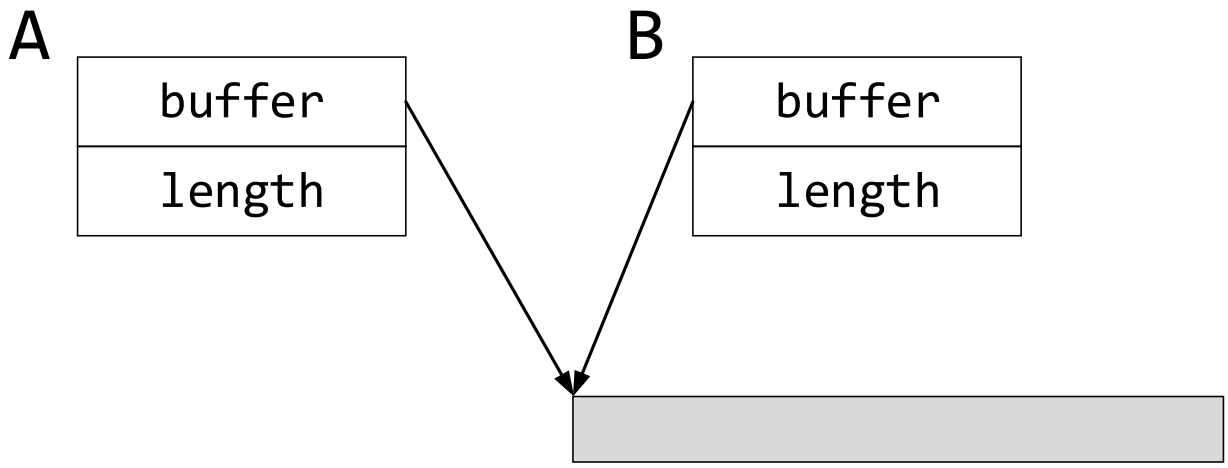

Рис. 2

В данном случае необходимо предоставить свою реализацию конструктора копирования, выполняющего выделение нового региона памяти и копирования туда данных, содержащихся в уже существующем объекте.

Реализация конструктора копирования:

```
MyString::MyString(const MyString& Str)
{
    this->length = Str.length;
    this->buffer = new char[this->length + 1];
    memset(this->buffer, 0, this->length + 1);
    strcpy(this->buffer, Str.buffer);
}
```

Конструктор копирования будет вызван в следующих случаях:

```
//MyString B определен ранее
MyString A(B);
MyString C = B;
MyString D = MyString(B);
MyString* pA = new MyString(B);
```

Также, конструктор копирования может быть вызван при передаче объекта в функцию по значению или возврата из функции по значению:

```
void Print(MyString Str);
MyString Concat(MyString& Str1, MyString& Str2);
...
MyString A = ...;
Print(A);    //Вызван конструктор копирования для копирования объекта A в аргумент функции Print
```

```
MyString C = Concat(A, B);  //Вызван конструктор копирования для копирования результата выполнения функции Concat в объект C. Поскольку A и B переданы по ссылке, то для них конструктор копирования не вызывается.
```

Move-конструктор

При передаче объектов, особенно анонимных, по значению в функции и возврата объектов из функций по значению будет вызываться конструктор копирования. Это может быть достаточно затратной операцией. При этом созданный анонимный объект вскоре будет уничтожен.

Избежать данных затрат можно при помощи конструктора move-копирования.

Объявление класса MyString с move-конструктором:

```
class MyString
{
    char* buffer;
    int length;
public:
    MyString(MyString&& Other);
};
```

Если класс содержит указатели на выделенные регионы памяти, то в move-конструкторе необходимо выполнить «быстрое» копирование. При этом указателю, содержащемуся в создаваемом объекте, присваивается значение указателя из существующего анонимного объекта. Поскольку анонимный объект вскоре после вызова move-конструктора будет уничтожен, то во избежание освобождения региона памяти, который теперь принадлежит созданному объекту, указателю в анонимном объекте необходимо присвоить значение nullptr.

Реализация move-конструктора (TmpStr – анонимный объект, который вскоре после вызова move-конструктора будет уничтожен):

```
MyString::MyString(MyString&& TmpStr)
{
    // Передаём данные новому объекту (не выполняется полного копирования области данных)
    this->length = TmpStr.length;
    this->buffer = TmpStr.buffer;

    // Обнуляем данные, чтобы при вызове деструктора для временного объекта и освобождении памяти не удалить данные из нового объекта
    TmpStr.buffer = nullptr;
    TmpStr.length = 0;
}
```

Move-конструктор вызывается в тех же самых случаях, что и обычный конструктор копирования, но только при условии, что новый объект создается на основе анонимного объекта.

Пример вызова move-конструктора:

```
MyString str = MyString(...);
MyString str2 = MyString(...);
//MyString operator+(const MyString& Str);
MyString str3 = str + str2;
```

Анонимный объект

Для инициализации str3 будет вызван move-конструктор.

Если в классе отсутствует реализация move-конструктора, то вместо него будет вызван обычный конструктор копирования.

Деструктор

При уничтожении экземпляра класса, например, при выходе из области видимости или вызове оператора delete, автоматически вызывается специальный метод, выполняющий работу по освобождению ресурсов, используемых экземпляром класса.

Данный метод называется **деструктором**.

Деструктор, как и конструкторы, не имеет возвращаемого значения, даже void.

Деструктор не принимает аргументов.

Имя деструктора должно совпадать с именем класса и предваряться символом **~**.

Класс может иметь **только один деструктор**.

Если деструктор не реализован программистом, то он будет автоматически сгенерирован компилятором. Такой деструктор лишь вызывает деструкторы для полей, имеющих тип класса, при их наличии, в порядке, обратном порядку объявления полей в классе, но не может корректно освободить, например, выделенные регионы памяти.

Таким образом, при наличии в составе класса **указателей на выделенные регионы памяти, некоторые виды файловых дескрипторов** (например, указатель на структуру FILE для работы с файлами в стиле C), **сокеты и т.д. необходимо реализовывать собственный деструктор**.

Объявление класса MyString с деструктором:

```
class MyString
{
    char* buffer;
    int length;
public:
    ~MyString();
};
```

Реализация деструктора:

```
MyString::~MyString()
{
    delete[] this->buffer;
    this->buffer = nullptr;
    this->length = 0;
}
```

Деструктор будет вызван в следующих случаях:

```
{
    MyString A = ...;
    ...
} // Вызван деструктор для объекта A, поскольку осуществлен выход из области существования данного объекта. Такая область ограничена фигурными скобками.
```

```
MyString* pA = new MyString(...);
...
delete pA;    //Вызван деструктор для объекта pA, созданного в динамической области памяти.
```

Перегрузка операторов

Для упрощения и улучшения читаемости программы для класса могут быть перегружены операторы, имеющиеся в C++.

При перегрузке оператора необходимо указать ключевое слово operator и знак операции, которая будет перегружена. Также, как и для других методов класса и глобальных функций, указывается тип возвращаемого значения и список аргументов.

Некоторые встроенные операторы эквивалентны некоторой комбинации встроенных операторов.

Например, a++ ⇔ a = a + 1 ⇔ a += 1.

Такое отношение между операторами не сохраняется при их перегрузке, только если программист об этом не позаботился.

Операторы могут быть перегружены как методом класса, так и глобальной функцией.

При этом глобальная функция не имеет доступа к **private** и **protected** полям и методам класса. Поэтому она взаимодействует с классом через открытый интерфейс.

При перегрузке методом класса, указатель this является левым параметром выражения.

Операторы бывают унарными и бинарными.

Унарный оператор производит действие над одним объектом, бинарный – над двумя.

В C и C++ также существует тернарный оператор, но он запрещен для перегрузки.

Перегрузка унарного оператора может быть осуществлена:

- Методом класса без параметров;
- Глобальной функцией с одним параметром.

Перегрузка бинарного оператора может быть осуществлена:

- Методом класса с одним параметром;
- Глобальной функцией с двумя параметрами.

Оператор может быть объявлен только с синтаксисом, существующим для него в грамматике языка.

Например, нельзя объявить унарный оператор / или %.

Кроме того, запрещено определять новую лексему оператора, например, **.

Операторы, запрещенные для перегрузки:

- . (доступ к полю)
- .* (доступ к указателю на поле)
- :: (разрешение области видимости)

- ?: (тернарное выражение)
- # (символ препроцессора)
- ## (символ препроцессора)
- sizeof
- typeid (идентификация типа во время выполнения программы)

Не все операторы могут быть перегружены с использованием глобальной функции.

Операторы, которые могут быть перегружены только методом класса:

- =
- []
- ->
- () (приведение типа)
- () (вызов функции)

Способы вызова оператора:

```
MyString str1(...), str2(...);
MyString str3 = str1 + str2;
```

Если оператор перегружен методом класса, то он может быть вызван следующим образом:

```
MyString str3 = str1.operator+(str2);
```

Если глобальной функцией, то:

```
MyString str3 = operator+(str1, str2);
```

Оператор присваивания

Оператор присваивания используется для **присвоения нового значения уже существующему объекту**.

В некоторых случаях, если оператор присваивания не определен, то компилятор создаст его реализацию. Такая реализация осуществляет побайтовое копирование полей объекта. Поэтому, в большинстве случаев, при наличии в составе полей класса указателей, необходимо реализовать собственный оператор присваивания.

Объявление класса MyString с оператором присваивания:

```
class MyString
{
    char* buffer;
    int length;
public:
    MyString& operator=(const MyString& Other);
};
```

Логика работы оператора присваивания в большинстве случаев аналогична логике работы конструктора копирования. При этом оператор присваивания, в отличие от конструктора копирования, обладает возвращаемым значением. Кроме того, перед присвоением необходимо убедиться, что объект не присваивается самому себе и, если это не так, удалить ресурсы, содержащиеся в объекте.

Реализация оператора присваивания:

```
MyString& MyString::operator = (const MyString& Str) {
    if (this != &Str) {
        delete[] this->buffer;
        this->length = Str.length;
        this->buffer = new char[this->length + 1];
        memset(this->buffer, 0, this->length + 1);
        strcpy(this->buffer, Str.buffer);
    }
    return *this;
}
```

Оператор присваивания будет вызван в следующих случаях:

```
//MyString B определен ранее
MyString A = ...;
...
A = B;          //вызов оператора присваивания

MyString* pA = new MyString(...);
*pA = B;        //вызов оператора присваивания
```

Оператор move-присваивания

Оператор присваивания, аналогично конструктору копирования, может иметь move-версию. Логика работы оператора move-присваивания аналогична логике работы move-конструктора с учетом специфики оператора присваивания (возврат значения, проверка на присваивания самому себе и удаление ресурсов, содержащихся в объекте).

Объявление класса MyString с оператором move-присваивания:

```
class MyString
{
    char* buffer;
    int length;
public:
    MyString& operator=(MyString&& Other);
};
```

Реализация оператора move-присваивания:

```
MyString& MyString::operator = (MyString&& Str)
{
     if (this != &Str) {
          delete[] this->buffer;
          this->length = Str.length;
          this->buffer = Str.buffer;
          Str.buffer = nullptr;
          Str.length = 0;
     }
     return *this;
}
```

Пример вызова оператора move-присваивания:

```
MyString str = MyString(...);
MyString str2 = MyString(...);
MyString str3 = MyString(...);
str3 = str + str2;     //move-присваивание
```

Если оператор move-присваивания не реализован, то вместо него будет вызван обычный оператор присваивания.

Дружественные функции

Дружественная функция – это глобальная функция, которая может обращаться к закрытым (**private** и **protected**) полям и методам класса, в котором она объявлена.

При объявлении дружественной функции необходимо указать ключевое слово friend.

Не имеет значения, в какой секции (**private**, **protected** или **public**) будет объявлена дружественная функция, поскольку она не является членом класса.

Объявление класса MyString с дружественной функцией, осуществляющей перегрузку операции чтения из потока ввода:

```
class MyString {
    char* buffer;
    int length;
public:
    friend std::istream& operator>>(std::istream& In,
                                    MyString& Str);
};
```

Реализация дружественной функции записи в поток ввода:

```
std::istream& operator>>(std::istream& In, MyString& Str){
    std::string res;
    In >> res;
    Str.length = res.length();
    delete[] Str.buffer;
    Str.buffer = new char[Str.length + 1];
    memset(Str.buffer, 0, Str.length + 1);
    strcpy(Str.buffer, res.c_str());
    return In;
}
```

Задание

Реализовать класс, описывающий структуру данных «двусвязный список». В качестве данных использовать целые числа. Список имеет «головной» (**Head**) и «хвостовой» (**Tail**) элементы. Данные элементы используются для обозначения границ списка и не используются для хранения данных в поле **Data**. Элемент **Head** располагается перед первым элементом списка (**Pos = 0**), содержащим данные. Элемент **Tail** содержится после последнего элемента списка (**Pos = *N* – 1**), содержащего данные. Пример списка из ***N*** элементов представлен на рис. 3.

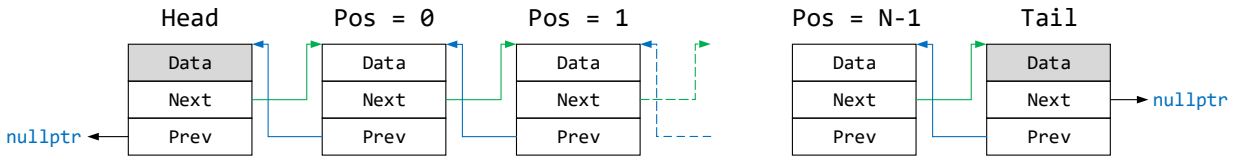

Рис. 3

Список состоит из структур типа ListNode:

```
struct ListNode
{
    int Data;
    ListNode* Next;
    ListNode* Prev;
};
```

Прототип класса:

```
class List
{
     ListNode Head;
     ListNode Tail;
public:
     List();
     List(int Val);
     List(const List& Lst);
     List(List&& Lst);
     ~List();
     List& operator = (const List& Lst);
     List& operator = (List&& Lst);
     void Add(int Val);
     void AddAfter(int Val, unsigned int Pos);
     void AddBefore(int Val, unsigned int Pos);
     void Print();
     void PrintReverse();
     void Delete(int Val);
     void DeleteAllWithVal(int Val);
     void Clear();
     bool IsEmpty();
     friend std::ofstream& operator << (std::ofstream& File,
                                        List& Lst);
};


std::ifstream&  operator  >>  (std::ifstream&  File,  List&
                               Lst);
```

Конструктор по умолчанию – создает пустой список. У пустого списка поле **Next** элемента **Head** содержит адрес элемента **Tail**. Поле **Prev** элемента **Tail** содержит адрес элемента **Head**. **Head.Prev = Tail.Next = nullptr** (рис. 4). Поле **Data** элементов **Head** и **Tail** может содержать любое значение.

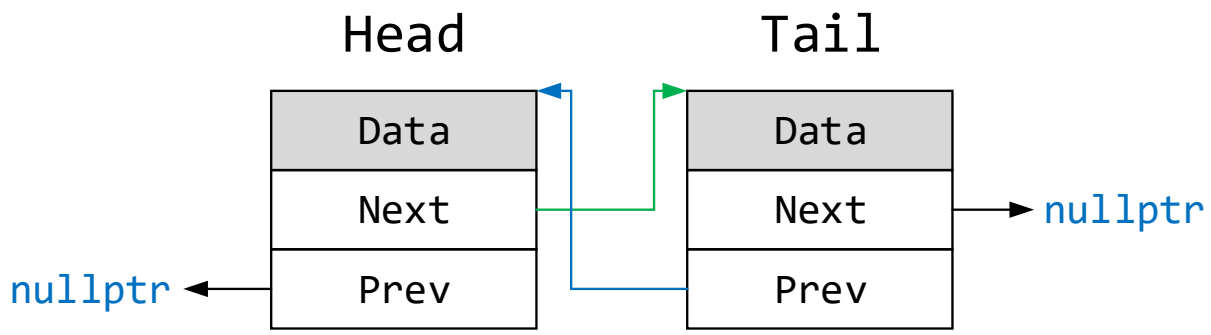

Рис. 4

Конструктор с параметром – создает список с единственным элементом, значение которого содержит значение, переданное в качестве параметра (рис. 5).

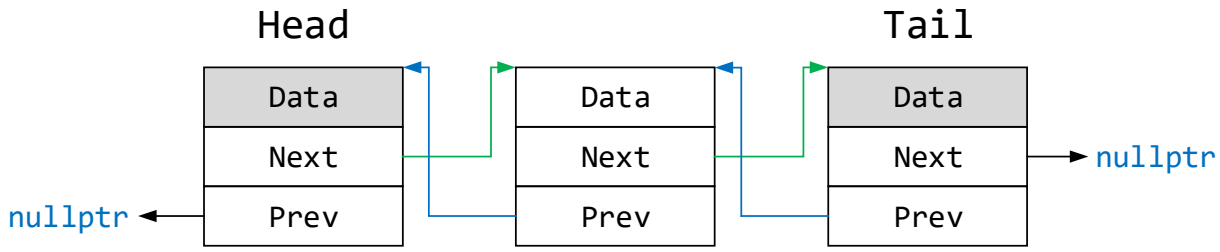

Рис. 5

Конструктор копирования – копирует содержимое из списка, переданного в качестве параметра в текущий список.

Move-конструктор – осуществляет «быстрое» копирование содержимого списка, переданного в качестве параметра, в текущий список.

Деструктор – корректно освобождает ресурсы, занятые списком. После отработки деструктора список должен быть пустым (рис. 4).

Оператор присваивания – осуществляет копирование содержимого из списка, переданного в качестве параметра в текущий список.

Ресурсы, содержащиеся в текущем списке перед выполнением присваивания должны быть корректно удалены.

Оператор move-присваивания – осуществляет «быстрое» копирование содержимого из списка, переданного в качестве параметра в текущий список. Ресурсы, содержащиеся в текущем списке перед выполнением присваивания должны быть корректно удалены.

Метод Add – добавляет новое значение в конец текущего списка (перед элементом **Tail**).

Метод AddAfter – добавляет элемент со значением ***Val*** после элемента с заданной позицией (порядковым номером) ***Pos***. Порядковый номер начинается с нуля и отсчитывается от начала списка. Если элемента с такой позицией не существует (т.е., если список содержит меньше элементов, чем ***Pos***) добавление производится в конец списка (перед элементом **Tail**).

Метод AddBefore – добавляет элемент со значением ***Val*** перед элементом с заданной позицией (порядковым номером) ***Pos***. Порядковый номер начинается с нуля и отсчитывается от начала списка. Если элемента с такой позицией не существует (то есть, если список содержит меньше элементов, чем ***Pos***) добавление производится в конец списка (перед элементом **Tail**).

Метод Print – выводит на экран все элементы списка в прямом порядке (то есть от начала списка к концу).

Метод PrintReverse – выводит на экран все элементы списка в обратном порядке (то есть от конца списка к началу).

Метод Delete – удаляет элемент, поле **Data** которого имеет значение ***Val***. При наличии в списке нескольких элементов со значением ***Val***, удаляется первый из них, начиная с начала списка.

Метод DeleteAllWithVal – удаляет из списка все элементы с заданным значением (то есть элементы, поля **Data** которых имеют значение ***Val***).

Метод Clear – удаляет все элементы из списка.

Метод IsEmpty – возвращает `true`, если список пуст (рис. 4), в противном случае – `false`.

operator << - записывает **содержимое** списка в файл.

operator >> - считывает данные из файла и помещает их в список.

Класс может содержать дополнительные методы, упрощающие реализацию требуемого функционала. Данные методы должны быть помещены в закрытую часть класса (то есть объявлены со спецификатором **private**).

Конструкторы, деструктор и операторы присваивания должны содержать отладочную печать (вывод на экран) с указанием своего имени.

Объявление структуры ListNode и класса List должно содержаться в заголовочном файле List.h, реализация методов – в файле исходного кода List.cpp. Файл List.h должен быть подключен в раздел «Файлы заголовков» («Header Files»), List.cpp – в раздел «Исходные файлы» («Source Files»). К файлу, содержащему функцию main должен быть подключен директивой #include только файл List.h.

Main.cpp

```
...
#include “List.h”


int main()
{
    ...
    return 0;
}
```

Разработанное приложение должно содержать код, иллюстрирующий работу всего реализованного функционала (конструкторов, деструктора, методов и операторов).

После каждого изменения содержимого списка должны быть вызваны методы **Print** и **Printreverse** для контроля целостности списка.

Для принудительного вызова move-конструктора или оператора move-присваивания можно воспользоваться алгоритмом **std::move**.

Рекомендации по реализации методов класса

Создание элементов списка (структур типа ListNode), кроме **Head** и **Tail**, должно производится в динамической области памяти (**heap**) для обеспечения требуемого времени жизни элементов списка.

Копирование содержимого одного списка в другой, требуемое при реализации конструктора копирования и оператора присваивания в цикле может осуществлять вручную путем создания новой структуры в динамической области памяти и связывания ее с оставшейся частью списка. Однако может оказаться более удобным вызов метода **Add**, в который будет передаваться значение из очередного элемента списка, из которого осуществляется копирование.

При реализации move-конструктора и оператора move-присваивания нужно скорректировать лишь указатели в полях **Head** и **Tail** создаваемого списка на значение полей **Next** и **Prev** из временного списка (переданного в качестве аргумента). Временный список необходимо сделать пустым (рис. 4).

Метод **Clear** удаляет все элементы из списка. Такое же поведение необходимо при реализации деструктора. Как следствие, в деструкторе можно просто вызвать метод **Clear**. **Обратное неверно!**

При реализации методов **Add**, **AddAfter** и **AddBefore** не стоит забывать о тщательном проектировании вставки элемента в начало или конец списка. При этом элемент **Head** располагается перед первым элементом списка (элементом с номером 0), а элемент **Tail** располагается после последнего элемента списка (см. рис. 3). Методы **Add**** выделяют память под новый элемент в в динамической области памяти (**heap**).

При реализации методов **Print** и **PrintReverse** перебор элементов нужно начинать с элемента, следующего за **Head** (т.е. с элемента,

на который указывает поле **Next** элемента **Head**), и заканчивать на элементе, предшествующем элементу **Tail**.

Примерный код выглядит следующим образом:

```
ListNode* tmp = Head.Next;
while (tmp->Next != &Tail)
{
    ...
    tmp = tmp->Next;
}
```

При реализации методов **Delete** и **DeleteAllWithVal** не стоит забывать о корректной обработке граничных случаев (удаление из начала или конца списка) и поддержании целостности списка. Методы **Delete**** осуществляют освобождение региона памяти, занятого удаляемыми структурами ListNode.

При реализации метода **Clear**, каждый элемент списка, за исключением **Head** и **Tail** должен быть корректно удален из динамической памяти. После этого список должен стать пустым. То есть поля **Head** и **Tail** должны быть инициализированы согласно рис. 4.

При записи списка в файл записывать имеет смысл лишь значения элементов списка (поле **Data** структур ListNode).

При чтении значений элементов списка из файла добавление в список удобно реализовать посредством вызова метода **Add**.

Вопросы для самопроверки:

1. Когда вызывается конструктор по умолчанию?
2. Когда происходит вызов конструктора копирования?
3. Когда происходит вызов move-конструктора?
4. Когда необходимо предоставлять собственную реализацию конструктора копирования?
5. Когда происходит вызов деструктора?
6. Когда необходимо предоставлять собственную реализацию деструктора?
7. Сколько деструкторов может быть у класса?
8. Когда происходит вызов оператора присваивания?
9. Когда необходимо предоставлять собственную реализацию оператора присваивания?
10. Что такое дружественная функция?
11. Какие операторы нельзя перегрузить?
12. Какие операторы можно перегрузить только методом класса?
13. Какой метод класса (конструктор копирования или оператор присваивания) будет вызван в следующем фрагменте кода?
```
//Объект a определен ранее
MyClass b = a;
```
14. Какой метод класса (конструктор копирования или оператор присваивания) будет вызван в следующем фрагменте кода?
```
//Объект a определен ранее
MyClass b;
b = a;
```