UNIVERSITATEA TEHNICĂ A MOLDOVEI

FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ
DEPARTAMENTUL INFORMATICĂ ȘI INGINERIA SISTEMELOR

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Методические указания к лабораторным работам

Chişinău
Editura „Tehnica-UTM”
2024

CZU

Lucrarea a fost discutată și aprobată pentru editare la ședința Consiliului Facultății Calculatoare, Informatică şi Microelectronică din 13.12.2023, proces-verbal nr.3.

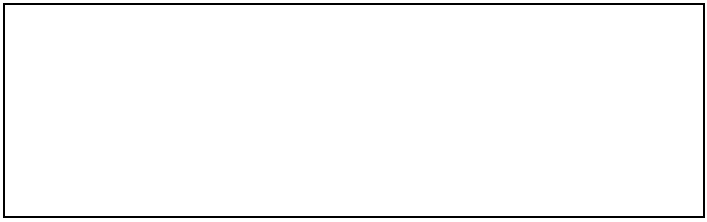
Данные методические указания предназначены для того, чтобы помочь студентам раскрыть суть основных принципов, входящих в понятие “объектно-ориентированное программирование”. В тексте содержится как теоретический материал, необходимый для работы с этими принципами, так и практические задания с примерами решения для иллюстрации принципов наследования, полиморфизма и абстракции в действии.

Работа в первую очередь предназначена для студентов второго курса, изучивших основы программирования на языке C, а также владеющих базовыми знаниями о фундаментальных структурах данных и алгоритмах.

Текст делится на главы, которые содержат теоретическое обоснование практических заданий и технические рекомендации по их выполнению с помощью языка C++. Также студентам предлагается семь лабораторных работ для закрепления изученного материала и проверки полученных навыков на практике.

Автор: ассистент, Брынзан, Л. В.

Рецензент: доцент, доктор Фалько, Н. С.



© UTM, 2024

Введение

Когда говорят об абстракции в обыденности, подразумевают что-то эфемерное, идеальное, не существующее в физическом мире. Между тем абстракция является очень важным инструментом в программировании. Вот знакомая всем запись, $y = f(x)$. Здесь есть несколько абстрактных элементов. Переменные x и y сами по себе являются абстракциями возможных конкретных значений. Функция f является абстрактным механизмом преобразования значения переменной x в значение переменной y. Знак равенства указывает на то, что в данном контексте применение функции f к аргументу равносильно значению переменной y. Это позволяет сокращать записи математических выражений или значительно их упрощать. Например, $4x^4 - x^2 - x + 2$, выражение выше можно упростить с помощью абстракции, $y = 2x^2 + x + 1$, тогда изначальное выражение можно переписать, $y^2 - y + 2$.

Скрыв подробности функции в переменной y можно сосредоточиться на работе с квадратичной функцией, при необходимости раскрывая определение переменной y. Использование абстракции в выражениях называется *комбинацией* [Башкин].

В программировании абстракцию приходится применять похожим образом. Код ниже это иллюстрирует:

```
push    rbp
mov     rbp, rsp
mov     edi, OFFSET FLAT:.string "Hello, World!"
mov     eax, 0
call    printf
mov     eax, 0
pop     rbp
```

Неопытным взглядом сложно понять что именно происходит, особенно если не знать инструкции, которые используются в этом фрагменте. При этом, существует

механизм, создающий абстракцию этих инструкций, который приводит весь фрагмент к другому виду:

```
printf(“Hello, World!”);
```

Новая запись предельно краткая, но полностью скрывает происходящее в момент применения функции **printf** к аргументу. Тем не менее, такая форма является более предпочтительной, потому что в подавляющем большинстве случаев программиста не столько интересует конкретный набор инструкций, сколько окончательный результат. Под таким углом полезность абстракций очевидна.

Эту концепцию поначалу трудно представить, когда речь идет о *типах данных*. И тем не менее, программирование в значительной степени состоит из взаимодействий с различными уровнями абстракций типов. Для того, чтобы в этом убедиться, следует вернуться к понятию “переменная”, но теперь уже – в контексте программирования. Кто-то может сказать, что переменная – это имя в программе. Что будет не совсем точно, потому что “имя”, обладает единственным свойством – уникальностью. Очевидно, что переменные обладают и другими свойствами. Кто-то скажет, что это “хранилище”, которое принимает различные значения. Это опять не совсем точно в отношении переменных в программах, так как часто переменная не хранит значение, которое ей присваивается, при этом значение не может существовать в программе само по себе, ведь оно тоже абстрактно. Оно будет иметь некоторое конкретное представление. Например, число “42” является значением, оно будет представлено в памяти компьютера в виде двоичного ($2^5 + 2^3 + 2^1$), а в программе – еще и в виде десятичного ($4 * 10^1 + 2 * 10^0$) или шестнадцатеричного ($2 * 16^1 + 10 * 16^0$, $2A_{16}$) набора данных.

На основании этих уточнений можно выделить три основных момента, присущих любой переменной. Переменные обладают способом представления в конкретном языке (целое число, дробное число, текст и т.д.). Переменные обладают порядком хранения в памяти компьютера (занимаемые байты, а

также назначение каждого бита в соответствующих байтах). Наконец, переменные обладают непосредственно значением, которое определяет их полезность в программе. Таким образом мы столкнулись с тремя уровнями абстракции, совершая переход от математических концепций к их воплощению в памяти компьютера посредством какого-то языка программирования[1].

Так как математические абстракции принято классифицировать по каким-либо признакам для удобства (например, переменные разделяются на такие, которые представляют целые или дробные числа), такой подход применяется и в языках программирования. Классификация производится по свойствам, перечисленным выше, а именно: какое множество значений принимают данные, какую форму принимают значения из этого множества, какие операции возможны для значений из этого множества [Wirth]. То есть, *тип данных* – это совокупность представления какого-то значения в программе и в памяти компьютера, которая позволяет работать с конкретным значением в программе с помощью соответствующих операций. Внутри самой программы также есть “конкретные” сущности и абстрактные сущности. Первые ассоциируются с *объектами*, вторые – со значениями [McJones]. В общем смысле, объект – это некоторое отношение между типом данных и самими данными в памяти компьютера. Свойства объекта определяются его типом.

Другой смысл слова “абстрактный” – “удаленный”, то есть, рассматриваемый в отрыве от своего источника. Так разделение подходов в программировании на парадигмы (программирование может быть структурное, объектно-ориентированное, функциональное и т.д.) является абстракцией, потому что с точки зрения целевого устройства все эти парадигмы производят один и тот же машинный код, который исполняется независимо от того, какую идею

[1] Программирование можно считать одной из областей математики, языки программирования можно называть абстрактными компьютерами [Strachey].

вкладывал в него программист. Предназначение абстракций в программировании заключается в том, чтобы *проектировать программы, понимать их и проверять их логику на правильность на основании законов, которые управляют абстракциями* (например, на основании законов математики), не отвлекаясь на подробности реализации этих абстракций в конкретном компьютере.

1. Абстрактные типы данных

1.1. Теоретическая информация

Допуская, что поставлена задача разработать тип данных в языке **C**, который будет использоваться для представления рациональных[2] чисел в различных программах, такое число можно хранить в виде структуры:

```
struct rational {
    int p;
    int q;
};
```

Тогда атрибут p – это числитель, а q – знаменатель. В этом случае создание рационального числа в программе могло бы выглядеть так:

```
rational a = {1, 2};
rational b;
b.p = 5; b.q = 6;
rational c;
```

Все три способа являются правильными с точки зрения компилятора (в данном примере используется GNU g++[3]). Но внимательный читатель может заметить, что тут есть проблемы. Во-первых, в качестве числителя и знаменателя можно передавать любые значения, в том числе 0. Это, конечно, противоречит правилам работы с рациональными числами, так как 0 в числителе должен обнулять все число, а в знаменателе 0 недопустим. Во-вторых, и числитель, и знаменатель можно свободно менять уже после создания переменной. В-третьих, оба атрибута можно оставить без значений, что делает такую переменную бесполезной, а поведение программы, которая ее использует, непредсказуемым.

---

[2] Действительное число, которое можно представить как пару целых чисел, записанных одно над другим через прочерк, в виде дроби.

[3] *https://gcc.gnu.org/*.

Общепринятый способ решения похожих проблем заключается в создании специальных функций, которые устанавливают значения согласно определенному набору требований (в данном случае – допустимые значения числителя и знаменателя). Для рациональных чисел эта функция могла бы выглядеть так:

```
rational cons(int x, int y) {
    assert(y != 0);
    if(x == 0) return {0, 0};
    rational r;
    if(y < 0) {
        r.p = -x; r.q = -y;
    }
    else {
        r.p = x; r.q = y;
    }
    return r;
}
```

Функция проверяет: числитель на нулевое значение (тогда вся дробь обнуляется), знаменатель на нулевое значение (тогда даже нет смысла продолжать, и работа программы прерывается; в одной из следующих глав показан более элегантный способ справляться с этой ситуацией), знаменатель на отрицательное значение (тогда отрицание переносится на числитель):

```
rational a = cons(1, 2);
```

Конечно, те, кто разбирается в нюансах создания переменных в **C**, понимают, что эта функция никак не решает проблемы изменения значений числителя и знаменателя в дальнейшем. В целом, ограничения, которые эта функция накладывает, условны. Никак не запрещается создавать переменные такого типа без использования этой функции. Другими словами, она рассчитана на то, что разработчики будут руководствоваться здравым смыслом при работе с такими рациональными числами. Тем не менее, компилятор позволяет установить более жесткие ограничения, которые полностью

контролируют создание переменных и изменение атрибутов каждого создаваемого объекта типа **`rational`**. О них речь пойдет в следующих главах.

Тут интересно другое. Когда было принято решение создать функцию, задача которой – заботиться о правильности инициирования рациональных чисел, появилась абстракция реализации рациональных чисел в программе. Использование рациональных чисел в своих программах является основным критерием для пользователя. Аналогичные функции можно написать для всех операций, в которых будут участвовать такие объекты. Результатом становится *абстрактный тип данных*. Такой тип данных предоставляет не только механизм для создания объектов с данными, но и механизмы для полноценной работы с этими объектами – стандартные операции над ними.

1.2. Практические задачи для закрепления теории

а. Абстрактные типы данных будут составлять основу большинства концепций, обсуждаемых в этом тексте, поэтому с ними нужно освоиться как можно лучше. Для этого читателю предлагается дополнить тип данных, описанный выше, несколькими операциями. Во-первых, было бы неплохо поддерживать сокращенную форму всех создаваемых в программе рациональных чисел (при заданных числителе и знаменателе, которые кратны одному и тому же числу, делить их на это число по умолчанию: $^{5}/_{10}$ превращать в ½, $^{9}/_{6}$ – в $^{3}/_{2}$). Во-вторых, было бы удобно отображать такие числа в текстовом виде как дроби, а не пару целых чисел. Пример этих дополнений будет в разделе с решениями упражнений, но настоятельно рекомендуется сделать их самостоятельно, и только в случае полной неудачи посмотреть реализацию из соответствующего раздела.

Лабораторная работа 1
Тема: Абстрактные типы данных

Цель работы: Для заданного варианта[4] с помощью структуры реализовать абстрактный тип данных, который будет скрывать указатель на базовую коллекцию данных (массив, список, дерево и т.д.), избавляя пользователя от необходимости работать с указателями напрямую для получения данных из коллекции. Ниже показаны примеры прототипов функций, из которых складывается интерфейс абстрактного типа данных.

Условие для выполнения лабораторной работы:

```
#include <assert.h>
// остальные необходимые библиотеки здесь
typedef enum { false, true } bool;
typedef struct {
    // необходимые поля с данными здесь
} abstract_data_t;
// рекомендуемые прототипы функций:
abstract_data_t make_copy(const abstract_data_t*);
abstract_data_t make_from_array(const int[], size_t);
abstract_data_t make_of_size(size_t);
abstract_data_t make_empty();
void clear(abstract_data_t*);
int at(const abstract_data_t*, int);
bool is_equal(
    const abstract_data_t*, const abstract_data_t*);
// код для проверки правильности выполнения задания:
int main() {
    int array[] = {1, 4, 7, 9};
    abstract_data_t a = make_from_array(array, 4);
    assert(9 == at(&a, 3));
    assert(9 == at(&a, -1));
    assert(9 == at(&a, 13));
    assert(1 == at(&a, -13));
    abstract_data_t b = make_copy(&a);
    assert(is_equal(&a, &b));
    clear(&a); clear(&b);
}
```

[4] Варианты см. в приложении 1.2.

Для самопроверки в пример включены утверждения `assert`. Работа не будет выполнена полностью до тех пор, пока эти утверждения не будут проходить. Таким образом для нового типа данных необходимо реализовать следующие операции.

Функция **`make_empty`** не имеет параметров, возвращает объект заданной структуры с внутренними полями, установленными в правильные значения по умолчанию.

Функция **`make_of_size`** имеет один параметр – целое беззнаковое число, которое указывает на желаемый размер коллекции, которую необходимо создать. Функция возвращает объект заданной структуры с внутренними полями установленными в правильные значения для того, чтобы сохранить созданный в функции контейнер.

Функция **`make_from_array`** имеет два параметра: указатель – адрес массива целых чисел, целое беззнаковое число – размер этого массива. Функция должна создать контейнер такого же размера, содержащий те же значения, что и массив-аргумент функции. Функция возвращает объект заданной структуры с внутренними полями установленными в правильные значения для того, чтобы сохранить созданный в функции контейнер.

Функция **`make_copy`** имеет один параметр – объект заданной структуры. Функция создает идентичную копию (*deep copy* – изменение копии объекта не должно изменять первоначальный объект) заданной структуры и возвращает объект заданной структуры с внутренними полями установленными в правильные значения.

Функция **`clear`** имеет один параметр – ссылку на объект заданной структуры, функция ничего не возвращает. По ссылке необходимо освободить память, занимаемую контейнером и установить поля объекта в правильные значения. Все существующие на момент вызова указатели становятся

недействительными. Асимптотичность функции линейная – ***O****(n)*.

Функция **`is_equal`** имеет два параметра – два объекта заданной структуры, которые необходимо сравнить. Функция возвращает “истинность”[5], если аргументы равны, и “ложь” в противном случае. Объекты считаются равными, если равны их контейнеры. Контейнеры равны, если они содержат равное количество элементов, а также – если значение каждого элемента одного контейнера равно элементу другого контейнера на том же месте.

Функция **`at`** имеет два параметра[6]: объект заданной структуры и целое число (которое может принимать отрицательные значения). Функция возвращает элемент массива из переданной структуры, находящийся по индексу, заданному вторым аргументом. Если второй аргумент отрицательный, необходимо предварительно преобразовать его в соответствующее положительное число, в зависимости от размера массива. Если значение второго аргумента не входит в диапазон возможных индексов массива, необходимо вернуть элемент массива, находящийся на первом или последнем месте, в зависимости от знака индекса. Так для списка `{1, 3, 5, 7, 9}` аргумент со значением 1 вернет второй элемент (3), аргумент со значением -1 вернет последний элемент (9), аргумент со значением 10 также вернет последний элемент, а аргумент со значением -10 вернет первый элемент (1).

Тестироваться код будет с помощью компилятора gcc, который не поддерживает функциональность языка **C++**. По этой причине рекомендуется компилировать сделанную

---

[5] Другие версии стандарта языка **C** содержат описание встроенного типа _Bool, который представляет самые маленькие беззнаковые целые числа (гарантирует объем памяти, достаточный для хранения значений 1 и 0) [WG14].

[6] Функция **`at`** реализуется только для тех вариантов, которые подразумевают возможность доступа к элементу контейнера по индексу.

лабораторную работу с помощью приложения `gcc` в файле с расширением `.c`, установив флаги `-Wall`, `-Wpedantic`, `-std=c17`.

2. Сокрытие данных и инкапсуляция

2.1. Теоретическая информация

В предыдущей главе была создана структура, которая представляет в программах дробные числа. Почти сразу же стало очевидно, что одной из проблем этой структуры является полный доступ к ее полям из любого места в программе. Это позволяет перезаписывать значения числителя и знаменателя в любой момент, в обход всех правил, которые были ранее установлены для создания рациональных чисел. Это также противоречит сути типа данных, который является абстрактным согласно установленному выше дизайну. Из того, что это абстрактный тип данных, следует, что пользователь не должен изменять эти поля напрямую. А это значит, что их нужно от пользователя как-то скрыть.

В **C++** для этого есть специальный механизм [Stroustrup]. Этот механизм позволяет запретить прямое изменение полей структуры одним из двух способов. Первый способ заключается в добавлении специального слова `private` в структуру, непосредственно над полями, которые необходимо скрыть:

```
struct rational {
private:
    int p; int q;
};
```

Стоит обратить внимание на двоеточие после данного слова, это напоминает *метки* в **C**, которые можно использовать для возврата к нужной строке в программе. Метка `private` делит структуру на две области: все, что находится над ней, доступно вне структуры, а все, что – под ней, скрыто и доступно только внутри структуры. В примере выше ничего кроме двух полей нет, поэтому поля *p* и *q* не доступны нигде.

Второй способ достичь такого же эффекта – использовать ключевое слово **`class`** вместо **`struct`** в определении структуры:

```
class rational { int p; int q; };
```

Оба подхода дают одинаковый результат, поэтому на практике можно использовать любой из них, но, как правило, **struct** используется, когда какие-то поля с данными все-таки планируется сделать открытыми для изменений, а **class** – когда поля должны быть скрыты от внешнего мира[7].

Независимо от того, какой метод будет выбран[8], эффект будет один: программа из предыдущей главы перестанет работать, так как создать переменную типа **rational** – ни напрямую, ни с помощью функции **cons** – будет нельзя. Доступ к полям *p* и *q* теперь возможен только внутри класса, а никакого другого кода там нет, поэтому и сами поля будут оставаться неизменными в течение жизни созданного объекта такого типа. Если писать программу в строгом соответствии с правилами языка **C**, на этом история типа **rational** бы и закончилась, так как никакой пользы в программах такие объекты бы не приносили. В **C** нет прямых механизмов сокрытия данных, потому что **C** создавался совсем для других целей (для написания ядра операционной системы Unix, например [Ritchie]). Конечно, добавив такую возможность в **C++**, его авторы добавили и способы работы с внутренними полями. Хорошая новость заключается в том, что для этого почти ничего не надо менять. Достаточно взять функцию **cons** и перенести ее внутрь самого класса:

```
class rational {
public:
    static rational cons(int x, int y) {
        assert(y != 0);
        rational r;
        if(x == 0) { r.p = 0; r.q = 0; }
```

[7] Если говорить конкретней, слово “класс” используется в ситуациях, когда необходимо подчеркнуть особенность способа управления процессов в программе, когда классы используются для создания новых типов данных, которые управляют выполнением команд через обмен сообщениями между объектами с помощью их методов [Goldberg].

[8] С этого момента в тексте будет использоваться слово **class**, все области доступа будут помечены соответствующей меткой.

```
        else if(y < 0) { r.p = -x; r.q = -y; }
        else { r.p = x; r.q = y; }
        return r;
    }
private:
    int p; int q;
};
```

Особое внимание стоит обратить на ключевое слово **static**, которое появилось перед определением функции в классе. Если читатель сталкивался с этим словом в языке **C**, то знает, что оно отвечает за доступ к отмеченному им идентификатору, *статические локальные переменные* можно использовать в любом месте файла, в котором они объявлены, а *статические функции* не видны за пределами файла, в котором объявлены (это верно и для *глобальных* переменных).

В **C++** это слово приобретает новые свойства, которые легче будет понять на конкретном примере. Допустим, что этого слова там нет. Попробуем теперь создать новую переменную, используя эту функцию. Первый же вопрос, который встанет перед программистом: как вызвать функцию, которая находится внутри какого-то класса? Известно, что к полям структуры необходимо обращаться либо через *оператор выбора* (**.**), либо через *оператор косвенного выбора* (**->**), в зависимости от типа данных объекта. Так как новый объект представлен типом **rational**, вызов функции мог бы выглядеть так:

```
rational b; b.cons(20, 50);
```

Интуиция в этом случае оказалась верной. Авторы **C++** не стали изобретать велосипед и сохранили привычный синтаксис языка **C** для этого случая. Но при этом код немного потерял в лаконичности: необходимо сначала создать ничем не инициализированный объект, а потом использовать функцию, чтобы изменить его значения. Более того, это даже вводит в заблуждение, потому что такой вызов функции не изменит саму переменную *b*. Функция вернет копию нового объекта, который надо будет сохранить в *b* с помощью присваивания:

```
rational b = b.cons(20, 50);
```

Такая запись выглядит странно. Ключевое слово **static** в примере выше как раз и нужно для того, чтобы исключить двоякое чтение кода. Почему *b* находится и справа, и слева от оператора присваивания? Что происходит раньше: создание переменной или вызов функции? Эту запись стоило бы сделать более понятной.

Но есть другой нюанс. Согласно правилам языка статические функции ничего не знают об объектах того типа, которому принадлежат. Они знают только внутреннюю структуру самого типа данных. Как следствие, форма записи **b.cons** или **b->cons** для статических функций не нужна. Но и запись вида **rational b = cons(10, 50)** невозможна, потому что невозможно определить где в файле находится идентификатор **cons**. И здесь на помощь приходит *оператор расширения области видимости* (**::**), указывающий на область видимости, в которой нужно искать заданное имя. В примере выше область видимости для имени функции **cons** определена границами класса **rational**, а значит и обращаться к функции надо посредством имени класса:

```
rational b = rational::cons(10, 50);
```

То есть, функцию под таким именем нужно искать где-то внутри класса **rational**; так получилась желанная форма записи, при этом другие поля класса защищены от внешнего воздействия. Осталось обсудить возможные альтернативы такого подхода.

Один из самых распространенных способов – создать специальные функции внутри класса, которые дают возможность изменять поля класса при соблюдении определенных правил. Такие функции называются “мутаторами” (от слова mutate – “изменять”) или “сеттерами” (от слова set – “устанавливать”):

```
class rational {
```

```
public:
    void set_p(int x) {
        p = x;
    }
    void set_q(int x) {
        assert(x != 0);
        q = x;
    }
private:
    int p; int q;
};
```

Фрагмент ниже может показаться знакомым. Как и в самом первом примере, здесь объект под именем *c* не конструируется с этими значениями. Они задаются уже после создания переменной:

```
rational c; c.set_p(5); c.set_q(10);
```

Таким образом, в одном случае объект создается с уже заданными полями и может быть использован в других контекстах, в другом случае объект необходимо как-то инициализировать после создания – это два разных подхода к созданию объектов. Возникает закономерный вопрос, “какой из этих способов лучше?”. Ведь принципиальное отличие функции **cons** от сеттеров заключается в том, что она *производит новый объект*, в то время как сеттеры *изменяют уже существующий*.

На практике оба метода могут оказаться нужны, поэтому принято их комбинировать. Необходимость включения сеттеров в класс диктуется тем, какую функциональность должен предоставлять тип данных. Если нужно создать объект с заранее заданными полями, выбор очевиден. Если же нужно изменить свойства существующего объекта, проще вызвать сеттер, чем создавать новый объект. Хотя, на практике часто выбирают создание нового объекта, чтобы избежать *побочных эффектов*.

С этого момента введем два новых термина. Факт группирования данных и операций над ними внутри какой-то структуры данных мы будем называть *инкапсуляцией*.

Совокупность всех *открытых* операций (т.е. функций, которые не скрыты меткой `private`), определенных внутри класса, мы будем называть ее *интерфейсом*.

2.2. Практические задачи для закрепления теории

а. Выше были упомянуты возможные побочные эффекты при использовании сеттеров. Следует рассмотреть один такой эффект и подумать, как его избежать. В предыдущем задании была разработана функция, которая сокращает дробь. Очевидно, что если изменить значение числителя или знаменателя после сокращения, сокращать дробь придется еще раз. Было бы лучше, если бы это происходило само собой: как при конструировании объекта, так и при любых изменениях числителя или знаменателя. Это означает, что функцию **`reduce`** можно использовать по-разному. Можно сделать ее внешней относительно класса, используя сеттеры и геттеры[9] для реализации. Можно сделать ее внутренней скрытой функцией и вызывать в нужных местах. Наконец, можно сделать ее внутренней статической функцией, включив ее в интерфейс класса, чтобы ее можно было вызывать в любом месте программы. Рекомендуется попробовать самостоятельно реализовать все три варианта. В разделе с примерами представлен один из них.

Отдельно стоит отметить, что сеттеры по своей природе нарушают *принцип закрытости* класса. Подробней об этом и других принципах будет говориться в последующих главах, однако важно иметь ввиду, что делать поля закрытыми с помощью метки `private`, а затем разрешать их изменение с помощью специальной функции может противоречить общему решению.

б. Это упражнение призвано развенчать некоторые мифы, связанные с классами в **C++**. Для его выполнения понадобится

[9] Геттер (или “аксессор”) – (от слова get, англ.) внутренняя функция какой-либо структуры данных, которая возвращает значение внутренних полей структуры, часто – без возможности их последующего изменения.

некоторое знание указателей и того, как структурируется виртуальная память процесса в языке **C**. Необходимо создать переменную типа **`rational`**, а затем изменить ее скрытые поля данных *p* и *q*, не меняя класс **`rational`** и не используя сеттеры. Следует подумать о том, как это может быть связано с указателями и с тем, как они работают в **C**.

3. Конструкторы

3.1. Теоретическая информация

Понятие конструктора существует во многих языках программирования и также является абстракцией, основное предназначение которой – абстрагироваться от процедуры конструирования объекта в программе. Если пробежаться глазами по примерам из предыдущих глав, можно заметить постепенный переход от конструирования объектов типа **rational** вручную к делегированию этого конструирования специальной функции (**cons**). Благодаря этому были установлены конкретные правила конструирования объектов. Таким образом любой программист, который в будущем захотел бы воспользоваться типом данных **rational**, вынужден создавать переменные посредством функции **cons**.

Здесь есть несколько тонкостей. Во-первых, было бы неплохо добиться того, чтобы объекты можно было создавать без явного вызова специальной функции. Например, если пользователь не знает, какие первоначальные значения для числителя и знаменателя задать, конструктор мог бы определять их самостоятельно. Во-вторых, было бы удобно создавать новые объекты, используя уже существующие объекты напрямую, без явного вызова геттеров и последующей передачи их в конструктор, как было сделано в одном из примеров выше:

```
return cons(r.p, r.q);
```

Было бы логичней если бы оно выглядело, например, так:

```
return rational(r);
```

В этом случае намерение данной инструкции выглядит более явным: вызвать специальную функцию, которая носит имя того типа данных, который будет использоваться для создания нового объекта, при этом новый объект будет создан на базе существующего.

Это подразумевает установление для всех полей нового объекта значений на основании величин, хранящихся в соответствующих полях аргумента. Используя последний пример, новый объект, который будет создан в результате этой операции, в своем поле p будет содержать значение поля p, которое принадлежит объекту r, в своем поле q будет содержать значение поля q, которое принадлежит объекту r.

Существуют и другие типы конструкторов. Например, можно написать такую функцию, которая создает объект типа **`rational`** на основании переданной строки определенного вида (для строки "$^3/_2$" создать объект с полем p равным 3, и полем q равным 2). Или, как упоминалось ранее, можно написать функцию, которая создает рациональные числа, в которых значения полей p и q задаются по умолчанию в том случае, если программист при создании объекта не задал их явно (например, чтобы исключить возможность появления 0 в знаменателе для неинициализированного q).

Для всего этого можно сделать соответствующие функции внутри заданного класса, дав этим функциям названия, которые указывают на их предназначение. И в некоторых языках программирования это пришлось бы сделать, так как они не позволяют создавать больше одного конструктора. В **C++** конструкторов может быть сколько угодно, отличаться они должны только в том, какие аргументы они принимают. Например, ниже можно посмотреть на код, который конвертирует метод **`cons`** в конструктор с двумя аргументами – желаемыми значениями числителя и знаменателя:

```
rational(int x, int y) {
        assert(y != 0);
        if(y < 0) { p = -x; q = -y; }
        else { p = x; q = y; }
        if(x != 0) reduce(*this);
}
```

Пример выше содержит несколько синтаксических приемов, которые до сих пор в этом тексте не встречались. В

соответствии с правилами **C++** у конструкторов есть ряд неявных свойств, о которых следует знать перед тем, как начать ими пользоваться.

Первое, что бросается в глаза – отсутствие типа данных перед именем этой функции. Второе – это некий указатель **this**, используемый в последней строке функции. Третье – само имя функции повторяет имя класса, внутри которого функция задана. Это продиктовано условностями реализации стандартов языка **C++** на различных платформах. Как и в случае с меткой `private`, это условности, которые устанавливаются на уровне компилятора для удобства и соблюдения единого стандарта. Чтобы разобраться с этими нюансами, нужно в первую очередь запомнить, что конструктор – это такая же функция, как и другие функции, которая имеет ряд синтаксических особенностей.

Первая особенность заключается в том, что на уровне кода пользователя конструкторы имеют сокращенную форму. В первую очередь, это выражается в том, что у них, как и *у всех функций, находящихся внутри класса*, есть **скрытый первый аргумент**. У этого аргумента есть тип и название (речь идет о ключевом слове **this** из предыдущего фрагмента). Если бы этот аргумент задавался явно, выглядеть это могло бы следующим образом:

```
void cons(rational *_this, int x, int y) {
    assert(y != 0);
    if(y < 0) { _this->p = -x; _this->q = -y; }
    else { _this->p = x; _this->q = y; }
    if(x != 0) rational::reduce(*_this);
}
```

В этом фрагменте кода нет ничего загадочного. В функцию передается адрес, по которому хранятся данные рационального числа, и посредством этого адреса функция может изменять эти данные. Единственный вопрос, который закономерно возникает: как передать этот адрес в функцию, которая скрывает аргумент **this**? Здесь стоит вспомнить что

было сказано в предыдущей главе о вызове нестатичных функций, которые объявлены внутри класса. Как вы уже знаете, эти функции вызываются следующим образом (где *b* – это переменная типа **rational**):

```
b.cons(20, 50);
```

Вызов такой функции возможен только посредством существующего объекта, следовательно функция изменяет состояние объекта, который ее вызывает (при этом он явно не обозначается как один из аргументов), из чего можно сделать вывод, что объект туда передан неявно.

Этот объект передается как указатель, для удобства названный **this**, что намекает на природу изменяемого объекта. Слово "этот" *указывает на переменную, которая использовалась для вызова функции* (то есть, в контексте функции ее вызвал "этот объект"). В примере выше – это адрес переменной *b*, соответственно *b.p* примет значение 20, а *b.q* – 50. Другими словами, если вне класса к полям объекта надо обращаться через оператор-точку, применяемую к его имени в программе, то внутри класса – через оператор-стрелочку, применяемую к его адресу в программе. С этого момента *функции, объявленные внутри класса*, которые содержат неявный аргумент **this**, будут называться *методами*.

Для конструкторов это тоже верно, то есть, они являются специальными методами. Главная же особенность конструкторов заключается в том, что они *вызываются автоматически во время создания объекта* в программе. Иначе говоря, каждая переменная соответствующего типа, объявленная в программе, в момент своей инициализации вызовет подходящий конструктор. Это также верно для классов, в которых не объявлено ни одного явного конструктора. В **C++** *компилятор автоматически генерирует конструкторы* в меру своих способностей *в случае их отсутствия*. На практике на это полагаться не рекомендуется, поэтому если ваш конструктор должен соблюдать определенный набор правил, его надо задавать явно.

Конкретней, для типа **rational** следующая запись вызовет *конструктор по умолчанию*:

```
rational a;
```

Конструкторы по умолчанию выполняют действия по инициализации полей класса в тех случаях, когда для этого нет конкретных данных. В данном примере конструктор по умолчанию будет сгенерирован автоматически. Так как внутренние поля представлены примитивным типом **int**, сгенерированному конструктору ничего не придется делать. Но в предыдущих главах было установлено как минимум одно правило, которое должно выполняться даже для таких случаев: знаменатель не должен быть равен нулю. Для обработки такого сценария существует несколько возможных вариантов.

Первый способ заключается в предварительной инициализации полей внутри класса:

```
class rational {
private:
    int p{1}; int q{1};
};
```

Используя *инициализаторы* (фигурные скобки после имени поля с данными) можно задать первоначальные значения для всех переменных такого типа. Это делается еще до вызова конструктора. Так можно гарантировать правильные значения для всех полей во избежание странных ошибок в дальнейшем.

Другой способ заключается в задании значений по умолчанию для одного из конструкторов с параметрами:

```
class rational {
public:
    rational(int x = 1, int y = 1) {
        assert(y != 0);
        if(y < 0) { this->p = -x; this->q = -y; }
        else { this->p = x; this->q = y; }
        if(x != 0) reduce(*this);
    }
private:
```

```
    int p; int q;
};
```

Строка **rational(int** *x*=1, **int** *y*=1) использует свойство функций **C++** устанавливать значения по умолчанию для параметров (соблюдая определенные правила [IBM]). У этого свойства есть интересные последствия для конструкторов:

```
rational c = rational(9, 2);
assert(9 == c.get_p() && 2 == c.get_q());
rational d;
assert(1 == d.get_p() && 1 == d.get_q());
```

То есть, в данном случае компилятор считает, что конструктор с параметрами одновременно является и конструктором по умолчанию. Если вызвать этот конструктор с конкретными значениями, они заменят аргументы по умолчанию при вызове функции, если же не передавать ничего, этот конструктор все равно будет вызван, но аргументы будут иметь значения по умолчанию, определенные в заголовке функции.

Третий способ – это запрет на конструирование объектов заданного типа без параметров. Если в программе нет ситуаций, в которых рациональное число нужно было бы создавать без конкретных значений, лучше себя обезопасить от случайностей, указав на это в классе, чтобы компилятор мог распознавать все попытки так сделать:

```
class rational {
public:
    rational() = delete;
    rational(int x, int y) {
        assert(y != 0);
        if(y < 0) { this->p = -x; this->q = -y; }
        else { this->p = x; this->q = y; }
        if(x != 0) reduce(*this);
    }
private:
    int p; int q;
};
```

Здесь появляется новое ключевое слово **delete** (не путать с оператором **delete** [Whitney(a)]). В данном случае оно указывает на то, что конструктор без параметров для этого типа использовать нельзя. Если теперь попробовать создать переменную без явной инициализации: **rational** *a* – компилятор покажет ошибку, “error: use of deleted function 'rational::rational()”. Обратите внимание на то, что в конструкторе с параметрами больше нет значений по умолчанию. Ведь параметры по умолчанию превращают конструктор в *конструктор по умолчанию*, и последний фрагмент тогда имел бы два конструктора по умолчанию. То есть, нужно сделать выбор в пользу одного из этих двух вариантов.

Следующий важный конструктор, который стоит реализовать – это *конструктор копирования*. Он нужен для того, чтобы в программе можно было создавать новые переменные как копии уже существующих переменных этого же типа:

```
rational a(2, 3); rational b(a);
assert(2 == b.get_p() && 3 == b.get_q());
```

Здесь надо быть очень внимательным. Как и в случае с конструктором по умолчанию, компилятор генерирует конструктор копирования в определенных ситуациях, поэтому пример кода выше может быть вполне рабочим. Это происходит потому, что все поля в классе **rational** представлены примитивным типом данных **int**. Компилятор знает как копировать данные типа **int** и без чужой помощи. Для составных типов данных вроде массивов оставлять копирование на совести компилятора не следует. Для данного случая конструктор копирования мог бы выглядеть следующим образом:

```
class rational {
public:
```

```
    rational(const rational &other) {
        this->p = other.p; this->q = other.q;
    }
private:
    int p{1}; int q{1};
};
```

Как уже было установлено, в конструктор передается аргумент для скрытого параметра **rational *this**[10]. Явно в пару к нему заявляется параметр – *ссылка*[11] *на объект* (указатель, который не может принимать значения **NULL** или **nullptr**) такого же типа. Таким образом, необходимо поочередно скопировать значения всех полей второго аргумента (*other* в примере) в первый (**this**). Существует и другая форма записи, использующая *список инициализации*:

```
class rational {
public:
    rational(const rational &other) :
        p(other.p), q(other.q) {}
private:
    int p{1}; int q{1};
};
```

То есть, в случаях, когда достаточно скопировать данные из одной переменной в другую, язык позволяет сделать это до вызова тела конструктора.

---

[10] С этого момента указатель **this** в коде явно использоваться не будет, так как не является обязательным в случаях, в которых компилятор сам может определить принадлежность переменных в методе.

[11] Ссылки в **C++** отличаются от указателей в **C** тем, что они уже разыменованы в момент инициализации; если для какого-то **int a** ссылка создается с помощью оператора **&a**, то для **int& a** объект уже ведет себя как **int**. Другими словами, если к данным указателя вы обращаетесь через оператор разыменования *, для ссылок это делать не нужно, они ведут себя как обычные значения.

3.2. Практические задачи для закрепления теории

а. Используя примеры из этой главы необходимо написать конструктор, который создает объект типа `rational` на основании переданной строки определенного вида (например, для строки “$^3/_2$” создать объект с полем p равным 3, и полем q равным 2).

б. *Конструктор переноса* является еще одним “каноническим конструктором”, который в **C++** принято реализовывать из соображений быстродействия программ. Останавливаться на нем очень подробно сейчас не стоит, но познакомиться с принципами его работы полезно. Работает он следующим образом. Для значений особого типа такой конструктор не копирует передаваемый ему объект, а “крадет” существующие в памяти значения, переписывая адреса этих значений на свой объект. Таким образом переданный в этот конструктор объект остается без действительных значений, так как все они были перенесены в другой объект. В зависимости от конкретной реализации языка **C++** это может происходить автоматически[12]. В большинстве же случаев программисты самостоятельно отмечают переменные, которые они хотят перенести. Такие переменные принято называть ссылками на *R-значения*[13] и в коде отмечать как **&&**.

В одной из следующих глав будет рассмотрен более конкретный пример того, зачем такой конструктор нужен на практике. Главное не зацикливаться на этих нюансах если что-то пока не понятно. Задача данного текста заключается не в том,

---

[12] Компилятору позволяется самому выбирать между копированием и переносом в ситуациях, когда переменная не отмечена ключевым словом **volatile** или не является аргументом функции [Polacek].

[13] Это название исторически закрепилось за анонимными значениями в коде, так как они всегда находятся справа от оператора присваивания (отсюда и название, right-hand value – R-value). Примером такого значения может быть любое константное число (int x = **5**), которое указывает на присваиваемое в переменную первоначальное значение. В **C++** переменные тоже могут обладать свойствами R-значений, в этом случае их называют *X-значениями*. https://en.cppreference.com/w/cpp/language/value_category.

чтобы объяснить особенности реализации языка **C++**, который используется только в ознакомительных целях. Загвоздка же в том, что ссылки на R-значения являются способом реализации важных абстрактных идей, который выбрали разработчики **C++**, и следует иметь это ввиду. Следует написать такой конструктор самостоятельно, а затем свериться с готовым примером из соответствующего раздела.

в. Возвращаясь к конструктору переноса осталось сделать функцию по обмену значениями двух рациональных чисел. Наивная реализация могла бы выглядеть так:

```
void swap(rational &a, rational &b) {
    rational tmp = a; a = b; b = tmp;
}
```

Но в контексте реализованного конструктор переноса уже стоит иметь в виду существование функции `std::move`, потому что в данном случае нет смысла копировать значения. Отдельно стоит подчеркнуть, что значимость всех этих манипуляций с типами значений не сразу очевидна на таких простых типах данных, как **`rational`**. Все таки, он сам состоит их двух примитивных типов. Если бы он в себя включал указатели, конструктор переноса заиграл бы новыми красками. Следует смоделировать эту ситуацию и подумать, как конструктор переноса способствует более эффективной работе программы в таком случае.

Лабораторная работа 2
Тема: Конструкторы

Цель работы: Для заданного варианта[14] добавить в класс все нужные конструкторы и деструктор для инициализации внутренних полей. Ниже показаны примеры прототипов функций, из которых складывается интерфейс абстрактного типа данных.

Условие для выполнения лабораторной работы:

```
class abstract_data_t {
public:
    // рекомендуемые прототипы функций:
    ~abstract_data_t();
    abstract_data_t();
    abstract_data_t(size_t);
    abstract_data_t(const int[], size_t);
    abstract_data_t(const abstract_data_t&);
    void clear();
    int at(int);
    void resize(size_t);
    void assign(size_t, int);
    size_t length();
    bool empty();
    void swap(const abstract_data_t&);
private:
    // необходимые поля с данными здесь
};
```

Таким образом необходимо реализовать следующие операции.

Конструктор по умолчанию не имеет параметров, ничего не возвращает, инициализирует внутренние поля класса в правильные значения по умолчанию.

Конструктор с параметром типа **`size_t`** принимает целое беззнаковое число, которое указывает на желаемый размер

[14] Варианты см. в приложении 1.2.

коллекции, которую необходимо создать, чтобы сохранить созданный в функции контейнер.

Конструктор с двумя параметрами (указатель – адрес массива целых чисел, целое беззнаковое число – размер этого массива) должен создать контейнер такого же размера, содержащий те же значения, что и массив-аргумент функции.

Конструктор копирования имеет один параметр – объект заданного класса, создает идентичную копию (*deep copy* – изменение копии объекта не должно изменять первоначальный объект) заданного класса с внутренними полями, установленными в правильные значения.

Деструктор не имеет параметров. Его задача – освободить память, занимаемую контейнером и установить поля объекта в правильные значения. При этом функция **`clear`** из предыдущей лабораторной работы конвертируется в метод класса, поэтому параметров больше не имеет, ничего не возвращает. В этом методе также необходимо освободить память, занимаемую контейнером, и установить поля объекта в правильные значения.

Функция **`at`** из предыдущей лабораторной работы конвертируется в метод класса. Метод имеет один параметр[15] – целое число (которое может принимать отрицательные значения), возвращает элемент коллекции из текущего объекта, находящийся по индексу, заданному вторым аргументом. Если второй аргумент отрицательный, необходимо предварительно преобразовать его в соответствующее положительное число, в зависимости от размера коллекции. Если значение второго аргумента не входит в диапазон возможных индексов коллекции, необходимо вернуть элемент коллекции, находящийся на первом или последнем месте, в зависимости от знака индекса.

Метод **`resize`** принимает беззнаковое целое число в качестве аргумента – количество элементов, которые должен в

[15] Функция **at** реализуется только для тех вариантов, которые подразумевают возможность доступа к элементу контейнера по индексу.

себе содержать контейнер после операции. Метод оставляет нужное количество элементов, считая сначала, удаляя остальные. Если **length** меньше чем значение аргумента, оставшиеся места в массиве заполняются значениями по умолчанию определяемыми для конкретного *типа данных*. Асимптотичность метода линейна относительно разницы между величинами **length** и значением аргумента – ***O(n)***.

Метод **assign** принимает два аргумента: беззнаковое целое число, обозначающее количество элементов, целое число, содержащее новое значение этих элементов. Метод заменяет элементы коллекции на заданное количество с заданным значением для всех новых элементов.

Метод **empty** возвращает булевое значение: “истинность”, если коллекция не содержит ни одного элемента, и “ложь” в противном случае.

Метод **swap** принимает ссылку на объект заданного типа и обменивает значения элементов внутренней коллекции с элементами коллекции из аргумента. При этом метод не должен производить операции копирования или переноса над отдельными элементами коллекций, сохраняя все существующие в программе ссылки на отдельные элементы обеих коллекций. Асимптотичность метода константная – ***O(1)***.

Метод **length** возвращает текущий размер коллекции.

Функция **print** принимает константную ссылку на объект заданного типа и выводит элементы контейнера в порядке возрастания.

Для проверки в пример включены утверждения `assert`. Работа не будет выполнена полностью до тех пор, пока эти утверждения не будут проходить:

```
#include <cassert>
#include <iostream>
// остальные необходимые библиотеки здесь
void print(const abstract_data_t&);
// код для проверки правильности выполнения задания:
int main() {
```

```
    int array[] = {1, 4, 7, 9};
    abstract_data_t a(array, 4);
    assert(4 == a.length());
    assert(1 == a.at(0));
    assert(9 == a.at(10));
    print(a);
    abstract_data_t b(a);
    b.clear();
    assert(b.empty());
    abstract_data_t c;
    assert(c.empty());
    c.assign(3, 7);
    assert(3 == c.length());
    abstract_data_t d(5);
    assert(5 == d.length());
    d.resize(10); b.swap(d);
    assert(10 == b.length());
    assert(d.empty());
}
```

Также следует иметь ввиду, что тестироваться код будет с помощью компилятора G++, который может не поддерживать функциональность языка **C**, к которому студенты привыкли. По этой причине рекомендуется компилировать сделанную лабораторную работу с помощью приложения `g++` в файле с расширением `.cpp`, установив флаги `-Wall`, `-Wpedantic`, `-std=c++17`.

4. Перегрузка функций и операторов

4.1. Теоретическая информация

В большинстве примеров выше для проверки функциональности кода используется ключевое слово **assert**. В последнем примере мелькает следующая строка:

```
assert(1 == g.get_p() && 5 == g.get_q());
```

Логический оператор “и” (**&&**) указывает на то, что все выражение в скобках проверяется на истинность. Если результат выполнения метода **get_p** для объекта *g* равен единице и результат выполнения метода **get_q** для этого же объекта равен 5, то строка кода программой игнорируется, в противном случае работа программы будет прервана.

Стоит напомнить читателю, что метод **get_p** всего лишь возвращает значение поля *p* заданного объекта, и аналогично **get_q** – значение поля *q*. Выше уже говорилось о том, что данные методы нарушают инкапсуляцию, то есть их пришлось сделать конкретно для этого сравнения, потому что другого способа проверить текущие значения скрытых полей нет. Было бы лучше иметь под рукой какой-то механизм проверки всего этого условия без того, чтобы копаться в недрах объекта и высматривать значения отдельных полей.

Например, можно заменить проверку каждого поля в отдельности на один вызов специального метода, который отвечает на вопрос, “равны ли поля объекта переданным значениям?”:

```
class rational {
public:
	bool eq(int __p, int __q) {
	    return p == __p && q == __q;
	}
private:
	int p{1}; int q{1};
};
```

Тогда следующий фрагмент кода мог бы проверять поля заданного объекта.

```
rational e(1, 5);
assert(e.eq(1, 5));
```

Если позволить фантазии разгуляться, можно представить, что будут ситуации, где таким же образом надо сравнить два объекта типа **rational**. В предыдущих частях это бы делалось поочередным сравнением соответствующих полей двух объектов с помощью методов **get_p** и **get_q**. Но лучшим вариантом было бы сделать отдельный метод, который всю “кухню” скрывает внутри класса, аналогично примеру выше:

```
class rational {
public:
	bool eq(int __p, int __q) {
	    return p == __p && q == __q;
	}
	bool eq(const rational& other) {
	    return p == other.p && q == other.q;
	}
private:
	int p{1}; int q{1};
};
```

Тогда вызов выглядит так:

```
rational a(2, 3); rational b(a);
assert(b.eq(a));
```

Было бы удобно не только сохранить оба этих определения, но и если бы можно было делать сколько угодно таких методов с одинаковыми названиями. Разработчики **C++** именно так и думали, потому что оно так и работает. До тех пор, пока функции имеют *отличающиеся списки аргументов, они могут иметь одинаковое название и возвращаемый тип*.

Для читателя это не является новостью, потому что в предыдущих примерах это свойство функций в языке **C++** уже использовалось для конструкторов. Все конструкторы называются одинаково и отличаются только в количестве и типах параметров. Исходя из типов и порядка переданных аргументов компилятор выбирает какой конструктор вызвать. С другими методами это работает так же, потому что и конструкторы, и методы являются особым видом функций. Когда говорят о *перегрузке* (“overload”, не путать с ключевым словом **`override`**) именно это свойство и подразумевают.

Имея под рукой все методы, описанные выше, можно сравнивать любые рациональные числа друг с другом, что очень полезно само по себе, но не очень удобно. Например, имея массив рациональных чисел пользователь мог бы его отфильтровать, игнорируя все отрицательные числа в массиве:

```
if(a[i].lt(0)) { … }
```

Здесь **`lt`** – это метод из класса **`rational`**, который проверяет, если рациональное число слева строго меньше целочисленного аргумента метода. Даже на этом простом примере видно, как синтаксис языка делает достаточно простую операцию сложной для восприятия. Было бы намного проще понять что происходит, если бы выражение выглядело так:

```
if(a[i] < 0) { … }
```

Язык **C++** позволяет это сделать. *Операторы* являются функциями, а функции можно перегружать, следовательно, операторы тоже могут быть перегружены[16]. Для этого

[16] https://en.cppreference.com/w/cpp/language/operators.

достаточно заменить название функции на ключевое слово **operator**, с указанием символа нужного оператора. Например, можно переписать функцию **eq**:

```
class rational {
public:
       bool operator==(const rational &other) {
           return p == other.p && q == other.q;
       }
private:
       int p{1}; int q{1};
};
```

Тогда сравнение двух рациональных чисел могло бы выглядеть так:

```
assert(b == rational(2, 3));
```

Но не следует спешить переписывать все методы, сделанные в предыдущей главе, как операторы. Перегрузка операторов требует следовать определенным правилам, и не все методы в классе **rational** этим правилам соответствуют. Например, нельзя превратить в оператор метод **eq**, работающий с двумя целыми числами:

```
bool operator==(int __p, int __q);
```

Методы вызываются на объект с помощью оператора-точки, то есть метод всегда имеет аргумент **this**, а это значит, что для такой формы записи оператора он бы принимал три аргумента. Но читателю уже должно быть известно, что оператор сравнения принимает только два аргумента – слева и справа от оператора. Ошибка компилятора для такого варианта оператора это же и говорит, “error: bool rational::operator==(int, int)' must have exactly one argument”. То есть, в такой форме записи оператора уже есть параметр **this**, вдобавок к которому можно передавать только один дополнительный аргумент. Если попробовать исправить это, убрав скрытый аргумент **this** с помощью ключевого слова **static**:

```
static bool operator==(int __p, int __q);
```

компилятор сгенерирует другую ошибку, “error: static bool rational::operator==(int, int)' must be either a non-static member function or a non-member function”. Другими словами, убрав аргумент **this** и избавившись от переменных типа **rational** в этом методе совсем, пропала необходимость держать его внутри класса, так как он не оперирует данными класса. Если же явно задать аргумент типа **rational** для этого оператора, опять создается патовая ситуация с тремя аргументами.

Единственное, что позволяется сделать – это задать оператор сравнения с одним единственным целым числом:

```
bool operator==(int x);
```

Это позволит делать сравнения такого вида: `a == 1`; при этом сравнения вида `1 == a` будут невозможны, так как первоначальная форма оператора предполагает, что объект типа **rational** всегда слева (он является скрытым аргументом **this**, который всегда проверяется слева от оператора).

Данное ограничение тоже можно обойти с помощью ключевого слова **friend**:

```
class rational {
public:
       friend bool
       operator==(int lhs, const rational &rhs) {
           return rhs.p == lhs && rhs.q == lhs;
       }
private:
       int p{1}; int q{1};
};
```

Ключевое слово **friend** позволяет обойти ограничение на обязательное использование аргумента **this**, достаточно чтобы аргументов было адекватное количество (для бинарного оператора – два), и чтобы один из аргументов был типа **rational**:

```
rational a;
assert(1 == a);
```

Проблема только в том, чтобы запомнить все эти тонкости. Это очень показательная ситуация. Язык **C++** позволяет сделать значительную часть кода легко читаемой при условии, что программист готов затратить большие усилия на предварительную работу по подготовке всех нужных для этого инструментов.

Часто используемый прием при перегрузке функций – использовать уже готовые функции как основу реализации новых, если эти функции как-то связаны между собой. Например, с помощью оператора “равно” можно реализовать оператор “не равно”, используя *булевое отрицание*:

```
class rational {
public:
       friend bool
        operator==(int lhs, const rational &rhs) {
            return rhs.p == lhs && rhs.q == lhs;
        }
        friend bool
        operator!=(int lhs, const rational &rhs) {
            return !(lhs == rhs);
        }
private:
       int p{1}; int q{1};
};
```

Аналогичным способом можно перегрузить оператор “меньше”. Особое внимание следует обратить на версию этого оператора, которая сравнивает два рациональных числа. Этот оператор используется компилятором для сортировки, поэтому при необходимости отсортировать список рациональных чисел определение этого оператора придется предоставить.

“Наивная” версия алгоритма могла бы выглядеть так: если знаменатель левой части равен знаменателю правой части, меньше то число, числитель которого меньше; если знаменатели не равны, оба числа предварительно необходимо привести к

общему знаменателю и сравнить полученные таким образом два числа. На деле реализация может оказаться не такой тривиальной, так как алгоритм может зациклиться в бесконечной рекурсии; конструирование новых объектов и их последующее сравнение может организоваться в цикл последовательных вызовов самого оператора “меньше”.

Другой алгоритм предполагает использование следующего свойства для некоторых натуральных чисел, при условии, что $b, d \notin \{0\}$:

$$\frac{a}{b} < \frac{c}{d} \Leftrightarrow a \cdot d < c \cdot b$$

На первый взгляд такой метод предельно прост в реализации, не считая возможных проблем с отрицательными аргументами[17]. Тем не менее, он редко встречается в реализациях данной операции, в частности из-за того, что результат умножения числителя на знаменатель имеет тенденцию к переполнению[18]. В примере ниже используется метод сравнения рациональных чисел, конвертированных в числа с плавающей запятой:

```
friend bool
operator<(const rational &lhs, const rational &rhs) {
        auto x = (long double)lhs.p / (long double)lhs.q;
        auto y = (long double)rhs.p / (long double)rhs.q;
        return x < y;
}
```

Синтаксис уже должен быть понятен. Здесь сохраняется проблема переполнения в том случае, когда числитель или

[17] Например, сравнение двух чисел `rational(2, 3)` и `rational(-3, -4)` даст неправильный результат, так как он зависит от сравнения выражений 2 * -4 и -3 * 3, то есть, -8 и -9, где правая часть меньше левой.

[18] На практике применяется способ, который не вызывает переполнения при арифметических операциях с числителем и знаменателем; например, реализация оператора “меньше” в библиотеке Boost упрощает аргументы, приводя их к непрерывным дробям, используя алгоритм Евклида для нахождения наибольшего общего делителя [Moore].

знаменатель хотя бы одного из операндов представлены значениями, превышающими максимальное значение типа **long double**. Соответственно, для данной реализации *инвариантой*[19] является диапазон допустимых значений для числителя и знаменателя аргументов. В данном случае это определяется на уровне класса **rational**, а следовательно может контролироваться выбранным изначально типом, единственное требование к которому – быть подмножеством (или приводиться к нему) множества чисел, представляемых типом **long double**. Для такой реализации код ниже будет рабочим:

```
rational b(2, 3); rational c(3, 4);
assert(b < c);
```

Реализовать все остальные операторы сравнения достаточно просто, они могут быть выражены через уже сделанный оператор “меньше”:

```
friend bool
operator>(const rational &lhs, const rational &rhs) {
        return rhs < lhs;
}
friend bool
operator>=(const rational &lhs, const rational &rhs) {
        return !(lhs < rhs);
}
friend bool
operator<=(const rational &lhs, const rational &rhs) {
        return !(rhs < lhs);
}
```

4.2. Практические задачи для закрепления теории

а. Внимательные читатели успели заметить одну деталь в фрагментах кода, представленных выше, которая в тексте пристально не рассматривалась. До того момента, как конструкторы были введены в класс **rational**, создавать объекты все равно было можно. Более того, можно было

[19] Свойство какой-либо сущности в выражении, которое остается неизменным независимо от условий.

создавать объекты на базе других объектов, например, таким образом:

```
rational b = a;
```

Здесь объект *a* тоже имеет тип **rational**. Это возможно благодаря сгенерированному автоматически конструктору копирования. Нюанс же заключается в следующем. Что произойдет, если дополнить строку выше такой инструкцией: *a = b*? Здесь конструирования не происходит, потому что оба объекта уже были инициализированы выше, но данные должны копироваться из области, связанной с именем справа от оператора присваивания, в область имени слева от оператора присваивания.

Секрет как раз и состоит в том, что оператор присваивания для класса **rational** тоже будет сгенерирован автоматически. Как можно догадаться, этот оператор тоже можно перегрузить. Он во многом повторяет логику соответствующего конструктора копирования за одним исключением. Если при конструировании объекта не нужно беспокоиться о его состоянии, то при вызове оператора присваивания объект, владеющий этим оператором (то есть, аргумент **this**), может уже существовать в программе какое-то время, и его состояние этим оператором не устанавливается начисто, а изменяется с одного на другое. В таких ситуациях предварительно приходится объект готовить к присваиванию, для чего оператор должен делать определенные проверки. Конкретно для типа **rational** такая проверка проста: желательно убедиться что присваивание не производится объектом над самим собой (*b = b*). В более сложных случаях нужны дополнительные проверки текущего состояния объекта. Имея это ввиду необходимо написать перегрузку для оператора присваивания как для аргументов L-значений, так и для R-значений.

б. Операция сложения двух рациональных чисел может быть хорошим кандидатом на перегрузку. Оператор сложения

также является двоичным оператором, поэтому его можно перегрузить по той же схеме, что и любой из операторов сравнения. Само сложение двух дробных чисел требует нахождения наименьшего общего множителя в случае отличающихся знаменателей. Затем обе дроби нужно привести к общему знаменателю и уже после этого сложить их числители.

Лабораторная работа 3
Тема: Перегрузка операторов

Цель работы: Для заданного варианта[20] добавить в класс все нужные операторы. Ниже показаны примеры прототипов, из которых складывается интерфейс абстрактного типа данных.

Условие для выполнения лабораторной работы:

```
class abstract_data_t {
public:
    // методы из предыдущей лабораторной работы здесь
    abstract_data_t &operator=(const abstract_data_t&);
    friend bool operator==(
        const abstract_data_t&, const abstract_data_t&);
    friend bool operator!=(
        const abstract_data_t&, const abstract_data_t&);
    friend bool operator<(
        const abstract_data_t&, const abstract_data_t&);
    friend bool operator>(
        const abstract_data_t&, const abstract_data_t&);
    friend bool operator<=(
        const abstract_data_t&, const abstract_data_t&);
    friend bool operator>=(
        const abstract_data_t&, const abstract_data_t&);
    int &operator[](size_t);
    friend std::ostream &operator<<(
        std::ostream&, const abstract_data_t&);
    friend std::istream &operator>>(
        std::istream&, abstract_data_t&);
    bool contains(int);
    size_t count(int);
private:
    // необходимые поля здесь
};
```

Таким образом необходимо реализовать следующие операции.

[20] Варианты см. в приложении 1.2.

Оператор присваивания **operator=** принимает один аргумент – ссылку на объект заданного класса. Оператор выполняет замену всех элементов коллекции объекта слева от оператора на копии элементов коллекции объекта справа от оператора. Данный оператор семантически схож с конструктором копирования, поэтому в некоторых ситуациях они могут быть взаимозаменяемы. Оператор присваивания возвращает ссылку на измененный объект. Это необходимо для того, чтобы данный оператор можно было вызывать на собственный результат.

Оператор сравнения **operator==** принимает два аргумента: ссылку на объект заданного класса слева от оператора и ссылку на объект заданного класса справа от оператора. Оператор возвращает булевое значение: “истинность”, если аргументы равны, и “ложь” в противном случае. Объекты считаются равными, если равны их коллекции. Коллекции равны, если они содержат равное количество элементов, а также – если значение каждого элемента одной коллекции равно элементу другой коллекции на том же месте. Данный оператор повторяет логику метода **is_equal** из первой лабораторной работы. Аналогично реализуется обратный оператор сравнения **operator!=**.

Оператор сравнения **operator<** принимает два аргумента: ссылку на объект заданного класса слева от оператора и ссылку на объект заданного класса справа от оператора. Оператор возвращает булевое значение: “истинность”, если аргумент слева от оператора строго меньше аргумента справа от оператора, и “ложь” в противном случае. Сравнение производится лексикографически. Аналогично реализуются операторы **operator<=**, **operator>**, **operator>=**.

Оператор ввода из потока данных **operator>>** принимает два аргумента: ссылку на поток данных слева от оператора и ссылку на объект заданного класса справа от оператора. Результатом вызова оператора должно быть копирование информации из потока данных слева в объект

справа от оператора. Оператор возвращает ссылку на измененный поток ввода. Это необходимо для того, чтобы данный оператор можно было вызывать на собственный результат.

Оператор вывода в поток данных **operator<<** принимает два аргумента: ссылку на поток данных слева от оператора и ссылку на объект заданного класса справа от оператора. Результатом вызова оператора должно быть копирование информации из объекта справа в поток данных слева от оператора. Оператор возвращает ссылку на измененный поток вывода. Это необходимо для того, чтобы данный оператор можно было вызывать на собственный результат.

Оператор доступа по индексу **operator[]** принимает единственный аргумент – беззнаковое целое число, обозначающее позицию элемента в массиве. возвращает ссылку на элемент коллекции, находящийся по заданному индексу. В отличие от метода **at** не проверяет индекс на несоответствие размеру коллекции. В случаях, когда **pos >= length()**, поведение метода не определено. Асимптотичность метода постоянная – ***O(1)***.

Метод **count** принимает один аргумент – целое число – и возвращает количество элементов коллекции, значение которых равно значению аргумента.

Метод **contains** принимает один аргумент – целое число – и возвращает булевое значение: “истинность”, если значение аргумента равно значению хотя бы одного элемента коллекции, и “ложь” в противном случае.

Для проверки в пример включены утверждения `assert`. Работа не будет выполнена полностью до тех пор, пока эти утверждения не будут проходить:

```
#include <cassert>
#include <iostream>
#include <sstream>
// остальные необходимые библиотеки здесь
// код для проверки правильности выполнения задания:
```

```
int main() {
    std::stringstream ss{"1 3 5 7 9"};
    abstract_data_t a(5);
    ss >> a;
    assert(5 == a.length());
    assert(1 == a[0]);
    assert(9 == a[4]);
    std::cout << a;
    abstract_data_t b{a};
    assert(a == b);
    assert(3 == b[1]);
    assert(7 == b[3]);
    b[4] = 0;
    assert(0 == b[4]);
    assert(!b.contains(9));
    assert(b < a);
    assert(a > b);
    std::cout << b;
    abstract_data_t c;
    assert(0 == c.length());
    c = b;
    assert(b == c);
    c[1] = c[2] = 7;
    assert(7 == c[1]);
    assert(7 == c[2]);
    assert(3 == c.count(7));
    std::cout << c;
}
```

Рекомендуется компилировать сделанную лабораторную работу с помощью приложения `g++` в файле с расширением `.cpp`, установив флаги `-Wall`, `-Wpedantic`, `-std=c++17`.

5. Итераторы

5.1. Теоретическая информация

Перегрузка функций часто используется для получения специализированных версий алгоритмов. Так, алгоритмы сортировки могут работать с множеством различных видов контейнеров, каждый со своей особой структурой. Наивная версия могла бы выглядеть так:

```
void sort(array &);
void sort(vector &);
void sort(list &);
void sort(btree &);
```

Перегрузив функцию **sort** пользователям предоставляется алгоритм, который может сортировать массивы, векторы, списки или двоичные деревья. Такой подход требует написать четыре версии реализации данной функции – для каждого типа контейнера. Со временем был разработан другой подход, который позволяет использовать одну версию реализации функции для всех типов.

Этот подход основан на поведении объектов, являющихся составной частью заданного контейнера. Таким объектам достаточно обладать следующими свойствами: предоставлять *доступ к значению*, хранящемуся в конкретном элементе контейнера; *переходить* к соседнему (следующему или предыдущему) элементу контейнера; *сравнивать* свое значение с значениями других таких же объектов.

В языке **C** этим требованиям удовлетворяют указатели массива. Их можно вычитать друг из друга, прибавлять к ним целые числа, сравнивать адреса, которые они хранят, разыменовывать их для получения доступа к значению по адресу. Тем не менее, для работы алгоритмов в языке **C++** “голые” указатели использовать не принято. Взамен классы-контейнеры предоставляют методы типа **begin** и **end**, которые могут выглядеть так:

```
int *begin() { return &data[0]; }
int *end() { return &data[size()]; }
```

Пример реализации методов **begin** и **end**, показанный выше, является “наивным” в том смысле, что он использует указатели для получения доступа к содержимому памяти контейнера (массив **data**). Таким образом можно получить методы, которые скрывают сам массив внутри контейнера, но позволяют указать на начало или конец контейнера. Их результат можно разыменовывать, инкрементировать или декрементировать для получения соседних элементов. В стандартной библиотеке шаблонов для этого используются *итераторы*[21], которые являются полноценными объектами, скрывающими указатели от прямого доступа для большинства операций контейнеров. Причем, задача итераторов не только и не столько в том, чтобы скрыть указатели, а в том, чтобы *привести интерфейс всех контейнеров к одной форме, не зависящей от реализации контейнеров*. Что это подразумевает?

Например, читатель уже должен быть знаком с тем, как реализуются односвязные списки. В отличие от массивов односвязный список не хранит все свои элементы в соседних блоках памяти, поэтому указатель на каждый последующий элемент нужно хранить вместо того, чтобы арифметически расчитывать его позицию, как это делается для массивов. То есть, для односвязного списка реализация методов **begin** и **end** через обычные указатели, ничего бы не дала, так как алгоритмы на контейнерах используют инкрементирование итераторов для перебора элементов контейнера. Это значит, что для такого списка недостаточно сделать методы **begin** и **end**, а нужно также перегрузить операторы инкремента, декремента, сложения,

[21] Итератор (**iterator**, англ.) – специальный объект, который указывает на отдельный элемент из списка элементов; итератор также может переходить от одного элемента к другому, используя специальные операторы; в **C++** указатели обладают всеми свойствами итератора и полностью аналогичны по своей функциональности итераторам случайного доступа, https://en.cppreference.com/w/cpp/iterator.

вычитания и сравнения итераторов – создать новый тип данных, который *ведет себя как указатель*, определить для него все вышеперечисленные операции и возвращать методами **begin** и **end** объекты этого типа, а не простые указатели. Благодаря этому становится возможным универсальная реализация алгоритма сортировки:

```
void sort(Iterator a, Iterator b);
```

Тип **Iterator** обозначает итератор произвольного контейнера. Таким образом в функцию сортировки можно передавать не весь контейнер целиком, а какую-то его часть, заключенную в отрезок, ограниченный итераторами a и b. При этом контейнер может быть любым: списком, массивом, деревом, графом и т.п. – единственное требование к таким контейнерам заключается в том, чтобы их итераторы предоставляли все необходимые алгоритму операции.

Эта упрощенная реализация позволяет переписать методы **begin** и **end** с учетом итераторов:

```
Iterator begin() { return Iterator(&data[0]); }
Iterator end() { return Iterator(&data[size()]); }
```

Нетрудно себе представить, как реализованные таким же образом итераторы для односвязного или двусвязного списков могли бы работать с похожим эффектом, скрывая от программы детали того, как итерация устроена для отдельных элементов списка, “выставляя вперед” только интерфейс, базирующийся на итераторах. Для односвязного списка интерфейс итератора может выглядеть следующим образом:

```
class Iterator {
public:
        Iterator(pointer);
        Iterator(const Iterator &);
        Iterator &operator++();
        Iterator operator++(int);
        bool operator==(const Iterator &);
        bool operator!=(const Iterator &other)
```

```
        { return !(*this == other); }
        reference operator*();
        pointer operator->();
// private-секция здесь
};
```

Помимо стандартных конструкторов в итераторе присутствует конструктор, инициализирующий объект адресом нужного элемента (значения в массиве, узла в списке и т.п.). Перегружаются операторы инкрементирования (префиксная и постфиксная формы), которые будут использоваться циклами для итерации по контейнеру. Аналогично перегружается пара операторов “равно/не равно”. Первый сравнивает адреса двух итераторов контейнера, а второй реализован с помощью первого, опираясь на существующую абстракцию, где это возможно.

Наконец, для итератора необходимо перегрузить операторы разыменования, так как в языке **C++** итераторы ведут себя как указатели. Разыменование должно возвращать значение по адресу, хранящемуся в итераторе. Соответственно оператор ***** возвращает ссылку на это значение, а оператор **->** возвращает сам указатель.

Лабораторная работа 4
Итераторы

Цель работы: Для заданного варианта[22] реализовать класс-итератор, который сделает тип данных совместимым с стандартными алгоритмами языка **C++**.

Условие для выполнения лабораторной работы:

```
class iterator {
public:
    // рекомендуемые прототипы функций:
    iterator(int *);
    iterator(const iterator &);
    iterator &operator++();
    iterator operator++(int);
    iterator &operator--();
    iterator operator--(int);
    int& operator*();
    int* operator->();
    bool operator==(const iterator &);
    bool operator!=(const iterator &);
private:
    int *ptr{nullptr};
};
```

Таким образом необходимо реализовать следующие операции.

Конструктор копирования имеет один параметр – объект заданного класса, создает идентичную копию аргумента.

Конструктор, принимающий указатель на значение из контейнера в качестве аргумента, инициализирует внутренний указатель переданным адресом.

Операторы инкрементирования **`operator++`** изменяют адрес внутреннего итератора на адрес элемента контейнера, идущего непосредственно за текущим элементом, на который итератор указывал до этого. Префиксный оператор возвращает

[22] Варианты см. в приложении 1.2.

ссылку на итератор. Постфиксный возвращает копию итератора с состоянием до изменения адреса.

Операторы декрементирования итератора **operator--** изменяют адрес внутреннего итератора на адрес элемента контейнера, идущего непосредственно перед текущим элементом, на который итератор указывал до этого. Префиксный возвращает ссылку на итератор. Постфиксный возвращает копию итератора с состоянием до изменения адреса.

Оператор разыменования итератора **operator*** возвращает ссылку на значение по адресу, который хранится в указателе итератора.

Оператор доступа по указателю на итератор **operator->** возвращает адрес, который хранится в указателе итератора.

Оператор сравнения итераторов **operator==** принимает один аргумент: ссылку на объект заданного класса справа от оператора. Оператор возвращает булевое значение: “истинность”, если аргументы равны, и “ложь” в противном случае. Объекты считаются равными, если равны их указатели.

Оператор сравнения итераторов **operator!=** реализуется, используя предыдущий оператор.

Для контейнера необходимо реализовать следующие операции:

```
class abstract_data_t {
public:
    // методы из предыдущей лабораторной работы здесь

    // рекомендуемые прототипы функций:
    abstract_data_t(const iterator &, const iterator &);
    auto begin();
    auto end();
    auto rbegin();
    auto rend();
    void assign(const iterator &, const iterator &);
private:
    // необходимые поля с данными здесь
};
```

Конструктор с двумя параметрами-итераторами на другой контейнер такого же типа должен создать контейнер такого же размера, содержащий те же значения, что и отрезок, заключенный между итераторами-аргументами.

Метод **`begin`** возвращает итератор на первый элемент коллекции. Метод **`end`** возвращает итератор на несуществующий элемент коллекции, адрес которого совпадает с байтом, идущим непосредственно за последним элементом коллекции, как показано на рисунке 1.

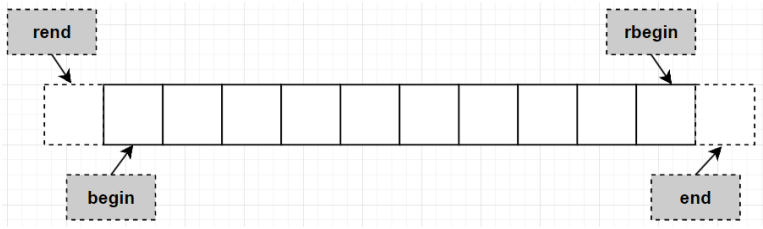


Рисунок 1, Расположение итераторов `begin`, `end`, `rbegin`, `rend` относительно элементов в контейнере, состоящем из 10 элементов.

Метод **`rbegin`** возвращает итератор на последний элемент коллекции. Метод **`rend`** возвращает итератор на несуществующий элемент коллекции, адрес которого совпадает с байтом, идущим непосредственно перед первым элементом коллекции.

Метод **`assign`** принимает два аргумента-итератора на другой контейнер такого же типа. Метод заменяет элементы коллекции на значения из отрезка, заключенного между итераторами-аргументами.

Функция **`is_equal`** из первой лабораторной работы конвертируется в статический метод класса, имеет два параметра – два объекта заданного класса, которые необходимо сравнить. Метод возвращает “истинность”, если аргументы равны, и “ложь” в противном случае. Объекты считаются равными, если равны их контейнеры. Контейнеры равны, если они содержат равное количество элементов, а также – если

значение каждого элемента одного контейнера равно элементу другого контейнера на том же месте. Для сравнения двух объектов рекомендуется использовать итераторы.

Для проверки в пример включены утверждения `assert`. Работа не будет выполнена полностью до тех пор, пока эти утверждения не будут проходить:

```
#include <algorithm>
#include <cassert>
#include <iostream>
// остальные необходимые библиотеки здесь
int main() {
    int array[] = {19, 47, 74, 91};
    abstract_data_t a(array, 4);
    for(auto it = a.begin(); it != a.end(); ++it)
        std::cout << *it << " ";
    abstract_data_t b(a.begin(), a.end());
    assert(a == b);
    for(auto &&it : b)
        std::cout << it << " ";
    abstract_data_t c(b.begin(), b.end());
    assert(std::equal(c.begin(), c.end(), b.begin()));
    for(auto it = c.rbegin(); it != c.rend(); ++it)
        std::cout << *it << " ";
    abstract_data_t d(c.rbegin(), c.rend());
    for(auto &&it : d)
        std::cout << it << " ";
}
```

Рекомендуется компилировать сделанную лабораторную работу с помощью приложения `g++` в файле с расширением `.cpp`, установив флаги `-Wall`, `-Wpedantic`, `-std=c++17`.

6. Наследование и полиморфизм

6.1. Теоретическая информация

6.1.1. Типы полиморфизма

Прежде чем обсуждать феномен полиморфизма следует рассмотреть его противоположность, с которой читатель уже хорошо знаком, но которой до сих пор не придавалось особого значения. Примеры реализации функций выше включали в себя только такие функции, которые работают с типами данных, известными заранее. Эти типы прописываются в списке параметров функции. Эта уникальность типов, которая устанавливается для аргументов и значений функций, обуславливает их *мономорфность*[23]. На этом контрасте легко понять, что такое полиморфность – это *свойство переменных и значений принимать более одного типа*. Такие примеры выше тоже были, но они не выделялись специально. Если присмотреться, например, к использованию конструкторов:

```
rational a(2, 3); rational b(a); rational c("3/15");
```

очевидно, что одна и та же функция (конструктор **rational**) которая принимает разные типы данных в качестве аргументов: то пару **int**, то объект **rational**, а то и строку **char***. Понятно, что это на самом деле достигается не полиморфными типами данных, а *перегрузкой* – наличием разных функций с одинаковым названием, но с точки зрения программы это тоже своего рода полиморфизм, а точнее один из примеров “ad hoc”[24] полиморфизма.

---

[23] От греческого “моно” – одна, “морфи” – форма; языки, основанные на идее функций и процедур, а также их операндов, которые имеют уникальный тип, принято называть мономорфными, так как каждое значение и каждую переменную можно интерпретировать единственным типом данных [Cardelli].

[24] Ad hoc – решение, специально придуманное под конкретную задачу, буквально “для этого” (лат.).

Другой пример “ad hoc” полиморфизма – *приведение типов* – читатель тоже видел, не подозревая, что это называют таким термином. Если в голову не приходит конкретный пример, следует внимательно посмотреть на следующую строку:

```
rational b(2.5, 3.8);
```

Как будет выглядеть конечный объект *b*? Очевидно, что оба аргумента являются значениями типа **double**, то есть не представлены целыми числами, как того требует интерфейс класса **rational**. Тем не менее, объект будет создан без ошибок, и если читатель предположил, что значением будет число $2/3$, он угадал. Так как при вызове подходящего конструктора вариант с дробными числами найден не будет, компилятор попробует выбрать ближайший из возможных[25], согласно правилам системы типов языка **C++**. Язык позволяет переводить числовые данные из целых в дробные и обратно, при условии, что часть данных будет утеряна, и программист это осознает. Благодаря этому компилятор знает, что возможна инициализация объекта **rational** дробными числами с помощью конструктора, который содержит целые числа в списке аргументов, так как дробные числа легко могут быть приведены к целым в момент инициализации аргументов.

Это неявное приведение типов является примером полиморфизма, так как конструктор может быть вызван типом данных, который изначально не предусматривался. Если в случае с перегрузкой функций вид полиморфизма выбирал сам программист, то при приведении типов это делается системой типов данных в языке. Тогда перед программистом встает вопрос: стоит такое поведение допускать или нет.

6.1.2. Наследование

Перед тем, как рассматривать другой тип полиморфизма на конкретных примерах, необходимо чуть более подробно

[25] Это поведение можно запретить с помощью ключевого слова **explicit** в объявлении конструктора.

поговорить о некоторых особенностях системы типов в **C++**. Эти особенности – *подтипы* и *шаблоны* – вводят в оборот новые инструменты, которые позволяют использовать полиморфные типы данных. А именно, полиморфные типы данных позволяют создавать *универсально полиморфные* функции – такие функции, которые могут работать с бесконечным количеством уникальных типов. Очевидно, что такого эффекта нельзя добиться перегрузкой или приведением типов, потому что физически невозможно создать бесконечно много перегруженных версий той или иной функции, так же физически невозможно описать бесконечно большой список правил приведения того или иного типа в другой. Здесь на помощь приходит *полиморфизм подтипов*.

Основной механизм для достижения такой полиморфности – *наследование* – в разных языках реализуется по-разному. Суть наследования состоит в расширении свойств и методов отдельно взятого типа данных за счет создания нового, “дочернего” типа данных, базирующегося на существующем типе. То есть, новый класс перенимает все поля другого класса без необходимости их копировать руками. Компилятор рассматривает новый тип как принадлежащий к иерархии базового типа, таким образом возникает ситуация, в которой любой объект базового типа можно легко заменить объектом нового подтипа без каких-либо ошибок. Это работает благодаря тому, что все поля базового типа в новом подтипе тоже присутствуют, соответственно те инструкции программы, которые обращаются к таким полям, найдут их и в новом подтипе гарантированно.

По этой причине иерархии наследования принято изображать как древовидную структуру, в которой базовый класс играет роль корня дерева. Например, на рисунке 2 ниже показана иерархия чисел, которая взята как фрагмент системы типов языка **Smalltalk**.

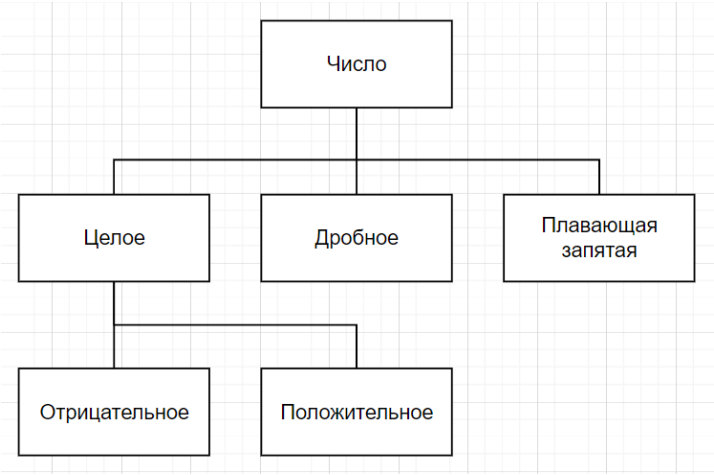


Рисунок 2, Фрагмент иерархии классов в языке **Smalltalk**.

Как говорилось чуть выше, основное предназначение классов, которые составляют систему типов, представленную на диаграмме – управлять поведением своих объектов. Тогда становится очевидным, что целое число и дробное число ведут себя примерно одинаково, потому что им присущи одни и те же операции (взятие по модулю, сложение, вычитание и т.д.) – следовательно, они оба принадлежат классу чисел. Отрицательное целое число ведет себя схожим образом с любым другим целым числом, а если продолжить аналогию – с любым другим числом, поэтому отрицательные целые числа являются подтипом целых чисел, которые в свою очередь являются подтипом чисел. При этом поведение подтипа становится чуть более специфичным по сравнению со своим базовым типом (“супер-типом”). Например, всем числам присущи операции сложения и вычитания, но только дробные числа имеют числитель и знаменатель и т.д.

Если рассматривать наследование как абстрактный механизм классификации, он выглядит понятно и местами элегантно. Реализация наследования – совсем другое дело. Например, в **Smalltalk** наследование реализуется через

инстанцирование объектов. Каждый класс в системе типов – это объект своего базового класса. Пользуясь иллюстрацией выше, “целое число” – это одновременно класс для “положительного числа” и объект класса “число”. Отсюда можно сделать вывод, что все классы – это объекты других классов, за исключением корня иерархии классов, потому что корень ничего не наследует и является чистым “абстрактным” классом. Благодаря этому “классы” можно изменять в ходе работы программы (дополнять новыми методами или изменять существующие), так как в конечном итоге они ведут себя как переменные.

В **C++** используется другой подход. Так как понятие “класса” тесно связано со структурами данных, делать подтипы в виде объектов своего базового класса не так удобно. Разработчики посчитали более рациональным обратный способ: сделать базовый класс объектом внутри своего дочернего класса. По аналогии с иллюстрацией выше эта иерархия выглядит, как показано на рисунке 3.

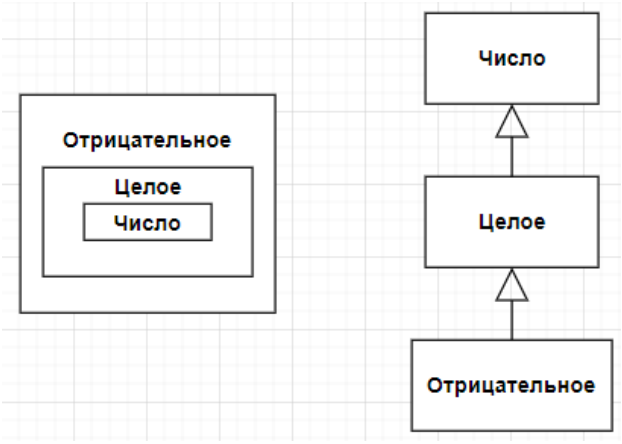


Рисунок 3, Наследование в языке **C++** через расширение (слева); в виде UML-диаграммы[26] (справа).

[26] *https://www.uml.org/*.

На такой диаграмме легче визуализировать принцип расширения одного класса другим, так как подкласс включает в себя объект своего базового класса целиком. В коде это выглядит следующим образом:

```
class number { … };
class integer : number { … };
class positive : integer { … };
```

Двоеточие после названия класса ожидает справа от себя имя базового класса. Это является сигналом для компилятора: при инициализации объекта типа **positive** будет инициализирован и объект типа **integer**, а следовательно – и объект **number**. Все они будут включены один в другой по принципу матрешки (этот прием также называется *композицией*).

За двоеточием может следовать указание *вида наследования*, который выбирается из трех возможных: *открытое*, *защищенное* или *закрытое* наследование:

```
class number { … };
class integer : public number { … };
class positive : public integer { … };
```

Тип наследования потенциально изменяет доступ к полям наследуемого класса. Открытый тип – `public` в примере выше – не имеет эффекта, а поэтому рекомендуется к использованию в большинстве случаев [Pomeranz]. Если тип наследования не указать явно, *по умолчанию устанавливается закрытое наследование*. Это означает, что все модификаторы доступа базового класса изменяются на `private` в дочернем классе, из чего следует, что *открытые поля базового класса будут недоступны объекту дочернего класса* (но будут доступны внутри самого дочернего класса).

Самый верхний тип (**number**) играет роль абстрактного (подробней об этом говорится в следующей главе), что не подразумевает создания объектов такого типа в своих

программах явно. Имея такую структуру наследования, фрагмент кода ниже должен быть “легальным”:

```
positive i = -1;
assert(i == 1);
```

Инициализация позитивного целого числа отрицательным числом должна либо конвертировать значение в положительное, либо принудительно останавливать работу программы. В примере выше предполагается, что программа должна продолжить работу. Это значит, что конструктор класса **positive** должен принимать любые целые числа и сохранять абсолютное значение их целой части.

Этого эффекта можно добиться как минимум двумя способами. Так как при наследовании дочерний класс при инициализации содержит в себе объект родительского класса, можно хранить значение для типа **positive** в его внутреннем объекте типа **integer**. Тогда задача конструкторов класса **positive** заключается в проверке передаваемых им значений и приведении их к нужной форме с последующим хранением в своем объекте типа **integer**, используя поля с данными этого объекта для хранения своих значений.

Второй способ предполагает игнорирование полей с данными родительского объекта и создание своих собственных атрибутов соответствующего типа (например, **unsigned int** для класса **positive**). Это подразумевает дублирование информации, так как наличие внутренних полей в родительских объектах никак не используется в дочернем классе. Чтобы визуализировать всю эту иерархию – следующий пример:

```
class number {
    // абстрактный класс без полей с данными
};
class integer {
public:
    integer(int a): x(a) {}
private:
```

```
    number n; // явное включение родительского класса для
доступа к его операциям
    int x{0};
};
class positive {
public:
    positive(int a): i(abs(a)) {}
private:
    integer i; // явное включение родительского класса для
доступа к его данным и операциям
};
```

Здесь вместо неявного включения родительских объектов через наследование используется явное – композиция. С точки зрения класса **positive** в данном случае принципиально ничего не меняется. Внимательным глазом можно заметить, что этот подход немного пересекается с тем, как наследование устроено в **Smalltalk**, классы тоже инициализируются как объекты (и это естественно, так как и **C++**, и **Smalltalk** разрабатывались с оглядкой на другой язык – **Simula-67**[27]).

6.2. Практические задачи для закрепления теории

а. В одной из предыдущих глав уже упоминался способ ограничения типов данных, которыми можно инициализировать объекты **rational**. Учитывая пример неявного ad hoc полиморфизма с приведением типов, который был показан выше, можно было бы изменить дизайн класса **rational** и сделать невозможным конструирование объектов с помощью дробных чисел.

б. В примере с композицией стоит обратить внимание на использование конструкторов. Так, при создании объекта типа **integer** вызовется не только соответствующий конструктор класса, но и конструкторы всех атрибутов этого класса, которые не представлены примитивными типами данных. В этом случае был бы вызван синтезированный конструктор по умолчанию класса **number**. По такой же цепочке происходят вызовы

[27] *http://www.simula67.info/.*

конструкторов при инициализации объекта **positive**. Если присмотреться, можно заметить, что в классе **positive** есть явный вызов конструктора **integer(int)**. Похожим образом эти конструкторы будут вызваны и в случае с наследованием.

Возвращаясь к строке **positive** `i = -1`, возможно, не каждому из прочитавших эту часть было понятно по какой цепи выполнялся код, который привел к присваиванию значения “1” переменной, которая инициализировалась числом -1. Чтобы явно увидеть связь между всеми операциями, вызванными инструкцией **positive** `i = -1`, следует дополнить все определения классов **number**, **integer** и **positive**. Затем, используя *дебаггер* или текстовую *трассировку*[28], подтвердить ход и конкретное место исполнения инструкций в программе. Для этого следует использовать конструкторы и их переопределение. Сделать это читателю предлагается с использованием наследования, чтобы убедиться, что порядок инициализации будет очень схож с тем, как это происходит для случая с композицией.

Напоследок, стоит отметить, что сейчас может быть непонятно, зачем нужно наследование, которое скрывает определенные операции, если практически такого же эффекта можно добиться применением композиции. Ответ на этот вопрос будет дан в одной из следующих глав.

[28] Code tracing – техника визуального подтверждения значения одной или нескольких переменных на каждом этапе работы программы, как правило выполняемая с помощью вывода значений в одно из “окон” программы, либо – вручную на бумаге.

7. Полиморфизм подтипов

7.1. Теоретическая информация

Любые операции, заданные уровнем выше в иерархии наследования, также будут доступны через объекты подтипов. Например, как было отмечено чуть ранее, операция сложения должна быть доступна для любых чисел, дробных и целых. При этом сама процедура проходит по-разному для целых и для дробных чисел. Перед разработчиком две задачи: реализовать операцию сложения для соответствующих типов чисел и убедиться, что *каждый новый тип, добавляемый в иерархию чисел, также будет реализовывать эту операцию обязательно*.

Для чего это нужно на практике? Чтобы понять причину этих требований можно рассмотреть следующий пример кода:

```
'a add('a x, 'a y) {
    return x + y;
}
```

Здесь запись **'a**[29] является переменной, которая представляет *неопределенный тип данных*. То есть, такой тип данных, который неизвестен компилятору до тех пор, пока программа не будет запущена, и в качестве аргументов для этой функции не будут переданы конкретные значения. На основании этих значений программа определит какие типы были переданы, и после этого попытается произвести операцию сложения между объектами *x* и *y*.

К примеру, если оба аргумента будут целыми числами, то в качестве операции сложения будет выбрана реализация для целых чисел, и ее результатом тоже будет целое число. Аналогично это сработает для двух дробных чисел. Но вот интересный вопрос: какая операция будет выбрана для случая, в котором аргумент *x* – целое число, а аргумент *y* – дробное?

Чуть более сложный пример ниже иллюстрирует еще одно ограничение:

[29] Читается как “альфа”.

```
'a add('a x, 'a y, 'b z) {
      if(z) return x + y;
      else return 0;
}
```

Так как последняя строка в функции возвращает целое число (0), то результат сложения аргументов *x* и *y* обязан вернуть или целое число, или значение типа данных, приводимого к целому числу. В противном случае возвращаемое значение функции было бы двояким, чего компилятор допустить не может.

Интересней дело обстоит с аргументом *z*. Он участвует в условной операции **if**, поэтому изначально можно предположить, что его тип данных – **bool**. Но из данного текста уже известно, что условные операции принимают значения любых типов, возвращая “false” или “true” в зависимости от того, равны они нулю или нет, поэтому аргумент *z* не обязательно имеет тип **bool**. Тем не менее, свободно можно переписать функцию так и не ошибиться:

```
int add(int x, int y, bool z) {
      if(z) return x + y;
      else return 0;
}
```

Теперь можно внести в нее еще одно изменение:

```
'a add('a x, 'a y, 'a z) {
      if(z) return x + z;
      else return x + y;
}
```

Здесь аргумент *z* участвует и в условной операции, и в операции сложения. Это не делает тип двояким (он может быть либо числовым, либо булевым значением, но не обоими сразу), при этом все равно оставляет его неопределенным.

Важно подчеркнуть, что типы данных функции не стали *динамическими* в классическом смысле этого слова. Динамические системы типов в языках программирования как

раз допускают некоторую двоякость при обращении с переменными (например, одной и той же переменной можно присваивать значения разных типов в разных инструкциях при условии, что последующие операции учитывают этот факт). **C++** является языком со *статической* системой типов, а это значит, что компилятор должен быть уверен в правильности написанного кода для всех участвующих в том или ином выражении значений.

Примеры выше возможны и в **C++** благодаря системе *выявления типов*[30] переменных. Выявление типов позволяет писать программы, не задавая явно типы данных для переменных. Хорошо спроектированная система типов в языке позволяет не отвлекаться на это. На программиста накладывается ответственность за правильность алгоритмов в рамках такой системы, так как необходимо заранее предвидеть возможные варианты трактовки системой типов того или иного выражения. Это может немного ограничить количество вариантов решения той или иной задачи. С другой стороны, использование системы неявных типов вовсе не обязательно. С помощью такой системы часто реализуется *универсальный полиморфизм* – способ реализации функций, поведение которых не привязано к типам данных переданных аргументов. В **C++** нет полноценной динамической системы выявления типов аргументов, поэтому в данном языке широко используется *полиморфизм подтипов*. Для этого используется *наследование* (а для статического полиморфизма – *шаблоны типов*).

С помощью наследования полиморфизм реализуется через отношение дочерних типов данных к базовому типу. Базовый тип данных играет роль полиморфного типа, то есть, такого типа, который может принимать разные формы в зависимости от типов переданных аргументов, аналогично

[30] Type inference – общее название семейства алгоритмов, которые частично или полностью определяют типы всех переменных на основании уже определенных ранее типов [Clarkson].

переменным типам **‘a** или **‘b**. Выглядеть это может следующим образом:

```
class Shape {
public:
    virtual double area() = 0;
};
class Circle : public Shape {
public:
    Circle(double x): r(x) {}
    double area() override { return 3.14159265359 * r; }
private:
    double r{.0};
};
class Square : public Shape {
public:
    Square(double x) : length(x) {}
    double area() override { return length * length; }
private:
    double length{.0};
};
```

Первое, что может вызвать вопросы – ключевое слово **virtual**. Для методов это слово помечает их как динамически или виртуально вызываемые[31]. Виртуальный вызов функции означает, что функция может быть вызвана для переменной любого типа из иерархии какого-то базового типа [Chu]. Другими словами, в этих случаях компилятору не обязательно знать как работает метод, он все равно позволит его вызвать, используя указатель на базовый класс, если в дочернем классе есть версия определения такого метода. Для примера выше это означает, что в программе можно будет создать переменную типа **Shape*** или **Shape&**[32], инициализировать ее объектом

[31] https://en.cppreference.com/w/cpp/language/virtual.

[32] Это важный момент, потому что он имеет серьезные последствия; наследование позволяет обращаться к данным базового класса через дочерний класс, что означает возможность приводить объекты дочернего типа к базовому; из этого следует что можно изначально объявить переменную как базовый тип, а присвоить ей значение дочернего типа, но

дочернего класса, и операция сложения будет работать так, как описано в дочернем классе.

Другой момент – присваивание нуля прототипу функции. Это помечает функцию как чисто виртуальную, что делает сам класс *абстрактным*[33]. Если вспомнить пример иерархии классов в языке **Smalltalk**, упомянутый выше (см. рисунок 2), там говорилось о том, что корень всего дерева типов является абстрактным классом, потому что он не инициализируется как объект какого-то родительского класса. По этой же схеме в других языках программирования абстрактные классы помещают в корень иерархии для того, чтобы они служили *описанием всех методов, доступных в каждом из объектов иерархии* – то есть, описанием *интерфейса*. В некоторых языках такие классы выделяют в отдельную категорию “интерфейсов” [Whitney(b)].

Наконец, ключевое слово **override** после списка параметров функции помечает функцию как виртуальную, *заменяющую определение такой же функции в базовом классе*. Таким образом, любой дочерний класс может быть использован для инициализации ссылки на его базовый класс, так как анонимный объект базового типа в ней тоже содержится. После этого, если функция базового класса будет вызвана посредством такой ссылки, компилятор будет знать, что ее определение нужно искать в дочернем классе, который использовался для инициализации объекта. В коде это могло бы выглядеть так:

```
void draw(Shape &fig, int x) {
       int density = x * fig.area();
       // вывод пикселей на экран
       return;
}
```

только если переменная является ссылкой или указателем, потому что компилятору заранее должен быть известен размер переменной в байтах, и только для указателей и ссылок размер не зависит от типа значения.

[33] Абстрактный базовый класс (**ABC**, англ.) – класс или структура, которая не может быть использована для создания переменных.

Следует обратить внимание на то, что первый аргумент представлен ссылкой на базовый тип **Shape**: то, какая версия метода **area** вызывается, зависит от вызова функции **draw**, то есть окончательно известно только после запуска программы:

```
Circle circ(15.5); draw(circ, 55);
```

Теперь можно подвести промежуточные итоги. Чтобы понять принципиальную разницу между двумя основными видами полиморфизма нужно иметь ввиду, что *универсальный* полиморфизм позволяет использовать один и тот же код для аргументов любого типа, тогда как *ad hoc* полиморфизм может использовать разный код для аргументов разных типов. Какой из видов полиморфизма использовать, как правило, диктует характер поставленной перед программистом задачи. Например, если перед разработчиком стоит задача определить длину произвольного массива, имеет ли для него значение, каким типом данных представлен каждый элемент массива? Ответив на этот вопрос можно определить то, каким видом полиморфизма можно было бы воспользоваться для решения.

7.2. Практические задачи для закрепления теории

а. Стоит вернуться к проблеме сложения двух полиморфных типов. Используя модель, представленную в предыдущей главе, можно определить основные требования к классу “число”. Первое, базовый класс не может быть объектом, поэтому нужно убедиться, что компилятор не разрешает создавать переменные такого типа. Второе, все дочерние классы должны включать в себя определенные операции (которые составляют *интерфейс* класса “число”). Третье, полиморфные функции и методы, работающие с числами, должны принимать любое число, дробное или целое. Учитывая эти требования можно написать первую версию класса “число”:

```
class number {
    // чистый абстрактный класс без полей с данными
public:
    virtual number &operator+(const number &other) = 0;
```

```
};
```

В **C++** абстрактный класс, реализованный через структуру данных, требует включения в него чистой виртуальной функции. Такая функция обязательно должна возвращать ссылку или указатель, потому что объект абстрактного класса возвращать будет невозможно, и возвращаемое значение будет к базовому типу приведено:

```
class integer : public number {
public:
    integer operator+(const number &other) override;
    // поля с данными и конструкторы
};
```

Целые числа можно представить абстрактным классом, так как инициализировать целое число без информации о знаке смысла не имеет. Но в данной упрощенной модели класса отрицательных чисел нет, поэтому перегруженный оператор должен быть отмечен ключевым словом **override**. На основании этого класса создается класс положительных целых чисел:

```
class positive : public integer {
public:
    positive &operator+(const number &other) override;
    // поля с данными и конструкторы
};
```

Если перед программой будет стоять выбор: какую версию метода **operator+** вызвать – он будет зависеть от типа данных объекта. Для объекта типа **positive** будет вызван оператор сложения, в нем же и переопределенный:

```
integer a = -5; positive b = 5;
auto c = a + b; auto d = b + a;
```

Например, стоит подумать о том, какого типа будут переменные *c* и *d* для фрагмента выше. Как можно видеть из этого сценария, работа с интерфейсами в **C++** часто сопряжена с решением нестандартных задач, которые могут вызвать

серьезные затруднения. Это частично связано с тем, что идея интерфейсов пришла в этот язык из других языков программирования, где с ее помощью решали проблемы, связанные с условностями реализации наследования в этих языках. В **C++** для этих целей используется *множественное наследование*. О нем речь пойдет в последней главе.

Лабораторная работа 5
Тема: Наследование

Цель работы: Для заданного варианта[34] создать иерархию наследования, где класс **abstract_data_t** выступает в роли интерфейса [Radford] (то есть, является *абстрактным базовым классом* для другого класса с реализацией). В коде ниже уже присутствует чисто виртуальный метод (деструктор класса **abstract_data_t**), это нужно для того, чтобы класс компилятором определялся как чисто абстрактный[35].

Условие для выполнения лабораторной работы:

```
class abstract_data_t {
public:
    // методы из предыдущей лабораторной работы здесь
    virtual ~abstract_data_t() = 0;
    virtual int &front() = 0;
    virtual int &back() = 0;
    virtual void push(int) = 0;
    virtual void pop() = 0;
    virtual void extend(const abstract_data_t&) = 0;
};
inline abstract_data_t::~abstract_data_t() {}
class название_вашего_варианта_здесь : public
abstract_data_t {
public:
    // конструкторы, деструктор, переопределение всех
нужных методов здесь
    // например:
    int &front() override;
    int &back() override;
    void push(int) override;
    void pop() override;
```

[34] Варианты см. в приложении 1.2.

[35] Правила языка **C++** требуют, чтобы в абстрактном классе был хотя бы один чисто виртуальный метод (отмеченный нулём справа от объявления метода). Статические и дружественные методы не считаются частью самого класса, поэтому не могут быть виртуальными, соответственно на роль единственного виртуального метода часто выбирается деструктор.

```
    void extend(const abstract_data_t&) override;
private:
    // необходимые поля здесь
};
```

Таким образом необходимо реализовать следующие операции.

Метод **front** возвращает значение первого элемента коллекции (не для всех вариантов[36]).

Метод **back** возвращает значение последнего элемента коллекции (не для всех вариантов).

Метод **push** принимает один аргумент – целое число – и добавляет значение аргумента в качестве нового элемента в начало (или конец) заданной коллекции (не для всех вариантов).

Метод **pop** удаляет первый (или последний) элемент из заданной коллекции (не для всех вариантов).

Метод **extend** с одним параметром (ссылкой на другую коллекцию того же типа, что и заданная) добавляет значения аргумента в конец заданной коллекции в том же порядке, в каком они встречаются в коллекции-аргументе.

Для проверки в пример включены утверждения `assert`. Работа не будет выполнена полностью до тех пор, пока эти утверждения не будут проходить:

```
#include <cassert>
#include <iostream>
// остальные необходимые библиотеки здесь
int main() {
    const int a[4] = {2, 3, 5, 7};
    abstract_data_t *v = new
название_вашего_варианта_здесь(a, 4);
    assert(!v->empty());
```

[36] В зависимости от варианта метод **front** или **back** может быть неуместным, так как суть контейнера не предполагает доступ к первому или последнему элементу (например, для “стека” или “очереди”). В таком случае следует заменить один или оба метода на соответствующие методы для своего варианта.

```
    v->push(11);
    assert(11 == v->back());
    assert(11 == (*v)[v->length() - 1]);
    const int b[3] = {13, 17, 19};
    v->extend(название_вашего_варианта_здесь(b, 3));
    assert(8 == v->length());
    abstract_data_t *w = new
название_вашего_варианта_здесь();
    assert(w->empty());
    *w = *v;
    assert(*w == *v);
    (*w)[0] = 0;
    assert(0 == w->front());
    w->pop();
    assert(17 == w->back());
    assert(7 == w->length());
}
```

Рекомендуется компилировать сделанную лабораторную работу с помощью приложения `g++` в файле с расширением `.cpp`, установив флаги `-Wall`, `-Wpedantic`, `-std=c++17`.

8. Шаблоны типов

8.1. Теоретическая информация

В предыдущей главе был задан вопрос: как найти длину массива, не зная заранее какого типа будут его отдельные элементы? Если рассматривать возможные варианты реализации этого в **C++**, ответа на вопрос будет как минимум два. Но один из них очень специфичен. Предполагая, что длина массива, в первую очередь, нужна для поэлементного поиска в массиве с помощью цикла (чтобы знать, где массив заканчивается), есть встроенный в язык способ:

```
const int arr[15]{0};
for(auto &i : arr) { ... }
```

Такая конструкция цикла называется **for-each**, и особенность ее в том, что она сама определяет длину переданного контейнера, если он отвечает определенным условиям. В примере выше массив был объявлен как статический, поэтому его размер известен компилятору, соответственно цикл может определить, где массив заканчивается, исходя из заданного размера с помощью арифметики указателей. Любопытно, что это не окончательный код, который будет скомпилирован:

```
{
     const int (&__range1)[15] = arr;
     const int * __begin1 = __range1;
     const int * __end1 = __range1 + 15L;
     for(; __begin1 != __end1; ++__begin1) {
         const int & i = *__begin1;
         ...
     }
}
```

Фрагмент кода выше – сгенерированная препроцессором реализация цикла для заданного массива. Как можно видеть, действительный цикл использует принцип итерации для прохода

по массиву, вычисляя конец массива (`__end1`) арифметикой указателей.

Особое внимание стоит обратить на ключевое слово **`auto`**, которого в сгенерированном коде нет. Это слово автоматически подменяется на тип каждого отдельного выражения. Оно является частью реализации *выявления типов* в **C++**.

Чтобы четко себе представить, как работает выявление типов в данном примере, следует разобраться с “шаблонами типов данных”, которые выполняют роль переменных типов[37]. Рассмотрим объявление произвольной функции:

```
template <typename T> size_t length(const T(&a)[15]);
```

Здесь встречается три новых синтаксических элемента. Ключевое слово **`template`** обозначает начало определения шаблона. В самом общем смысле шаблоны являются отдельными сущностями и делятся на несколько категорий: шаблоны классов, шаблоны функций, шаблоны переменных и т.п. Препроцессор генерирует код для шаблона тогда, когда шаблон инициализируется. В примере выше создается шаблон функции **`length`**, который имеет свой список аргументов в угловых скобках. Параметр у этого шаблона всего один – переменная `T`, которая принимает типы данных в качестве аргумента, что подчеркивается ключевым словом **`typename`**. Эта переменная затем используется в списке аргументов самой функции для обозначения того, что аргумент функции будет такого типа, который будет передан в качестве аргумента шаблона. Шаблоны также позволяют передавать в качестве аргументов константные значения, например, длину массива:

```
template <typename T, size_t N>
size_t length(const T(&a)[N]);
```

[37] https://en.cppreference.com/w/cpp/language/auto.

Здесь второй параметр в списке аргументов шаблона – имя *N*, которое должно быть целым беззнаковым числом – автоматически определенный размер массива.

Для массива **int** *arr*[15]{0} вызов выглядел бы так:

```
size_t l = length<int>(arr);
```

или так:

```
size_t l = length(arr);
```

Соответственно, в выводе препроцессора появится сгенерированный им код:

```
template<> size_t length<int, 15>(const int (&a)[15])
{ ... }
```

Этот код генерируется препроцессором только в случае вызова шаблонной функции, так как только вызов с конкретными аргументами позволяет препроцессору определить типы данных, которыми надо инициализировать шаблонные параметры. Такие версии шаблонных функций называются *специализациями*. Более того, для массива **double** *arr*[15]{.0} появится еще одна специализация:

```
template<>
size_t length<double, 15>(const double (&a)[15])
{ ... }
```

Можно заметить, что этот процесс инстанцирования шаблонных функций похож на перегрузку функций: при обнаружении вызова функции препроцессор генерирует перегруженную версию функции с заданным телом, но с заменой параметров-типов данных на конкретные типы. Другой очевидный вывод из этого – код программы заметно вырастает в объеме с каждым новым вызовом такой функции.

Синтаксис стал заметно сложнее для восприятия. Но в определенных ситуациях оно того стоит. Во-первых, это позволяет создавать полиморфные функции без использования наследования. Во-вторых это позволяет изменять тип аргумента

в будущем без необходимости изменять тело функции или все ее вызовы, которые уже есть в программе.

Шаблоны классов устроены по похожему принципу:

```
template <typename T> struct A { T m; };
A<int> x;
// инициализация генерирует код:
template<> struct A<int> {
  int m;
  inline A() noexcept = default;
};
```

Инстанцирование шаблона класса генерирует его специализированную версию под конкретный тип, переданный как шаблонный аргумент.

Стоит рассмотреть применение шаблонов на примере одного из самых востребованных классов в **C++**. В пространстве имен **std** содержится список различных ключевых слов и типов данных, которые определяются в различных стандартных библиотеках, поставляемых вместе с компилятором. Одна из таких библиотек – **vector** – представляет собой реализацию динамического массива, как альтернативы массивам из языка **C**, которые создаются на стеке и не изменяются в размерах в ходе работы программы. В отличие от них вектор может “расти” или “уменьшаться” в зависимости от контекста:

```
std::vector<int> a;
a.resize(10);
```

Если посмотреть на то, как класс **vector** объявлен в заголовочном файле своей библиотеки, некоторые элементы могут вызвать особый интерес:

```
template <typename _Tp,
          typename _Alloc = std::allocator<_Tp>>
class vector : protected _Vector_base<_Tp, _Alloc> { … };
```

Как и ожидалось, вектор объявлен как шаблон, причем список аргументов шаблона включает в себя другой шаблон –

необязательный параметр *_Alloc*, который определяет способ выделения памяти для нужд вектора. Вектор не является базовым классом, а наследует часть своих свойств от другого шаблона класса **_Vector_base**. Исходя из этого описания можно попробовать написать наивную реализацию вектора самостоятельно:

```
template <typename T> class vector {
public:
    vector() {}
    template <size_t N>
    vector(const T(&list)[N]) :
        length((N < 100) ? N : 100)
    {
        for(size_t i = 0; i < length; ++i) {
            arr[i] = list[i];
        }
    }
    vector(size_t N) : length((N < 100) ? N : 100) {}
    size_t size() { return length; }
    T *begin() { return arr; }
    T *end() { return arr + length; }
private:
    T arr[100]{0}; size_t length{0};
};
```

Такой вариант реализации включает методы **begin**, **end**, **size** для использования в циклах. В качестве основного механизма хранения данных вектора выступает статический массив на 100 значений заданного типа. В качестве итераторов используются указатели на массив. Конструкторы проверяют переданные аргументы для того, чтобы убедиться, что их значения не позволят массиву стать размером больше 100 элементов. Тогда становится возможным следующий фрагмент:

```
vector<int> my_vec({10, 20, 30});
for(auto i : my_vec) std::cout << i;
```

Благодаря использованию шаблонов созданный таким образом вектор может принимать любой тип данных в качестве основы, и это не повлияет на его работоспособность.

Лабораторная работа 6
Тема: Шаблоны типов

Цель работы: Для заданного варианта[38] создать шаблонный тип данных.

Условие для выполнения лабораторной работы:

```
template <typename T> class abstract_data_t {
public:
    // методы из предыдущей лабораторной работы здесь
    virtual iterator<T> find(const T&) = 0;
    virtual iterator<T> insert(iterator<T>, const T&) = 0;
    virtual iterator<T> erase(iterator<T>) = 0;
};
template <typename T> inline
abstract_data_t<T>::~abstract_data_t() {}
template <typename T, typename Iter = iterator<T>>
class название_вашего_варианта_здесь : public
abstract_data_t<T> {
private:
    // необходимые поля здесь
public:
    // конструкторы, деструктор, переопределение всех
нужных методов здесь, например:
    Iter find(const T&) override;
    Iter insert(Iter, const T&) override;
    Iter erase(Iter) override;
};
```

Таким образом необходимо реализовать следующие операции.

Метод **`find`** принимает аргумент заданного в шаблоне типа, соответствующий типу данных хранящихся в контейнере элементов, и возвращает итератор на элемент контейнера, значение которого равно значению аргумента.

Метод **`insert`** имеет два параметра: итератор, указывающий на существующий в коллекции элемент, и ссылку на значение заданного для коллекции типа. Метод возвращает

[38] Варианты см. в приложении 1.2.

итератор на новый элемент коллекции. Метод сохраняет элементы, которые уже существуют в коллекции на момент вызова, сдвигая их на один вправо относительно значения первого аргумента. Если новый размер превышает выделенный до сих пор под коллекцию объем памяти, метод вызывает выделение нового блока памяти. При этом все ссылки и итераторы, существующие в программе, которые указывают на элементы, идущие за значением первого аргумента, становятся недействительными Асимптотичность метода линейная относительно длины оставшейся за индексом части коллекции – ***O****(n)*.

Метод **`erase`** принимает один аргумент – итератор, обозначающий желаемую позицию элемента на удаление. Метод возвращает итератор на элемент, следующий за удаленным. Метод сохраняет элементы, которые уже существуют в коллекции на момент вызова, сдвигая их на один влево относительно *`pos`*. При этом все ссылки и итераторы, существующие в программе, которые указывают на элементы, идущие за *`pos`*, становятся недействительными Асимптотичность метода линейная относительно длины оставшейся за индексом части коллекции – ***O****(n)*.

Для проверки в пример включены утверждения `assert`. Работа не будет выполнена полностью до тех пор, пока эти утверждения не будут проходить:

```
#include <cassert>
#include <iostream>
#include <string>
// остальные необходимые библиотеки здесь
int main() {
    название_вашего_варианта_здесь<int> a({2, 3, 5, 7});
    a.push(11);
    assert(11 == a[a.length() - 1]);
    a.insert(a.begin(), 1);
    assert(1 == a[0]);
    a.extend(
        название_вашего_варианта_здесь<int>(
```

```
            {13, 17, 19}));
    a.erase(a.rbegin());
    assert(a.find(19) == a.end());
    assert(8 == a.length());
    название_вашего_варианта_здесь<char> b = "abra";
    assert(0 == b.back());
    b.erase(b.rbegin());
    b.push('c');
    assert('c' == *b.rbegin());
    b.insert(b.end(), 'a');
    assert('a' == *b.rbegin());
    b.extend(
        название_вашего_варианта_здесь<char>("dabra"));
    assert(b.rbegin() == b.find(0));
    b[0] = 'A';
    assert('A' == b.front());
    название_вашего_варианта_здесь<std::string> c(
        {"Hello", "world"});
    c[0].append(",");
    c.insert(c.find("world"), " ");
    c[c.length() - 1].append("!");
}
```

Рекомендуется компилировать сделанную лабораторную работу с помощью приложения `g++` в файле с расширением `.cpp`, установив флаги `-Wall`, `-Wpedantic`, `-std=c++17`.

9. Множественное наследование

9.1. Теоретическая информация

Напоследок можно было бы подумать о том, как сделать такой тип данных, который представляет рациональные числа во всей их полноте; например, этот тип самостоятельно определяет, показывать только целую часть или всю дробь, в зависимости от значений числителя и знаменателя.

Проверенный временем способ, использующийся для достижения такого уровня функциональности – “объединение”, тип данных **union**. Следует напомнить, что этот тип является специальным видом структуры, которая может содержать несколько переменных, как любая другая структура, но памяти потребляет ровно столько, сколько нужно для хранения самой “большой” из объявленных переменных. Например, для упрощенной формы типа **rational**:

```
class rational {
public:
    int p{1}; int q{1};
};
union r_int_variant { int a{0}; rational r; };
```

Здесь объект структуры **union r_int_variant** будет занимать в памяти не 12 байтов, как может сначала показаться (по 4 байта для каждого поля с типом **int**), а 8 – ровно столько, сколько нужно для хранения поля *r*. Благодаря этому для инициализации поля *a* будет использоваться тот же адрес, что и для инициализации поля *r*. Очевидно, что в таких условиях эти поля не могут существовать в памяти одновременно, изменение одного будет стирать значение другого.

Это свойство объединений можно использовать для цели, обозначенной выше. Так как одновременно в объединении может существовать только целое или дробное представление, проблема заключается в правильной инициализации и контролировании операции присваивания. Для этого нужно

следить за тем, какое из полей последний раз было инициализировано, благодаря чему можно определить к какому из полей нужно обращаться в тот или иной момент:

```
class r_int {
public:
    int _type; r_int_variant data;
};
#define r_int_cons(c, val) if(                \
    ((sizeof(val) == sizeof(int))             \
         ? (c._type = ((c.data.a = val), 2))  \
         : (c._type = ((c.data.r = val), 1))))
```

В примере выше код написан ближе к тому, как он бы выглядел в языке **C**. Роль непосредственного типа играет специальный класс, который контролирует доступ к объединению с данными. Специальный “флаг” – переменная *_type* – следит за тем, какое из полей объединения сейчас инициализировано. Для инициализации используется “макро” – именованная инструкция, которая автоматически выставляет “флаг” согласно размеру переданного аргумента:

```
r_int u; r_int_cons(u, 1);
```

Очень быстро становится очевидным главный минус такого подхода: следить за изменением значения “флага” нужно очень внимательно, иначе работа программы может нарушиться. Проверять значение “флага” придется при каждой операции над объектами такого класса. Не говоря уже о том, что такой метод контроля за значениями “флага” не будет работать, если размеры полей совпадают.

Все эти операции можно включить в новый класс в виде методов, скрыть “флаг” и данные от прямого внешнего воздействия. Создатели **C++** именно так и поступили, добавив в язык библиотеку **variant**[39]:

```
std::variant<int, rational> v(1);
std::cout << std::get<int>(v) << "\n";
```

[39] https://en.cppreference.com/w/cpp/utility/variant.

```
std::variant<int, rational> w(rational(4, 5));
std::cout << std::get<rational>(w) << "\n";
```

Класс `std::variant` использует шаблоны типов для того, чтобы правильно инициализировать память. Он следит за тем, какой из типов в каждый конкретный момент доступен для переменной. Чтобы получить значение, которое хранится в переменной варианта, необходимо использовать стандартную функцию `std::get`, которую необходимо вызвать с соответствующим типом в качестве шаблонного аргумента. Функция попытается привести данные в "варианте" к указанному типу. Таким образом, если тип указан правильно, приведение сработает без ошибок, и результат функции будет содержать нужное значение. Если же тип будет указан для того поля объекта, которое в данный момент не активно, это вызовет исключение[40] в программе.

В отличие от способа с объединением, данный метод позволяет работать только с абстракциями. Более того, исключения можно обработать и предотвратить ошибки программы даже в случае неправильно указанного типа. Тем не менее, в целом данный способ не намного лучше предыдущего, так как программисту все равно придется следить за тем, что он правильно указывает тип и обрабатывает исключения. В случае с объединением точно так же пришлось бы следить за тем, к какому полю класса он обращается, и какое значение "флага" действительно в тот момент.

Данную проблему можно решить как минимум еще одним способом – через *множественное наследование*. Работает оно именно так, как звучит: у конкретного класса может быть два и более родительских класса. Это значит, что при инициализации такого объекта в нем будет содержаться соответствующее количество анонимных объектов наследуемых

[40] Исключение (**exception**, англ.) – лингвистический механизм, который служит для взаимодействия процедур внутри программы, в первую очередь, при возникновении ошибок в логике программы.

классов со своими полями, а значит посредством одного дочернего объекта будут доступны все их операции.

Исходя из этого можно сделать один важный вывод. Если родительские объекты являются родственными, со множеством пересекающихся методов, их структура в дочернем объекте может стать очень запутанной. Поэтому множественное наследование как инструмент в основном используется в случаях, когда имеется *несколько никак друг с другом не связанных классов*[41], но при этом их интерфейсы необходимо объединить в одном типе данных. Похожего эффекта можно добиться либо через абстрактные классы-*интерфейсы* (так делается в языках программирования, где множественное наследование не поддерживается), либо через *композицию* объектов (явное включение требуемых классов в качестве полей нового класса).

Пример, описываемый в этой главе, к таким случаям не относится. Наоборот, цель заключается в том, чтобы объединить в рамках одного повернутого к пользователю класса (“рациональные числа”) возможности двух других родственных типов (“целые числа” и “дробные числа”). Это вызовет определенные трудности, но можно представить простой пример реализации такого отношения между типами данных:

```
class r_int : public rational, public integer {
public:
    r_int() = delete;
    r_int(int x) : rational(x, 1), integer(x) {}
    r_int(int x, int y) : rational(x, y), integer(x) {}
    friend std::ostream &operator<<(
        std::ostream &os, const r_int &obj)
    {
        if(obj.q == 1) os << obj.x;
        else os << obj.p << "/" << obj.q;
        return os;
    }
```

[41] Такие классы принято называть *ортогональными*, то есть, не пересекающимися в плоскости функциональности.

```
};
```

Тип **integer**, включенный в иерархию наследования, является простой “оберткой” встроенного типа **int** – поле нужного типа оборачивается в класс для того, чтобы добавить в один с полем контекст необходимые операции и наследовать примитивный тип. В качестве примера операции с переменным результатом реализован оператор вывода. Так как оператор вывода вызывается только на объекты типов, определенных в библиотеке **iostream**, его перегрузка внутри других классов обязана быть дружественной функцией (важно помнить, что для методов есть обязательно условие – неявный указатель на сам объект, **this**). Первый аргумент – это ссылка на объект потока, в который будет осуществлен вывод; второй – объект, который надо поместить в поток вывода:

```
r_int x(1); std::cout << x;
```

В этом случае будет вызван оператор вывода, перегруженный в классе **r_int**, первым аргументом для него будет объект `std::cout`, а вторым – объект *x*. Этот фрагмент кода выведет в поток строку “1”, а код ниже строку “4/5":

```
r_int y(4, 5); std::cout << y;
```

Как читатель наверняка помнит, дружественные функции не имеют доступа к скрытым полям анонимного объекта родительского типа, поэтому в классах **rational** и **integer** необходимо использовать метку `protected` вместо метки `private`:

```
class integer {
public:
    integer(int a) : x(a) {}
    // остальные методы
protected:
    int x{0};
};
class rational {
public:
```

```
    rational(int x, int y) { ... }
    // остальные методы
protected:
    int p{1}; int q{1};
};
```

Аналогично оператору вывода можно перегрузить все остальные операторы, либо использовать те, которые перегружены для родительских типов.

9.2. Практические задачи для закрепления теории

а. Если предположить, что перед программистом стоит задача интегрировать в иерархию наследования, показанную в этой главе, классы, которые рассматривались в предыдущих главах (**number**, **positive_integer** и т.д.), иерархия, реализованная в этой главе, может быть схематично представлена как сходящиеся к типу **r_int** ветви, как показано на рисунке 4.

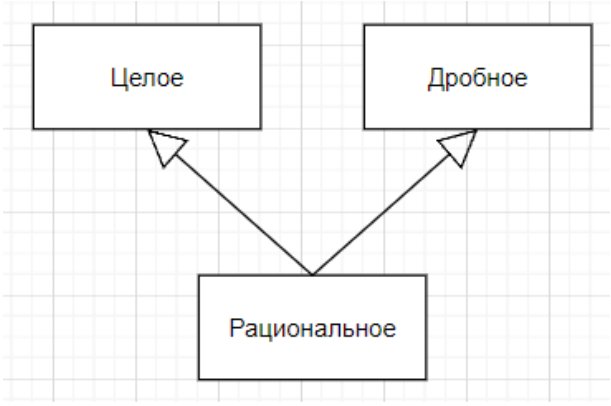


Рисунок 4, Диаграмма классов для иерархии множественного наследования.

Если в иерархию добавить общий для целых и дробных чисел базовый класс “Числа”, при инициализации объекта копия этого класса будет содержаться и в объекте типа “Целое”, и в объекте “Дробное”, как показано на рисунке 5.

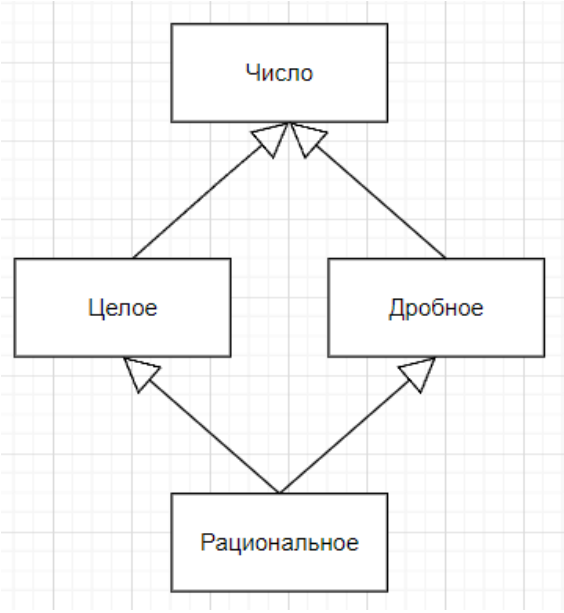


Рисунок 5, Диаграмма классов для иерархии ромбовидного наследования.

Но в объекте “Рациональное” при инициализации будут созданы копии объектов и “Целого”, и “Дробного” типов. Отсюда можно сделать вывод, что из объекта типа “Рациональное” будет доступ к двум разным объектам типа “Число”, так как *тип “Рациональное” по факту наследует тип “Число” дважды*. Если попытаться обратиться к одному из них изнутри класса “Рациональное”, какая из копий будет использоваться?

Такая ромбовидная[42] структура иерархии наследования намекает на возможные проблемы в программах в будущем. Необходимо заставить компилятор не создавать лишние копии объектов при построении объекта, который расположен внизу

[42] В специальной литературе этот феномен называется “**dreaded diamond**” [Standard].

иерархии. Конечно, в описании класса “Рациональное” можно явно указывать: к какой именно из двух копий обращаться, используя пространства имен. Но это нужно будет делать везде, что не очень удобно.

Лабораторная работа 7
Тема: Контейнеры

Цель работы: Для заданного варианта[43] реализовать шаблонный класс контейнера (*generic container*) на основании предыдущей лабораторной работы. Уже реализованные в предыдущих лабораторных работах методы при необходимости изменить согласно требованиям варианта. Добавить реализацию методов, которые должны присутствовать в классе, но отсутствовали в интерфейсе до этого момента. Все варианты должны реализовывать *конструктор по умолчанию*, *конструктор копирования*, оператор *присваивания* (**operator=**), а также набор статических операций: операторы *сравнения* (**operator==**, **operator!=**, **operator<**, **operator>**, **operator<=**, **operator>=**), оператор *ввода* (**operator>>**), оператор *вывода* (**operator<<**).

Array

Условие для выполнения лабораторной работы:

```
#include <algorithm>
#include <iostream>
#include <iterator>
int main() {
    array<int> a1 = {3, 2, 1};
    array<int> a2 = a1;
    std::sort(a1.begin(), a1.end());
    for(auto n : a1) std::cout << n << " ";
    std::reverse_copy(
        a2.begin(), a2.end(),
        std::ostream_iterator<int>(std::cout, " "));
}
```

Vector

Помимо стандартных конструкторов, деструктора и оператора присваивания, а также уже реализованных методов интерфейс включает в себя следующие методы.

[43] Варианты см. в приложении 1.2.

Метод **capacity** возвращает максимальное количество элементов, которое в данный момент вмещает вектор. Как правило, размер блока памяти, выделенной, под вектор превышает текущее количество элементов в векторе для того, чтобы при добавлении нового элемента каждый раз не выделять память заново. Асимптотичность метода постоянная – ***O(1)***.

Метод **push_back** принимает один аргумент – значение, которое необходимо добавить в конец вектора в качестве нового элемента. Если новый размер превышает **capacity**, метод вызывает выделение нового блока памяти. При этом все ссылки и итераторы, существующие в программе, становятся недействительными. Асимптотичность метода в среднем константная – ***O(1)***.

Метод **pop_back** удаляет последний элемент вектора. В зависимости от задачи, метод может выделять новую память под контейнер, когда пустых блоков становится слишком много. Асимптотичность метода в среднем константная – ***O(1)***.

Метод **shrink_to_fit** ничего не возвращает и не имеет параметров. Метод пытается удалить неиспользуемые контейнером байты из памяти, занимаемой им. Можно считать, что метод “подгоняет” значение **capacity** под значение **length**. В некоторых ситуациях метод может ничего не сделать (зависит от реализации контейнера). Асимптотичность метода в среднем константная, в худшем случае линейная – ***O(n)***.

Условие для выполнения лабораторной работы:

```
#include <iostream>
int main() {
    vector<int> v{0, 1, 2, 3, 4, 5, 6, 7};
    v.push_back(8);
    v.push_back(9);
    for(int n : v) std::cout << n << " ";
    std::cout << "\n";
    v.pop_back();
    v.erase(v.begin());
    v.insert(v.end(), 10);
    v.erase(v.begin() + v.size() - 1, v.end());
```

```
    for(auto it = v.begin(); it != v.end(); ) {
        if(*it % 2) it = v.erase(it);
        else ++it;
    }
    for(int n : v) std::cout << n << " ";
}
```

Priority queue

Помимо стандартных конструкторов, деструктора и оператора присваивания, а также уже реализованных методов интерфейс включает в себя следующие методы.

Метод **push** принимает один аргумент – значение, которое необходимо добавить в контейнер в качестве нового элемента. Все свойства метода зависят от реализации адаптируемого контейнера, но итоговая асимптотичность будет суммой с ***O****(log n)*.

Метод **pop** не имеет параметров и ничего не возвращает. Метод удаляет максимальный (по умолчанию) элемент контейнера. Все свойства метода зависят от реализации адаптируемого контейнера, но итоговая асимптотичность будет суммой с ***O****(log n)*.

Метод **top** не имеет параметров и возвращает ссылку на значение максимального (по умолчанию) элемента контейнера. Асимптотичность метода константная – ***O****(1)*.

Условие для выполнения лабораторной работы:

```
#include <iostream>
int main() {
    priority_queue<int> pq1;
    pq1.push(5);
    priority_queue<int> pq2{pq1};
    priority_queue<int> pq3({3, 1, 4, 1, 5});
    for(; !pq3.empty(); pq3.pop())
        std::cout << pq3.top() << " ";
}
```

Forward list

Помимо стандартных конструкторов, деструктора и оператора присваивания, а также уже реализованных методов интерфейс включает в себя следующие методы.

Метод **`insert_after`** принимает два аргумента: первый – итератор – указывает на позицию внутри коллекции, второй ссылается на значение, которое необходимо добавить в контейнер. Метод вставляет новый элемент в список следом за элементом, на который указывает первый аргумент. Операция не затрагивает уже существующие итераторы списка. Метод возвращает итератор на вставленный элемент. Так как место вставки известно заранее, асимптотичность метода константная – ***O**(1)*.

Метод **`erase_after`** принимает один аргумент – итератор на позицию внутри контейнера. Метод удаляет элемент, следующий за элементом, на который указывает аргумент `pos`, из списка. Операция не затрагивает уже существующие итераторы списка. Метод возвращает итератор на следующий после удаленного элемент (или итератор на `end`, если удаленный элемент был последним). Так как место удаления известно заранее, асимптотичность метода константная – ***O**(1)*.

Метод **`push_front`** принимает один аргумент – ссылку на значение заданного типа. Метод ничего не возвращает. Метод добавляет новый элемент со значением аргумента в начало списка, сохраняя все существующие на тот момент итераторы. Асимптотичность метода константная – ***O**(1)*.

Метод **`pop_front`** не имеет параметров и ничего не возвращает. Метод удаляет первый элемент списка, сохраняя все остальные итераторы. Асимптотичность метода константная – ***O**(1)*.

Метод `remove` принимает один аргумент – ссылку на значение заданного типа – и либо ничего не возвращает, либо возвращает количество удаленных элементов (в зависимости от поставленной задачи). Метод удаляет все элементы списка, значения которых соответствуют значению аргумента, сохраняя

все остальные итераторы. Асимптотичность метода линейная – ***O**(n)*.

Метод **`merge`** принимает ссылку на такой же список в качестве аргумента. Метод ничего не возвращает. Метод производит слияние двух отсортированных по возрастанию списков в один общий отсортированный по возрастанию список. После окончания операции список аргумента остается пустым. Для определения порядка новых элементов в списке используется оператор `<` типа `T`. Метод ничего не делает, если аргумент указывает на сам список, на который метод изначально вызывался. Если в списке аргумента встречается значение, которое уже есть в первоначальном списке, оно вставляется с сохранением порядка элемента с этим же значением из первоначального списка (то есть, операция является *стабильной*). Асимптотичность метода линейная – ***O**(n)*.

Метод **`reverse`** не имеет параметров и ничего не возвращает. Метод изменяет направление списка на противоположное, сохраняя существующие итераторы. Асимптотичность метода линейная – ***O**(n)*.

Метод **`unique`** не имеет параметров и либо ничего не возвращает, либо возвращает количество удаленных элементов (в зависимости от поставленной задачи). Метод удаляет все повторяющиеся в списке элементы, идущие подряд, оставляя в списке первый элемент из группы повторяющихся. Равенство элементов определяется с помощью оператора `==` для типа `T`. Асимптотичность метода линейная относительно количества произведенных сравнений – ***O**(n)*.

Метод **`sort`**[44] не имеет параметров и ничего не возвращает. Метод сортирует элементы списка по возрастанию. Операция стабильна: если элементы уже находятся в отсортированном порядке, их позиция не изменяется. Для определения порядка элементов в списке используется оператор

[44] Класс требует отдельный метод сортировки, так как `std::sort` в качестве аргументов ожидает итераторы случайного доступа (как указатели в массив), а список такой доступ к своим элементам не предоставляет.

< типа T. Асимптотичность метода относительно количества сравнений и количества элементов в списке – ***O****(n log n)*.

Метод **splice_after** принимает два аргумента: итератор на позицию в текущем списке и ссылку на другой список. Метод ничего не возвращает. Метод переносит элементы из списка-аргумента в текущий список, элементы вставляются в том же порядке за элементом в заданной первым аргументом позиции. Метод не совершает операции копирования, все существующие итераторы остаются действительными после операции. Асимптотичность метода линейная относительно количества элементов списка-аргумента – ***O****(n)*.

Условие для выполнения лабораторной работы:

```
#include <cassert>
#include <iostream>
int main() {
    forward_list<int> fl1{1, 2, 3, 4, 5};
    forward_list<int> fl2(fl1.begin(), fl1.end());
    for( ; !fl1.empty(); fl1.pop_front())
        std::cout << fl1.front() << " ";
    assert(fl1.empty());
    auto itbegin = fl2.begin();
    fl2.insert_after(itbegin, 0);
    fl2.pop_front();
    assert(0 == fl2.front());
}
```

List

Помимо стандартных конструкторов, деструктора и оператора присваивания, а также уже реализованных методов интерфейс включает в себя следующие методы.

Метод **push_front** принимает один аргумент – ссылку на значение заданного типа. Метод ничего не возвращает. Метод добавляет новый элемент со значением аргумента в начало списка, сохраняя все существующие на тот момент итераторы. Асимптотичность метода константная – ***O****(1)*.

Метод **pop_front** не имеет параметров и ничего не возвращает. Метод удаляет первый элемент списка, сохраняя все остальные итераторы. Асимптотичность метода константная – ***O**(1)*.

Метод **push_back** принимает один аргумент – значение, которое необходимо добавить в конец списка в качестве нового элемента, сохраняя все существующие на тот момент итераторы. Асимптотичность метода в среднем константная – ***O**(1)*.

Метод **pop_back** не имеет параметров и ничего не возвращает. Метод удаляет последний элемент списка, сохраняя все остальные итераторы. Асимптотичность метода в среднем константная – ***O**(1)*.

Метод **remove** принимает один аргумент – ссылку на значение заданного типа – и либо ничего не возвращает, либо возвращает количество удаленных элементов (в зависимости от поставленной задачи). Метод удаляет все элементы списка, значения которых соответствуют значению аргумента, сохраняя все остальные итераторы. Асимптотичность метода линейная – ***O**(n)*.

Метод **merge** принимает ссылку на такой же список в качестве аргумента. Метод ничего не возвращает. Метод производит слияние двух отсортированных по возрастанию списков в один общий отсортированный по возрастанию список. После окончания операции список аргумента остается пустым. Для определения порядка новых элементов в списке используется оператор `<` типа `T`. Метод ничего не делает, если аргумент указывает на сам список, на который метод изначально вызывался. Если в списке аргумента встречается значение, которое уже есть в первоначальном списке, оно вставляется с сохранением порядка элемента с этим же значением из первоначального списка (то есть, операция является *стабильной*). Асимптотичность метода линейная – ***O**(n)*.

Метод **reverse** не имеет параметров и ничего не возвращает. Метод изменяет направление списка на

противоположное, сохраняя существующие итераторы. Асимптотичность метода линейная – ***O(n)***.

Метод **unique** не имеет параметров и либо ничего не возвращает, либо возвращает количество удаленных элементов (в зависимости от поставленной задачи). Метод удаляет все повторяющиеся в списке элементы, идущие подряд, оставляя в списке первый элемент из группы повторяющихся. Равенство элементов определяется с помощью оператора **==** для типа T. Асимптотичность метода линейная относительно количества произведенных сравнений – ***O(n)***.

Метод **sort** не имеет параметров и ничего не возвращает. Метод сортирует элементы списка по возрастанию. Операция стабильна: если элементы уже находятся в отсортированном порядке, их позиция не изменяется. Для определения порядка элементов в списке используется оператор **<** типа T. Асимптотичность метода относительно количества сравнений и количества элементов в обоих списках – ***O(m + n - 1)***.

Метод **splice** принимает два аргумента: итератор на элемент внутри текущего списка и ссылку на другой список. Метод переносит элементы из списка-аргумента в текущий список, начиная с позиции, на которую указывает первый аргумент. Это делается без копирования элементов, только за счет переопределения ссылок на узлы списка, сохраняя все итераторы. Аргумент становится пустым по итогу операции. Асимптотичность метода константная – ***O(1)***.

Условие для выполнения лабораторной работы:

```
#include <algorithm>
#include <iostream>
int main() {
    list<int> l = {2, 4, 6, 8};
    l.push_front(1);
    l.push_back(9);
    auto it = std::find(l.begin(), l.end(), 6);
    if(it != l.end()) l.insert(it, 5);
    for(auto n : l) std::cout << n << " ";
```

```
}
```

Deque

Помимо стандартных конструкторов, деструктора и оператора присваивания, а также уже реализованных методов интерфейс включает в себя следующие методы.

Метод **`push_front`** принимает один аргумент – ссылку на значение заданного типа. Метод ничего не возвращает. Метод добавляет новый элемент со значением аргумента в начало контейнера, при этом все существующие на тот момент итераторы становятся недействительными. Асимптотичность метода константная – ***O(1)***.

Метод **`pop_front`** не имеет параметров и ничего не возвращает. Метод удаляет первый элемент контейнера, существующие на тот момент итераторы на удаленный элемент (а также на итератор `end`, если удаленный элемент был последним). Асимптотичность метода константная – ***O(1)***.

Метод **`push_back`** принимает один аргумент – значение, которое необходимо добавить в конец контейнера в качестве нового элемента, все существующие на тот момент итераторы становятся недействительными. Асимптотичность метода в среднем константная – ***O(1)***.

Метод **`pop_back`** не имеет параметров и ничего не возвращает. Метод удаляет последний элемент контейнера, существующие на тот момент итераторы на удаленный элемент (а также на итератор `end`) становятся недействительными. Асимптотичность метода в среднем константная – ***O(1)***.

Метод **`shrink_to_fit`** ничего не возвращает и не имеет параметров. Метод пытается удалить неиспользуемые контейнером байты из памяти, занимаемой им. В некоторых ситуациях метод может ничего не сделать (зависит от реализации контейнера). Асимптотичность метода в среднем константная, в худшем случае линейная – ***O(n)***.

Условие для выполнения лабораторной работы:

```
#include <iostream>
```

```
int main() {
    deque<int> q;
    q.push_back(1);
    q.push_back(2);
    q.push_back(3);
    for( ; q.size(); q.pop_front())
        std::cout << q.front() << " ";
    deque<int> s;
    s.push_front(1);
    s.push_front(2);
    s.push_front(3);
    for( ; s.size(); s.pop_back())
        std::cout << s.back() << " ";
}
```

Stack

Помимо стандартных конструкторов, деструктора и оператора присваивания, а также уже реализованных методов интерфейс включает в себя следующие методы.

Метод **`push`** принимает один аргумент – значение, которое необходимо добавить на верхушку стека в качестве нового элемента. Все свойства метода зависят от реализации адаптируемого контейнера.

Метод **`pop`** не имеет параметров и ничего не возвращает. Метод удаляет элемент с верхушки стека. Все свойства метода зависят от реализации адаптируемого контейнера.

Метод **`top`** не имеет параметров и возвращает ссылку на значение верхушки стека. Асимптотичность метода константная – ***O(1)***.

Для самопроверки можно использовать код, предоставленный ниже.

```
#include <iostream>
int main() {
    stack<int> s;
    s.push(1);
    s.push(2);
    s.push(3);
    for( ; s.size(); s.pop()) std::cout << s.top() << " ";
```

```
}
```

Queue

Помимо стандартных конструкторов, деструктора и оператора присваивания, а также уже реализованных методов интерфейс включает в себя следующие методы.

Метод **push** принимает один аргумент – значение, которое необходимо добавить в конец очереди в качестве нового элемента. Все свойства метода зависят от реализации адаптируемого контейнера.

Метод **pop** не имеет параметров и ничего не возвращает. Метод удаляет элемент из начала очереди. Все свойства метода зависят от реализации адаптируемого контейнера.

Условие для выполнения лабораторной работы:

```
#include <iostream>
int main() {
    queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);
    for( ; q.size(); q.pop())
        std::cout << q.front() << " ";
}
```

Set

Помимо стандартных конструкторов, деструктора и оператора присваивания, а также уже реализованных методов интерфейс включает в себя следующие методы.

Метод **merge** принимает ссылку на такое же множество в качестве аргумента. Метод ничего не возвращает. Метод производит слияние двух множеств, удаляя нужные узлы из аргумента и вставляя их в текущее множество. Если в множестве аргумента встречается значение, которое уже есть в первоначальном множестве, узел остается нетронутым. Операция производится без копирования, только за счет изменения ссылок внутри узлов “донора”. Таким образом существующие итераторы и ссылки на значения переносимых

узлов остаются действительными. Асимптотичность метода зависит от длины текущего множества и множества-“донора” – ***O****(n log(n + m))*.

Метод **`extract`** принимает итератор на значение внутри множества. Метод возвращает объект-узел с заданным значением. Метод удаляет узел, на который указывает аргумент, из множества. При этом не происходит операций копирования, при необходимости дерево балансируется заново. Асимптотичность метода в среднем константная – ***O****(1)*.

Метод **`lower_bound`** принимает ссылку на значение типа `T` и возвращает итератор на первый элемент множества, значение которого не меньше аргумента. Если в контейнере нет соответствующих значений, в качестве результата возвращается **`end`**. Асимптотичность метода логарифмическая – ***O****(log n)*.

Метод **`upper_bound`** принимает ссылку на значение типа `T` и возвращает итератор на первый элемент множества, значение которого строго больше аргумента. Если в контейнере нет значений больше аргумента, в качестве результата возвращается **`end`**. Асимптотичность метода логарифмическая – ***O****(log n)*.

Метод **`equal_range`** принимает ссылку на значение типа `T` и возвращает пару итераторов на элементы множества, где первый итератор указывает на первый элемент, значение которого меньше либо равно аргументу, второй итератор указывает на первый элемент, значение которого строго больше аргумента. Если в контейнере нет значений меньше или равных аргументу, в качестве первого итератора возвращается **`end`**. Если в контейнере нет значений больше аргумента, в качестве второго итератора возвращается **`end`**. Асимптотичность метода логарифмическая – ***O****(log n)*.

Условие для выполнения лабораторной работы:

```
#include <cassert>
#include <iostream>
int main() {
    set<int> s1;
```

```
    s1.insert(3); s1.insert(3);
    assert(1 == s1.count(3));
    set<int> s2 = {1, 2, 3, 4, 1, 2, 3, 4};
    assert(4 == s2.size());
    for(auto it = s2.begin(); it != s2.end(); ) {
        if(*it % 2) it = s2.erase(it);
        else ++it;
    }
    assert(0 == s2.erase(1));
}
```

Unordered map

Помимо стандартных конструкторов, деструктора и оператора присваивания, а также уже реализованных методов интерфейс включает в себя следующие методы.

Метод **merge** принимает ссылку на другой ассоциативный массив в качестве аргумента. Метод ничего не возвращает. Метод производит слияние двух массивов, удаляя нужные узлы из аргумента и вставляя их в текущий массив, используя хеш-функцию и соответствующий предикат. Если в массиве аргумента встречается значение, которое уже есть в первоначальном массиве, ключ остается нетронутым. Операция производится без копирования, только за счет изменения ссылок внутри узлов "донора". Таким образом существующие итераторы и ссылки на значения переносимых узлов остаются действительными. Асимптотичность метода зависит от длины множества-"донора", в среднем линейное – ***O****(n)* или ***O****(n * m + n)*.

Метод **extract** принимает итератор на значение внутри массива. Метод возвращает объект-узел с заданным значением. Метод удаляет узел, на который указывает аргумент, из массива. При этом не происходит операций копирования. Асимптотичность метода в среднем константная – ***O****(1)*, в худшем случае линейная – ***O****(n)*.

Условие для выполнения лабораторной работы:

```
#include <iostream>
```

```
#include <string>
int main() {
    unordered_map<std::string, std::string> u = {
        {"RED", "#FF0000"},
        {"GREEN", "#00FF00"},
        {"BLUE", "#0000FF"}};
    for(auto n : u)
        std::cout << n.first << ": " << n.second << " ";
    std::cout << std::endl;
    u["BLACK"] = "#000000";
    u["WHITE"] = "#FFFFFF";
    for(auto n : u)
        std::cout << n.first << ": " << n.second << " ";
}
```

Рекомендуется компилировать сделанную лабораторную работу с помощью приложения `g++` в файле с расширением `.cpp`, установив флаги `-Wall`, `-Wpedantic`, `-std=c++17`.

10. Примеры решений для практических задач

10.1. Задачи из пункта 1

а. Идея, стоящая за функцией сокращения дробей довольно проста. Во-первых, если числитель равен нулю или знаменатель – единице, делать сокращение нет необходимости. Во-вторых, если сокращение делать все таки надо, для этого достаточно определить значение наибольшего общего делителя для дроби и поделить на эту величину числитель и знаменатель:

```
void reduce(rational &r) {
    if(r.q == 1 || r.p == 0) return;
    int d = gcd(abs(r.p), r.q);
    r.p /= d;
    r.q /= d;
}
```

Важно отметить то, что тип данных для переменной *d* указан явно, что можно сделать только с несколькими допущениями: результат функции **gcd** имеет значение типа **int** или без потерь приводится к нему; числитель и знаменатель аргумента *r* могут участвовать в выражениях вместе со значениями типа **int**. Этот дефект алгоритма в последствии придется исправить. *Обобщенное программирование* [Musser] является одним из инструментов для этого и подробно обсуждается в главе, посвященной универсальному полиморфизму.

Реализация нахождения наибольшего общего делителя могла бы выглядеть так:

```
int gcd(int m, int n) {
       while(n != 0) { int t = m % n; m = n; n = t; }
       if(m < 0) m = -m;
       return m;
}
```

Здесь используется модифицированный алгоритм Евклида [Степанов]. Данный алгоритм легко переписывается с

помощью рекурсии для тех языков, которые не имеют встроенных циклов. У простых реализаций этого алгоритма есть некоторые недостатки. Алгоритм зависит от операции деления, которая является недостаточно эффективной. Алгоритм также зависит от типов данных аргументов, что привязано к архитектуре процессора и используемому компилятору, поэтому неудачная реализация может нарушать важные математические свойства алгоритма [Cormen]. На практике часто можно встретить использование расширенной версии алгоритма Евклида или двоичного алгоритма, разработанного Д. Штайном [Knuth]. Поэтому лучшим выбором для большинства разработчиков будет использование *встроенного в стандартную библиотеку* алгоритма, реализация которого остается на совести авторов компилятора. Например, в **C++**:

```
template<class M, class N>
constexpr std::common_type_t<M, N> gcd(M m, N n);
```

Стандартная функция использует *шаблоны типов* для того, чтобы в качестве аргументов можно было передавать любые целые числа. С этого момента текст предполагает, что все упоминания функции **gcd** используют версию алгоритма, поставляемую с языком **C++**.

В случае с выводом строковой формы дроби реализация будет чуть сложнее. Для этого необходимо решить две задачи. Первая задача заключается в определении всех цифр, которые составляют числа в числителе и знаменателе. Вторая задача состоит в составлении строки, которая содержит в себе символы, отвечающие полученным числовым значениям. Достичь этого можно несколькими способами, и ниже представлен лишь один из них:

```
char *rstr(const rational &r, char *ret) {
    int i = 0;
    auto f = [&] (int n) {
        char *start = ret + i;
        while(n) { ret[i++] = (n % 10) + '0'; n /= 10; }
        char *end = ret + i - 1;
```

```
        while(start != end) {
            char tmp = *start;
            *start = *end;
            *end = tmp;
            start++; end--;
        }
    };
    f(r.p);
    if(r.q > 1) { ret[i++] = '/'; f(r.q); }
    return ret;
}
```

Тонкий момент в коде выше связан с включением в функцию другой функции. Это сделано исходя из того факта, что внутренняя функция никогда не будет использоваться отдельно от родительской функции, соответственно нет причины захламлять пространство имен, связанное с этим файлом, еще одним именем. Сама внутренняя функция **f** сделана с помощью лямбда-выражения[45]. Эта функция определяет цифры, составляющие числитель и знаменатель, с помощью остатка от деления на 10 и записывает эти цифры в виде печатных символов в предоставленный для этого массив. Сами печатные символы получаются прибавлением ASCII-кода [ANSI] символа “0” к числовому значению, что даст ASCII-код искомой цифры (так как все коды в таблице идут по порядку, начиная с нуля).

Символы поглощаются справа налево, следовательно запись в массив происходит по мере их получения, то есть, в порядке обратном тому, который нужен для правильного отображения всей дроби. Поэтому часть массива, которая только что была заполнена, должна быть перевернута (это делается во втором цикле внутренней функции **f**). В конечном итоге, если знаменатель не равен единице, строковое представление должно в себя включать символ дроби и сам знаменатель после.

[45] https://neerc.ifmo.ru/wiki/index.php?title=Лямбда-исчисление.

Наконец, важно отметить несколько неочевидных моментов. Во-первых, функция не знает заранее, какой величины должна быть результирующая строка, и для таких случаев строка либо поставляется в функцию вместе с дробью (то есть, размер уже определен до вызова функции), либо строка создается динамически в самой функции, что подразумевает несколько дополнительных операций (подсчет общего количества символов и выделение нужного участка памяти в соответствии с результатом подсчета). В примере выше используется первый способ. Но если читатель чувствует уверенность в себе, автор рекомендует попробовать реализовать второй способ самостоятельно.

Во-вторых, функция возвращает строку, которая передается в нее изначально, что может показаться бесполезным. Причина состоит в том, что строки в **C** и **C++** являются указателями, что подразумевает отсутствие процесса “конструирования” объекта внутри функции. Это значит, что данная функция не создает новых объектов, а только лишь дублирует адрес строки, в которой были произведены необходимые манипуляции с данными. Другими словами, можно считать, что у функции нет возвращаемого значения:

```
rational a = cons(10, 20); reduce(a);
char tmp[100]{}; rstr(a, tmp);
assert(!strcmp("1/2", tmp));
```

Тем не менее, согласно первоначальной идее того, как эта функция должна работать, она возвращает строковое представление:

```
char tmp[100]{};
assert(!strcmp(“1/2”, rstr(a, tmp)));
```

То есть, результат работы этой функции можно передавать как аргумент в другие функции, ожидающие указатель на строку в качестве аргумента. Из этого следует, что функция должна возвращать не адрес той строки, которую в нее передали, а непосредственно символы, составляющие такую

строку, которые можно скопировать в новую переменную типа **char***:

```
tmp = rstr(a);
```

Это можно сделать с помощью абстрактного типа данных, о чем речь пойдет дальше.

10.2. Задачи из пункта 2

а. Функция **reduce** использует ранее написанный код, таким образом опять применяется *абстракция* алгоритма (создание отдельной функции) и его *комбинация* (вызов функции):

```
#include <cassert>
#include <numeric>
class rational {
public:
    static void reduce(rational &r) {
        if(r.q == 1 || r.p == 0) return;
        int d = gcd(abs(r.p), r.q);
        r.p /= d; r.q /= d;
    }
    static rational cons(int x, int y) {
        assert(y != 0);
        rational r;
        if(x == 0) { r.p = 0; r.q = 0; }
        else if(y < 0) { r.p = -x; r.q = -y; }
        else { r.p = x; r.q = y; }
        reduce(r);
        return r;
    }
    int get_p() { return p; }
    int get_q() { return q; }
    void set_p(int x) { p = x; }
    void set_q(int x) {
        assert(x != 0);
        q = x;
    }
private:
    int p; int q;
```

```
};
```

Проверить работу можно с помощью следующего блока кода:

```
rational a = rational::cons(10, 50);
a.set_p(15); a.set_q(45);
rational::reduce(a);
assert(1 == a.get_p() && 3 == a.get_q());
rational b = rational::cons(10, 50);
assert(1 == b.get_p() && 5 == b.get_q());
```

Другим вариантом решения этой проблемы может быть полный отказ от сеттеров. Тогда остается только один способ работы с рациональными числами – создание новых объектов каждый раз, когда какое-то значение должно поменяться. Это значит, что придется переписать и функцию **reduce**:

```
static rational reduce(rational &r) {
    if(r.q == 1 || r.p == 0) return r;
    int d = gcd(abs(r.p), r.q);
    if(d == 1) return r;
    return cons(r.p / d, r.q / d);
}
```

Здесь из соображений краткости используется функция **cons** для построения нового объекта. Читатель помнит, что внутри **cons** как раз используется функция **reduce**, а это чревато бесконечной взаимной рекурсией.

Данный подход менее эффективен в контексте языка **C**, потому что делает несколько дополнительных вызовов функций и создает новые объекты, без которых можно было бы обойтись. Но в других языках программирования (например, в Haskell[46] или OCaml[47]) запрет на изменение существующих переменных – “мутации” – часто является обязательным, поэтому такой подход к решению задач надо иметь ввиду.

[46] *https://www.haskell.org/*.
[47] *https://ocaml.org/*.

б. Пример решения представлен ниже:

```
rational b = rational::cons(20, 50);
int *p = (int *)&b; int *q = p + 1;
*p = 5; *q = 6;
assert(5 == b.get_p() && 6 == b.get_q());
```

Если не получилось достичь такого же результата, следует проанализировать пример выше. Особенно внимательно стоит посмотреть на строку `int *p = (int *)&b`.

О чем говорит этот пример? Во-первых, о том, что метка `private` не дает какой-то “железной” защиты данных, которые под ней скрываются. Во-вторых, о том, что сокрытие данных и инкапсуляция, в основном, призваны не защитить данные от изменения, а защитить программистов от соблазна изменять данные. Эта идея проходит красной нитью через основание всех принципов объектно-ориентированного программирования. Не стоит забывать, что в конечном итоге программы пишутся для компьютера, а не для других программистов, и все стилистические решения в программах должны, в первую очередь, диктоваться требованиями компонентов компьютера, для которого они пишутся.

10.3. Задачи из пункта 3

а. Пример конструктора со строкой в качестве единственного параметра:

```
class rational {
public:
    rational(const char *str) {
        int __p, __q;
        if(sscanf(str, "%d/%d", &__p, &__q) >= 1) {
            p = __p; q = __q;
            if(p != 0) reduce(*this);
        }
    }
private:
    int p{1}; int q{1};
};
```

Первый вариант мог бы выглядеть так. В этом случае для конвертирования строк в целые числа используется функция **sscanf** из библиотеки **stdio**:

```
rational e("3/15");
assert(1 == e.get_p() && 5 == e.get_q());
```

Используя тот факт, что функция умеет обращаться со строками в качестве чисел, можно заранее подготовить две переменные, куда будет скопировано числовое представление правильно отформатированной строки. Но это же свойство функции означает, что любую другую строку так обработать не получится, аргумент должен включать в себя символ “/”, иначе нет гарантии, что в числитель и знаменатель будут записаны правильные значения.

В этой же функции есть небольшая хитрость, которую можно использовать как желаемое поведение: функция **sscanf** из библиотеки **stdio** возвращает в качестве результата число, указывающее на то, сколько переменных было успешно конвертировано. Если это результат больше нуля, значит считывание строки в числитель прошло успешно. Например, такой вызов конструктора:

```
rational e(“3 15”);
```

дал бы в виде результата рациональное число с 3 в числителе и 1 в знаменателе, то есть, все содержимое исходной строки после пробела было бы проигнорировано. В зависимости от дизайна абстрактного типа данных это может быть как желаемым свойством данного конструктора, так и “*багом*”, который нужно исправить.

Если отказаться от использования **sscanf** для достижения желаемого эффекта, можно попробовать другие варианты конвертирования, с точным определением положения символа “/” в исходной строке. Например, имея некую целочисленную переменную `offset = 0`, проверить наличие дробной черты можно было бы так:

```
while(str[offset++] != '/' && strlen(str) > offset);
```

Если значение переменной после цикла равно длине строки, исходная строка либо содержит целое число, либо не содержит подходящего значения вовсе. Исходя из этого, можно разделить исходную строку на числитель и знаменатель и конвертировать их по отдельности, что вам и предлагается сделать в качестве самопроверки.

б. *Конструктор переноса* – это такой конструктор, который принимает *ссылку на R-значение* в качестве аргумента. Например, для типа **rational** он мог бы выглядеть так:

```
rational(rational &&other) { … }
```

Оставив тело конструктора пустым для начала, следует посмотреть на параметр. Очевидно, что этот конструктор будет вызван только в случаях, когда объект заданного типа создается с помощью ссылки на R-значение, иначе будет вызван конструктор копирования. При этом в коде еще ни разу не встречались объекты, которые изначально были бы ссылками на R-значение. На практике принято создавать (или использовать имеющуюся в библиотеке **utility**) функцию **move**, которая конвертирует значения, переданные в нее, в ссылки на R-значения[48]:

```
class rational {
public:
    rational(rational &&other) {
        p = other.p; q = other.q;
        other.p = 1; other.q = 1;
    }
private:
    int p{1}; int q{1};
};
```

[48] Точнее, функция **move** возвращает R-значение без вызова соответствующего конструктора копирования [Hinnant].

Порядок действий достаточно прост. Берутся значения всех полей объекта-источника и копируются в объект-приемник, затем значения полей объекта источника “сбрасываются”:

```
rational g(std::move(e));
assert(1 == e.get_p() && 1 == e.get_q());
assert(1 == g.get_p() && 5 == g.get_q());
```

Здесь важно отметить два момента. Во-первых, оба поля переданного объекта представлены примитивными типами данных. Если бы эти поля были представлены составными типами, каждое поле в отдельности следовало бы предварительно конвертировать в ссылку на R-значение (с помощью вызова на них функции **move**). Во-вторых, компилятор не дает никаких гарантий относительно того, чему будут равны значения полей объекта-источника после переноса, поэтому в конструкторе следует их установить в какие-то адекватные значения. Перенос делается для экономии памяти, то есть, объект источник использовать больше не планируется, но если про это забыть и после переноса его применить в каком-то выражении, лучше бы он не содержал какой-то мусор.

в. Пример измененной функции можно посмотреть ниже:

```
class rational {
public:
    static void swap(rational &a, rational &b) {
        rational tmp(std::move(a));
        a = std::move(b);
        b = std::move(tmp);
    }
private:
    int p{1}; int q{1};
};
```

Кому-то из читателей эта реализация может показаться лишней. В конечном итоге, зачем добавлять вызов функции туда, где и без нее можно обойтись:

```
rational a(1, 2);
assert(rational(1, 2) == a);
```

```
rational b(2, 3);
assert(rational(2, 3) == b);
rational::swap(a, b);
assert(rational(2, 3) == a);
assert(rational(1, 2) == b);
```

По большому счету это правильный ход мыслей, так как на уровне процессора новая версия функции **swap** точно так же копирует данные как и наивная. Если сравнить генерируемый GCC код для двух реализаций выше, они отличаются только следующими строчками:

```
movq -24(%rbp), %rax
movq %rax, %rdi
call std::remove_reference<
    rational&>::type&& std::move<rational&>(rational&)
```

То есть, все выполняемые процессором инструкции для первой версии **swap** остаются и во второй версии, вдобавок к трем инструкциям выше (если внимательно к ним присмотреться, можно заметить, что это всего-навсего вызов функции `std::move`). Для того, чтобы понять механику происходящего в новой версии стоит взглянуть на реализацию функции `std::move`:

```
return static_cast<
    typename std::remove_reference<_Tp>::type&&>(__t);
```

Не обращая внимания на синтаксические элементы, которые раньше в тексте не встречались, нужно выделить три момента. Ключевое слово **_Tp** – это *псевдоним* (alias), присвоенный компилятором типу данных **rational**. Это видно из “дизассемблированного” фрагмента кода, где тип данных упоминается явно. Интересно же в функции `std::move` то, что она делает последовательные вызовы функций **remove_reference** и **static_cast(&&)** на переданный ей объект. Другими словами, она сначала избавляется от ссылки на переданный объект, получая L-значение, а потом явно приводит полученный тип к ссылке на R-значение. Это, в свою очередь,

заставляет компилятор *обратиться к конструктору переноса*, а не копирования.

10.4. Задачи из пункта 4

а. Следует свериться с примером ниже:

```
class rational {
public:
    rational &operator=(const rational &other) {
        if(this != &other) { p = other.p; q = other.q; }
        return *this;
    }
    rational &operator=(rational &&other) {
        if(this != &other) {
            p = other.p; q = other.q;
            other.p = 1; other.q = 1;
        }
        return *this;
    }
private:
    int p{1}; int q{1};
};
```

Единственное, что может вызвать вопросы – это возвращаемое значение. Так как оператор может быть использован в цепочке вызовов (например, `a = b = c`), а вызовы эти делаются один за другим, возникает ситуация, при которой каждый последующий вызов делается на результат предыдущего вызова оператора присваивания. Очевидно, что результатом должно быть L-значение того же типа, который находится справа от оператора. Поэтому оператор присваивания в качестве результата своего вызова возвращает то, с чем он сам совместим – ссылку на объект, который его вызвал (ссылку, потому что экономится память, чтобы не пришлось копировать данные из существующей области памяти в новую и потом оттуда копировать данные обратно).

б. Ниже представлен готовый пример для сравнения:

```
int lcm(int a, int b) { return (a * b) / gcd(a, b); }
class rational {
```

```
public:
    friend rational operator+(
        const rational &lhs, const rational &rhs)
    {
        rational res;
        res.p = lhs.p + rhs.p; res.q = lhs.q;
        if(lhs.q != rhs.q) {
            res.q = lcm(lhs.q, rhs.q);
            res.p = lhs.p * (res.q / lhs.q) +
                    rhs.p * (res.q / rhs.q);
        }
        return res;
    }
private:
    int p{1}; int q{1};
};
```

Функция **lcm** находит наименьший общий множитель для знаменателей, если знаменатели отличаются. В последнем случае она используется оператором для приведения обеих дробей к общему знаменателю:

```
rational e(1, 2); rational f(1, 3);
assert(rational(5, 6) == (e + f));
```

Аналогичным образом можно реализовать остальные арифметические операции, что читателю предлагается сделать самостоятельно в качестве дополнительного упражнения.

10.5. Задачи из пункта 6

а. Пример решения можно посмотреть ниже:

```
class rational {
public:
    rational(float x, float y) = delete;
    rational(double x, double y) = delete;
private:
    int p{1}; int q{1};
};
```

При такой версии этих конструкторов любая попытка создать объект заменой целых чисел на дробные будет вызывать

ошибку компиляции примерно следующего содержания, “error: use of deleted function 'rational::rational(double, double)'”. Это сообщение напомнит программисту, а пользователям подскажет о необходимости более точно определять тип аргументов.

Есть как минимум еще один способ добиться такого эффекта: можно перенести оба конструктора под метку `private`. Это также не позволит создавать объекты данного типа в программе, вызывая эти конструкторы, но сообщения об ошибке, которые компилятор в таком случае, будут отличаться.

б. Пример создания конструкторов в иерархии наследования:

```
class number {
public:
    // условно абстрактный класс без полей с данными
    number() { std::cout << "number\n"; }
};
class integer : number {
public:
    integer(int a): x(a) {
        std::cout << "integer: " << x << "\n";
    }
protected:
    int x{0};
};
class positive : integer {
public:
    positive(int a): integer(abs(a)) {
        std::cout << "positive: " << x << "\n";
    }
    friend bool operator==(const positive &lhs, int rhs) {
        return lhs.x == rhs;
    }
};
```

Если теперь выполнить инструкцию **`positive i = -1`**, в окне вывода появится три строки:

number

integer: 1

positive: 1

Сначала будет вызван конструктор **number**; затем – конструктор **integer**, в качестве значения ему будет передано число 1 (из-за зависимости **abs**(*a*)); в последнюю очередь вызовется конструктор **positive**. Если же, например, выполнить инструкцию **integer** *i* = -1, сработает только конструктор **integer**. Он инициализирует поле *x* значением "-1", и на этом все закончится.

Сам класс **positive** не имеет поля *x*, но может получить доступ к соответствующему полю своего материнского класса (при условии что поле находится под меткой protected). Это отношение очевидно, если посмотреть на полное описание типа переменной *x*: **int** integer::*x* – то есть, эта переменная изменяется в объекте типа **integer**, доступ к которому осуществляется через переменную типа **positive**, о чем компилятор бы сообщил (error: class 'positive' does not have any field named 'x'). Так как поле *x* принадлежит классу **integer**, этот анонимный объект должен быть инициализирован раньше, чем его дочерний класс. Чтобы его инициализировать вручную надо указать какой именно конструктор использовать для этого.

10.6. Задачи из пункта 7

а. Базовый класс остается без изменений, так как ни конструкторов, ни полей с данными в нем быть не должно, если он играет роль интерфейса. Класс **integer** должен содержать реализацию оператора, заявленного в базовом классе. И здесь есть несколько нюансов. Во-первых, поле данных класса **integer** – **int** *x* – объявлено внутри него, а соответственно базовый класс о нем ничего не знает. Поэтому аргумент типа **number&** не предоставит информации о значении аргумента. Это можно обойти, объявив "геттер" (аксессор) в интерфейсе. При этом всплывает другая проблема: тип данных, который должен возвращаться при обращении к аксессору, заранее не известен. Эту проблему тоже можно решить, но дабы не усложнять задачу

предполагается, что все дочерние классы в данной иерархии будут работать с целыми числами:

```
class number {
    // чистый абстрактный класс без полей с данными
public:
    virtual number &operator+(const number &other) = 0;
    virtual int value() const = 0;
};
```

Теперь появляется возможность написать первую реализацию оператора:

```
class integer : number {
public:
    integer() {}
    integer(int a): x(a) {}
    int value() const override { return x; }
    integer &operator+(const number &other) override {
        x += other.value();
        return *this;
    }
protected:
    int x{0};
};
```

Второй нюанс заключается в том, что согласно свойству замкнутости[49] операция сложения должна производиться между двумя числами и производить какое-то третье число. Для функций в **C++** это предполагает новую область памяти для хранения результата (возвращаемое “по значению”). В случае с виртуальными функциями новое значение возвращать напрямую нельзя, как уже неоднократно упоминалось выше. Это значит, что можно либо создать новый объект внутри метода, либо вернуть объект, которому принадлежит метод.

В примере реализации выше инструкция `integer a = b + c` не просто присваивала бы в *a* сумму *b* и *c*, но и изменяла значение *b* на величину *c*, так как оператор присваивания в

[49] Для любых двух элементов подмножества результат операции между ними даст элемент этого же подмножества [Овчинников].

данном случае вызывается именно для *b* (находится справа от него). Такой сценарий противоречит характеру операции сложения в арифметике, что вынуждает обратиться к другому способу: созданию нового объекта внутри метода с последующим возвратом ссылки на него. Это могло бы выглядеть примерно так:

```
integer &operator+(const number &other) override {
       integer *p = new integer(x + other.value());
       return *p;
}
```

Здесь проглядывается новая проблема: какой объект или функция отвечает за освобождение памяти, занятой новым объектом *p*? Переменная, к которой он привязан, не существует вне метода, поэтому после операции сложения будет удалена. Можно было сохранить адрес этой переменной и потом вручную вызвать на него оператор **delete**, но делать так каждый раз после операции сложения было бы неразумно (следить за всеми такими указателями, которые создаются при сложении, было бы трудоемкой задачей[50]).

Читателю стоит подумать над тем, как обойти этот трудный момент, сохранив семантику операции сложения. Ниже будет показан один из способов автоматической очистки памяти, встроенный в язык **C++**:

```
class integer : number {
public:
    // конструкторы и другие операторы
    integer &operator+(const number &other) override {
        p = std::shared_ptr<integer>(
            new integer(x + other.value()));
        return *(p);
    }
```

[50] Подсчет ссылок (**reference counting**, англ.) – техника отслеживания всех используемых указателей в программе, с немедленным освобождением памяти в тот момент, когда объект, использующий такой указатель, уничтожается [Gill]

```
protected:
    int x{0}; std::shared_ptr<integer> p;
};
```

Данный фрагмент использует объект типа `std::shared_ptr`, который объявлен в стандартной библиотеке **memory**. Этот объект представляет собой сущность, которая будет уничтожена вместе с объектом, которому она принадлежит[51]. В данном случае этот объект инициализируется значением суммы двух аргументов, после чего ссылка на него возвращается из метода в место вызова, где новое значение будет скопировано в другой объект типа **integer**:

```
integer x;
{
      integer a = -5; integer b = 5; x = a + b;
}
std::cout << x.value();
```

Фрагмент выше ожидаемо поместит в окно вывода число “0”. Читатель мог заметить, что в такой последовательности есть слабое место. Что происходит с адресом созданного в недрах `integer::operator+` при последующих попытках сложить его объект с другим? В случае обычного указателя переменная *p* внутри него перезаписывалась бы новым адресом, теряя старый. Создатели класса `std::shared_ptr` предусмотрели и этот момент: объект уничтожается автоматически в случае, если переменной типа `std::shared_ptr` присваивается какое-то другое значение[52].

И, конечно, есть простой способ привести код к более читаемой форме: отказаться от виртуальности оператора сложения в пользу “дружественности”:

```
friend integer
operator+(const number& a, const number& b) {
      return integer(a.value() + b.value());
}
```

[51] https://en.cppreference.com/book/intro/smart_pointers
[52] https://en.cppreference.com/w/cpp/memory/shared_ptr.

Такая форма позволяет сохранить симметричность операции, потому что и аргумент справа, и аргумент слева представлены абстрактным типом данных, в отличие от виртуального метода. Очевидным плюсом этой реализации также является возвращаемое значение, которое может быть просто скопировано без использования указателей.

Пример выше является типичным, несмотря на то, что программисты стараются избегать таких ситуаций. То, что такая ситуация сложилась при построении данной иерархии наследования, косвенно указывает на плохо продуманную структуру наследования. В данном конкретном случае всего лишь имитировалась иерархия, взятая из другого языка, и на практике она бы выглядела совсем по-другому. Тем не менее, этот пример позволил познакомить читателей с неочевидными моментами реализации полиморфизма подтипов в **C++**.

10.7. Задачи из пункта 9

а. Для того, чтобы компилятор взял на себя решение этой проблемы, нужно использовать ключевое слово **`virtual`**, только теперь не для методов, а для объявленных родительских классов. Например, без виртуального наследования иерархия классов[53] выглядела бы так:

```
Class number
number (0x0x39ae480) 0 empty

Class rational
rational (0x0x3731340) 0 empty
  number (0x0x39ae4e0) 0 empty

Class integer
integer (0x0x37313a8) 0 empty
  number (0x0x39ae540) 0 empty
```

[53] В компиляторе gcc иерархию классов можно сгенерировать, собрав программу с флагом `-fdump-lang-class`.

```
Class r_int
r_int (0x0x39c0000) 0 empty
  rational (0x0x3731410) 0 empty
    number (0x0x39ae5a0) 0 empty
  integer (0x0x3731478) 1 empty
    number (0x0x39ae600) 1 empty
```

Как видно из сгенерированной компилятором иерархии, в классе **r_int** будет два объекта (два уникальных адреса) **number**. Отсюда и двоякость интерпретации.

Виртуальное наследование (в сокращенном виде) будет выглядеть примерно следующим образом:

```
class number {
       // тело класса
};
class rational : virtual number {
       // тело класса
};
class integer : virtual number {
       // тело класса
};
class r_int : rational, integer {
       // тело класса
};
```

Правильная реализация использует виртуальное наследование для тех классов, которые находятся непосредственно под общим базовым классом. То есть, в данном случае класс **number** объявляется “виртуальным” с точки зрения наследования, поэтому такой класс в иерархии наследования будет инициализироваться только один раз независимо от того, сколько раз он наследуется дочерними классами. А это именно тот эффект, которого следовало добиться:

```
Class r_int
r_int (0x0x3942000) 0
  rational (0x0x36b1410) 0 nearly-empty
      primary-for r_int (0x0x3942000)
```

```
  number (0x0x392e5a0) 0 empty virtual
integer (0x0x36b1478) 8 nearly-empty
  number (0x0x392e5a0) alternative-path
```

В иерархии с виртуальным наследованием класс **r_int** содержит только один объект **number** (оба упоминания указывают на один и тот же адрес), что снимает проблему с неопределенностью обращения к полям этого объекта из дочернего класса.

Виртуальное наследование широко используется в стандартной библиотеке, так как иерархия классов часто имеет сегменты ромбовидной формы. Один из таких примеров – библиотека **iostream**, иерархия классов которой показана на рисунке 6 [Brinkerhoff].

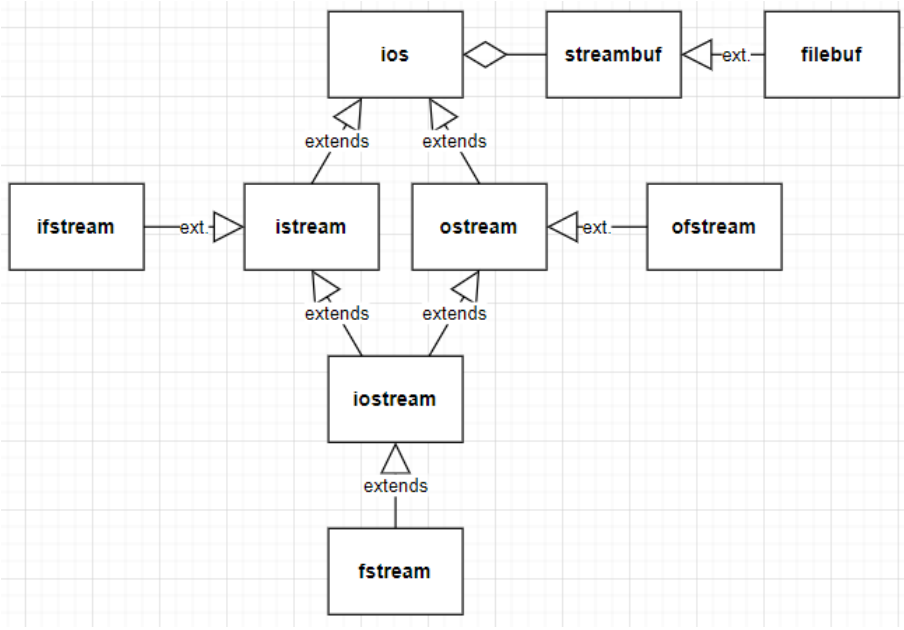


Рисунок 6, Диаграмма классов для фрагмента иерархии библиотеки iostream.

Буфер ввода/вывода (**streambuf**) принадлежит базовому классу **ios**, работа с буфером описывается в дочерних классах, но все они имеют к нему доступ через общий источник – класс **ios**. Чтобы избежать двоякости с адресами этого объекта при двойном наследовании, в классы **istream** и **ostream** он наследуется виртуально.

Как следствие такой структуры, в случае с вводом и выводом операции по чтению из файла и записи в файл на диске ничем внешне не отличаются от чтения и записи в окно терминала. Благодаря общему “интерфейсу” **ios** все операторы будут перегружены похожим образом, и инструкция:

```
r_int x(1);
std::cout << x;
```

может быть заменена на другую:

```
std::ofstream out(“database.txt”);
r_int x(1);
out << x;
```

Семантика операций при этом сохраняется, что делает объекты взаимозаменяемыми. Читателю рекомендуется закончить реализацию всей иерархии рациональных чисел самостоятельно, используя код из предыдущих упражнений.

Библиография

1. Башкин, В. А., “Лямбда-выражения”, Лямбда-исчисление, ЯрГУ, Ярославль, 2018, с. 8.
2. Овчинников, А. В., “Множество, замкнутое относительно операции”, Теория групп. Лекционный курс, Москва, 2016, с. 10.
3. Степанов, А. А., “Наибольшая общая мера: последние 2500 лет” [сетевой ресурс], 2010, [посещен 24.08.2023]. Ссылка: https://www.youtube.com/watch?v=NfGeVRebiio.
4. ANSI, “ISO-IR-6: ASCII Graphic Character Set”, ANSI, December 1, 1975.
5. Brinkerhoff, D. A., “Introduction to files and I/O streams”, Streams, Object-Oriented Programming Using C++ [сетевой ресурс], Weber State University, 2015-2022, [посещен 24.08.2023]. Ссылка: https://icarus.cs.weber.edu/~dab/cs1410/textbook/14.Streams/introduction.html.
6. Cardelli, L., Wegner, P., “On Understanding Types, Data Abstraction, and Polymorphism”, Computing Surveys, Vol 17 n. 4, 1985, pp. 471-522.
7. Chu, I., “Dynamic Dispatch in Object Oriented Languages”, Concepts of Programming Languages [сетевой ресурс], Course notes, [посещен 24.08.2023]. Ссылка: https://condor.depaul.edu/ichu/csc447/notes/wk10/Dynamic2.htm.
8. Clarkson, M. R. et al., “Type Inference”, OCaml Programming: Correct + Efficient + Beautiful [сетевой ресурс], [посещен 24.08.2023]. Ссылка: https://cs3110.github.io/textbook/cover.html.
9. Cormen, T. H., et al., “Common divisors and greatest common divisors”, Number-Theoretic Algorithms, Introduction to Algorithms, 4th ed., 2022, p. 906.
10. Gill, P. E., et al., “Reference Counting”, Iotr documentation [сетевой ресурс], August 27, 2003, [посещен 24.08.2023].

Ссылка:
http://www.ccom.ucsd.edu/~peg/papers/iotrdoc/reference_counting.html.
11. Goldberg, A., Kay, A., “The Smalltalk World and Its Primitives”, Smalltalk-72 instruction manual, Palo Alto Research Center, Xerox, March, 1976, p.48.
12. Hinnant, H. E., et al., “A Brief Introduction to Rvalue References” [сетевой ресурс], June 12, 2006, [посещен 24.08.2023]. Ссылка:
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html.
13. IBM, “Restrictions on default arguments (C++ only)” [сетевой ресурс], March 23, 2021, [посещен 24.08.2023]. Ссылка:
https://www.ibm.com/docs/en/zos/2.1.0?topic=only-restrictions-default-arguments-c.
14. Knuth, D. E., “The Greatest Common Divisor”, Rational Arithmetic, Arithmetic, The Art of Computer Programming, vol. 2, 3rd ed., 1998, p. 338.
15. McJones, P., Stepanov, A. A., “Foundations. Values”, Elements of Programming, Semigroup Press, 2019, p. 2.
16. Moore, P., “Rational.hpp”, Boost Libraries Documentation [сетевой ресурс], [посещен 24.08.2023]. Ссылка:
https://www.boost.org/doc/libs/1_55_0/boost/rational.hpp.
17. Musser, D. R., Stepanov, A. A., “Generic Programming”, First International Joint Conference of ISAAC 88 and AAECC 6, 1988.
18. Polacek, M., “Understanding when not to std::move in C++” [сетевой ресурс], [посещен 24.08.2023]. Ссылка:
https://developers.redhat.com/blog/2019/04/12/understanding-when-not-to-stdmove-in-c.
19. Pomeranz, A., “Inheritance and access specifiers”, Inheritance [сетевой ресурс], Learn C++, March 24, 2022, [посещен 24.08.2023]. Ссылка:
https://www.learncpp.com/cpp-tutorial/inheritance-and-access-specifiers/.

20. Radford, M., ”C++ Interface Classes - An Introduction”, Overload 12(62) [сетевой ресурс], ACCU, August, 2004, [посещен 24.08.2023]. Ссылка: https://accu.org/journals/overload/12/62/radford_233/.
21. Ritchie, D.M., “The Development of the C Language”, History of Programming Languages, 2nd ed., ACM Press, 1996.
22. Standard C++ Foundation, “Inheritance – Multiple and Virtual Inheritance” [сетевой ресурс], Standard C++ Foundation Wiki, [посещен 24.08.2023]. Ссылка: https://isocpp.org/wiki/faq/multiple-inheritance.
23. Strachey, C., “Fundamental Concepts in Programming Languages”, Higher-Order and Symbolic Computation, 13, 2000, p. 13.
24. Stroustrup, B., Sutter, H., “C++ Core Guidelines” [сетевой ресурс], International Standardization Organization C++, 2022, [посещен 24.08.2023]. Ссылка: http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.
25. WG14 / N1256, “Types”, Concepts, Language, ISO/IEC 9899:TC3, Committee Draft, 2007, p. 33.
26. Whitney(a), T., et al., “delete operator”, Built-in operators, precedence and associativity [сетевой ресурс], C++ Language Reference, Microsoft Docs, August 3, 2021, [посещен 24.08.2023]. Ссылка: https://docs.microsoft.com/en-us/cpp/cpp/delete-operator-cpp?view=msvc-170.
27. Whitney(b), T., et al., “__interface”, Classes and Inheritance, C++ Language Reference [сетевой ресурс], Microsoft Docs, August 3, 2021, [посещен 24.08.2023]. Ссылка: https://docs.microsoft.com/en-us/cpp/cpp/interface?view=msvc-170.
28. Wirth, N., “Representation of Arrays, Records, and Sets”, Fundamental Data Structures, Algorithms and Data Structures, 1985, p. 21.

Приложения

Приложение 1.1

Требования к выполнению лабораторных работ

Лабораторные работы выполняются в аудитории, оборудованной компьютерами с установленным компилятором языка программирования **C++,** который поддерживает стандарт версии C++17. Каждая лабораторная работа рассчитана на четыре академических часа. За каждую лабораторную работу выставляются две оценки: первая оценка – за результат, полученный в ходе работы в аудитории; вторая оценка – по результатам защиты работы студентом.

Например, студент не смог получить никаких положительных результатов в ходе первого занятия, за что получил оценку “5”. Затем студент сделал лабораторную работу дома и на второе занятие пришел с законченной работой. Во время защиты получил оценку “9”. Итоговая оценка за данную лабораторную работу – “7”.

В другом сценарии студент досрочно выполнил лабораторную работу за одно занятие и защитил ее в тот же день, получив итоговую оценку “10”. На втором занятии он приступает к выполнению следующей лабораторной работы.

Процесс проверки результатов выполнения лабораторных работ проходит в три этапа. Первый этап заключается в проверке компиляции сделанного задания в специальном разделе страницы курса на платформе ELSE (либо на другой платформе – github, gitlab и т.п. – на усмотрение преподавателя). Второй этап заключается в устном опросе студента касаемо мотивации, стоящей за выбором использованных техник в ходе выполнения работы. На третьем этапе проверяется соответствие выбранных вариантов решения требованиям (асимптотичность алгоритмов, полнота интерфейса, выход за рамки ограничений и т.д.). Студент может предоставить теоретическое обоснование

либо в виде письменного отчета, либо в виде небольшой презентации в ходе практических занятий.

Критерии оценки результатов выполнения лабораторных работ различаются в зависимости от выбранного варианта. За выполнение заданий согласно вариантам “массив”, “вектор”, “односвязный список”, “очередь с приоритетом” максимальная оценка – “8”. За выполнение заданий согласно вариантам “двусвязный список”, “двухсторонняя очередь”, “стэк”, “очередь” максимальная оценка – “9”. За выполнение заданий согласно вариантам “множество”, “словарь” максимальная оценка – “10”.

За выполнение задания с использованием контейнеров (с префиксом std::), включенных в стандартные библиотеки либо в многочисленные расширения (boost, Qt и т.п.) максимальная оценка – “5”, независимо от варианта.

На усмотрение преподавателя максимальную оценку можно повысить или снизить в зависимости от сложности выбранной реализации. Например, если студент с вариантом “стэк” реализует свой контейнер как адаптер другого контейнера “двухсторонняя очередь”, который он также реализует самостоятельно, при этом двухсторонняя очередь реализуется нетривиально, сложность реализации сопоставима с реализацией варианта “словарь”, и максимальная оценка повышается до “10”.

Варианты[54]

1. Тип данных **array** (массив).
2. Тип данных **vector** (вектор).
3. Тип данных **forward list** (односвязный список).
4. Тип данных **list** (двусвязный список).
5. Тип данных **priority queue** (очередь с приоритетом).
6. Тип данных **deque** (двухсторонняя очередь).
7. Тип данных **stack** (стэк).
8. Тип данных **queue** (очередь).
9. Тип данных **set** (множество).
10. Тип данных **unordered map** (ассоциативный массив, словарь).

Array

Массив – тип данных, представляющий собой коллекцию последовательно хранящихся данных (*sequence container*). Характерной особенностью этого типа данных является использование единого блока статической памяти (*stack-allocated*) для реализации. В **C++**, как правило, массив реализуется с использованием **C**-массива заданного типа и заданного размера. Если при создании объекта этого типа размер задан как 0, адрес начала массива будет совпадать с адресом конца массива, при этом значения в начале и конце массива не определены.

Vector

Вектор – тип данных, представляющий собой коллекцию последовательно хранящихся данных (*sequence container*). Характерной особенностью этого типа данных является использование единого блока динамической памяти (*heap-allocated*) для реализации. Благодаря этому вектор ведет себя как массив с динамическим размером. Память, выделяемая

[54] В качестве источника использовался ресурс *https://en.cppreference.com/w/cpp/container*.

под нужды вектора, автоматически очищается и заново выделяется при необходимости увеличить размер вектора.

Priority queue

Очередь с приоритетом – тип данных, представляющий собой *адаптер*[55] произвольного контейнера[56], реализующий быстрый доступ (с постоянной средней асимптотичностью, ***O****(1)*) к наибольшему (или наименьшему) элементу контейнера. Это достигается с помощью внедрения специальных алгоритмов вставки новых элементов в контейнер и удаления элементов из контейнера (с логарифмической асимптотичностью, ***O****(log n)*). Также принято добавлять возможность выбирать критерий приоритета в контейнере с помощью операции сравнения в качестве необязательного аргумента (по умолчанию используется оператор “меньше”). Таким образом работа с очередью похожа на работу с “пирамидой”[57].

Forward list

Однонаправленный связный список – тип данных, представляющий собой коллекцию последовательно хранящихся данных (*sequence container*). Характерной особенностью этого типа данных является использование разрозненных блоков памяти для реализации. Это позволяет реализовать быстрые операции добавления новых элементов в список и удаления элементов из списка. С этим же связана невозможность мгновенного доступа к произвольному элементу списка. Добавление и удаление элементов списка не затрагивает итераторы остальных элементов, в отличие от массива или вектора.

List

Двунаправленный связный список – тип данных, представляющий собой коллекцию последовательно

---

[55] Адаптирует интерфейс другого существующего контейнера под свои нужды, включая другой контейнер как поле с данными в свою реализацию.

[56] В стандартной библиотеке **C++** для этой цели по умолчанию используется `std::vector`.

[57] https://neerc.ifmo.ru/wiki/index.php?title=Двоичная_куча.

хранящихся данных (*sequence container*). Характерной особенностью этого типа данных является использование разрозненных блоков памяти для реализации. Аналогично односвязному списку это позволяет реализовать быстрые операции добавления новых элементов в список и удаления элементов из списка. В отличие от односвязного списка двусвязный список может быть пройден как в прямом, так и в обратном направлении благодаря дополнительной ссылке на предыдущий элемент в каждом элементе списка.

Deque

Двусторонняя очередь (“дек”) – тип данных, представляющий собой индексированную коллекцию последовательно хранящихся данных (*sequence container*). Дек позволяет вставку новых элементов в начало и удаление элементов с конца за константное время. При этом, итераторы на элементы, оставшиеся в деке, остаются действительными. В отличие от вектора, дек не реализуется при помощи массива. Как правило, для реализации используется связный список, состоящий из статических массивов. Из этого следует, что при обращении к элементу списка по индексу производится две операции разыменования. Благодаря этому в большинстве сценариев дек обладает хорошим быстродействием как для доступа к элементам, так и для добавления (удаления) элементов.

Stack

Стек – тип данных, представляющий собой *адаптер*[58] произвольного контейнера, реализующий хранение данных по схеме “последний вошел, первый вышел” (*LIFO*). В качестве класса-адаптера (*wrapper*) стек предоставляет ограниченное число методов, так как определяющими его операциями являются добавление элементов на “верхушку” стека (начало или конец контейнера) и последующее удаление элементов с нее.

[58] В стандартной библиотеке **C++** для этой цели по умолчанию используется std::**deque**.

Queue

Очередь – тип данных, представляющий собой *адаптер*[59] произвольного контейнера, реализующий хранение данных по схеме “первый вошел, первый вышел” (*FIFO*). В качестве класса-адаптера (*wrapper*) очередь предоставляет ограниченное число методов, так как определяющими его операциями являются добавление элементов в конец очереди (начало или конец контейнера) и последующее удаление элементов из начала очереди.

Set

Множество – тип данных, представляющий собой *ассоциативный контейнер* данных, который позволяет хранить упорядоченные данные с возможностью быстрого поиска (с логарифмической асимптотичностью, ***O****(log n)*). Как правило такое множество реализуется на основании “красно-черного дерева”[60].

Unordered map

Ассоциативный массив – тип данных, представляющий собой *ассоциативный контейнер* данных, который позволяет хранить неупорядоченные пары “ключ-значение” с возможностью быстрого поиска значений по заданному ключу, быстрого добавления значений по ключу и быстрого удаления по ключу (с постоянной средней асимптотичностью, ***O****(1)*). Как правило, контейнер реализуется с помощью “хеш-таблицы”[61].

[59] В стандартной библиотеке **C++** для этой цели по умолчанию используется std::**deque**.

[60] https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево.

[61] https://neerc.ifmo.ru/wiki/index.php?title=Хеш-таблица.

Полный список кодов

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
struct rational { int p; int q; };
rational cons(const int x, const int y) {
    assert(y != 0);
    rational r;
    if(x == 0) return {0, 0};
    if(y < 0) {
        r.p = -x; r.q = -y;
    } else {
        r.p = x; r.q = y;
    }
    return r;
}
int gcd(int m, int n) {
    while(n != 0) {
        int t = m % n; m = n; n = t;
    }
    if(m < 0) m = -m;
    return m;
}
void reduce(rational &r) {
    if(r.q == 1 || r.p == 0) return;
    int d = gcd(abs(r.p), r.q);
    r.p /= d; r.q /= d;
}
char *rstr(const rational &r, char *ret) {
    int i = 0;
    auto f = [&](int n) {
        char *start = ret + i;
        while(n) {
            ret[i++] = (n % 10) + '0';
            n /= 10;
        }
        char *end = ret + i - 1;
        while(start < end) {
            char tmp = *start;
            *start = *end;
            *end = tmp;
```

```
            start++; end--;
        }
    };
    f(r.p);
    if(r.q > 1) { ret[i++] = '/'; f(r.q); }
    return ret;
}
int main() {
    rational a = cons(10, 20);
    assert(a.p == 10 && a.q == 20);
    reduce(a);
    assert(a.p == 1 && a.q == 2);
    char tmp1[100]{};
    assert(!strcmp("1/2", rstr(a, tmp1)));
    rational b = cons(100, 20);
    assert(b.p == 100 && b.q == 20);
    reduce(b);
    assert(b.p == 5 && b.q == 1);
    char tmp2[100]{};
    assert(!strcmp("5", rstr(b, tmp2)));
    rational c = cons(9, 6);
    assert(c.p == 9 && c.q == 6);
    reduce(c);
    assert(c.p == 3 && c.q == 2);
    char tmp3[100]{};
    assert(!strcmp("3/2", rstr(c, tmp3)));
    rational d = cons(12, 37);
    assert(d.p == 12 && d.q == 37);
    reduce(d);
    assert(d.p == 12 && d.q == 37);
    char tmp4[100]{};
    assert(!strcmp("12/37", rstr(d, tmp4)));
}
```

```
#include <assert.h>
class rational {
public:
    static rational reduce(rational &r) {
        if(r.q == 1 || r.p == 0) return r;
        int d = gcd(abs(r.p), r.q);
        if(d == 1) return r;
        return cons(r.p / d, r.q / d);
    }
    static rational cons(const int x, const int y) {
        assert(y != 0);
        rational r;
        if(x == 0) {
            r.p = 0; r.q = 0;
        } else if (y < 0) {
            r.p = -x; r.q = -y;
        } else {
            r.p = x; r.q = y;
        }
        return reduce(r);
    }
    void set_p(int x) { p = x; }
    void set_q(int x) { assert(x != 0); q = x; }
    const int get_p() { return p; }
    const int get_q() { return q; }
private:
    int p;
    int q;
};
int main() {
    rational a = rational::cons(10, 50);
    assert(1 == a.get_p() && 5 == a.get_q());
    a.set_p(15); a.set_q(45);
    a = rational::reduce(a);
    assert(1 == a.get_p() && 3 == a.get_q());
    rational b = rational::cons(20, 50);
    assert(2 == b.get_p() && 5 == b.get_q());
    int *p = (int *)&b;
    int *q = p + 1; *p = 5; *q = 6;
    assert(5 == b.get_p() && 6 == b.get_q());
}
```

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <utility>
class rational {
public:
    static void reduce(rational&);
    rational() = delete;
    rational(const int x, const int y) {
        assert(y != 0);
        if(y < 0) {
            p = -x; q = -y;
        } else {
            p = x; q = y;
        }
        if(x != 0) reduce(*this);
    }
    rational(const rational &other) : p(other.p), q(other.q) {}
    rational(rational &&other) :
        p(std::move(other.p)), q(std::move(other.q))
    {
        other.p = 1; other.q = 1;
    }
    rational(const char *str) {
        int __p = 1, __q = 1;
        if (sscanf(str, "%d/%d", &__p, &__q) >= 1) {
            this->p = __p; this->q = __q;
            if (this->p != 0) reduce(*this);
        }
    }
    void set_p(int);
    void set_q(int);
    const int get_p();
    const int get_q();
private:
    int p{1};
    int q{1};
};
void cons(rational *_this, const int x, const int y) {
    assert(y != 0);
    if (y < 0) {
        _this->set_p(-x); _this->set_q(-y);
    } else {
```

```
        _this->set_p(x); _this->set_q(y);
    }
    if(x != 0) rational::reduce(*_this);
}
int main() {
    rational a(2, 3);
    rational b(a);
    cons(&b, 6, 9);
    assert(2 == b.get_p() && 3 == b.get_q());
    rational c = rational(9, 2);
    assert(9 == c.get_p() && 2 == c.get_q());
    rational d(10, 50);
    assert(1 == d.get_p() && 5 == d.get_q());
    rational e("3/15");
    assert(1 == e.get_p() && 5 == e.get_q());
    rational f("315");
    assert(315 == f.get_p() && 1 == f.get_q());
    rational g(std::move(e));
    assert(1 == e.get_p() && 1 == e.get_q());
    assert(1 == g.get_p() && 5 == g.get_q());
}
```

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <utility>
class rational {
public:
    static void reduce(rational&);
    rational() = delete;
    rational(const int, const int);
    rational(const rational&);
    rational(rational&&);
    rational(const char*);
    bool eq(int __p, int __q) {
        return p == __p && q == __q;
    }
    bool eq(const rational &other) {
        return p == other.p && q == other.q;
    }
    bool gte(const rational &other) {
        if(q == other.q) return p > other.p || p == other.p;
        int c = q * other.q;
        rational x, y;
        x.p = p * other.q; x.q = c;
        y.p = other.p * q; y.q = c;
        return x.gte(y);
    }
    static bool __simplify(
        const rational &lhs, const rational &rhs,
        bool (*f)(const rational &a, const rational &b))
    {
        int c = lhs.q * rhs.q;
        rational x, y;
        x.p = lhs.p * rhs.q; x.q = c;
        y.p = rhs.p * lhs.q; y.q = c;
        return f(x, y);
    }
    static bool lte(const rational &lhs, const rational &rhs) {
        if(lhs.q == rhs.q) return lhs.p < rhs.p || lhs.p ==
rhs.p;
        return __simplify(lhs, rhs, lte);
    }
    bool ne(const rational &other) { return !eq(other); }
    bool gt(const rational &other) { return !lte(*this, other);
```

```
}
    bool lt(const rational &other) { return !gte(other); }
private:
    int p{1};
    int q{1};
};
int main() {
    rational a(2, 3);
    rational b = std::move(a);
    assert(a.eq(1, 1));
    assert(b.eq(rational(2, 3)));
    rational c(3, 4);
    assert(b.lt(c));
    assert(c.gt(b));
    rational d(c);
    assert(rational::lte(c, d));
}
```

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <utility>
int lcm(int a, int b) { return (a * b) / gcd(a, b); }
class rational {
public:
    static void reduce(rational&);
    rational() = delete;
    rational(const int, const int);
    rational(const char*);
    rational(const rational&);
    rational(rational&&);
    rational &operator=(const rational &other) {
        if(this != &other) { p = other.p; q = other.q; }
        return *this;
    }
    rational &operator=(rational &&other) {
        if(this != &other) {
            p = other.p; q = other.q;
            other.p = 1; other.q = 1;
        }
        return *this;
    }
    bool operator==(const int x) { return p == x && q == x; }
    friend bool operator==(const int lhs, const rational &rhs) {
        return rhs.p == lhs && rhs.q == lhs;
    }
    bool operator==(const rational &other) {
        return p == other.p && q == other.q;
    }
    bool operator!=(const rational &other) {
        return !(*this == other);
    }
    friend bool operator<(const rational &lhs, const rational
&rhs) {
        auto x = (long double)lhs.p / (long double)lhs.q;
        auto y = (long double)rhs.p / (long double)rhs.q;
        return x < y;
    }
    friend bool operator>(const rational &lhs, const rational
&rhs) {
        return rhs < lhs;
```

```
    }
    friend bool operator>=(const rational &lhs, const rational
&rhs) {
        return !(lhs < rhs);
    }
    friend bool operator<=(const rational &lhs, const rational
&rhs) {
        return !(rhs < lhs);
    }
    friend rational operator+(const rational &lhs, const
rational &rhs) {
        rational res; res.p = lhs.p + rhs.p; res.q = lhs.q;
        if(lhs.q != rhs.q) {
            res.q = lcm(lhs.q, rhs.q);
            res.p = lhs.p * (res.q / lhs.q) + rhs.p * (res.q /
rhs.q);
        }
        return res;
    }
private:
    int p{1}; int q{1};
};
int main() {
    rational a(2, 3);
    rational b = std::move(a);
    assert(a == 1);
    assert(1 == a);
    assert(b == rational(2, 3));
    assert(rational(2, 3) == b);
    assert(a >= b);
    rational c(3, 4);
    assert(b < c);
    assert(c > b);
    rational d(c);
    assert(c <= d);
    rational e(1, 2);
    rational f(1, 3);
    assert(rational(5, 6) == e + f);
    rational g(1, 4);
    rational h(7, 12);
    assert(h == f + g);
    rational i(5, 18);
    assert(rational(31, 36) == h + i);
}
```

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <utility>
int lcm(int, int);
class rational
{
public:
    static void reduce(rational&);
    rational() = delete;
    rational(const float, const float) = delete;
    rational(const double, const double) = delete;
    rational(const int, const int);
    rational(const char*);
    rational(const rational&);
    rational(rational&&);
    rational &operator=(const rational&);
    rational &operator=(rational&&);
    bool operator==(const int);
    friend bool operator==(const int, const rational&);
    bool operator==(const rational&);
    bool operator!=(const rational&);
    friend bool operator<(const rational&, const rational&);
    friend bool operator>(const rational&, const rational&);
    friend bool operator>=(const rational&, const rational&);
    friend bool operator<=(const rational&, const rational&);
    friend rational operator+(const rational&, const rational&);
    static void swap(rational &a, rational &b) {
        rational tmp(std::move(a));
        a = std::move(b); b = std::move(tmp);
    }
private:
    int p{1}; int q{1};
};
int main() {
    rational a(1, 2);
    assert(rational(1, 2) == a);
    rational b(2, 3);
    assert(rational(2, 3) == b);
    rational::swap(a, b);
    assert(rational(2, 3) == a);
    assert(rational(1, 2) == b);
}
```

```
#include <assert.h>
#include <stdio.h>
struct number {};
class integer : public number {
public:
    integer(long long int a) : x(a) { printf("integer: %d\n",
x); }
    long long int x{0};
};
class positive : public integer {
public:
    positive(long long int a) : integer((a < 0) ? -a : a) {
        printf("positive: %d\n", x);
    }
};
int main() {
    integer i = -1;
    assert(i.x == -1);
    positive j = -1;
    assert(j.x == 1);
}
```

```
#include <assert.h>
#include <memory>
class Shape {
public:
    virtual float area() = 0;
};
class Circle : public Shape {
public:
    Circle(float x) : r(x) {}
    float area() override { return 3.14159265359 * r; }
private:
    float r{.0};
};
class Square : public Shape {
public:
    Square(float x) : length(x) {}
    float area() override { return length * length; }
private:
    float length{.0};
};
void draw(Shape &fig, const int x) { assert(x ==
int(fig.area())); }
class positive_integer;
class negative_integer;
class number {
public:
    friend positive_integer;
    friend negative_integer;
    virtual ~number() {}
    virtual number &operator+(const number &other) = 0;
private:
    long long int x{0};
};
class integer : public number {
public:
    ~integer() {}
    virtual integer &operator+(const number &other) = 0;
};
class positive_integer : public integer {
public:
    ~positive_integer() {}
    positive_integer(long long int a) { x = (a < 0) ? -a : a; }
    positive_integer &operator+(const number &other) override {
```

```
        p = std::shared_ptr<positive_integer>(
            new positive_integer(
                x + ((other.x < 0) ? -other.x : other.x)));
        return *(p);
    }
    void assert_eq(const long long int a) { assert(a == x); }
private:
    positive_integer() {}
    std::shared_ptr<positive_integer> p;
};
positive_integer *test_scope() {
    positive_integer x(1);
    positive_integer y(-1);
    return new positive_integer(x + y);
}
int main() {
    Square s(2.);
    Circle c(0.7);
    draw(s, 4);
    draw(c, 2);
    positive_integer *z = test_scope();
    z->assert_eq(2);
}
```

```
#include <assert.h>
#include <stdio.h>
#include <vector>
template <typename T, unsigned int N>
unsigned int length(const T (&a)[N]) {
    unsigned int i = 0; for(; i < N; ++i); return i;
}
template <typename T> class vector {
public:
    ~vector() { delete[] arr; }
    vector() {}
    template <unsigned int N>
    vector(const T (&list)[N]) : length(N), arr(new T[N]) {
        for(unsigned int i = 0; i < length; ++i) arr[i] =
list[i];
    }
    vector(const unsigned int N) : length(N), arr(new T[N]) {}
    vector(const vector &other) :
        length(other.length), arr(new T[other.length])
    {
        for(unsigned int i = 0; i < length; ++i) arr[i] =
other.arr[i];
    }
    unsigned int size() { return length; }
    T *begin() { return arr; }
    T *end() { return arr + length; }
private:
    T *arr{nullptr};
    unsigned int length{0};
};
int main() {
    int arr[15]{0};
    assert(15 == length(arr));
    int x = 0; for(auto i : arr) x++;
    assert(15 == x);
    float brr[7]{0.};
    assert(7 == length(brr));
    std::vector<int> a;
    std::vector<float> b(10);
    std::vector<char> c({101, 103, 105});
    a.push_back(1);
    a.insert(a.begin(), 2);
    a.pop_back();
```

```
    a.erase(a.begin());
    for(int i = 0; i < c.size(); ++i) c[i];
    for(auto i = b.begin(); i != b.end(); ++i) *i;
    for(auto i : a) i;
    vector<int> my_vec({10, 20, 30});
    for (auto i : my_vec) printf("%d ", i);
}
```

```
#include <assert.h>
#include <iostream>
#include <variant>
class number {
private:
    long long int x{0};
};
class r_int;
class rational : public virtual number {
public:
    friend r_int;
    rational(){};
    rational(const int x) : p(x) {}
    rational(const int x, const int y) {
        assert(y != 0);
        if(y < 0) {
            p = -x; q = -y;
        } else {
            p = x; q = y;
        }
    }
    rational &operator=(const rational &other) {
        if(this != &other) { p = other.p; q = other.q; }
        return *this;
    }
    friend std::ostream &operator<<(
        std::ostream &os, const rational &rhs)
    {
        os << rhs.p << "/" << rhs.q;
        return os;
    }
private:
    int p{1}; int q{1};
};
class integer : public virtual number {
public:
    friend r_int;
    integer() {}
    integer(const int x) : a(x) {}
private:
    int a{0};
};
class r_int : public rational, public integer {
```

```
public:
    r_int() = delete;
    r_int(const int x) : rational(x, 1), integer(x) {}
    r_int(const int x, const int y) : rational(x, y), integer(x)
{}
    friend std::ostream &operator<<(std::ostream &os, const
r_int &rhs) {
        return r_int::build_os(os, rhs);
    }
private:
    static std::ostream &build_os(std::ostream &os, const r_int
&obj) {
        if(obj.q == 1) os << obj.a;
        else os << obj.p << "/" << obj.q;
        return os;
    }
};
union r_int_variant {
    r_int_variant() {}
    int a{0};
    rational r;
};
class r_int_c {
public:
    int _type; r_int_variant data;
};
#define r_int_cons(c, val) if (              \
    ((sizeof(val) == sizeof(int))            \
         ? (c._type = ((c.data.a = val), 2)) \
         : (c._type = ((c.data.r = val), 1))))
int main() {
    r_int x(1);
    std::cout << x << "\n";
    r_int y(4, 5);
    std::cout << y << "\n";
    r_int z(6, 1);
    std::cout << z << "\n";
    r_int_c u;
    r_int_cons(u, 1);
    std::cout << ((u._type == 1) ? u.data.r : u.data.a) << "\n";
    std::variant<int, rational> v(1);
    std::cout << std::get<int>(v) << "\n";
    std::variant<int, rational> w(rational(4, 5));
    std::cout << std::get<rational>(w) << "\n";
}
```

Содержание

Bun de tipar 03.01.24 | Formatul hârtiei 60x84 1/16
Hârtie ofset. Tipar RISO | Tirajul 100 ex.
Coli de tipar 9,75 | Comanda nr. 04

MD-2004, Chișinău, bd. Ștefan cel Mare și Sfânt, 168, UTM
MD-2045, Chișinău, str. Studenților, 9/9, Editura „Tehnica-UTM”