

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

ПРАКТИКУМ

по объектно-ориентированному программированию

Учебно-методическое пособие

Часть 1. Введение в классы

Озерск, 2019 г.

УДК 681.3.06
П 56

Вл. Пономарев. Практикум по объектно-ориентированному программированию. Учебно-методическое пособие. Часть 1. Введение в классы. Озерск: ОТИ НИЯУ МИФИ, 2019. — 44 с.

В пособии излагается, как выполнять практические работы по дисциплине «Объектно-ориентированное программирование».

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:

- 1.
- 2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

1. Работа ООП-101. Основы классов.....	5
1.1. Подготовка проекта.....	5
1.2. Лампочки до и после ООП.....	5
1.3. Конструктор.....	7
1.4. Идентичность и абстракция.....	9
1.5. Инкапсуляция.....	10
1.6. Конструктор с параметрами.....	11
1.7. Подготовка домашней работы.....	12
1.8. Вопросы и упражнения.....	14
2. Работа ООП-102. Интерфейс и реализация.....	15
2.1. Интерфейс и реализация класса.....	15
2.2. Подготовка проекта.....	16
2.3. Класс стека, вариант 1.....	16
2.4. Тестирование стека.....	18
2.5. Критика стека, вариант 1.....	19
2.6. Класс стека, вариант 2.....	19
2.7. Критика стека, вариант 2.....	22
2.8. Класс стека, вариант 3.....	22
2.9. Вопросы и упражнения.....	23
3. Работа ООП-103. Статические и константные элементы классов.....	24
3.1. Подготовка проекта.....	24
3.2. Статические элементы данных класса.....	24
3.3. Статические функции-элементы.....	26
3.4. Классы статических функций.....	27
3.5. Константные объекты и элементы класса.....	28
3.6. Вопросы и упражнения.....	30
4. Работа ООП-104. Друзья классов.....	31
4.1. Подготовка проекта.....	31
4.2. Друзья класса.....	31
4.3. Сильно инкапсулированные объекты.....	31
4.4. Дружественные функции.....	33
4.5. Перегрузка операций.....	34
4.6. Дружественные классы.....	36
4.7. Вопросы и упражнения.....	37
5. Работа ООП-105. Деструктор и специальные функции класса.....	38
5.1. Подготовка проекта.....	38
5.2. Деструктор.....	38
5.3. Операция присваивания.....	40
5.4. Конструктор копии.....	41
5.5. Подготовка домашней работы.....	43
Вопросы и упражнения.....	44

Общие цели занятий

В ходе практических работ изучаются основы использования классов в рамках методологии объектно-ориентированного программирования.

В этой части работ рассматриваются следующие темы:

- 1) абстракция объектов предметной области;
- 2) инкапсуляция свойств;
- 3) внедрение операций;
- 4) создание и использование объектов;
- 5) статические и константные элементы классов;
- 6) друзья классов;
- 7) деструктор и другие специальные функции классов.

К практическим работам приписаны контрольные вопросы и упражнения. Контрольные вопросы могут быть заданы преподавателем в ходе защиты работы, однако преподаватель может задавать и другие вопросы, не указанные в списке.

Для защиты знаний и навыков, полученных в ходе выполнения практических работ студент должен иметь тетрадь (12-18 листов). Для каждой выполненной работы в тетрадь записывается отчет.

Отчет по работе начинается с заголовка:

1. Фамилия Имя Отчество
2. Группа
3. Дата начала выполнения работы
4. Код работы
5. Название работы
6. Цели работы
7. Задачи работы

При необходимости при выполнении работы в отчет записываются контрольные значения.

Во время защиты тетрадь используется для вопросов преподавателя и ответов студента.

1. Работа ООП-101. Основы классов

Цели:

- описание классов простых объектов.

Задачи:

- абстракция объектов предметной области;
- инкапсуляция свойств;
- добавление операций;
- создание и использование объектов.

1.1. Подготовка проекта

Скачайте архив работы ООР100. Извлеките каталог ООР100 из архива в корневой каталог диска C:. Переименуйте каталог в ООР101. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Модуль main.cpp содержит основную функцию. Укажите в начале этого модуля сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1по-00д
// Студент Имя Отчество
// Объектно-ориентированное программирование
// ООР-101. Основы классы
// 01.01.2000
```

Модуль class.h предназначен для описания классов.

1.2. Лампочки до и после ООП

Начнем с классического примера «Лампочка», но сначала без ООП.

Пусть нам нужно описать множество лампочек, которые можно только включать или выключать.

Если у нас нет средств ООП, то уж наверняка есть средства создания пользовательского типа данных, в терминологии языка Си — структура данных struct. Модуль class.h:

```
// class.h
#pragma once
. . .
// структура лампочки
struct lamp_a {
    int turnedon;
};
```

Модуль main.cpp.

Описываем здесь создание лампочки (одной, хотя и нет никаких препятствий создать множество их):

```

01 int main() {
02     lamp_a a;
03     a.turnedon = 0;
04     cout << "lamp_a: " << a.turnedon << endl;
05 }

```

Ничего нового пока, это просто структура данных.

В строке 2 создаем объект типа `lamp_a`, в строке 3 устанавливаем начальное значение элемента данных (лампочка выключена), в строке 4 выводим в консоль значение этого элемента.

Попробуем создать класс, описывающий лампочку.

Модуль `class.h`:

```

// класс лампочки
struct lamp_b {
    int turnedon;
};

```

Я бы сказал, никакой разницы. Вроде это и не класс, а структура.

На самом деле это и есть структура, поскольку функций-элементов в ней нет. Добавим функцию, которая будет инициализировать лампочку в выключенное состояние:

```

// класс лампочки
struct lamp_b {
    int turnedon;
    void init() {
        turnedon = 0;
    }
};

```

Соответственно, в функции `main` добавим еще одну лампочку:

```

01 int main() {
02     lamp_a a;
03     a.turnedon = 0;
04     cout << "lamp_a: " << a.turnedon << endl;
05     lamp_b b;
06     b.init();
07     cout << "lamp_b: " << b.turnedon << endl;
08 }

```

Интересно, конечно. И в чем смысл?

Ну вот, например.

Есть у нас элемент данных, который мы назвали `turnedon` (*включена*).

То есть, когда мы пишем программу, мы можем написать только следующие предложения:

```

лампочка.включена присвоить ноль
лампочка.включена присвоить один
если лампочка.включена что-то-сделаем
если не лампочка.включена что-то-сделаем

```

Класс же позволит написать другие предложения, более понятные, используя глаголы, а не только существительные, например:

```
лампочка. ВЫКЛЮЧИТЬ  
лампочка. ВКЛЮЧИТЬ  
ЕСЛИ лампочка. ВКЛЮЧЕНА что-то-сделаем  
ЕСЛИ лампочка. ВЫКЛЮЧЕНА что-то-сделаем
```

Модуль class.h.

Описываем новый вариант лампочки:

```
// класс лампочки  
class lamp_c {  
    // текущее состояние  
    int state;  
public:  
    // инициализирует  
    void init() {  
        state = 0;  
    }  
    // выключает  
    void off() {  
        state = 0;  
    }  
    // включает  
    void on() {  
        state = 1;  
    }  
    // выключена?  
    int is_off() {  
        return (state == 0);  
    }  
    // включена?  
    int is_on() {  
        return (state != 0);  
    }  
};
```

Мы добавили в новый класс четыре функции, чтобы сделать семантику работы с объектом лампочки более понятной. При этом элемент данных получил у нас тоже более понятное название — «состояние». Это элемент данных никому не виден, поскольку находится в закрытой секции класса. В открытой же секции только функции, определяющие семантику.

Пробуем использовать новую лампочку, основная функция:

```
01 int main() {  
02     . . .  
08     lamp_c c;  
09     c.init();  
10     cout << "lamp_c turned-on: " << c.is_on() << endl;  
11     c.on();  
12     cout << "lamp_c turned-on: " << c.is_on() << endl;  
13 }
```

1.3. Конструктор

Все хорошо с классом лампочки, вот только всегда нужно помнить о том, что после создания лампочку нужно инициализировать функцией `init`.

На самом деле в классах есть несколько специальных функций, которые вызываются автоматически.

Одна из таких функций называется *конструктором*, потому что она вызывается автоматически в момент создания объекта. В C++ конструктор описывается специальным образом.

Во-первых, название функции конструктора должно в точности совпадать с именем класса. Во-вторых, функция конструктора не может возвращать никакого значения, и при этом не может быть описана, как функция, которая возвращает `int`.

Напишем новый класс лампочки `d`, в котором роль функции `init` будет выполнять конструктор. Модуль `class.h`:

```
// класс лампочки
class lamp_d {
    // текущее состояние
    int state;
public:
    // конструктор
    lamp_d() {
        state = 0;
    }
    // выключает
    void off() {
        state = 0;
    }
    // включает
    void on() {
        state = 1;
    }
    // выключена?
    int is_off() {
        return state == 0;
    }
    // включена?
    int is_on() {
        return state != 0;
    }
};
```

Соответственно, пробуем использовать объект этого класса.

Модуль `main.cpp`, основная функция:

```
01 int main() {
02     . . .
13     lamp_d d;
14     cout << "lamp_d turned-on: " << d.is_on() << endl;
15     d.on();
16     cout << "lamp_d turned-on: " << d.is_on() << endl;
17 }
```

Убеждаемся, что лампочка инициализируется должным образом, отслеживая выполнение кода построчно, с заходом в конструктор.

1.4. Идентичность и абстракция

Рассмотрим два одинаковых стакана.

Они одинаковые потому, что обладают похожими характеристиками, такими, как форма, размер или вес. Но на самом деле двух одинаковых стаканов не бывает. Всегда найдется какое-либо отличие в форме или размере, может быть не очень значительное. С этой точки зрения два стакана обладают идентичностью, то есть уникальностью.

При помощи абстракции выявляются общие характеристики объектов одного типа (или вида), а идентичность позволяет отделить один объект от другого.

Абстракция лежит в основе объектно-ориентированного подхода.

В результате абстракции формируется абстрактный тип данных, который и называют классом. Экземпляры классов, называемые объектами, обладают идентичностью, но не в смысле одинаковости, а в смысле отделения одного объекта от другого.

В объектно-ориентированном программировании основной задачей является выявление абстракций и определение их взаимодействия.

Абстрагирование, — это процесс отделения существенных характеристик предмета или явления от несущественных его характеристик для данной предметной области.

Возьмем ту же лампочку.

До сих пор мы полагали, что единственная характеристика лампочки, которая существенна, это состояние «включена» или «выключена». Это потому, что предметная область не была нами уточнена. Если же уточнить предметную область, то может оказаться, что некоторые другие характеристики лампочки имеют значение.

Прежде попробуем перечислить возможные характеристики.

Это мощность, напряжение, форма, цоколь, размер, вес, тип и т.д.

Предположим, что лампочки нам нужны для хранения на складе.

Какие характеристики нужны при этом?

Вероятно, имеют значение размер и вес, в то время как мощность или тип нас в данном случае не интересуют.

Если же мы предполагаем учитывать лампочки для продажи, то, вероятно, мощность, цоколь и тип будут иметь значение, а вес неинтересен.

Получается, что в разных случаях абстрагирование будет приводить к формированию разных абстрактных типов данных.

После формирования абстрактного типа данных, или класса, можно использовать его для формирования конкретных экземпляров. При этом два совершенно одинаковых экземпляра будут обладать идентичностью в том смысле, что будут занимать в памяти каждый свое место.

1.5. Инкапсуляция

Одним из важнейших свойств объектно-ориентированного подхода является *инкапсуляция*. Она обозначает заключение элементов данных класса в закрытую секцию (*in capsule*).

При этом элементы данных становятся недоступными для использования, и управлять ими можно только при помощи открытых методов.

Это дает возможность поддерживать целостность класса, то есть такое состояние объектов, в котором все элементы данных в любой момент времени имеют только допустимые значения.

Если какой-то элемент данных инкапсулируется, то для управления им требуется внедрить в класс одну или две функции-элемента.

Рассмотрим пример абстрактного типа данных «Лампочка». Будем учитывать следующие характеристики лампочки: мощность и напряжение.

Модуль class.h.

Сначала опишем перечисление возможных мощностей и напряжений:

```
// перечисление мощностей
enum powers { P10 = 10, P15 = 15, P20 = 20 };
// перечисление напряжений
enum voltages { V12 = 12, V36 = 36, V220 = 220 };
```

Теперь будем описывать класс лампочки e:

```
// класс лампочки
class lamp_e {
    // мощность
    powers power;
    // напряжение
    voltages voltage;
public:
    // конструктор
    lamp_e() {
        power = P10;
        voltage = V12;
    }
};
```

Поскольку мы инкапсулировали два элемента данных, требуется четыре функции для управления ими.

Добавляем в класс функции для управления значением мощности:

```
class lamp_e {
    . . .
    // возвращает мощность
    powers get_power() {
        return power;
    }
    // устанавливает мощность
    void set_power(powers newValue) {
        power = newValue;
    }
};
```

Переходим в основную функцию и создаем новые лампочки e1 и e2:

```
int main() {
    . . .
    lamp_e e1, e2;
    e1.set_power(P10);
    e2.set_power(P20);
    cout << "e1 power = " << e1.get_power() << endl;
    cout << "e2 power = " << e2.get_power() << endl;
}
```

Аналогичным образом описываем функции для управления значением напряжения. Делаем это самостоятельно. Далее задаем значения напряжений для лампочек e1 и e2, выводим значения в консоль.

1.6. Конструктор с параметрами

Замечательные лампочки e, но есть у них недостаток, если заметили.

После создания лампочки можно установить любые значения мощности и напряжения, а впоследствии легко изменить начальные значения на любые другие. Правильно это или нет, нужно решать в каждом конкретном случае предметной области отдельно.

Мне кажется, что так делать нельзя. Поэтому я думаю, что класс лампочки не должен давать возможность изменить заданное при инициализации значение, и сделать это можно при помощи конструктора с параметрами и ограничением доступа к закрытым элементам.

Модуль class.h.

Описываем класс лампочки f:

```
// класс лампочки
class lamp_f {
    // мощность
    powers _power;
    // напряжение
    voltages _voltage;
public:
    // конструктор
    lamp_f(powers p, voltages v) {
        _power = p;
        _voltage = v;
    }
    // возвращает мощность
    powers power() {
        return _power;
    }
    // возвращает напряжение
    voltages voltage() {
        return _voltage;
    }
};
```

В основной функции создаем новые лампочки:

```

int main() {
    . . .
    lamp_f f1(P10, V220), f2(P20, V36);
    cout << "f1 power = " << f1.power()
         << " voltage = " << f1.voltage() << endl;
    cout << "f2 power = " << f2.power()
         << " voltage = " << f2.voltage() << endl;
}

```

Обратим внимание, что при объявлении новых лампочек f1 и f2 требуется обязательно указывать мощность и напряжение, при этом срабатывает конструктор. Нельзя создать лампочку типа f, не указывая характеристики. Обязательно попробуйте это:

```

int main() {
    . . .
    lamp_f f3;
}

```

Компилятор не должен разрешать делать это и сообщать о том, что нельзя создать объект, поскольку для типа lamp_f не определен конструктор по умолчанию (*default constructor*). Конструктором по умолчанию называется конструктор без параметров. Он позволяет компилятору создавать объекты самостоятельно при необходимости.

1.7. Подготовка домашней работы

Скачайте архив домашней работы OORHW1. Извлеките из архива каталог OORHW1 в корневой каталог диска C:. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

В проекте один модуль, main.cpp. В этом модуле выполняется тестирование функций класса. Выясните у преподавателя, класс какого объекта вам задан. Варианты приведены в следующей таблице.

Вариант	Объект	Имя-класса
1	положительное число	pos
2	натуральное число	nat
3	отрицательное число	neg
4	рациональное число	rat
5	комплексное число	complex
6	число Фибоначчи	fibs
7		
8		

Далее предполагается, что задан вариант 1, класс pos. Добавьте в проект модули имя-класса.h и имя-класса.cpp. Для класса pos это модули pos.h и pos.cpp. Укажите в начале каждого модуля сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Фмилия Имя Отчество
// Объектно-ориентированное программирование
// ООРНВ1. Перегрузка операций
// 01.01.2000
```

Включите модуль имя-класса.h в основной модуль main.cpp и в модуль имя-класса.cpp:

```
#include "pos.h"
```

В модуле .h опишите класс и конструктор по умолчанию:

```
class pos {
    // хранитель значения
    int num;
public:
    // конструктор по умолчанию
    pos();
};
```

В модуле .cpp определите конструктор по умолчанию:

```
// конструктор по умолчанию
pos::pos() : num(0) {}
```

В интерфейсе класса опишите метод, возвращающий целое значение:

```
class pos {
    // хранитель значения
    int num;
public:
    // конструктор по умолчанию
    pos();
    // возвращает целое значение
    int intVal();
};
```

В реализации класса опишите реализацию метода:

```
// возвращает целое значение
int pos::intVal() {
    return num;
}
```

Заметим, что для классов `rat` и `complex` в этом случае также требуется вывести целочисленное значение, даже если оно не соответствует истинному значению объекта. Используйте, например, приведение числителя.

В основной функции объявите объект и выведите его значение:

```
int main() {
    pos a;
    cout << "a.intVal() = " << a.intVal() << endl;
}
```

В интерфейсе класса опишите конструктор приведения или преобразования (конструктор с одним параметром, — конструктор приведения, с несколькими параметрами, — конструктор преобразования):

```

class pos {
    . . .
public:
    // конструктор по умолчанию
    pos();
    // конструктор приведения
    pos(int Value);
    // возвращает целое значение
    int intVal();
};

```

Соответственно, в реализации класса опишите этот конструктор:

```

// конструктор приведения
pos::pos(int Value) {
    num = Value;
}

```

В интерфейсе класса опишите операцию ++ (инкремент):

```

// операция инкремент (префиксная)
pos operator ++ (void);
// операция инкремент (постфиксная)
pos operator ++ (int);

```

В реализации класса опишите реализацию операции «инкремент»:

```

// операция инкремента префиксная
pos pos::operator ++ (void) {
    return pos(++num);
}

// операция инкремента постфиксная
pos pos::operator ++ (int) {
    return pos(num++);
}

```

В основной функции протестируйте новые операции:

```

cout << "a++ = " << (a++).intVal() << endl; // 0
cout << "++a = " << (++a).intVal() << endl; // 2

```

Трассируйте программу с заходом во все функции класса. Вам нужно разобраться с префиксной и постфиксной формой операции ++. Читайте литературу. При необходимости сохраните проект на съемный диск.

1.8. Вопросы и упражнения

1. Дайте определение понятиям «класс» и «объект класса».
2. В чем заключается абстрагирование при конструировании абстрактного типа данных?
3. В чем заключается «идентичность» экземпляра класса?
4. В чем заключается инкапсуляция в ООП?
5. Дайте определение конструктору класса.
6. Сконструируйте класс, описывающий точку на плоскости.
7. Сконструируйте класс, описывающий кошку или собаку.

2. Работа ООП-102. Интерфейс и реализация

Цели:

- исследование неизменности интерфейса;

Задачи:

- формирование классов стека с одинаковым открытым интерфейсом;
- формирование класса стека с альтернативным интерфейсом.

2.1. Интерфейс и реализация класса

Одна из замечательных особенностей класса заключается в том, что два класса являются одинаковыми до тех пор, пока они имеют одинаковый интерфейс.

Понятие «интерфейс» многократно перегружено, поэтому при его применении почти всегда нужно оговариваться, что имеется в виду.

Здесь под интерфейсом класса понимается все, что записано в определении класса, то есть текст от открывающей после `struct` или `class` скобки до закрывающей скобки.

Вот пример интерфейса класса:

```
class cat {  
    // мяу  
    int meow();  
};
```

Два класса будут считаться одинаковыми, если в точности совпадают описания всех открытых (на самом деле также и защищенных) элементов.

Это означает, что два таких класса можно использовать в программе без какой-либо ее модификации. (При этом также неявно предполагается, что не закрытые функции-элементы имеют одну и ту же семантику).

В связи с этим можно говорить о совпадении открытых интерфейсов.

Заметим, что закрытые элементы класса C++ не оказывают никакого влияния на использование класса, но в C++ они видны, и с этим ничего не поделать.

Под реализацией класса в C++ понимается текст функций-элементов класса. Этот текст в C++ можно включить в интерфейс, но обычно так не поступают, а помещают этот текст в отдельный файл. Это позволяет скрыть реализацию от потенциального пользователя. Никакого другого смысла в этом нет.

Как поступить в конкретном случае, зависит от цели класса. Если вы не предполагаете распространять его, то реализацию можно описывать непосредственно в интерфейсе.

2.2. Подготовка проекта

Скачайте архив работы OOP100. Извлеките каталог OOP100 из архива в корневой каталог диска C:. Переименуйте каталог в OOP102. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Укажите в начале модуля main.cpp сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Студент Имя Отчество
// Объектно-ориентированное программирование
// OOP-102. Интерфейс и реализация
// 01.01.2000
```

Добавьте в проект три новых заголовочных модуля: stack1.h, stack2.h и stack3.h — классы стека, интерфейс и реализация.

Включите эти модули в модуль main.cpp:

```
#include "class.h"
#include "stack1.h"
// #include "stack2.h"
// #include "stack3.h"
```

2.3. Класс стека, вариант 1

Стеки очень часто используются в программах. Поэтому имеет смысл иметь под рукой подходящий класс, описывающий стек. Для простоты будем разрабатывать стек, хранящий целые числа типа int.

Модуль stack1.h.

Начинаем с того, что описываем класс стека:

```
// stack1.h
#pragma once
#include <assert.h>

// класс стека целых чисел
class stack {
    // счетчик элементов
    int size;
public:
    // конструктор по умолчанию
    stack();
};
```

Мы пока не знаем, из чего состоит интерфейс, но:

- количество элементов в стеке потребуется,
- конструктор по умолчанию лучше всегда иметь.

Поэтому именно так начинается наш класс. Дальше начинаем думать об операциях. Две операции должны быть определены для стека:

- проталкивание элемента в стек;
- выталкивание элемента с вершины стека.

Часто требуется также знать, какой элемент находится на вершине стека, а также, сколько в стеке элементов.

Дальше думаем, как операции назвать. Со стеком все просто:

- проталкивание — push,
- выталкивание — pop,
- элемент на вершине — top.
- количество элементов в стеке — count.

Поэтому дальше описываем операции стека:

```
class stack {  
    . . .  
public:  
    . . .  
    // проталкивание  
    void push(int element);  
    // выталкивание  
    int pop();  
    // элемент на вершине  
    int top();  
    // количество элементов  
    int count();  
};
```

На самом деле мы определили открытый интерфейс стека, состоящий из четырех методов. А дальше нужно думать над реализацией.

Чем плох стек, так это тем, что трудно во всех случаях предвидеть максимальную его глубину. Однако часто в программах заранее известно, что в стек будет помещаться максимум N элементов. Рассчитывая на эту определенность, можно очень просто реализовать стек в виде массива соответствующих структур. Тогда нам нужна константа, задающая максимальный размер стека:

```
// stack1.h  
#pragma once  
#include <assert.h>  
#define MAX_STACK_SIZE 10
```

Для определенности мы взяли число 10, с таким же успехом его можно поменять на 16384, например, но при большом количестве элементов могут возникнуть сложности.

Теперь можно определить массив, который будет хранить значения:

```
// класс стека целых чисел  
class stack {  
    // массив стека  
    int st[MAX_STACK_SIZE];  
    // счетчик элементов  
    int size;  
public:  
    . . .  
};
```

Дело остается за реализацией.

Размещаем реализацию ниже интерфейса в модуле `stack1.h`.

Конструктор должен обнулять размер стека:

```
// конструктор по умолчанию
stack::stack() {
    size = 0;
}
```

Метод, возвращающий количество элементов, тривиален:

```
// количество элементов
int stack::count() {
    return size;
}
```

При проталкивании элемента важно не превысить размер массива.

Для исключения такой ситуации используем функцию `assert`.

Если аргумент функции ложный, она останавливает программу и выбрасывает ее после того, как будет закрыто сообщение об условии, которое вызвало остановку:

```
// проталкивание
int stack::push(int element) {
    assert(size < MAX_STACK_SIZE);
    st[size++] = element;
}
```

При выталкивании может возникнуть ситуация, когда мы попытаемся вытолкнуть элемент из пустого стека.

Опять воспользуемся `assert`:

```
// выталкивание
int stack::pop() {
    assert(size > 0);
    return st[--size];
}
```

Получение элемента на вершине также проверяет наличие элементов:

```
// элемент на вершине
int stack::top() {
    assert(size > 0);
    return st[size - 1];
}
```

Собственно, это весь класс.

2.4. Тестирование стека

Теперь, когда класс стека готов, нужно тщательно протестировать его.

Переходим в модуль `main.cpp`, объявляем в основной функции переменную стека `s`:

```
stack s;
```

Если ошибок не возникает, то пока все хорошо.

Далее мы будем наполнять стек значениями от нуля до максимально возможного, сделать это проще в цикле, поэтому нужна константа, задающая максимальное количество операций проталкивания:

```
// main.cpp
. . .

// максимальное количество проталкиваний в стек
#define MAX_PUSH 11
```

Мы намеренно взяли количество проталкиваний на единицу больше, чем размер стека. Описываем проталкивания в стек в цикле:

```
int main(int) {
    stack s;
    for (int i = 0; i < MAX_PUSH; i++) {
        s.push(i);
        printf("top=%d\tsize=%d\n", s.top(), s.count());
    }
}
```

Запускаем программу Ctrl+F5 и убеждаемся, что 10 элементов проталкиваются, а 11-й элемент вызывает остановку.

Полезно также выполнять программу пошагово, чтобы удостовериться в исполнении правильных действий.

2.5. Критика стека, вариант 1

Очевидных недостатков стека версии 1 два:

- ограниченная глубина;
- жестко заданный тип хранимых значений.

Неочевидный недостаток, — операции со стеком могут вызывать остановку программы.

Увеличить глубину стека можно, если использовать иной подход к реализации. Что мы и попытаемся сделать.

2.6. Класс стека, вариант 2

Для реализации другого варианта стека перейдем в модуль stack2.h.

Нужно понимать, что программный стек бесконечного размера не удастся разработать никаким способом. Однако, если мы хотим получить стек с неограниченной глубиной, то есть такие варианты реализации:

- использовать связные структуры данных;
- использовать динамический массив.

Оба варианта используют для хранения элементов стека динамически выделяемую из кучи память, которая представляется достаточно большой.

Мы будем использовать первый вариант, — связные структуры.

Нам нужно обеспечить эквивалентность открытого интерфейса, поэтому мы описываем интерфейс, совпадающий со стеком варианта 1:

```

// класс стека целых чисел
class stack {
    // счетчик элементов
    int size;
public:
    // конструктор по умолчанию
    stack();
    // проталкивание
    void push(int element);
    // выталкивание
    int pop();
    // элемент на вершине
    int top();
    // количество элементов
    int count();
};

```

Если мы теперь переключим включаемый стек на этот вариант, программа должна оставаться правильной, что и требуется:

```

#include "class.h"
// #include "stack1.h"
#include "stack2.h"
// #include "stack3.h"

```

А далее мы разрабатываем элемент стека в виде внутреннего класса:

```

class stack {
    struct stack_el {
        // значение элемента
        int value;
        // указатель на следующий элемент
        stack_el * next;
        // конструктор
        stack_el(int element) : value(element), next(0) {}
    };
};

```

Класс элемента стека определен полностью в своем интерфейсе.

Заметим, что в конструкторе мы использовали список инициализации, следующий за двоеточием. Такой способ инициализации элементов данных немного ускоряет конструирование объекта.

Поскольку стек будет представлять собой односвязный список, нужен элемент, указывающий на начало:

```

class stack {
    . . .
    // первый элемент
    stack_el * home;
    // счетчик элементов
    int size;
    . . .
};

```

Приступаем к реализации функций-элементов.

Размещаем их в модуле stack2.h ниже интерфейса.

Конструктор обнуляет элементы данных home и size:

```
// конструктор по умолчанию
stack::stack() {
    home = 0;
    size = 0;
}
```

Операция проталкивания выполняется теперь следующим образом. Сначала создаем новый элемент стека при помощи генератора `new`. Далее проверяем, что генератор вернул указатель на новый элемент, и переполнения памяти не произошло. Затем формируем цепочку списка в зависимости от того, создается первый или последующий элемент:

```
// проталкивание
void stack::push(int element) {
    stack_el * e = 0;
    e = new stack_el(element);
    assert(e);
    if (size > 0) e->next = home;
    home = e;
    size++;
}
```

Если элемент первый, то `home` указывает на него, иначе `home`, — это последующий элемент списка (рисунок 1).

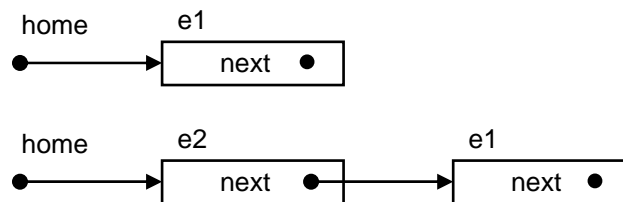


Рисунок 1 — Формирование цепочки

Переходим к операции выталкивания.

Сначала, как и в стеке 1, удостоверяемся, что стек не пуст.

Затем создаем дополнительный указатель `e` на первый элемент `home`.

Далее выполняем переключение указателя `home` на второй элемент, и уничтожаем первый элемент, на который указывает теперь указатель `e`. По ходу дела запоминаем выталкиваемое значение `value`:

```
// выталкивание
int stack::pop() {
    assert(size > 0);
    int value = home->value;
    stack_el * e = home;
    home = home->next;
    delete e;
    size--;
    return value;
}
```

В этот момент наша программа становится полностью эквивалентной предыдущей, когда использовался стек 1.

Реализация функций-элементов, возвращающих значение на вершине и количество элементов, тривиальна.

Делаем это самостоятельно.

Не забываем про `assert` в функции `top`.

2.7. Критика стека, вариант 2

По-прежнему остается тот недостаток, что программа может быть остановлена функцией `assert`.

Однако хочу заметить, что этого не должно происходить в правильно написанной программе. Например, можно просто проверять количество элементов на стеке перед операцией выталкивания. Кроме того, функция `assert` может сработать и помимо вашей воли, поскольку она используется в коде подключаемых библиотек. Ну и, наконец, программа в принципе может быть остановлена. Это же не повод не писать программы.

Хочется заметить также, что в данной реализации вообще не нужен счетчик элементов, поскольку он не несет никакой полезной нагрузки. Поэтому, если в программе, использующей данный стек, знать количество элементов не требуется, то вместо функции-элемента `count` лучше использовать функцию-элемент `empty`, которая возвращает 1, если стек пуст, и 0 в противном случае. Определяется это по значению элемента `home`:

```
// возвращает 1 если стек пуст
int stack::empty() {
    return (home == 0);
}
```

2.8. Класс стека, вариант 3

Лично мне больше нравится стек вариант 1. Он очень простой, самый простой, а простота, — залог надежности.

Тем не менее, если думать о стеке, который не создает проблем своей работой, то есть и другие варианты.

При этом интерфейс стека должен быть другим, например, таким:

```
// класс стека целых чисел
class stack {
public:
    // конструктор по умолчанию
    stack();
    // операция проталкивания
    int push(int element);
    // операция выталкивания
    int pop(int & element);
    // элемент на вершине
    int top(int & element);
    // количество элементов
    int count();
};
```

Отличительная особенность функций-элементов, описывающих операции стека в этом случае заключается в том, что возвращаемым значением функций является признак успешности операции, а исключительные ситуации обрабатываются самим стеком.

Вот, например, как это происходит при выталкивании:

```
int stack::pop(int & element) {  
    if (size == 0) return 0;  
    element = st[--size];  
    return 1;  
}
```

При использовании этого стека программа должна проверять результат выполнения каждой операции. Заметим, что в этом случае полезно определить константы возвращаемых значений.

Реализацию и тестирование стека 3 выполняем самостоятельно.

В заключение.

Все рассмотренные варианты стека нацелены на использование целого типа данных `int`. Это, разумеется, недостаток. Если нам потребуется проталкивать в стек, например, структуру данных, то класс придется переделывать, меняя тип `int` на другой тип во множестве мест.

На данном этапе изучения предмета мы не можем изменить эту ситуацию, однако решение существует, и мы его используем в последующих работах.

2.9. Вопросы и упражнения

1. Что называется интерфейсом класса?
2. Что называется реализацией класса?
3. Зачем реализацию отделяют от интерфейса?
4. Что называется списком инициализации?
5. Поясните различие между конструкторами:
 - конструктором по умолчанию,
 - конструктором приведения и
 - конструктором преобразования.
6. Как работает функция `assert`?
7. Сконструируйте как минимум один из классов:
 - класс `list`, реализующий список,
 - класс `dequeue`, описывающий дек,
 - класс `set`, реализующий множество.

3. Работа ООП-103. Статические и константные элементы классов

Цели:

- исследование статических и константных элементов класса.

Задачи:

- исследование статических элементов данных;
- исследование статических функций-элементов;
- исследование константных объектов и элементов класса.

3.1. Подготовка проекта

Скачайте архив работы ООР100. Извлеките каталог ООР100 из архива в корневой каталог диска C:. Переименуйте каталог в ООР103. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Укажите в начале модуля main.cpp сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1по-00д
// Студент Имя Отчество
// Объектно-ориентированное программирование
// ООР-103. Статические и константные элементы классов
// 01.01.2000
```

3.2. Статические элементы данных класса

Статическим называется элемент класса, объявленный с ключевым словом static. Статические элементы существуют независимо от наличия экземпляров класса и принадлежат классу в целом.

В языке C++ память для статического элемента данных должна быть выделена вне описания класса, но в пределах видимости модуля (файла).

Статический элемент данных виден любому объекту и может быть им изменен. В отсутствие экземпляров класса статический элемент данных доступен через операцию разрешения видимости класса.

Модуль class.h.

Описываем класс со статическим элементом данных:

```
class lamp {
public:
    // счетчик объектов
    static int counter;
};

// определение статического элемента данных
// здесь выделяется память для него
// и задается начальное значение
int lamp::counter = 0;
```

Мы намеренно все элементы поместили в открытую секцию.

Переходим в основную функцию main.

Сначала убеждаемся, что статический элемент доступен без создания экземпляров, то есть существует сам по себе:

```
// основная функция
int main(int) {
    cout << "lamp::counter=" << lamp::counter << endl;
}
```

Программа выводит значение 0.

Его можно также и изменять:

```
int main(void) {
    lamp::counter = 1;
    cout << "lamp::counter=" << lamp::counter << endl;
}
```

Программа выводит значение 1.

Статический элемент доступен не только при помощи операции разрешения видимости класса `class::`, но и через любой экземпляр класса:

```
int main(int) {
    lamp::counter = 1;
    cout << "lamp::counter=" << lamp::counter << endl;
    lamp x;
    cout << "x.counter = " << x.counter << endl;
}
```

Хотя такой способ доступа к статическому элементу данных возможен, он не рекомендуется. Способ с использованием операции разрешения видимости класса предпочтительнее, потому что он явно показывает, что элемент данных является единственным.

Статический элемент сейчас доступен потому, что он объявлен в открытой секции класса. Перенесем этот элемент в закрытую секцию:

```
class lamp {
    // счетчик объектов
    static int counter;
public:
};
```

Теперь статический элемент недоступен, о чем компилятор должен предупреждать.

Доопределим класс следующим образом:

```
class lamp {
    // счетчик объектов
    static int counter;
public:
    // идентификатор объекта
    int id;
    // конструктор по умолчанию
    lamp();
};
```

У лампочки есть только идентификатор.

Конструктор по умолчанию:

```
// конструктор по умолчанию
lamp::lamp() {
    id = ++counter;
}
```

В основной функции создаем объекты, предварительно поместив в комментарий имеющийся код:

```
lamp a, b, c;
cout << "a.id = " << a.id << endl;
cout << "b.id = " << b.id << endl;
cout << "c.id = " << c.id << endl;
```

Наблюдаем, как создаются идентификаторы при помощи счетчика, последовательно заходя в конструктор три раза подряд.

3.3. Статические функции-элементы

Статические функции-элементы класса подобны статическим элементам данных в смысле способа доступа к ним (их вызова).

Но статические функции-элементы отличаются от обычных функций-элементов тем, что им не передается указатель на объект, даже если они вызываются через представителя класса.

Поэтому статическим функциям-элементам недоступны нестатические элементы данных, так как они идентичны в смысле принадлежности к конкретным и разным объектам. Поэтому же в коде статической функции-элемента доступны только статические функции-элементы, — статическая функция не может передать нестатической функции указатель на объект.

Для доступа к закрытому статическому элементу данных можно использовать статические функции, которые мы дописываем в класс:

```
class lamp {
    . . .
    // устанавливает статический счетчик
    static void put_counter(int newValue) {
        counter = newValue;
    }
    // возвращает статический счетчик
    static int get_counter(void) {
        return counter;
    }
};
```

В основной функции проверяем действие статических функций:

```
cout << "lamp::get_counter = " << lamp::get_counter() << endl;
cout << "    a::get_counter = " << a::get_counter() << endl;
```

Как видим, статическая функция может вызываться при помощи операции разрешения видимости, и как вызов метода объекта.

Проверяем, как устанавливается новое значение счетчика:

```

lamp::put_counter(5);
cout << "lamp::get_counter = " << lamp::get_counter() << endl;
a.put_counter(10);
cout << "lamp::get_counter = " << lamp::get_counter() << endl;

```

Создается ощущение, что статические функции-элементы только и могут, что работать со статическими элементами. Это не совсем так.

В статическую функцию можно передавать объект, вот пример такой функции:

```

class lamp {
    . . .
public:
    // идентификатор объекта
    int id;
    . . .
    // изменяет идентификатор объекта
    static void change_id(lamp & o, int newValue) {
        o.id = newValue;
    }
};

```

В основной функции удостоверяемся, что идентификатор объекта действительно изменяется:

```

lamp::change_id(a, 22);
cout << "a.id = " << a.id << endl;

```

3.4. Классы статических функций

Статические функции можно также использовать для создания пакетов функций, которые не применяются к объектам, а используются сами по себе. Это особенно актуально для систем программирования, в которых нельзя создавать отдельно стоящие функции, поскольку в таких системах код пишется только внутри методов классов. Примером является класс Math в языке Java, определяющий математические функции.

Мы сконструируем класс, состоящих только из статических функций, определяющих разные полезные методы. Модуль class.h:

```

struct util {
    // меняет местами значения двух переменных
    static void swap(int & a, int & b);
    // сортирует массив целых чисел
    static void sort(int * arr, int count);
    // обращает массив целых чисел
    static void reverse(int * arr, int count);
};

```

Это реализация методов:

```

// меняет значения двух переменных
void util::swap(int & a, int & b) {
    b ^= a ^= b ^= a;
}

```

```

// сортирует массив целых чисел
void util::sort(int * A, int size) {
    for (int k = 1; k < size; k++) {
        for (int j = 1; j < size; j++) {
            if (A[j - 1] > A[j]) {
                swap(A[j - 1], A[j]);
            }
        }
    }
}

// обращает массив целых чисел
void util::reverse(int * A, int size) {
    for (int j = 0; j < size / 2; j++) {
        swap(A[j], A[size - 1 - j]);
    }
}

```

В основной функции проверяем, как это работает:

```

int arr[] = {5, 2, 3, 4, 1};
util::sort(arr, sizeof(arr) / sizeof(int));
util::reverse(arr, sizeof(arr) / sizeof(int));

```

Данный пример приведен только для демонстрации, поскольку в программе на языке C++ можно определять отдельно стоящие функции для этих целей. Заметим также, что операции для работы с массивами лучше, наверное, определять в классе, описывающем массив.

Сейчас удалите или поместите в комментарий код функции main.

3.5. Константные объекты и элементы класса

Можно создать представителя класса с модификатором const.

В этом случае компилятор не может изменять такой объект после его инициализации. В случае, если константный объект используется с не-константной функцией-элементом, компилятор генерирует сообщение об ошибке. Например:

```

const lamp e;
lamp::change_id(e, 33);
e.id = 0;

```

Убедитесь, что компилятор запрещает изменять свойство id объекта как прямо, так и при помощи статической функции.

Инкапсулируем идентификатор id, перенеся его в закрытую секцию.

У меня при этом статическая функция-элемент по-прежнему изменяет идентификатор не-константного объекта на вот таком примере:

```

lamp e;
lamp::change_id(e, 33);

```

Иначе говоря, статическая функция видит закрытые не статические элементы данных конкретного объекта.

Добавим две функции для чтения и установки значения id.

Модуль class.h:

```
class lamp {
    . . .
public:
    . . .
    // устанавливает id
    void put_id(int newValue) {
        id = newValue;
    }
    // читает id
    int get_id(void) const {
        return id;
    }
};
```

Обратим внимание, что функция-элемент для чтения идентификатора `get_id` объявлена как константная. Это означает, что она может применяться как к константным, так и к не-константным объектам, в то время, как функция установки идентификатора `put_id` может применяться только к не-константным объектам.

Убеждаемся в этом в основной функции:

```
lamp e;
cout << "e.id = " << e.get_id() << endl;
const lamp f;
cout << "f.id = " << f.get_id() << endl;
// изменение объекта
e.put_id(44);
// не работает с константным объектом
//f.put_id(55);
```

Элемент данных может быть константным. Его начальное значение тогда может быть задано только в списке инициализации конструктора.

Опишем для класса `lamp` константный элемент данных «мощность»:

```
class lamp {
    . . .
    // мощность
    const int pwr;
public:
    . . .
};
```

Инициализируем константный элемент в списке инициализации:

```
// конструктор по умолчанию
lamp::lamp() : pwr(0) {
    id = ++counter;
}
```

Самостоятельно определите конструктор приведения, задающий ненулевое значение мощности параметром конструктора, а также функцию-элемент `power`, возвращающую значение мощности.

Протестируйте конструктор приведения и функцию-элемент `power`.

3.6. Вопросы и упражнения

1. Какой элемент класса называется статическим?
2. Чем статический элемент данных отличается от обычного?
3. Чем статическая функция-элемент отличается от обычной?
4. Где хранится значение статического элемента данных?
5. Как можно получить доступ к статическому элементу?
6. Какой элемент класса называется константным?
7. С какими объектами могут быть использованы не-константные и константные функции-элементы?
8. Как инициализируется константный элемент данных?
9. Спроектируйте класс Array (Массив), использующий статические и константные элементы.
10. Сравните варианты задания неизменяемой мощности лампочки:
 - а) конструктором приведения,
 - б) константным объектом,
 - в) константным элементом данных.

4. Работа ООП-104. Друзья классов

Цели:

- инкапсуляция и дружественные функции и классы.

Задачи:

- исследование сильной инкапсуляции,
- исследование дружественных функций класса,
- исследование перегрузки операций,
- исследование дружественных классов класса.

4.1. Подготовка проекта

Скачайте архив работы ООР100. Извлеките каталог ООР100 из архива в корневой каталог диска С:. Переименуйте каталог в ООР104. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Укажите в начале модуля main.cpp сведения об организации, о себе, о проекте, тему и дату начала работы.

4.2. Друзья класса

Друзья — те, что приходят к вам домой и делают что хотят. Так же и в программировании с классами на С++.

Любой дружественный объект, будь то функция или класс, имеют доступ к любым элементам класса, независимо от секции класса, в которой элементы находятся, будь то закрытая или иная секция.

Иначе говоря, друзья *видят* класс как сам класс.

Различие лишь в том, что класс видит элементы через указатель this, то есть непосредственно, а друг видит элементы через объект класса, который должен быть передан другу (как параметр).

Дружественные функции и классы *улучшают инкапсуляцию* класса, потому что класс, при использовании друзей, может не иметь открытых функций-элементов для управления инкапсулированными элементами данных, то есть не давать никакой возможности их изменять.

Класс должен явным образом объявлять друзей при помощи специального ключевого слова friend. При этом класс не становится другом того, кого признает другом. Если некто А объявляет своим другом некоего В, это не означает, что А становится другом В. Некто В сам должен разрешать друзьям (если я считаю тебя своим другом, это вовсе не значит, что ты считаешь другом меня, «дружить семьями» в С++ не удастся).

4.3. Сильно инкапсулированные объекты

Для начала рассмотрим класс с сильной инкапсуляцией.

Опять класс лампочки с одним инкапсулированным элементом.

Описываем класс в модуле class.h:

```
// сильная инкапсуляция
class lamp {
    int state;
public:
    lamp() {
        state = 0;
    }
    int is_off() {
        return . . .;
    }
    int is_on() {
        return . . .;
    }
};
```

Функция `is_off` должна возвращать 0, если лампочка включена, и 1, если лампочка выключена. Функция `is_on`, наоборот, должна возвращать 1, если включена, и 0, если лампочка выключена.

Чтобы убедиться в правильности реализации функций, в основной функции добавляем тестирующий код:

```
int main(void) {
    lamp a;
    cout << "on = " << a.is_on() << endl;
    cout << "off = " << a.is_off() << endl;
}
```

Убеждаемся, что выводятся значения 0 и 1.

Теперь заменяем в конструкторе лампочки начальное значение `state` на, например, 11, и убеждаемся, что выводятся значения 1 и 0.

Если не так, то доводим функции-элементы класса лампочки до правильной реализации.

Возвращаем в конструкторе значение `state` на нулевое.

Очевидно, лампочку нельзя включить обычным образом, поскольку объекты получают сильно инкапсулированными.

Попробуем жульничество.

Поскольку структура лампочки состоит только из одного элемента типа `int`, попробуем привести `lamp*` к `int*` и изменить состояние лампочки, разыменовывая полученный указатель.

Основная функция (модуль `main.cpp`), дописываем после объявления лампочки а следующую конструкцию:

```
int main() {
    lamp a;
    *(int*)&a = 9;
    . . .
}
```

Запускаем программу и убеждаемся, что мы взломали инкапсуляцию сильно инкапсулированного объекта.

В связи с этим интересно, можем ли мы так же легко взломать инкапсуляцию объекта, в котором две переменных, например, типа `int`.

Да легко.

В модуле `class.h` опишите класс `rat` с двумя инкапсулированными элементами данных `num` и `den` типа `int`. Пусть конструктор по умолчанию устанавливает элемент `num` в значение 0, элемент `den` в значение 1.

В основной функции опишем похожую структуру данных:

```
struct RAT { int num, den; };
```

Теперь объявим объект `x` типа `rat`, а также объект `X` типа `RAT*`, которому присвоим адрес объекта `x`, приведенный к типу `RAT*`. После этого объекта `X` становится объектом `x` и можно легко изменять инкапсулированные элементы данных:

```
rat x;  
RAT * X = (RAT*)&x;  
(*X).num = 1;
```

Все это интересно, но никак не способствует написанию приличного кода. Нужны стандартные механизмы, позволяющие писать понятный код.

4.4. Дружественные функции

Раз друзья имеют доступ к любым элементам класса, попробуем написать пару функций, устанавливающих значение элемента данных класса лампочки в нулевое и единичное состояние.

Функцию, которая включает лампочку, назовем `turn_on_lamp`, а функцию, которая выключает лампочку, — `turn_off_lamp`. Модуль `class.h`:

```
void turn_on_lamp(lamp & a) {  
    lamp.state = 1;  
}  
void turn_off_lamp(lamp & a) {  
    lamp.state = 0;  
}
```

Функции ничего не возвращают и имеют только один параметр, — объект класса `lamp`. Обращаясь к этому объекту, функции изменяют значение элемента `state` на значение 1 или 0.

Увы, компилятор не признает функции правильными, так как элемент `state` находится в закрытой секции класса. Так и должно быть.

Теперь объявим функции друзьями класса `lamp`. Модуль `class.h`:

```
class lamp {  
    // я разрешаю этим функциям всё  
    friend void turn_on_lamp(lamp & a);  
    friend void turn_off_lamp(lamp & a);  
    int state;  
public:  
    . . .  
};
```

Теперь при помощи этих функций можно управлять лампочками.

Удостоверяемся, что дружественные функции действительно имеют доступ к закрытым элементам данных, основная функция:

```
void main(void) {  
    . . .  
    turn_on_lamp(a);  
    cout << "on = " << a.is_on() << endl;  
    cout << "off = " << a.is_off() << endl;  
    turn_off_lamp(a);  
    cout << "on = " << a.is_on() << endl;  
    cout << "off = " << a.is_off() << endl;  
}
```

4.5. Перегрузка операций

Дружественность часто используется при перегрузке операций. Прежде рассмотрим функцию, не элемент класса, которая является операцией.

Для продолжения немного изменим имеющийся класс `rat`.

- 1) перенесем элементы данных в открытую секцию;
- 2) добавим в класс конструктор преобразования с двумя параметрами:

```
rat(int n, int d) {  
    num = n;  
    den = d < 1 ? 1 : d;  
}
```

Теперь с классом можно выполнять операции при помощи функций операций, таких, в названии которых есть ключевое слово `operator`. Это называется перегрузкой операций.

Модуль `class.h`, класс `rat`:

Сначала для удобства опишем в классе сокращение дроби как функцию-элемент `cancel`, используя алгоритм Евклида. Например:

```
class rat {  
public:  
    . . .  
    rat cancel() {  
        int a = abs(num), b = abs(den);  
        while (b) { b ^= a ^= b ^= a %= b; }  
        num /= a;  
        den /= a;  
        return * this;  
    }  
}
```

Опишем операцию сложения двух рациональных чисел (вне класса):

```
// сложение рациональных чисел  
rat operator + (rat a, rat b) {  
    return rat(a.num * b.den + b.num * a.den, a.den * b.den).cancel();  
}
```

В основной функции комментируем имеющийся код и тестируем операцию, например, так:

```
rat a, b(1, 3), c(2, 3);  
a = b + c;
```

Пошаговым исполнением удостоверяемся, что сложение происходит.

Недостаток перегрузки операций таким образом должен быть очевиден: элементы данных класса не могут быть инкапсулированными. Хорошо это или плохо (или правильно или неправильно), — это другой вопрос, который придется решать в каждом конкретном случае индивидуально.

Сейчас нам кажется, что лучше бы операцию определял класс.

Это требует исследования. Описываем в классе операцию вычитания:

```
// операция вычитания  
rat operator - (rat b) {  
    return rat(num * b.den - b.num * den, den * b.den).cancel();  
}
```

Элементы данных num и den перемещаем в закрытую секцию.

В основной функции временно комментируем операцию сложения, и заменяем оператор сложения оператором вычитания:

```
a = b - c;
```

Можно убедиться, что и эта операция успешно выполняется, при этом мы обеспечили инкапсуляцию.

Однако есть одна тонкость, которую не сразу и разглядишь.

Можно ли вычесть из рационального числа не рациональное число, а целое, например, 1?

Учитывая, что в этом случае единица передается в качестве параметра функции операции в классе, в принципе ее можно преобразовать в объект типа rat, конструктором приведения или операцией приведения.

Поскольку в классе rat нет конструктора приведения, определим его:

```
// конструктор приведения  
rat(int n) {  
    num = n;  
    den = 1;  
}
```

Установим в конструкторе точку остановки.

В основной функции описываем операцию вычитания:

```
a = b - 1;
```

Убеждаемся, что данная операция выполняется, а перед выполнением операции управление переходит к конструктору приведения для того, чтобы преобразовать единичный операнд в объект типа rat.

Попробуем теперь выполнить операцию наоборот:

```
a = 1 - b;
```

Увы, компилятор не допускает этот код, потому что операция, описываемая в классе, предполагает, что левым операндом является объект rat, а поскольку левый операнд, — целое число, то операция в классе даже и не рассматривается компилятором.

В связи с этим использование дружественных функций для перегрузки операций представляется в данном случае предпочтительнее. Поэтому мы объявляем функцию операции сложения другом `rat`:

```
class rat {  
    friend rat operator + (rat a, rat b);  
};
```

Извлекаем функцию сложения из комментария, и тестируем сложение на примере кода:

```
a = 1 + b;
```

Удостоверяемся, что вычисляется значение переменной `a`. Однако вычитание выполнить таким же образом не удастся, здесь потребуется явный вызов конструктора приведения, попробуйте:

```
a = rat(1) - b;
```

4.6. Дружественные классы

Дружественные классы, как и дружественные функции, получают доступ к любым элементам класса. Рассмотрим пример, в котором используется наш класс лампочки. Он сильно инкапсулирован, и в отсутствие дружественных функций представляется бесполезным.

Так ли это? Будем из лампочек составлять гирлянду, для чего описываем в модуле `class.h` класс гирлянды `garland`. Будем считать, что количество лампочек в гирлянде постоянно и равно восьми. Начальная версия класса может иметь следующий вид:

```
// размер гирлянды  
#define MAX_GARLAND 8  
  
// гирлянда лампочек  
class garland {  
    // лампочки гирлянды  
    lamp L[MAX_GARLAND];  
public:  
    // включает гирлянду  
    void on() {}  
    // выключает гирлянду  
    void off() {}  
};
```

Чтобы гирлянда могла включить лампочку, нужно объявить класс гирлянды другом класса лампочки. Поскольку класс гирлянды определен в модуле ниже класса лампочки, нужно перед классом лампочки сделать предварительное объявление класса гирлянды.

Вот оба действия:

```
// предварительное объявление класса  
class garland;
```

```
// сильная инкапсуляция
class lamp {
    // я разрешаю этому классу всё
    friend garland;
    . . .
};
```

Заметим, что объявление "friend class garland" тоже допускается.

Теперь можно приступить к реализации включения гирлянды.

Например, так. Функция-элемент `on` перебирает в цикле все лампочки и устанавливает значения их элемента `state` в единицу прямым образом. Одновременно она выводит в поток значение элемента `state` после установки. В конце функция добавляет в поток элемент `endl` (конец строки), так что функция в конечном итоге выводит строку единиц.

Реализуем данный алгоритм. После этого в основной функции включаем гирлянду при помощи функции-элемента `on`, и удостоверяемся, что в консоль выводится "1111111".

Аналогичным образом описывается и функция-элемент `off`.

Если все получилось, то хотелось бы увидеть гирлянду типа «бегущий огонь» или, например, перемигивание лампочек. Это тоже можно сделать.

Опишем в классе гирлянды функцию-элемент `start`. Она будет делать следующее. Сначала описываем переменную `index`, — текущая лампочка, равна 0. Далее формируем цикл с 16 итерациями.

Следующие действия выполняются внутри этого цикла.

- включаем лампочку `index`;
- выводим в консоль всю гирлянду;
- выключаем лампочку `index`;
- вычисляем новое значение `index`:

```
index = ++index % MAX_GARLAND;
```

После этого в основной функции включаем гирлянду методом `start`.

4.7. Вопросы и упражнения

1. Что означает сильная инкапсуляция?
2. Кто такие друзья класса, кто может выступать в качестве друга?
3. Как объявить другом функцию, класс или метод класса?
4. К каким элементам класса имеют доступ друзья?
5. Почему дружественные функции усиливают инкапсуляцию?
6. В каких случаях дружественная операция лучше операции класса?
7. Какое отношение возникает между классами `lamp` и `garland`?
8. Спроектируйте сильно инкапсулированный класс циклического счетчика по модулю N , задаваемому во время создания, и дружественную функцию, возвращающую следующее значение счетчика.

5. Работа ООП-105. Деструктор и специальные функции класса

Цели:

- исследование классов с динамически выделяемой памятью.

Задачи:

- выявление необходимости деструктора;
- исследование операции присваивания;
- исследование конструктора копии.

5.1. Подготовка проекта

Скачайте архив работы ООР100. Извлеките каталог ООР100 из архива в корневой каталог диска C:. Переименуйте каталог в ООР105. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Укажите в начале модуля main.cpp сведения об организации, о себе, о проекте, тему и дату начала работы. Модуль class.h предназначен для описания классов. В этом модуле должна быть закомментирована следующая строка: using namespace std;.

5.2. Деструктор

В модуле class.h будем описывать класс string строки символов.

Память для строки будет выделяться динамически, и присваиваться указателю p. Первоначальная версия класса имеет следующий вид:

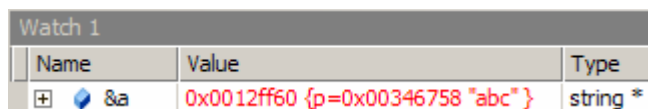
```
class string {
public:
    char * p;
    // конструктор приведения
    string(char * s) {
        int n = strlen(s);
        p = new char[1 + n];
        strcpy(p, s);
    }
};
```

Экспериментируем с классом в основной функции. Сначала создадим один объект и посмотрим, как выделяется память:

```
int main() {
    string a("abc");
}
```

Чтобы исследовать выделение памяти, нужно выполнить строку кода основной функции при помощи F10 и остановиться сразу после этого.

Далее введем в окно просмотра Watch ссылку &a:



Watch 1		
Name	Value	Type
&a	0x0012ff60 {p=0x00346758 "abc"}	string *

Видим, что переменная a имеет адрес 0x0012ff60, а строка "abc", на которую ссылается указатель p, хранится по адресу 0x00346758.

Очевидно, что это совершенно разные области памяти. В вашей среде конкретные числа будут другими.

Таким образом, объект a занимает в памяти два разных участка.

Остановим программу Shift+F5 или исполним до конца F5.

К чему это может привести, покажет нам следующее исследование.

Объявим два объекта и попробуем присвоить один объект другому:

```
int main(void) {
    string a("", b("abc"))
    a = b;
    strcpy(b.p, "xyz");
}
```

Выполним первую строчку кода и остановимся.

Посмотрим, как выделилась память:

Watch 1			
Name	Value	Type	
&a	0x0012ff60 {p=0x00346758 ""}	string *	
&b	0x0012ff54 {p=0x00346798 "abc"}	string *	

Видим, что переменные a и b располагаются в одной области памяти, на небольшом расстоянии друг от друга. Значения строк располагаются в другой области памяти на расстоянии 64 байта.

Выполним вторую строчку кода:

Watch 1			
Name	Value	Type	
&a	0x0012ff60 {p=0x00346798 "abc"}	string *	
&b	0x0012ff54 {p=0x00346798 "abc"}	string *	

Оба объекта указывают на одну область, память объекта a потерялась.

Выполним третью строчку кода:

Watch 1			
Name	Value	Type	
&a	0x0012ff60 {p=0x00346798 "xyz"}	string *	
&b	0x0012ff54 {p=0x00346798 "xyz"}	string *	

Изменили один объект, а изменились оба.

Остановим программу Shift+F5 или исполним до конца F5.

Кроме того, по завершении программы переменные уничтожаются автоматически, а вот память, которая была выделена для хранения строк, теряется, и происходит утечка памяти.

В этом тоже можно убедиться.

Заклучим переменные в блок, который создаст локальную область видимости, по выходе из которой переменные a и b будут уничтожены:

```

int main(void) {
    char *pa = 0, *pb = 0;
    {
        string a("123"), b("abc");
        pa = a.p;
        pb = b.p;
        a = b;
        strcpy(a.p, "---");
        strcpy(b.p, "xyz");
    }
    std::cout << pa << std::endl;
    std::cout << pb << std::endl;
}

```

В окно Watch дополнительно нужно ввести переменные pa и pb.

Исполняя программу шаг за шагом, можно наблюдать, как постепенно изменяются значения объектов, при этом значения переменных pa и pb остаются неизменными. После выхода из блока эти переменные по-прежнему указывают на выделенные области памяти.

Как мы знаем, в этом случае требуется деструктор.

Поэтому мы описываем его в классе string.

Модуль class.h, улучшенная версия класса:

```

class string {
public:
    char * p;
    string(char * s) {
        int n = strlen(s);
        p = new char[1 + n];
        strcpy(p, s);
    }
    // деструктор
    ~string() {
        delete[] p;
    }
};

```

Шаг за шагом выполняем ту же самую программу.

По завершении блока происходит разрушение объектов a и b, кроме того, выделенные им области памяти освобождаются, а поскольку на них указывают действительные еще указатели pa и pb, выполнение программы прерывается функцией assert.

5.3. Операция присваивания

Для исследования операции присваивания запишем другой код в основной функции main:

```

int main(void) {
    string a("123"), b("abc"), c("xyz");
    a = b;
}

```

Для начала запретим операцию присваивания от string.

Класс string, модуль class.h:

```
class string {
    // запрет операции присваивания
    string operator = (string & s);
public:
    . . .
};
```

После этого уже на этапе компиляции обнаруживается ошибка, поскольку присваивание `a = b` запрещено.

Однако не запрещено присваивать `char*`. Поэтому описываем в классе string операцию присваивания от `char*`:

```
class string {
    . . .
    // операция присваивания
    char * operator = (char * s) {
        delete[] p;
        int n = strlen(s);
        p = new char[1 + n];
        strcpy(p, s);
        return p;
    }
};
```

Соответственно, в основной функции можно заменить значения всех объектов сразу:

```
c = b = a = "---";
```

Отлаживаем, убеждаемся, что операция присваивания от `char*` вызывается три раза.

5.4. Конструктор копии

Конструктор копии вызывается в случае:

- инициализации объекта другим объектом этого же класса;
- передачи объекта функции в качестве параметра;
- возврата объекта из функции.

Рассмотрим первый вариант.

Основная функция main.

Описываем строки и присваивание:

```
int main(void) {
    string a("123");
    string b = a;
}
```

Исполняем программу шаг за шагом и убеждаемся, что в этом случае также происходит нарушение памяти. Об этом свидетельствует функция `assert`, которая в этом случае вызывается. Оно происходит при выходе из области видимости. Когда разрушается первый объект, память, на которую указывает его указатель `p`, освобождается.

Поскольку указатель `p` второго объекта указывает на ту же область памяти, то при уничтожении второго объекта возникает исключительная ситуация.

Рассмотрим второй вариант.

Нужна вспомогательная функция `foo` (перед функцией `main`):

```
void foo(string x) {
    x = "xyz";
}
```

В основной функции вызываем функцию `foo`:

```
int main(void) {
    string a("123");
    foo(a);
}
```

Выполняя эту программу шаг за шагом, можно убедиться, что также происходит нарушение памяти. В функцию `foo` передается точная копия `x` переменной `a`. По завершении `foo` срабатывает деструктор копии и память, на которую указывает `x.p`, очищается. Но поскольку оба указателя `p` указывают на одну и ту же область памяти, при выходе из `main` возникает нарушение, — указатель `a.p` уже освобожден.

Для тестирования третьего случая используем еще одну функцию:

```
string get() {
    string y("abc");
    return y;
}
```

В основной функции вызываем функцию `get`:

```
int main(void) {
    //string a("123");
    //foo(a);
    string b = get();
}
```

Здесь также происходит нарушение памяти по той же причине, что и в случае два. Поэтому требуется конструктор копии. Определяем его в классе `string`, модуль `class.h`:

```
class string {
    . . .
    // конструктор копии
    string(string & s) {
        int n = strlen(s);
        p = new char[1 + n];
        strcpy(p, s.p);
    }
};
```

Как только конструктор копии будет определен, во всех трех случаях нарушений памяти происходить не будет, в чем вы удостоверитесь сами.

5.5. Подготовка домашней работы

Скачайте архив домашней работы OORHW2. Извлеките из архива каталог OORHW2 в корневой каталог диска C:. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

В проекте один модуль, main.cpp. В этом модуле выполняется тестирование функций класса. Выясните у преподавателя, класс какого объекта вам задан. Варианты приведены в следующей таблице.

Вариант	Объект	Имя-класса
1	строка	dstring
2	динамический массив	darray
3	динамический список	dlist
4		
5		
6		
7		

Далее предполагается, что задан вариант 1, класс dstring.

Добавьте в проект модули имя-класса.h и имя-класса.cpp.

Укажите в начале каждого модуля сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Фамилия Имя Отчество
// Объектно-ориентированное программирование
// OORHW2. Класс с динамической памятью
// 01.01.2000
```

Включите модуль имя-класса.h в основной модуль main.cpp и в модуль имя-класса.cpp. В модуле .h опишите динамический элемент данных, конструкторы и деструктор:

```
class dstring {
    // динамический элемент данных
    char * p;
public:
    // деструктор
    ~dstring();
    // конструктор по умолчанию
    dstring();
    // конструктор приведения
    dstring(char * s);
    // конструктор копии
    dstring(dstring & s);
    // операция присваивания
    dstring operator = (const char * s);
    // операция присваивания
    dstring operator = (dstring & s);
};
```

В модуле .cpp определите методы:

```
#include "dstring.h"

// деструктор
~dstring() {}
// конструктор по умолчанию
dstring() {}
// конструктор приведения
dstring(char * s) {}
// конструктор копии
dstring(dstring & s) {}
// операция присваивания
dstring operator = (const char * s) {
    return *this;
}
// операция присваивания
dstring operator = (dstring & s) {
    return *this;
}
```

Опишите функциональность конструкторов. Тестируйте код, анализируя, как выделяется и освобождается память.

Вопросы и упражнения

1. Что называется деструктором?
2. Сколько деструкторов может быть у класса и почему?
3. Когда и как происходит нарушение памяти, если в классе нет деструктора?
4. Как определяется операция присваивания и когда она вызывается?
5. Когда и как происходит нарушение памяти, если в классе не определена операция присваивания?
6. Что называется конструктором копии? Какова его сигнатура?
7. Когда вызывается конструктор копии?
8. Когда и как происходит нарушение памяти, если в классе не определен конструктор копии?
9. Вернитесь к заданию ООП-102 и опишите деструктор в классе стека два. Удостоверьтесь в том, что при использовании этого класса не возникает нарушения памяти.

Владимир Вадимович Пономарев
Практикум по объектно-ориентированному программированию
Учебно-методическое пособие
Наиболее актуальную версию пособия см. revol.ponosom.ru

Отпечатано с готового оригинал-макета
Издательство ОТИ НИЯУ МИФИ, 2019
Тираж 11 экз.