МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ВОЛОГОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра автоматики и вычислительной техники

СОВРЕМЕННЫЕ ПРОБЛЕМЫ ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Методические указания к практическим занятиям

Факультет электроэнергетический

Направление 09.04.01 «Информатика и вычислительная техника»

Вологда
2014

УДК 621.374.1

Современные проблемы информатики и вычислительной техники: методические указания к практическим занятиям. – Вологда: ВоГУ, 2014. – 19 с.

Методические указания предназначены для выполнения практических работ по дисциплине «Современные проблемы информатики и вычислительной техники» и служат для успешного освоения данного предмета и получения навыков решения разных типов задач. Для каждой практической работы приводятся краткие теоретические сведения, разбираются примеры и предлагается набор заданий для самостоятельного решения.

Утверждено редакционно-издательским советом ВоГУ

Составитель И.А. Андрианов, канд. тех. наук, доцент

Рецензент А.Н. Швецов, д-р техн. наук, профессор

Введение

Данные методические указания предназначены для магистрантов направления 09.04.01 – Информатика и вычислительная техника.

Методические указания разработаны для поддержки курса «Современные проблемы информатики и вычислительной техники» (третий семестр) и содержат 4 четырёхчасовые практические работы (либо 8 двухчасовых работ), в которых рассматриваются вопросы проектирования и программной реализации сложных автоматизированных информационных систем с использованием технологий объектно-ориентированного и обобщенного программирования.

В результате выполнения работ студенты получат навыки проектирования и реализации информационных систем с использованием шаблонов проектирования в современной интегрированной среде разработки на языке C++.

Практическая работа 1

Проектирование и реализация структуры классов

В данной работе требуется спроектировать и реализовать на языке C++ иерархию классов геометрических фигур на плоскости, продемонстрировав при этом такие ключевые понятия объектно-ориентированного программирования, как инкапсуляция, наследование и полиморфизм.

Каждая фигура (окружность, треугольник и др.) обладает набором уникальных свойств – например, у окружности есть координаты её центра и радиус, а у треугольника – координаты трёх вершин. Но у всех фигур могут быть и общие свойства – например, цвет границы.

Пусть у каждой фигуры будет общая функция *place* для вычисления площади. Однако реализовываться эта функция для каждого конкретного класса должна, конечно, по-разному.

Решение данной задачи выглядит следующим образом. Вначале создадим абстрактный базовый класс *Figure*, в нём объявим общие свойства и функцию (метод) *place*. Поскольку реализацию данной функции мы пока написать не можем, объявим её как чисто виртуальную (*pure virtual*). Далее создадим соответствующие классы-наследники, в каждом из них объявим дополнительные поля и выполним свою реализацию *place*.

Чтобы продемонстрировать полиморфизм, создадим вектор, каждый элемент которого содержит указатель на фигуру. Однако на какую именно фигуру, на момент компиляции ещё неизвестно – это будет определяться случайным образом при выполнении программы. Теперь посчитаем суммарную площадь всех фигур – для этого нужно пройти в цикле по всем элементам вектора и для каждого вызвать функцию *place*. Обратите внимание, что при каждом вызове функции *place* реально будет вызываться соответствующая функция либо для окружности, либо для треугольника – смотря на какой объект ссылается указатель в этот момент. То есть адрес функции определяется не на этапе компиляции по статическому типу указателя, а на этапе вы-

полнения по динамическому типу объекта, на который он указывает. Такой способ получения адреса называется *динамическим* или *поздним связыванием* и является одной из разновидностью полиморфизма.

Построенная диаграмма классов в формате UML [4] изображена на рисунке 1.

Рис.1. Диаграмма классов

Далее рассмотрим особенности программной реализации. Объявления классов вынесем в заголовочный файл *figures.h*. Программная реализация функций классов будет находиться в файле *figures.cpp*. Наконец, создание экземпляров классов и работа с ними будет выполняться в исходном файле *main.cpp*.

figures.h

```
#ifndef __FIGURES_H
#define __FIGURES_H
//абстрактный базовый класс
class Figure{
     int color;  //общие свойства для всех потомков
public:
     Figure(int color) :
```

```
        color(color) //список инициализации полей
        {}
        //чисто виртуальная функция вычисления площади
        virtual double place() const = 0;
        //вложенная структура для координат точки
        struct Point{
             int x, y;
             Point(int x, int y):
             x(x), y(y) {}
        };
};

class Circle: public Figure{
     Point center; //координаты центра
     int rad; //радиус
public:
     Circle(Point center, int rad, int color);
     //перегружаем виртуальную функцию
     virtual double place() const;
};

class Triangle : public Figure{
     Point p1, p2, p3; //координаты вершин
public:
     Triangle(Point p1, Point p2, Point p3, int color);
     //перегружаем виртуальную функцию
     virtual double place() const;
};
#endif
```

figures.cpp

```
#define _USE_MATH_DEFINES
#include <math.h>
#include "figures.h"
#include <iostream>

Circle::Circle(Point center, int rad, int color):
 Figure(color), //вызываем конструктор базового класса
 center(center), rad(rad) //инициализация полей
 {}

double Circle::place() const {
    #ifdef _DEBUG
       std::cerr << "In Circle::place()\n";
```

```
    #endif
     return M_PI * rad * rad;
}

//=================================================

Triangle::Triangle(Point p1, Point p2, Point p3, int color):
  Figure(color),
  p1(p1), p2(p2), p3(p3) {}

double Triangle::place() const {
    #ifdef _DEBUG
       std::cerr << "In Triangle::place()\n";
    #endif
     double ans = 0.5 * ( (p1.x-p3.x)*(p2.y-p3.y)
                    - (p2.x-p3.x)*(p1.y-p3.y) );
     if (ans < 0) ans = -ans;
     return ans;
}
```

main.cpp

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <vector>
#include <ctime>
#include "figures.h"

int main() {
   //создаём вектор указателей на фигуры и заполняем его
случайно
   srand((unsigned int)time(NULL));
   const int nFigures = 10;
   std::vector<Figure *> figures (nFigures);
   for (int i = 0; i < nFigures; i++) {
     if (rand() % 2 == 0) {
     figures[i] = new Circle(Figure::Point(rand()%100,
                      rand()%100), rand()%100+1, rand()%100);
     } else {
        figures[i] = new Triangle(Figure::Point(rand()%100,
            rand()%100), Figure::Point(rand()%100,
rand()%100),
```

```
            Figure::Point(rand()%100, rand()%100),
rand()%100);
      }
    }
    //найдём суммарную площадь всех фигур
    double sumPlace = 0.0;
    for (int i = 0; i < nFigures; i++) {
       sumPlace += figures[i]->place();
      }
      std::cout << std::fixed << std::setprecision(6)
                <<"Sum place = " << sumPlace << std::endl;
      //уничтожим все объекты
      for (int i = 0; i < nFigures; i++) delete figures[i];
      return 0;
   }
```

Вопросы для самопроверки

1. Что будет выведено в результате выполнения следующего фрагмента программы?

```
struct A {
  virtual void f() {
    std::cout << "A";
  }
};
struct B : public A {
  virtual void f() {
    std::cout << "B";
  }
};
...
A* a = new B;
a->f();
```

2. Что выведет следующий фрагмент программы?

```
struct A {
  virtual void f() {
    std::cout<<"A";
   }
} a;
struct B : public A {
 virtual void f() {
   std::cout<<"B";
 }
} b;
...
```

```
A *x = &b;
x->f();
a.f();
```

Самостоятельные задания

1. Добавьте в данную иерархию ещё один класс *Polygon* – произвольный многоугольник. *Подсказка*: площадь многоугольника можно найти через сумму площадей трапеций, взятых со знаками "+" или "–" (по сути, через интеграл).

Подумайте над вопросом: а нужен ли нам теперь класс *Triangle*, если треугольник – это частный случай многоугольника? Приведите доводы за и против.

2. Добавьте в базовый класс чисто виртуальную функцию вычисления периметра и реализуйте её во всех потомках.

3. Сделайте так, чтобы объекты каждого типа можно было выводить в поток операцией "<<", например:

```
Figure *f = new Circle(Figure::Point(100, 100), 50, 1);
std::cout << *f;
```

При этом должно вывестись: `<Circle: x=100, y=200, r=50, color=1>`

4. Создайте в базовом классе чисто виртуальную функцию *intersectTo*:

```
virtual bool intersectTo(const Figure &x) = 0;
```

Функция будет возвращать *true*, если наша фигура пересекается с указанной. Реализуйте её так, чтобы она работала для любой пары объектов.

5*. Реализуйте предыдущее задание так, чтобы при добавлении нового вида фигур в реализацию старых никаких изменений вносить не требовалось.

ПРАКТИЧЕСКАЯ РАБОТА 2

Шаблон проектирования Singleton

Достаточно часто встречаются ситуации, когда для класса должен быть создан только один (точнее, не более чем один) объект во всей программе. Примеры: класс *Logger* (запись сообщений в файл журнала), *Keyboard* (взаимодействие с клавиатурой), *Display* (работа с дисплеем), *Application* («приложение»), *SystemClock* (работа с системным временем), *PrintManager* (управление печатью) и др.

Описанный шаблон проектирования носит название *Singleton* [2, 5]. На русский язык данный термин иногда переводится как "одиночка" [6], но мы будет использовать более часто применяемую прямую транслитерацию "синглтон".

На первый взгляд, решение кажется очевидным: создать класс, в котором все поля и все методы являются статическими. Однако у такого подхода есть по меньшей мере два минуса. Во-первых, статические функции не могут быть виртуальными. Во-вторых, нет возможности управлять инициализацией и удалением объекта.

Чуть более сложное решение состоит в следующем [6]. Добавим в реализуемый класс статическую функцию *instance*, которая будет возвращать ссылку на экземпляр класса. При этом если экземпляр ещё не создан, то функция его создаст, а в противном случае возвратит ссылку на уже созданный объект.

Чтобы не реализовывать каждый класс-синглтон с нуля, воспользуемся возможностями обобщенного программирования и наследования языка C++. Вначале создадим базовый шаблонный класс *Singleton*, в котором определим основную функциональность. Конкретные же классы-синглтоны (*Logger*, *Keyboard* и т.п.) будут создаваться как наследники класса *Singleton*. Приведём пример программной реализации класса *Singleton* и его наследника – класса *Logger*.

singleton.h

```
#ifndef _SINGLETON_H_
#define _SINGLETON_H_
template <class T>
class Singleton {
  public:
      static T& instance() {
          if (myInstance == 0) {
              myInstance = new T;
          }
          return *myInstance;
      }
  protected:
        static T* myInstance;
        Singleton() {}
        Singleton(const Singleton&) {}
        Singleton& operator=(const Singleton&) {return
*this;}
};
template <class T>
T* Singleton<T>::myInstance = 0;
#endif
```

logger.h

```
#ifndef _LOGGER_H_
#define _LOGGER_H_
#include "singleton.h"
#include <fstream>
#include <string>
class Logger : public Singleton<Logger> {
    std::ofstream logFile;
    Logger();
```

```
  public:
    void writeToLogFile(const std::string &msg);
    friend class Singleton<Logger>;
};
#endif
```

logger.cpp

```
#include "logger.h"
#include <fstream>
#include <string>
static const char logFileName[] = "log.txt";
Logger::Logger() {
  logFile.open(logFileName);
}
void Logger::writeToLogFile(const std::string &msg){
  logFile << msg << std::endl;
}
```

main.cpp

```
#include <iostream>
#include "logger.h"
int main() {
    Logger::instance().writeToLogFile("Message 1");
    Logger::instance().writeToLogFile("Message 2");
    return 0;
}
```

Рассмотрим теперь вопрос уничтожения синглтона. В вышеприведённом коде разрушение объекта вообще не предусмотрено. Конечно, оперативная память будет освобождена операционной системой по завершении программы. Однако представим, что синглтон создаёт какие-то временные файлы на диске. Тогда при завершении программы они не будет удалены. Несложно привести и другие подобные примеры.

Для уничтожения класса можно добавить соответствующую статическую функцию *terminate* в класс Singleton:

```
void terminate() {
  if (myInstance != 0) {
    delete myInstance;
    myInstance = 0;
  }
}
```

Кроме того, в класс *Logger* нужно добавить деструктор, который и будет освобождать выделенные в конструкторе ресурсы.

У такого подхода есть определённые минусы. Один из основных состоит в том, что объект не удаляется автоматически – функцию *terminate* необходимо вызывать вручную.

Самостоятельные задания

1. Добавьте в класс *Logger* деструктор, в котором сначала в файл журнала пишется сообщение о том, что файл закрывается, после чего файл и на самом деле закрывается. Проверьте, будет ли это работать в случае вызова описанной выше функции *terminate* и без неё.

2. Сделайте так, чтобы деструктор в классе *Logger* вызывался автоматически при завершении работы программы без необходимости вызывать функцию *terminate*. Подсказка: создайте вспомогательный класс, в деструкторе которого вызовите метод *terminate*. Объявите статическую переменную типа данного класса – тогда деструктор будет вызываться автоматически.

3. После реализации предыдущего пункта возможна ситуация, что во время завершения работы программы произойдёт обращение к объекту-логгеру, который уже успел уничтожиться. Воспроизведите такую ситуацию.

4. Проверьте, что при обращении к уже уничтоженному синглтону (предыдущий пункт) ошибки не происходит, поскольку объект создаётся заново. В результате у нас получился так называемый "феникс" – то есть самовоссоздаваемый объект.

5*. Повторно воссозданный объект (см. пункт 4) уже не будет корректно удаляться автоматически. Проверьте, так ли это, и исправьте данную ошибку. *Подсказка*: посмотрите описание стандартной функции *atexit* языка *C*. Возможны и другие способы.

6. Сделайте так, чтобы можно было создать сразу несколько объектов-логгеров, при этом каждый будет писать в свой собственный файл. Двух объектов, пишущих в файл с одинаковым именем, существовать не должно.

ПРАКТИЧЕСКАЯ РАБОТА 3

Фабрики объектов

Достаточно часто встречается ситуация, когда нужно создать объект, тип которого неизвестен на этапе компиляции. Например, пусть мы разрабатываем графический редактор. При открытии файла в зависимости от его типа (JPG, PNG и др.) нужно создать объект соответствующего класса.

На первый взгляд, решение достаточно очевидно: написать несколько операторов *if*. Однако этот подход имеет недостатки. Названия конкретных классов на момент написания данного куска кода может быть неизвестно. Кроме того, в будущем может понадобиться добавить поддержку новых типов файлов. Это не должно требовать внесения изменений в написанный код, иначе значительно ухудшится его сопровождаемость.

В некоторых языках программирования (например, Delphi и Smalltalk) имеется понятие *метакласса,* что несколько облегчает решение задачи. В языке C++ метаклассов нет. Тем не менее, данную задачу можно решить достаточно гибко и эффективно.

Для примера рассмотрим иерархию классов геометрических фигур (см. практическую работу 1). Пусть каждый тип фигуры имеет уникальный числовой идентификатор (окружность – 0, треугольник – 1 и т.д.). Создадим от-

дельный класс *FigureFactory* – так называемую фабрику объектов [1, 2]. Каждый класс-фигура будет регистрировать себя в фабрике, передавая свой идентификатор и функцию для создания объектов этого класса.

При обращении к фабрике по указанному идентификатору будет происходить поиск нужной функции, а вызванная функция создаст и возвратит указатель на объект нужного типа. Заметим, что класс фабрики будет являться синглтоном (см. предыдущую работу).

Следующий код иллюстрирует данный подход.

Дополнения в файл figures.h

```
#include <map>

class FigureFactory: public Singleton<FigureFactory> {
public:
  typedef Figure* (*CreateFigureCallback)();
private:
  typedef std::map<int, CreateFigureCallback> CallbackMap;
  CallbackMap callbacks;
public:
  bool RegisterFigure(int FigureID, CreateFigureCallback
CreateFN);
  bool UnregisterFigure(int FigureID);
  Figure *CreateFigure(int FigureID);
  friend class Singleton<FigureFactory>;
};
```

Дополнения в файл figures.cpp

```
bool FigureFactory::RegisterFigure(int FigureID, CreateFigu-
reCallback CreateFN) {
   return callbacks.insert(CallbackMap::value_type(FigureID,
     CreateFN)).second;
}

bool FigureFactory::UnregisterFigure(int FigureID) {
   return callbacks.erase(FigureID) == 1;
}

Figure* FigureFactory::CreateFigure(int FigureID){
   CallbackMap::const_iterator i = callbacks.find(FigureID);
   if (i == callbacks.end()) {
      throw std::runtime_error("Unkonwn typeID");
   }
   return (i->second)();
}
```

```
//безымянное пространство имён
namespace {
  Figure* CreateCircle() {
    //создаём окружность со случайными параметрами
    return new Circle(Figure::Point(rand()%100, rand()%100),
                      rand()%100+1, rand()%100);
  }
  const int CIRCLE = 0; //идентификатор типа
  //регистрируем новый тип фигуры
  const bool circleRegistered = FigureFactory::instance()
             .RegisterFigure(CIRCLE, CreateCircle);
  };

namespace {
  Figure* CreateTriangle(){
    //создаём треугольник со случайными параметрами
    return new Triangle(Figure::Point(rand()%100,
rand()%100),
                        Figure::Point(rand()%100,
rand()%100),
                        Figure::Point(rand()%100,
rand()%100),
                        rand()%100);
  }
  const int TRIANGLE = 1; //идентификатор типа
  //регистрируем новый тип фигуры
  const bool triangleRegistered = FigureFactory::instance()
             .RegisterFigure(TRIANGLE, CreateTriangle);
};
```

Изменения в файле main.cpp

```
for (int i = 0; i < nFigures; i++) {
   //получаем случайный тип фигуры (0 или 1)
   int figureID = rand() % 2;
   //создаём фигуру, зная лишь идентификатор типа
   figures[i] = FigureFacto-
ry::instance().CreateFigure(figureID);
}
```

На рисунке 2 показаны получившиеся дополнения к диаграмме классов из практической работы 1.

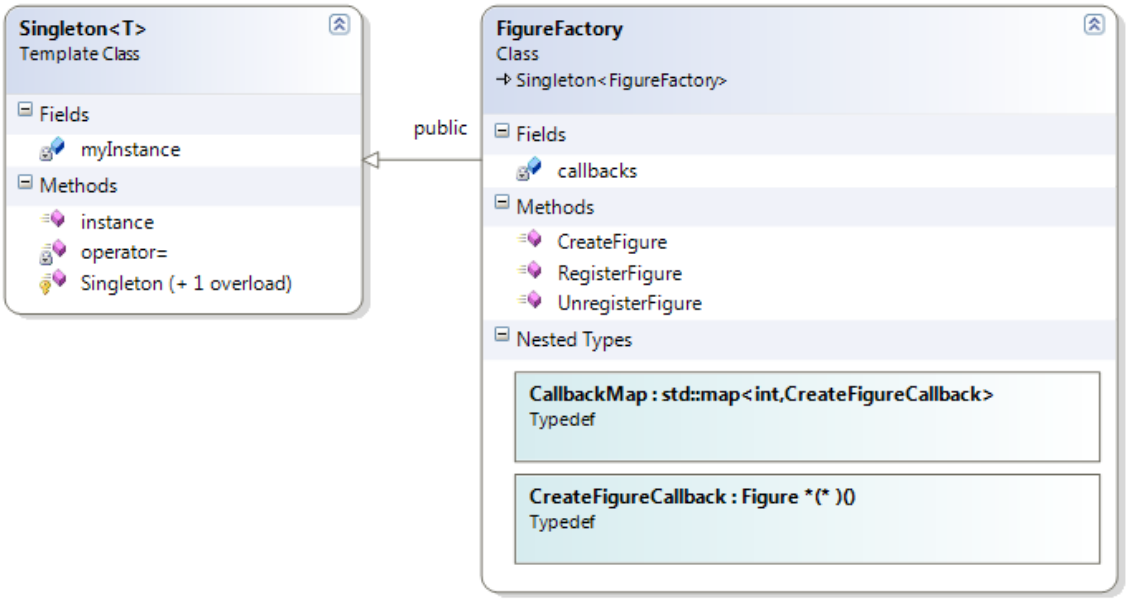

Рис.2. Дополнения к диаграмме классов из работы №1

Самостоятельные задания

1. Реализуйте возможность создания объектов-многоугольников со случайными параметрами с помощью вышеописанной фабрики.

2. Добавьте в каждый класс фигуры функцию записи в тестовый файл. Функции должен передаваться один параметр – ссылка на объект класса *ofstream* (при вызове функции предполагается, что файл уже открыт для записи). Функция вначале должна записать в файл идентификатор типа, а затем все его параметры.

3. Добавьте в каждый класс фигуры функцию чтения из текстового файла. Функции должен передаваться один параметр – ссылка на объект класса *ifstream* (при вызове функции предполагается, что файл уже открыт для чтения).

4. Добавьте в фабрику объектов функцию *CreateFigureFromFile*, которая создаёт объект, читая его из файла. Проверьте корректность работы: запишите последовательно в файл элементы вектора со случайными фигурами, затем прочитайте фигуры из файла и сравните с исходными данными.

5. Создайте класс Drawing, представляющий собой сложный рисунок – то есть набор простых фигур. Реализуйте для него функции записи в файл и чтения из файла.

6*. Реализуйте отображение фигур на экране с использованием любых средств работы с графикой.

7*. Реализуйте запись и чтение объектов в файл формата XML.

ПРАКТИЧЕСКАЯ РАБОТА 4

Разработка классов с настраиваемой функциональностью

Достаточно часто встречается ситуация, когда один и тот же класс должен действовать немного по-разному в зависимости от текущих потребностей. В качестве примера рассмотрим класс *PasArray*, позволяющий работать с массивами в "паскалевском" стиле – то есть при создании массива можно указывать его левую и правую границы. При обращении к элементу массива по индексу выход за границы массива может либо проверяться, либо нет.

Нужна ли такая проверка, вопрос неоднозначный. Наличие проверки замедляет скорость работы. Отсутствие же может привести к непредсказуемым результатам в случае ошибки. Можно найти и много других схожих примеров. В подобных случаях можно дать самому программисту возможность настроить класс с учётом решаемой задачи.

На первый взгляд, сделать это достаточно просто: с помощью оператора *if* проверять определённое поле класса и либо выполнять проверку выхода за границу, либо не выполнять. Однако это не лучшее решение – ведь само выполнение оператора *if* отнимает процессорное время, сравнимое с временем проверки выхода за границы.

Ещё один вариант – использование директив условной компиляции *#ifdef* и *#ifndef*. С помощью директивы *#define* тогда можно указать, какие части кода будут компилироваться, а какие – нет. Однако такой подход тоже не всегда подходит – например, когда в одной программе нужно сделать два объекта-массива, причём в одном из них выполнять проверку выхода за границы, а в другом – нет.

Хорошим вариантом является использование механизма наследования. Создаётся абстрактный базовый класс и его конкретные наследники, в одном из которых проверка реализуется, а в другом – нет. Но такой способ также имеет недостаток. Предположим, что кроме правила проверки выхода индекса за границы, имеется целый ряд других правил – например, проверять ли неотрицательность элементов, разрешать ли модифицировать элементы и др. В этом случае количество классов-наследников будет экспоненциально зависеть от количества правил.

Данный недостаток можно преодолеть, если использовать сочетание обобщенного и объектно-ориентированного программирования. Рассмотрим сначала решение для исходной задачи, когда речь идёт лишь об одном правиле – проверять ли выход за границы или нет.

Создадим два класса *CheckBounds* и *NoCheckBounds*. Каждый класс содержит функцию *check*, где первая функция действительно выполняет проверку, а тело второй пусто. В [1] такие классы называются *стратегиями* (в оригинале – *policy*).

Создадим шаблонный класс *PasArray*, где в качестве одного из шаблонных параметров будет передаваться выбранная стратегия *S*. Класс *PasArray*

будет наследоваться от *S* (в общем случае вместо наследования класс также может включать поле типа S).

Файл pasarray.h

```
#ifndef PasArray_h_
#define PasArray_h_
#include <iostream>
#include <vector>
#include <cassert>

//стратегия "проверять выход за границы"
struct CheckBounds {
  class IndexOutOfBoundsException: public
std::exception {};
  static void Check(int index, int left, int size) {
     if (index < left || index > left + size - 1) {
        throw IndexOutOfBoundsException();
     }
  }
};
//стратегия "не проверять выход за границы"
struct NoCheckBounds {
  static void Check(int, int, int) {}
};

//класс - массив в стиле языка Pascal
//по умолчанию выход за границы не проверяется
template<class T, class CheckingPolicy = NoCheckBounds>
class PasArray : public CheckingPolicy {
  int left;
  std::vector<T> data;
public:
  PasArray(int left, int right) :
    left(left),
    data (right - left + 1) {
    assert (right >= left);
  }
  T& operator [] (int index) {
    Check(index, left, data.size());
    return data[index - left];
  }
};
#endif // PasArray_h_
```

Файл main.cpp

```
#include <iostream>
#include "PasArray.h"
int main() {
  const int left = -10, right = 10;
  const int i = 11; //вызовет выход за границу
  PasArray<int, CheckBounds> a(left, right); //с
проверкой
  try{
    //будет исключительная ситуация, мы ее перехватим
    std::cout << a[i] << std::endl;
  }
  catch(CheckBounds::IndexOutOfBoundsException) {
    std::cerr << "Index is out of bounds" << std::endl;
  }
  PasArray<int, NoCheckBounds> b(left, right); //без
проверки
  try {
    //будет фатальная ошибка или непредсказуемый
результат
    std::cout << b[i] << std::endl;
  }
  catch(...) {
    std::cerr << "You will NOT see this message" <<
std::endl;
  }
  return 0;
}
```

Самостоятельные задания

1. Добавьте в класс *PasArray* конструктор-копировщик, позволяющий писать такой код:

```
PasArray<int, CheckBounds> a(left, right);
PasArray<int, NoCheckBounds> b(a);
```

Реализация копировщика должна быть в одном экземпляре.

Подсказка 1: используйте ключевое слово *template*.

Подсказка 2: для доступа из копировщика к закрытым классам объекта-параметра добавьте в разделе *public* класса *PasArray* строчку:

```
friend class PasArray;
```

2. Добавьте ещё один набор стратегий *CheckMax* / *NoCheckMax*, позволяющих проверять либо (не проверять), что присваиваемое значение не превышает некого *MaxValue*, где *MaxValue* передаётся в конструктор класса (только при использовании данной стратегии). Пример программного кода:

```
// индексы от 1 до 3, значения не проверяются
PasArray<int, CheckBounds, NoCheckMax> a(1, 3);
```

```
  a[2] = 1000; // ошибки нет
  // индексы от 1 до 3, значения не больше 100
  PasArray<int, CheckBounds, CheckMax> b(1, 3, 100);
  b[2] = 1000; // возникнет исключение
```

Подсказка 1. Стратегии *CheckMax* и *NoCheckMax* сами являются шаблонными классами. Поэтому правильное объявление класса PasArray теперь будет следующим:

```
template<class T, class CheckingPolicy = NoCheckBounds,
         template<typename> class CheckMaxPolicy = NoCheck-
Max >
class PasArray : public CheckingPolicy, CheckMaxPolicy<T> {
...
```

Подсказка 2. Внутри класса *PasArray* напишите класс-обёртку над *T* с именем *ProxyT*, в котором реализуйте следующие две функции и конструктор:

```
T& operator = (const T value)
operator T()
ProxyT(T &t, PasArray &pa)
```

В классе *PasArray* измените функцию *operator []*, чтобы она возвращала не *T&*, а *ProxyT*.

Заметим, что возможны и другие правильные варианты реализации.

Заключение

Тематика практических работ в данных методических указаниях была подобрана таким образом, чтобы в процессе их выполнения студенты получили навыки проектирования и разработки информационных систем с использованием современных методик объектно-ориентированного и обобщенного проектирования и программирования, познакомились с шаблонами проектирования, развили навыки программирования на языке высокого уровня C++. Внимательное и добросовестное выполнение практических работ является необходимым условием для успешной сдачи экзамена. Полученные при этом знания, умения и навыки пригодятся в последующих учебных дисциплинах и при подготовке магистерской диссертации, а также будут полезны в профессиональной деятельности в области программирования и информационных технологий.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Александреску, А. Современное проектирование на C++. Обобщенное программирование и прикладные шаблоны проектирования. / А. Александреску. – СПб.: Вильямс, 2002. – 326 с.
2.Приёмы объектно-ориентированного проектирования. Паттерны проектирования. / Э. Гамма, Р. Холм, Р. Джонсон, Дж. Влиссидес. – СПб.: Питер, 2010. – 366 с.
3. Полянский, А. М. Моделирование предметной области автоматизированных систем: учеб. пособие: Ч. 1 / А. М. Полянский. - Вологда: ВоГТУ, 2011. - 83 с.
4. Рамбо, Дж. UML 2.0: объектно-ориентиров. моделирование и разработка / Дж. Рамбо, М. Блаха. – 2-е изд. – СПб. [и др.]: Питер, 2007. – 540 с.
5. Паттерны проектирования / Эрик Фримен, Элизабет Фримен, Кэтти Сьерра, Берт Бейс. – СПб.: Питер, 2011. – 656 с.
6. Академия современного программирования. Язык программирования C++ [Электронный ресурс]. – Режим доступа: http://www.amse.ru/courses/cpp1.
7. The Standard C++ [Электронный ресурс]. – Режим доступа: https://isocpp.org/std/the-standard.

Подписано в печать . Усл. печ. л. . Тираж экз.
Печать офсетная. Бумага офисная. Заказ № _____

Отпечатано: РИО ВоГТУ, г. Вологда, ул. Орлова, 6