## Задание.

1. Перечислите все проблемы, которые вы видите в данном коде:

class Foo

{

public:

/\* j может иметь отрицательное значение и будет выброшено исключение \*/

Foo(int j) { i = new int[j]; }

/\* должно быть ‘delete[] i’ \*/

/\* деструктор должен быть виртуальным, так как предполагается наследование и удаление

объекта-наследника по указателю на базовый класс Foo \*/

~Foo() { delete i; }

private:

int\* i;

};

/\* будет приватное наследование \*/

class Bar: Foo {

public:

/\* j может иметь отрицательное значение и будет выброшено исключение \*/

/\* нет вызова конструктора родительского класса Foo с обязательным параметром j

(будет ошибка компиляции) \*/

Bar(int j) { i = new char[j]; }

/\* должно быть ‘delete[] i’ \*/

~Bar() { delete i; }

private:

char\* i;

};

/\* main должен возвращать код завершения c типом int (будет ошибка компиляции) \*/

void main()

{

Foo\* f=new Foo(100);

/\* из-за приватного наследования ‘Foo->Bar’ не получится создать объект b (будет ошибка компиляции) \*/

Foo\* b=new Bar(200);

/\* ‘затирается’ значение указателя 'int \*i' в объекте f (утечка памяти) \*/

\*f=\*b;

delete f;

/\* так как:

- после копирования '\*f=\*b' у объектов f и b значения указателей i имеют одинаковые значения

- объект f в процессе своего удаления уже освободил память, адресуемую по указателю i

То при удалении b будет двойное освобождение памяти по указателю i \*/

/\* из-за отсутствия виртуального деструктора в Foo, деструктор Bar вызван не будет (утечка памяти) \*/

delete b;

/\* отсутствие ‘return 0’ не является ошибкой, но, на мой взгляд, лучше явно прописать код возврата \*/

}

2. Есть класс CodeGenerator, который умеет генерить код на разных языках.

Предложите рефакторинг с учетом, что количество языков будет расширяться

class CodeGenerator

{

public:

enum Lang {JAVA, C\_PLUS\_PLUS, PHP};

CodeGenerator(Lang language) { \_language=language; }

std::string generateCode()

{

switch(\_language) {

case JAVA: //return generated java code

case C\_PLUS\_PLUS: //return generated C++ code

case PHP: //return generated PHP code

}

throw new std::logic\_error("Bad language");

}

std::string someCodeRelatedThing() // used in generateCode()

{

switch(\_language) {

case JAVA: //return generated java-related stuff

case C\_PLUS\_PLUS: //return generated C++-related stuff

case PHP: //return generated PHP-related stuff

}

throw new std::logic\_error("Bad language");

}

private:

Lang \_language;

}

Можно предложить:

- создать интерфейс ILang

- создать классы для различных языков, поддерживающие интерфейс ILang

- переработать код CodeGenerator так, чтобы класс CodeGenerator работал с объектами языков через интерфейс ILang

#include <iostream>

#include <memory>

class ILang

{

public:

virtual ~ILang() = default;

virtual std::string generateCode() = 0;

virtual std::string someCodeRelatedThing() = 0;

};

class C\_Plus\_Plus : public ILang

{

public:

virtual ~C\_Plus\_Plus() = default;

std::string generateCode() override

{

std::string res("CPP");

// ...

return res;

}

virtual std::string someCodeRelatedThing() override

{

std::string res("CPP");

// ...

return res;

}

};

class CodeGenerator

{

std::unique\_ptr<ILang> lang;

public:

CodeGenerator(std::unique\_ptr<ILang> lang)

: lang(std::move(lang))

{

}

virtual ~CodeGenerator() = default;

std::string generateCode()

{

if (!lang)

throw std::logic\_error("Bad language");

return lang->generateCode();

}

std::string someCodeRelatedThing()

{

if (!lang)

throw std::logic\_error("Bad language");

return lang->someCodeRelatedThing();

}

};

int main()

{

CodeGenerator codeGenerator(std:: unique\_ptr<ILang>(new C\_Plus\_Plus));

std::cout << codeGenerator.generateCode() << std::endl;

return 0;

}

Для создания объектов с интерфейсом ILang можно написать фабрику. Пример реализации фабрики есть в третьем задании.

3. Что не так в этом коде? Перечислите, какие недостатки вы видите. Предложите свой вариант рефакторинга.

#include <stdio.h>

class Feature

{

public:

enum FeatureType {eUnknown, eCircle, eTriangle, eSquare};

Feature() : type(eUnknown), points(0) { }

~Feature()

{

if (points)

delete points;

}

bool isValid()

{

return type != eUnknown;

}

bool read(FILE\* file)

{

if (fread(&type, sizeof(FeatureType), 1, file) != sizeof(FeatureType))

return false;

short n = 0;

switch (type)

{

case eCircle: n = 3; break;

case eTriangle: n = 6; break;

case eSquare: n = 8; break;

default: type = eUnknown; return false;

}

points = new double[n];

if (!points)

return false;

return fread(&points, sizeof(double), n, file) == n\*sizeof(double);

}

void draw()

{

switch (type)

{

case eCircle: drawCircle(points[0], points[1], points[2]); break;

case eTriangle: drawPoligon(points, 6); break;

case eSquare: drawPoligon(points, 8); break;

}

}

protected:

void drawCircle(double centerX, double centerY, double radius);

void drawPoligon(double\* points, int size);

double\* points;

FeatureType type;

};

int main(int argc, char\* argv[])

{

Feature feature;

FILE\* file = fopen("features.dat", "r");

feature.read(file);

if (!feature.isValid())

return 1;

return 0;

}

Архитектурные проблемы:

- класс Feature работает с конкретными фигурами

При добавлении новых типов фигур необходима доработка класса Feature

- большая 'осведомленность' класса Feature

Класс знает как происходит отрисовка для каждой фигуры, сколько нужно точек для отрисовки

- привязка к конкретному типу получения информации

Чтение из файла

- большое количество обязанностей

Сложно тестировать

Технические проблемы

- чтение из файла открывается в режиме 'r'

Более уместным выглядит режим 'rb'

- функция fread используется так, как будто возвращает количество прочитанных байт

Но fread возвращает количество прочитанных блоков

- после выделения памяти 'points = new double[n];' идет проверка на успешность выделения памяти

В общем случае это не нужно (варианты когда код работает без поддержки исключений

не рассматриваем)

- ручное управление памятью в формате 'new/delete'

- в методе read нет проверки на валидность указателя 'File \*file'

- после открытия файл не закрывается

Важной проблемой является то, что Feature работает с конкретными типами фигур. То есть при появлении новой фигуры придется дорабатывать код Feature.

Для исправления этой ситуации можно доработать код так, чтобы Feature работал с объектами обобщенного типа Figure, которые он будет получать от фабрики FigureFactory.

Фабрика FigureFactory должна уметь регистрировать новые типы объектов для создания.

Для отвязки от конкретного источника данных в функции read() следует использовать объект Reader.

Для низкоуровневой отрисовки можно создать объект Drawer.

В качестве числового идентификатора типа фигуры можно использовать значения из enum.

Предлагается следующий алгоритм работы метода read() объекта Feature:

- получение при помощи объекта Reader числового идентификатора фигуры, которую

необходимо использовать

- проверка, есть ли объект Figure для этого идентификатора

- если нет, то запрос у FigureFactory такого объекта и сохранение его

- запрос у полученного объекта Figure количества параметров, которое необходимо вычитать

- получение параметров при помощи объекта Reader

- сохранение объекта Figure и массива параметров в качестве текущих данных

В методе draw() происходит вызов метода draw() у текущего объекта Figure (если таковой имеется)

Объект Feature считается валидным, если определен текущий объект Figure.

Детально можно посмотреть в реализации (файл ‘main.cpp’).

Данный подход имеет следующие преимущества:

- при добавлении новой фигуры не требуется дорабатывать внутреннюю реализацию Feature и FigureFactory

- при необходимости чтения не из файла можно добавить новый Reader без доработки внутренней реализации Feature

- можно легко тестировать Feature путем использования mock объектов

Предлагаемый вариант выполнен в рабочем стиле – с namespace, комментариями в стиле doxigen, mock объектами для тестирования – листинг в отдельном файле (‘main.cpp’).

Логирование ошибок не добавлялось.