ФЕДЕРАЛЬНОЕ АГЕНСТВО ЖЕЛЕЗНОДОРОЖНОГО ТРАНСПОРТА

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ПУТЕЙ СООБЩЕНИЯ Императора Александра I»

Кафедра «Информационные и вычислительные системы»

Дисциплина «Программирование С++»

**ОТЧЁТ**

**ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3**

ВАРИАНТ 19

|  |  |
| --- | --- |
| Выполнил студент  Факультет: АИТ  Группа: ИВБ-211 | Шефнер А. |
| Проверил: | Проузин О.В. |

**Санкт-Петербург**

**2023**

Оценочный лист результатов ЛР № 3

Ф.И.О. студента \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Шефнер Альберт\_\_\_\_\_

Группа \_\_\_\_\_\_\_\_\_\_\_\_\_ИВБ-211\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№**  **п/п** | **Материалы необходимые для оценки знаний, умений**  **и навыков** | **Показатель**  **оценивания** | **Критерии**  **Оценивания** | **Шкала оценивания** | **Оценка** |
| 1 | Лабораторная работа№3 | Соответствие методике выполнения | Соответствует | 7 |  |
| Не соответствует | 0 |
| Срок выполнения | Выполнена в срок | 2 |  |
| Выполнена с опозданием на 2 недели | 0 |
| оформление | Соответствует требованиям | 1  0 |  |
| Не соответствует |  |
|  | **ИТОГО количество баллов** |  |  | 10 |  |

Доцент кафедры

«Информационные и вычислительные

системы» Проурзин О.В. «\_\_» \_\_\_\_\_\_\_\_\_\_2023 г.

**Цели работы:**

* Использовать структуры.
* Применить наследование и полиморфизм подтипов.

**Задание**

Написать приложение для работы с сотрудниками предприятия. Сотрудники бывают разных видов с разными характеристиками и способы расчёта зарплаты. В приложении можно создать новый список сотрудников, добавлять и удалять сотрудников, посчитать зарплату каждого сотрудника и вывести таблицу зарплат с общей суммой всех зарплат.

**Используемые средства**

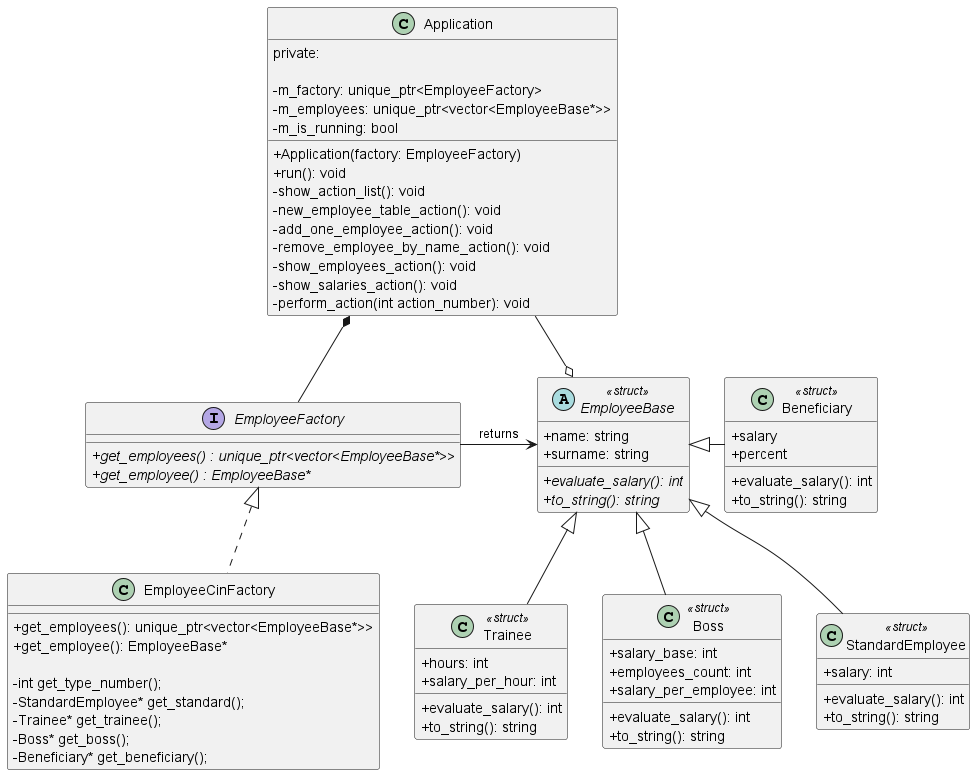
В качестве интегрированной среды разработки использовалась JetBrains CLion.

Для работы в консоли с потоками ввода-вывода использовалась стандартная библиотека <iostream>, а также библиотека <iomanip> инструментов для работы с форматированием.

Для управления памятью используется стандартная библиотека memory и умные указатели (unique\_ptr).

В программе есть несколько файлов, для сборки проекта использовался CMake в связке с Ninja.

Для создания UML-диаграммы использовался плагин с поддержкой языка разметки PlantUML и отрисовки диаграммы на основе кода на языке разметки.

**UML-диаграмма программы:**

**Код программы с комментариями**

**main.cpp**

#include "Application.h"  
#include "employees/EmployeeCinFactory.h"  
  
int main() {  
 // Создание нового приложения с фабрикой, которая получает работников из консоли  
 auto app = new Application(new EmployeeCinFactory);  
 app->run();  
  
 return 0;  
}

**Application.h**

#ifndef LAB\_3\_APPLICATION\_H  
#define LAB\_3\_APPLICATION\_H  
  
#include "employees/EmployeeFactory.h"  
#include "array"  
#include "memory"  
  
// Главный класс программы  
class Application {  
public:  
 // У класса application есть всего одно поле, которое задаётся через конструктор.  
 // Делать ApplicationBuilder будет избыточно.  
 explicit Application(EmployeeFactory\* factory) {  
 m\_factory = std::unique\_ptr<EmployeeFactory>(factory);  
 }  
  
 // Отправная точка приложения  
 void run();  
  
private:  
 // Абстрактная фабрика, которая отвечает за получение сотрудников.  
 // Это может быть консоль, файл, база данных, api сервера, в  
 // зависимости от реализации фабрики  
 std::unique\_ptr<EmployeeFactory> m\_factory = nullptr;

// Список работников  
 std::unique\_ptr<std::vector<EmployeeBase\*>> m\_employees = std::unique\_ptr<std::vector<EmployeeBase\*>>(new std::vector<EmployeeBase\*>);

bool m\_is\_running = false;  
  
 void show\_action\_list() const;  
  
 // action означает один из 5 пунктов в меню приложения.  
 void new\_employee\_table\_action();  
  
 void add\_one\_employee\_action();  
  
 void remove\_employee\_by\_name\_action();  
  
 void show\_employees\_action();  
  
 void show\_salaries\_action();  
  
 void perform\_action(int action\_number);  
};  
  
#endif //LAB\_3\_APPLICATION\_H

**Application.cpp**

#include <iostream>  
#include <iomanip>  
#include "Application.h"  
  
// Application решил сильно не усложнять объектно-ориентированной логикой,  
// так как в конечном счёте Я интерфейс просто навелосипедил.  
void Application::run() {  
 if(m\_factory == nullptr) {  
 std::cout << "EmployeeFactory must be not null!";  
 return;  
 }  
  
 m\_is\_running = true;  
 while (m\_is\_running) {  
 system("cls");  
  
 show\_action\_list();  
 int action\_number;  
 std::cin >> action\_number;  
 perform\_action(action\_number);  
  
 system("pause");  
 }  
}  
  
void Application::perform\_action(int action\_number) {  
 system("cls");  
 switch (action\_number) {  
 case 1:  
 new\_employee\_table\_action();  
 break;  
 case 2:  
 add\_one\_employee\_action();  
 break;  
 case 3:  
 remove\_employee\_by\_name\_action();  
 break;  
 case 4:  
 show\_employees\_action();  
 break;  
 case 5:  
 show\_salaries\_action();  
 break;  
 case 6:  
 m\_is\_running = false;  
 break;  
 default:  
 std::cout << "Wrong action number! Must be from 1 to 4\n";  
 }  
}  
  
void Application::show\_action\_list() const {  
 std::cout << "Enter action number you want to perform:\n"  
 << "1 - new employees table.\n"  
 << "2 - add one employee.\n"  
 << "3 - remove employee by name.\n"  
 << "4 - show all employees.\n"  
 << "5 - show only salaries.\n"  
 << "6 - exit from program.\n";  
}  
  
void Application::new\_employee\_table\_action() {  
 m\_employees = m\_factory->get\_employees();  
}  
  
void Application::add\_one\_employee\_action() {  
 m\_employees->push\_back(m\_factory->get\_employee());  
}  
  
void Application::show\_employees\_action() {  
 std::cout << "Employees:\n";  
 for(int i = 0; i < m\_employees->size(); i++) {  
 std::cout << i + 1 << ". " << (\*m\_employees)[i]->to\_string() << '\n';  
 }  
 std::cout << std::endl;  
}  
  
void Application::show\_salaries\_action() {  
 long sum = 0;  
 std::cout << "Employees salaries:\n" <<  
 "n | name | salary\n" <<  
 "----------------------------------------\n";  
 for(int i = 0; i < m\_employees->size(); i++) {  
 int salary = (\*m\_employees)[i]->evaluate\_salary();  
 sum += salary;  
 std::cout << i + 1 << " | " <<  
 std::setw(30) << std::left <<  
 (\*m\_employees)[i]->name + " " + (\*m\_employees)[i]->surname << "| " <<  
 std::setw(10) << std::right <<  
 salary << ".\n";  
 }  
 std::cout << "----------------------------------------\n" <<  
 "Total: " << sum << '\n' << std::endl;  
}  
  
void Application::remove\_employee\_by\_name\_action() {  
 std::cout << "Enter name and surname divided by whitespace:\n";  
 std::string name, surname;  
 std::cin >> name >> surname;  
 int index = -1;  
 for(int i = 0; i < m\_employees->size(); i++) {  
 if((\*m\_employees)[i]->name == name && (\*m\_employees)[i]->surname == surname) {  
 index = i;  
 break;  
 }  
 }  
 if(index != -1) {  
 m\_employees->erase(m\_employees->begin() + index);  
 std::cout << "Employee removed successfully.\n";  
 } else {  
 std::cout << "No employee with this name.\n";  
 }  
}

**employee/employees.cpp**

#include <sstream>  
#include <string>  
  
// Базовая структура работника  
struct EmployeeBase {  
 std::string name;  
 std::string surname;  
  
 // Абстрактный метод вычисления зарплаты работника  
 // по его характеристикам. Переопределяется в  
 // дочерних структурах  
 virtual int evaluate\_salary() = 0;  
  
 // Абстрактный метод преобразования характеристик структуры в string  
 virtual std::string to\_string() = 0;  
};  
  
// Дочерняя структура стандартного работника  
struct StandardEmployee : public EmployeeBase {  
 int salary;  
  
 explicit StandardEmployee(int salary) {  
 this->salary = salary;  
 }  
  
 std::string to\_string() override {  
 std::stringstream stringstream;  
 stringstream  
 << "Standard employee "  
 << name << " " << surname  
 << ". Overall salary: "  
 << salary;  
 return std::string(stringstream.str());  
 }  
  
 int evaluate\_salary() override {  
 return salary;  
 }  
};  
  
// Дочерняя структура стажёра  
struct Trainee : public EmployeeBase {  
 int hours;  
 int salary\_per\_hour;  
  
 Trainee(int hours, int salary\_per\_hour) {  
 this->hours = hours;  
 this->salary\_per\_hour = salary\_per\_hour;  
 }  
  
 std::string to\_string() override {  
 std::stringstream stringstream;  
 stringstream  
 << "Trainee "  
 << name << " " << surname  
 << ". Hours: " << hours  
 << ", salary per hour: " << salary\_per\_hour  
 << ". Overall salary: " << evaluate\_salary();  
 return std::string(stringstream.str());  
 }  
  
 int evaluate\_salary() override {  
 return hours \* salary\_per\_hour;  
 }  
};  
  
// Дочерняя структура начальника  
struct Boss : public EmployeeBase {  
 int salary\_base;  
 int employees\_count;  
 int salary\_per\_employee;  
  
 Boss(int salary\_base, int employees\_count, int salary\_per\_employee) {  
 this->salary\_base = salary\_base;  
 this->employees\_count = employees\_count;  
 this->salary\_per\_employee = salary\_per\_employee;  
 }  
  
 std::string to\_string() override {  
 std::stringstream stringstream;  
 stringstream  
 << "Boss "  
 << name << " " << surname  
 << ". Salary base: " << salary\_base  
 << ", employees count: " << employees\_count  
 << ", salary per employee: " << salary\_per\_employee  
 << ". Overall salary: " << evaluate\_salary();  
 return std::string(stringstream.str());  
 }  
  
 int evaluate\_salary() override {  
 return salary\_base + employees\_count \* salary\_per\_employee;  
 }  
};  
  
// Дочерняя структура льготника  
struct Beneficiary : public EmployeeBase {  
 int salary;  
 int percent;  
  
 Beneficiary(int salary, int percent) {  
 this->salary = salary;  
 this->percent = percent;  
 }  
  
 std::string to\_string() override {  
 std::stringstream stringstream;  
 stringstream  
 << "Beneficiary "  
 << name << " " << surname  
 << ". Salary: " << salary  
 << ", percent: " << percent << '%'  
 << ". Overall salary: " << evaluate\_salary();  
 return std::string(stringstream.str());  
 }  
  
 int evaluate\_salary() override {  
 return (int)(salary \* (1 + percent \* 0.01));  
 }  
};

**employee/EmployeeFactory.h**

#ifndef LAB\_3\_EMPLOYEEFACTORY\_H  
#define LAB\_3\_EMPLOYEEFACTORY\_H  
  
#include <vector>  
#include <memory>  
  
#include "employees.cpp"  
  
// Интерфейс фабрики работников. Ключевого слова interface в C++  
// нет, и Я не знаю как заставить дочерние классы переопределять эти  
// чистые виртуальные методы. Как я понял, если не переопределить,  
// то просто нельзя будет создать объект дочернего класса.  
class EmployeeFactory {  
public:  
 // Получить много работников  
 virtual std::unique\_ptr<std::vector<EmployeeBase\*>> get\_employees() = 0;  
  
 //получить одного работника  
 virtual EmployeeBase\* get\_employee() = 0;  
};  
  
  
#endif //LAB\_3\_EMPLOYEEFACTORY\_H

**employee/EmployeeCinFactory.h**

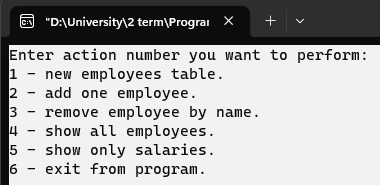
#ifndef LAB\_3\_EMPLOYEECINFACTORY\_H  
#define LAB\_3\_EMPLOYEECINFACTORY\_H  
  
#include <array>  
#include <vector>  
#include "EmployeeFactory.h"  
  
// "Консольная" фабрика работников, реализующая интерфейс EmployeeFactory  
class EmployeeCinFactory : public EmployeeFactory {  
public:  
 std::unique\_ptr<std::vector<EmployeeBase\*>> get\_employees() override;  
  
 EmployeeBase\* get\_employee() override;  
  
private:  
 int get\_type\_number();  
  
 StandardEmployee\* get\_standard();  
  
 Trainee\* get\_trainee();  
  
 Boss\* get\_boss();  
  
 Beneficiary\* get\_beneficiary();  
};  
  
#endif //LAB\_3\_EMPLOYEECINFACTORY\_H

**employee/EmployeeCinFactory.cpp**

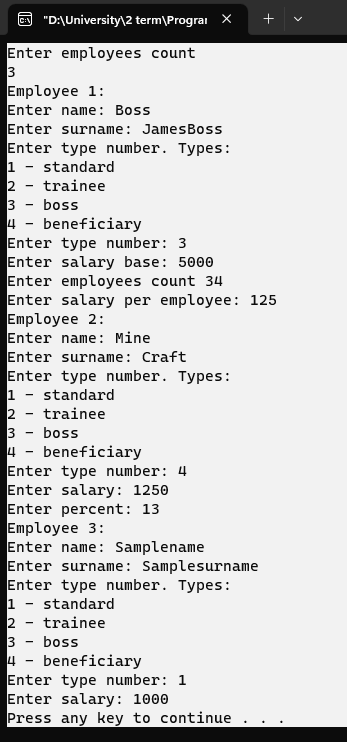
#include <iostream>  
#include "EmployeeCinFactory.h"  
#include "memory.h"  
  
// Я решил реализовать фабрику в функциональном стиле  
  
// Получение вектора работников  
std::unique\_ptr<std::vector<EmployeeBase\*>> EmployeeCinFactory::get\_employees() {  
 int employee\_count;  
 std::cout << "Enter employees count\n";  
 std::cin >> employee\_count;  
  
 // Делаю умный указатель на вектор, пусть он там сам памятью занимается.  
 auto employees = std::make\_unique<std::vector<EmployeeBase\*>>();  
  
 // заполнение вектора  
 for (int i = 0; i < employee\_count; i++) {  
 std::cout << "Employee " << i + 1 << ":\n";  
 employees->push\_back(get\_employee());  
 }  
  
 return employees;  
}  
  
EmployeeBase \*EmployeeCinFactory::get\_employee() {  
 EmployeeBase\* employee;  
  
 // Имя и фамилия считываются заранее, чтобы избежать дублирование кода,  
 // но задаются в самом конце так как структуры на данный момент нет  
 auto name = std::string();  
 auto surname = std::string();  
 std::cout << "Enter name: ";  
 std::cin >> name;  
 std::cout << "Enter surname: ";  
 std::cin >> surname;  
  
 // Получение типа  
 std::cout << "Enter type number. Types:\n";  
 std::cout << "1 - standard\n" << "2 - trainee\n" << "3 - boss\n" << "4 - beneficiary\n";  
 int type\_number = get\_type\_number();  
  
 // Получение сотрудника нужного типа через switch.  
 switch (type\_number) {  
 case 1:  
 employee = get\_standard();  
 break;  
 case 2:  
 employee = get\_trainee();  
 break;  
 case 3:  
 employee = get\_boss();  
 break;  
 case 4:  
 employee = get\_beneficiary();  
 break;  
 default:  
 // IDE ругается, что я дефолт не учитываю, хотя я его учёл в другой функции.  
 exit(1);  
 }  
  
 //После получения структуры можно установить имя и фамилию  
 employee->name = name;  
 employee->surname = surname;  
  
 return employee;  
}  
  
// Получение номера типа сотрудника.  
int EmployeeCinFactory::get\_type\_number() {  
 int num;  
 while(true) {  
 std::cout << "Enter type number: ";  
 std::cin >> num;  
 if(num >= 1 && num <= 4) break;  
 else std::cout << "Wrong type number.\n";  
 }  
 return num;  
}  
  
// Методы на каждый тип структуры, работают предельно просто  
StandardEmployee\* EmployeeCinFactory::get\_standard() {  
 std::cout << "Enter salary: ";  
 int salary;  
 std::cin >> salary;  
 return new StandardEmployee(salary);  
}  
  
Trainee \*EmployeeCinFactory::get\_trainee() {  
 int hours;  
 int salary\_per\_hour;  
 std::cout<<"Enter hours count: ";  
 std::cin >> hours;  
 std::cout << "Enter salary per hour: ";  
 std::cin >> salary\_per\_hour;  
 return new Trainee(hours, salary\_per\_hour);  
}  
  
Boss \*EmployeeCinFactory::get\_boss() {  
 int salary\_base;  
 int employees\_count;  
 int salary\_per\_employee;  
 std::cout << "Enter salary base: ";  
 std::cin >> salary\_base;  
 std::cout << "Enter employees count ";  
 std::cin >> employees\_count;  
 std::cout << "Enter salary per employee: ";  
 std::cin >> salary\_per\_employee;  
 return new Boss(salary\_base, employees\_count, salary\_per\_employee);  
}  
  
Beneficiary \*EmployeeCinFactory::get\_beneficiary() {  
 int salary;  
 int percent;  
 std::cout << "Enter salary: ";  
 std::cin >> salary;  
 std::cout << "Enter percent: ";  
 std::cin >>percent;  
 return new Beneficiary(salary, percent);  
}

**Тестовые примеры**

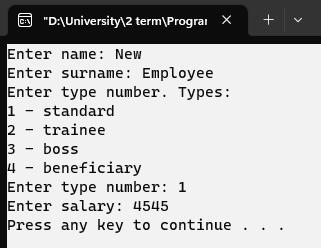
**Меню приложения:**

****

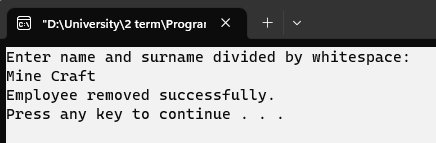
**Создание новой таблицы сотрудников:**

****

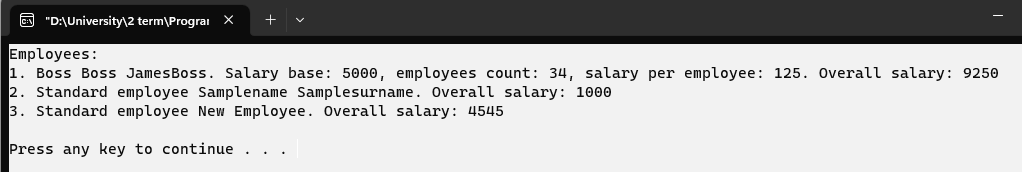
**Добавление одного сотрудника:**

****

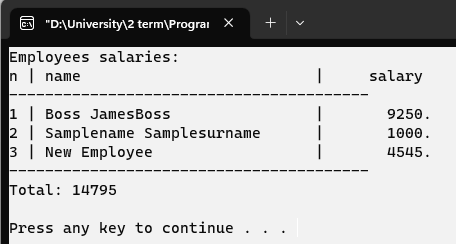
**Удаление сотрудника из списка:**

****

**Вывод полной информации о всех сотрудниках:**

****

**Вывод таблицы зарплат:**

****