Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационных технологий и управления

ОТЧЕТ

к лабораторной работе №6

**«Особенности использования ООПиП и паттернов проектирования»**

|  |  |
| --- | --- |
|  | |
|  |
|  |
|  |
|  |  | | |
|  | | |  | |  |
|  | | |  | |  |
| Студент гр. 324402 | | |  | | Цевелюк А.И. |
| Проверила | | |  | | Купрейчик А.С. |

Минск 2024

**1 Цель работы**

Цель работы:изучить особенности использования ООПиП и паттернов проектирования.

**2 Теоретические сведения по лабораторной работе**

Основы объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) — это парадигма программирования, в которой данные и код объединяются в объекты. Основные принципы ООП включают:

- Инкапсуляция — сокрытие деталей реализации объекта и предоставление к ним доступа только через публичные методы.

- Наследование — возможность создания новых классов на основе уже существующих, что упрощает повторное использование кода.

- Полиморфизм — способность объекта обрабатывать вызовы методов по-разному в зависимости от своего типа.

- Абстракция — создание абстрактных классов или интерфейсов для описания общих характеристик без указания конкретной реализации.

При проектировании объектно-ориентированных программ важно продумать классы, их методы и взаимодействия, а также минимизировать необходимость перепроектирования в будущем. Для этого используются паттерны проектирования.

Паттерны проектирования

Паттерны проектирования — это проверенные решения часто возникающих задач проектирования программных систем. Они делятся на три основные категории:

1 Порождающие паттерны — отвечают за процесс создания объектов:

- Singleton — гарантирует, что класс имеет только один экземпляр.

- Factory Method — предоставляет интерфейс для создания объекта, но оставляет подклассам решение о том, какой именно объект создавать.

- Abstract Factory — позволяет создавать семейства связанных объектов без указания их конкретных классов.

- Builder — отделяет процесс создания сложного объекта от его представления.

- Prototype — создает новый объект путем копирования существующего объекта (прототипа).

2 Структурные паттерны — определяют отношения между объектами, облегчают их взаимодействие и улучшение структуры программы:

- Adapter — преобразует интерфейс одного класса в интерфейс, который ожидает клиент.

- Bridge — разделяет абстракцию и реализацию, что позволяет изменять их независимо.

- Composite — объединяет объекты в древовидную структуру, чтобы работать с ними как с единым целым.

- Decorator — добавляет новые функциональные возможности объекту динамически.

- Facade — предоставляет простой интерфейс к сложной системе классов.

- Flyweight — минимизирует использование памяти за счет совместного использования объектов.

- Proxy — предоставляет объект-заместитель, который контролирует доступ к реальному объекту.

3 Поведенческие паттерны — описывают способы взаимодействия объектов, передачи данных и управления поведением программы:

- Chain of Responsibility — передает запрос по цепочке обработчиков.

- Command — инкапсулирует запрос в виде объекта, позволяя параметризовать клиентов запросами.

- Iterator — предоставляет последовательный доступ к элементам коллекции.

- Mediator — централизует взаимодействие между объектами.

- Observer — создает механизм подписки для оповещения множества объектов о событиях.

- State — позволяет объекту изменять свое поведение при изменении его внутреннего состояния.

- Strategy — определяет набор алгоритмов и помещает их в отдельные классы, что позволяет подменять алгоритмы в ходе работы программы.

- Template Method — задает общую структуру алгоритма, оставляя детали подклассам.

- Visitor — добавляет новую операцию к объектам, не изменяя их классы.

Порождающие паттерны: особенности реализации в C++

Одной из важных задач при проектировании программ является создание объектов в зависимости от условий выполнения программы. Например, тип объекта может быть неизвестен на этапе компиляции и определяться только во время выполнения. Для таких случаев применяются порождающие паттерны, которые делегируют решение о том, какой объект создавать, фабричным методам.

Фабричный метод (Factory Method)

Фабричный метод — это паттерн, который предоставляет интерфейс для создания объектов в суперклассе, но позволяет подклассам изменять тип создаваемого объекта. Он скрывает процесс создания объектов от клиента, предоставляя ему базовый интерфейс для работы с этими объектами.

Пример:

enum Buildings {FARM, FORGE, MILL};

class Building {

public:

virtual ~Building() {}

};

class Farm : public Building {};

class Forge : public Building {};

Building\* makeBuilding(Buildings type) {

switch (type) {

case FARM: return new Farm();

case FORGE: return new Forge();

// другие типы зданий

}

}

В этом примере тип создаваемого объекта выбирается на основе значения перечисления Buildings, а сам процесс создания скрыт внутри фабричного метода.

Фабричный метод с аргументами

Этот вариант фабричного метода позволяет передавать параметры конструкторам создаваемых объектов, что делает его более универсальным.

Полиморфная фабрика

Полиморфная фабрика — это вариант фабричного метода, при котором процесс создания объектов зависит от конкретного класса. Это позволяет делегировать создание объектов их производным классам, что упрощает код и делает его более гибким.

Пример полиморфной фабрики:

class Unit {};

class Knight : public Unit {};

class Building {

public:

virtual Unit\* makeUnit() const = 0;

};

class Castle : public Building {

public:

Knight\* makeUnit() const override { return new Knight; }

};

Виртуальные конструкторы не поддерживаются в C++, так как компилятор должен знать точный размер объекта на этапе компиляции. Однако паттерн полиморфное копирование позволяет добиться похожего поведения, создавая копии объектов во время выполнения с помощью фабричных методов.

Пример полиморфного копирования:

class Base {

public:

virtual Base\* clone() const = 0;

};

class Derived : public Base {

public:

Derived\* clone() const override { return new Derived(\*this); }

};

Шаблонный метод и фабрика

Шаблонный метод — это паттерн, который определяет алгоритм, оставляя подклассам возможность реализовать отдельные его шаги. В сочетании с фабрикой это позволяет гибко создавать объекты и управлять процессами в программе.

Таким образом, использование паттернов проектирования и принципов ООП делает программы более гибкими, модульными и простыми для поддержки и расширения.

**3 Формулировка индивидуального задания**

Во всех вариантах индивидуальных заданий необходимо создать возможность конструирования в приложении 5 различных типов объектов классов по предметной области, указанной в таблице ниже, с использованием паттернов «фабричный метод», «фабричный метод с аргументами», «фабричный конструктор», «полиморфная фабрика», «полиморфное копирование». Для всех типов необходимо реализовать возможность добавления, удаления, редактирования, вывода содержимого объектов на экран и в файл.

**4 Код решения индивидуального задания**

// ReSharper disable CppClangTidyCppcoreguidelinesSpecialMemberFunctions

// ReSharper disable CppClangTidyClangDiagnosticDeprecatedCopyWithDtor

// ReSharper disable CppClangTidyConcurrencyMtUnsafe

#include <iostream>

#include <vector>

#include <string>

#include <memory>

#include <fstream>

#include <unordered\_map>

class document\_lab6

{

public:

virtual void display() const = 0;

virtual std::unique\_ptr<document\_lab6> clone() const = 0;

virtual ~document\_lab6() = default;

};

class passport final : public document\_lab6

{

std::string passport\_number\_;

public:

explicit passport(std::string number) : passport\_number\_(std::move(number)) {}

void display() const override

{

std::cout << "Passport: " << passport\_number\_ << '\n';

}

std::unique\_ptr<document\_lab6> clone() const override

{

return std::make\_unique<passport>(\*this);

}

};

class visa final : public document\_lab6

{

std::string visa\_type\_;

int duration\_;

public:

visa(std::string type, const int duration) : visa\_type\_(std::move(type)), duration\_(duration) {}

void display() const override

{

std::cout << "Visa: " << visa\_type\_ << " for " << duration\_ << " days" << '\n';

}

std::unique\_ptr<document\_lab6> clone() const override

{

return std::make\_unique<visa>(\*this);

}

};

class residence\_permit final : public document\_lab6

{

std::string permit\_number\_;

public:

explicit residence\_permit(std::string number) : permit\_number\_(std::move(number)) {}

void display() const override

{

std::cout << "Residence Permit: " << permit\_number\_ << '\n';

}

std::unique\_ptr<document\_lab6> clone() const override

{

return std::make\_unique<residence\_permit>(\*this);

}

};

class application\_form final : public document\_lab6

{

std::string application\_id\_;

public:

explicit application\_form(std::string id) : application\_id\_(std::move(id)) {}

void display() const override

{

std::cout << "Application Form ID: " << application\_id\_ << '\n';

}

std::unique\_ptr<document\_lab6> clone() const override

{

return std::make\_unique<application\_form>(\*this);

}

};

class migration\_card final : public document\_lab6

{

std::string card\_number\_;

public:

explicit migration\_card(std::string number) : card\_number\_(std::move(number)) {}

void display() const override

{

std::cout << "Migration Card: " << card\_number\_ << '\n';

}

std::unique\_ptr<document\_lab6> clone() const override

{

return std::make\_unique<migration\_card>(\*this);

}

};

class document\_factory

{

public:

virtual std::unique\_ptr<document\_lab6> create\_document(const std::vector<std::string>& args) const = 0;

virtual ~document\_factory() = default;

};

class passport\_factory final : public document\_factory

{

public:

std::unique\_ptr<document\_lab6> create\_document(const std::vector<std::string>& args) const override

{

return std::make\_unique<passport>(args[0]);

}

};

class visa\_factory final : public document\_factory

{

public:

std::unique\_ptr<document\_lab6> create\_document(const std::vector<std::string>& args) const override

{

return std::make\_unique<visa>(args[0], std::stoi(args[1]));

}

};

class residence\_permit\_factory final : public document\_factory

{

public:

std::unique\_ptr<document\_lab6> create\_document(const std::vector<std::string>& args) const override

{

return std::make\_unique<residence\_permit>(args[0]);

}

};

class application\_form\_factory final : public document\_factory

{

public:

std::unique\_ptr<document\_lab6> create\_document(const std::vector<std::string>& args) const override

{

return std::make\_unique<application\_form>(args[0]);

}

};

class migration\_card\_factory final : public document\_factory

{

public:

std::unique\_ptr<document\_lab6> create\_document(const std::vector<std::string>& args) const override

{

return std::make\_unique<migration\_card>(args[0]);

}

};

class document\_registry

{

std::unordered\_map<std::string, std::unique\_ptr<document\_factory>> factories\_;

public:

void register\_factory(const std::string& document\_type, std::unique\_ptr<document\_factory> factory)

{

factories\_[document\_type] = std::move(factory);

}

std::unique\_ptr<document\_lab6> create\_document(const std::string& document\_type, const std::vector<std::string>& args) const

{

const auto it = factories\_.find(document\_type);

if (it != factories\_.end())

return it->second->create\_document(args);

throw std::runtime\_error("Unknown document type: " + document\_type);

}

};

class document\_manager

{

std::vector<std::unique\_ptr<document\_lab6>> documents\_;

public:

void add\_document(std::unique\_ptr<document\_lab6> doc)

{

documents\_.push\_back(std::move(doc));

}

void remove\_document(const size\_t index)

{

if (index < documents\_.size())

documents\_.erase(documents\_.begin() + static\_cast<long long>(index));

}

void edit\_document(const size\_t index, std::unique\_ptr<document\_lab6> doc)

{

if (index < documents\_.size())

documents\_[index] = std::move(doc);

}

void display\_all() const

{

for (size\_t i = 0; i < documents\_.size(); ++i)

{

std::cout << i + 1 << ". ";

documents\_[i]->display();

}

}

void save\_to\_file(const std::string& filename) const

{

std::ofstream out\_file(filename);

if (out\_file.is\_open())

{

for (const auto& doc : documents\_)

{

out\_file << "Document: ";

doc->display();

}

}

}

};

void show\_menu()

{

std::cout << "\n===== Document Management System =====\n";

std::cout << "1. Add document\n";

std::cout << "2. Remove document\n";

std::cout << "3. Edit document\n";

std::cout << "4. Display all documents\n";

std::cout << "5. Save documents to file\n";

std::cout << "6. Exit\n";

std::cout << "Choose an option: ";

}

std::string choose\_document\_type()

{

std::cout << "\nChoose document type:\n";

std::cout << "passport, visa, residence\_permit, application\_form, migration\_card\n";

std::cout << "Enter document type: ";

std::string doc\_type;

std::cin >> doc\_type;

return doc\_type;

}

std::vector<std::string> get\_document\_data(const std::string& document\_type)

{

std::vector<std::string> data;

std::string input;

if (document\_type == "passport")

{

std::cout << "Enter passport number: ";

std::cin >> input;

data.push\_back(input);

}

else if (document\_type == "visa")

{

std::cout << "Enter visa type: ";

std::cin >> input;

data.push\_back(input);

std::cout << "Enter visa duration (in days): ";

std::cin >> input;

data.push\_back(input);

}

else if (document\_type == "residence\_permit")

{

std::cout << "Enter permit number: ";

std::cin >> input;

data.push\_back(input);

}

else if (document\_type == "application\_form")

{

std::cout << "Enter application form ID: ";

std::cin >> input;

data.push\_back(input);

}

else if (document\_type == "migration\_card")

{

std::cout << "Enter migration card number: ";

std::cin >> input;

data.push\_back(input);

}

return data;

}

void end()

{

system("pause");

system("cls");

}

int main()

{

document\_manager manager;

document\_registry registry;

registry.register\_factory("passport", std::make\_unique<passport\_factory>());

registry.register\_factory("visa", std::make\_unique<visa\_factory>());

registry.register\_factory("residence\_permit", std::make\_unique<residence\_permit\_factory>());

registry.register\_factory("application\_form", std::make\_unique<application\_form\_factory>());

registry.register\_factory("migration\_card", std::make\_unique<migration\_card\_factory>());

bool running = true;

while (running)

{

show\_menu();

int option;

std::cin >> option;

switch (option)

{

case 1:

{

std::string doc\_type = choose\_document\_type();

auto data = get\_document\_data(doc\_type);

try

{

manager.add\_document(registry.create\_document(doc\_type, data));

}

catch (const std::exception& e)

{

std::cout << e.what() << '\n';

}

break;

}

case 2:

{

size\_t index;

manager.display\_all();

std::cout << "Enter document number to remove: ";

std::cin >> index;

manager.remove\_document(index - 1);

break;

}

case 3:

{

size\_t index;

manager.display\_all();

std::cout << "Enter document number to edit: ";

std::cin >> index;

std::string doc\_type = choose\_document\_type();

auto data = get\_document\_data(doc\_type);

try

{

manager.edit\_document(index - 1, registry.create\_document(doc\_type, data));

}

catch (const std::exception& e)

{

std::cout << e.what() << '\n';

}

break;

}

case 4:

manager.display\_all();

break;

case 5:

{

std::string filename;

std::cout << "Enter filename: ";

std::cin >> filename;

manager.save\_to\_file(filename);

break;

}

case 6:

running = false;

std::cout << "Have a nice day!\n";

break;

default:

std::cout << "Invalid option!\n";

break;

}

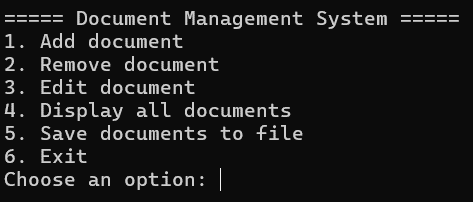
if (running) end();

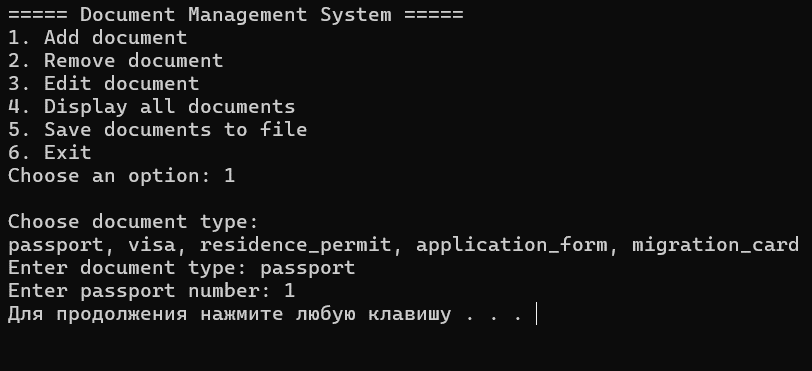
}

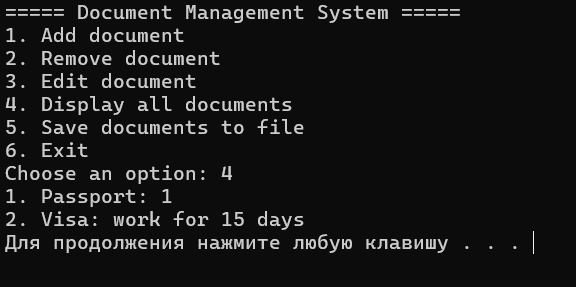
return 0;

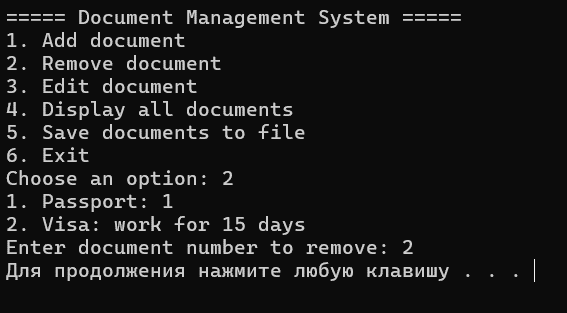
}

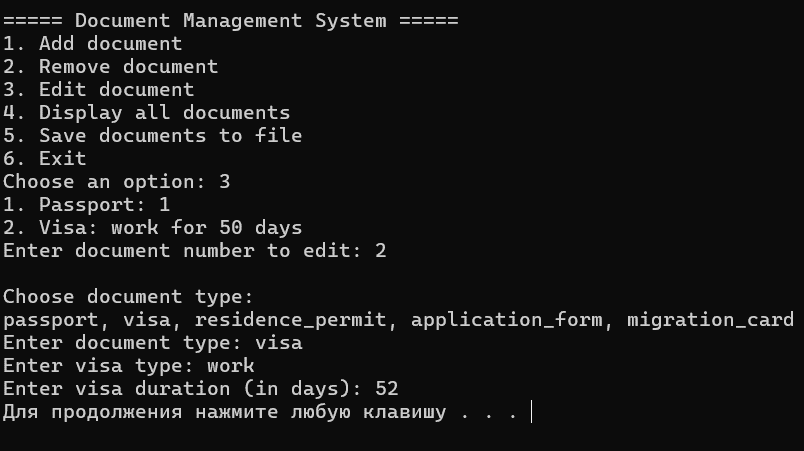
**5 Скриншоты выполнения индивидуального задания**

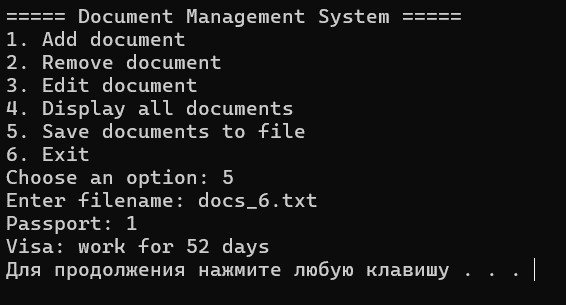


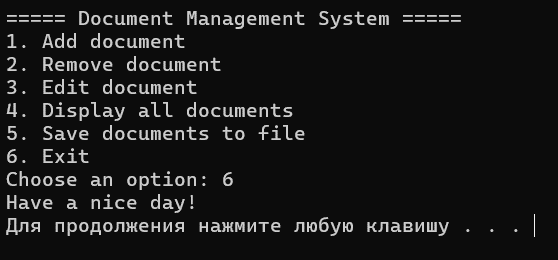












**6 Ответы на контрольные вопросы**

1. В чем преимущество объектно-ориентированного проектирования?

Основные преимущества объектно-ориентированного проектирования (ООП) заключаются в следующем:

- Модульность: ООП позволяет разделить программный код на отдельные модули — классы, которые инкапсулируют данные и методы. Это способствует улучшению читаемости кода и облегчает его поддержку.

Повторное использование кода: Благодаря наследованию и полиморфизму классы можно использовать повторно, создавая на их основе новые классы.

- Гибкость: Изменения в одной части программы минимально влияют на другие части. Это достигается благодаря инкапсуляции, которая скрывает реализацию от внешнего мира и позволяет легко изменять детали, не нарушая работу других компонентов.

- Поддержка масштабирования: Использование ООП упрощает добавление новых функций, объектов и классов, что важно для крупных программных систем.

- Повышенная безопасность: Инкапсуляция данных защищает их от несанкционированного доступа и изменений.

2. Напишите пример прототипа, реализации и вызова функции с переменным количеством аргументов.

Функции с переменным количеством аргументов могут принимать произвольное число параметров различных типов. Пример на языке C++ с использованием библиотеки <cstdarg>:

#include <iostream>

#include <cstdarg>

using namespace std;

// Прототип функции

void printValues(const char\* format, ...) {

va\_list args; // Структура для хранения списка аргументов

va\_start(args, format); // Инициализация списка

while (\*format != '\0') {

if (\*format == 'd') {

int i = va\_arg(args, int); // Получаем целое число

cout << "int: " << i << endl;

} else if (\*format == 'f') {

double d = va\_arg(args, double); // Получаем число с плавающей точкой

cout << "float: " << d << endl;

} else if (\*format == 's') {

char\* s = va\_arg(args, char\*); // Получаем строку

cout << "string: " << s << endl;

}

++format;

}

va\_end(args); // Завершаем работу с аргументами

}

int main() {

// Вызов функции с переменным числом аргументов

printValues("dffs", 10, 3.14, 7.89, "Hello");

return 0;

}

3. Напишите пример создания и использования перечисления.

Перечисление (enum) в C++ позволяет задавать набор символических имен для числовых констант:

#include <iostream>

using namespace std;

// Создание перечисления

enum Days { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };

int main() {

Days today = MONDAY;

// Использование перечисления

if (today == MONDAY) {

cout << "Today is Monday" << endl;

}

return 0;

}

4. Почему в языке C++ не разрешены виртуальные конструкторы?

В C++ конструкторы не могут быть виртуальными, потому что процесс создания объекта требует, чтобы тип создаваемого объекта был известен компилятору на этапе компиляции. Конструктор отвечает за выделение памяти для объекта, и для этого необходимо знать его точный тип. Виртуальные функции, наоборот, требуют динамического определения типа объекта на этапе выполнения, когда объект уже создан. Поскольку объект еще не создан, компилятор не может определить, какой конструктор нужно вызвать, что делает виртуальные конструкторы невозможными.

5. Для чего предназначен паттерн Фабрика?

Паттерн Фабрика (Factory) относится к порождающим паттернам и используется для создания объектов. Он предназначен для делегирования решения о том, какой объект создавать, подклассам. Это позволяет избежать жесткого связывания кода с конкретными классами и облегчает расширение программы новыми типами объектов. Фабрика инкапсулирует логику создания объектов, делая процесс более гибким и контролируемым.

6. Как паттерн Фабрика используется для создания эффекта виртуального конструктора?

Паттерн Фабрика помогает реализовать эффект виртуального конструктора, который сам по себе невозможен в C++. Виртуальный конструктор необходим, когда тип объекта известен только во время выполнения. Фабричный метод решает эту задачу, принимая решение о том, какой объект создавать на основании переданных параметров:

Пример:

class Base {

public:

virtual ~Base() {}

};

class Derived1 : public Base {};

class Derived2 : public Base {};

Base\* createObject(int type) {

if (type == 1)

return new Derived1();

else if (type == 2)

return new Derived2();

else

return nullptr;

}

7. Как добиться эффекта виртуального копирующего конструктора?

Для создания эффекта виртуального копирующего конструктора используется паттерн Прототип. Он позволяет клонировать объект, даже если тип объекта неизвестен на этапе компиляции. Каждый класс реализует метод clone(), который создает копию объекта:

Пример:

class Base {

public:

virtual Base\* clone() const = 0;

};

class Derived : public Base {

public:

Derived\* clone() const override { return new Derived(\*this); }

};

int main() {

Base\* obj = new Derived();

Base\* copy = obj->clone(); // Создание копии объекта

return 0;

}

8. Как паттерны Шаблонный метод и Фабрика используются совместно?

Паттерн Шаблонный метод определяет общий алгоритм, но позволяет подклассам реализовать отдельные его шаги. Этот паттерн может использоваться совместно с паттерном Фабрика, когда базовый класс определяет общий процесс создания объектов, а подклассы обеспечивают конкретную реализацию.

Пример:

class Building {

public:

virtual Unit\* makeUnit() const = 0; // Шаблонный метод для создания сущностей

};

class Castle : public Building {

public:

Unit\* makeUnit() const override { return new Knight(); } // Конкретная реализация фабрики

};

9. Что такое константа времени компиляции?

Константа времени компиляции (constexpr) — это значение, которое может быть вычислено на этапе компиляции. Такие константы позволяют компилятору выполнять вычисления до запуска программы, что повышает производительность. В C++ константы времени компиляции объявляются с помощью ключевого слова constexpr.

Пример:

constexpr int square(int x) {

return x \* x;

}

int main() {

constexpr int result = square(5); // result вычисляется на этапе компиляции

return 0;

}

10. Как можно узнать имя типа?

Имя типа объекта можно узнать с помощью оператора typeid и функции std::type\_info::name() из библиотеки <typeinfo>. Пример:

#include <iostream>

#include <typeinfo>

using namespace std;

class Base {};

class Derived : public Base {};

int main() {

Derived d;

cout << "Type of object: " << typeid(d).name() << endl;

return 0;

}

Этот код выведет искаженное имя типа объекта, но его можно использовать для отладки и анализа типов объектов во время выполнения программы.

**7 Выводы по лабораторной работе**

Вывод: я изучил особенности использования ООПиП и паттернов проектирования.