**УДК 004.4**

**МЕТОДИКА СОЗДАНИЯ И ПРИМЕНЕНИЯ МАССИВА КЛАССОВ В ОБЬЕКТНО-ОРИЕНТИРОВАННОМ ПРОГРАММИРОВАНИИ**

**Молодых Никита Андреевич,**

*студент гр. РИС-23-1б,*

*e-mail:* [*nikitamolodyh520@gmail.com*](mailto:nikitamolodyh520@gmail.com)

*1Пермский национальный исследовательский политехнический университет, г. Пермь, Россия*

***Аннотация***

*В статье рассматривается методика создания и применения массива классов в объектно-ориентированном программировании (ООП), которая представляет собой один из ключевых аспектов разработки эффективных и масштабируемых программных решений. В рамках работы анализируются основные этапы проектирования классов, включая выбор атрибутов, методов и конструкторов, а также их взаимоотношения, такие как наследование и полиморфизм. Особое внимание уделено вопросам использования массивов и коллекций объектов, позволяющих управлять множеством экземпляров различных классов. Рассмотрены примеры реализации массивов классов с полиморфизмом, а также преимущества и ограничения при их применении. Описываются методы работы с контейнерами стандартной библиотеки (например, vector и list в C++). В заключение подчеркивается важность соблюдения принципов ООП для создания гибких и поддерживаемых программных систем. Статья адресована разработчикам, исследователям и студентам, занимающимся вопросами объектно-ориентированного программирования и проектирования сложных программных решений.*

**Ключевые слова:**  объектно-ориентированное программирование (ООП) ,массив классов, создание классов ,применение классов ,программирование ,инкапсуляция ,наследование ,полиморфизм ,объекты и классы ,структуры данных

THE METHODOLOGY OF CREATING AND APPLYING AN ARRAY OF CLASSES IN OBJECT-ORIENTED PROGRAMMING

Nikita Andreevich Molodykh,

student gr. RIS-23-1b,

e-mail: nikitamolodyh520@gmail.com

1Perm National Research Polytechnic University, Perm, Russia

*The article discusses the methodology for creating and applying an array of classes in object-oriented programming (OOP), which is one of the key aspects of developing effective and scalable software solutions. The paper analyzes the main stages of class design, including the selection of attributes, methods and constructors, as well as their relationships, such as inheritance and polymorphism. Special attention is paid to the use of arrays and collections of objects that allow you to manage multiple instances of different classes. Examples of the implementation of arrays of classes with polymorphism are considered, as well as advantages and limitations in their application. The methods of working with standard library containers (for example, vector and list in C++) are described. In conclusion, the importance of adhering to the principles of OOP is emphasized in order to create flexible and supported software systems. The article is addressed to developers, researchers and students dealing with the issues of object-oriented programming and the design of complex software solutions.*

**Введение**

В последние десятилетия объектно-ориентированное программирование (ООП) стало основой большинства современных языков программирования, таких как Java, C++, Python и других. В современных системах программирования часто требуется эффективное управление большим количеством объектов, которые обладают схожей структурой или функциональностью. Одним из способов решения этой задачи является создание массива классов, который представляет собой коллекцию объектов одного или нескольких типов, взаимодействующих между собой. Однако, создание и управление массивами классов вызывает ряд проблем, связанных с производительностью, гибкостью, масштабируемостью и поддержкой кода.

Массив классов, как одна из форм представления множества объектов одного типа, играет важную роль в организации и хранении данных. В данной статье рассматриваются теоретические основы массивов классов, методика их создания, проектирование структуры, а также преимущества и недостатки их использования в различных сценариях.

Проблема заключается в создании эффективных механизмов для манипуляции и управления массивами классов, а также в обеспечении их оптимизации для работы с большими объемами данных.

**Основы объектно-ориентированного программирования**

Объектно-ориентированное программирование основывается на нескольких фундаментальных принципах, таких как инкапсуляция, наследование, полиморфизм и абстракция. Класс, как основной строительный блок ООП, объединяет данные и методы, работающие с этими данными, в единое целое. Это позволяет создавать структуры, которые инкапсулируют логику работы с данными и обеспечивают их взаимодействие.

Инкапсуляция: механизм, который ограничивает доступ к внутренним данным объекта и позволяет изменять его состояние только через определённые методы (обычно называемые геттерами и сеттерами). Это помогает скрыть детали реализации и защищает данные от случайных изменений, делая интерфейс работы с объектом более контролируемым.

Наследование: механизм, позволяющий создавать новый класс на основе существующего, при этом новый класс (потомок) наследует свойства и методы родительского класса. Это позволяет повторно использовать код и создавать иерархии классов.

Полиморфизм: Полиморфизм позволяет использовать объекты разных типов через одинаковый интерфейс. Это значит, что один и тот же метод может работать с объектами разных классов, а поведение этого метода будет зависеть от того, с каким объектом он взаимодействует.

Абстракция: Абстракция заключается в скрытии сложных деталей реализации и предоставлении только необходимой информации. Это позволяет пользователям работать с объектами на более высоком уровне, не вникая в детали их реализации.

Абстракция обычно достигается через абстрактные классы или интерфейсы, которые задают только необходимые методы, оставляя их реализацию на усмотрение подклассов.

**Понятие класса и массива классов**

Класс в ООП — это абстракция, описывающая свойства и поведение объектов. Массив классов представляет собой коллекцию объектов, каждый из которых является экземпляром того или иного класса. Использование массива классов позволяет организовать эффективное хранение и доступ к множеству объектов, при этом гарантируя удобство работы с ними через индексы или другие способы организации.

**Методика создания массива классов**

**Процесс проектирования массива классов**

Процесс создания массива классов начинается с определения структуры данных, которую будет содержать массив, а также спецификации методов, доступных для этих данных. Сначала необходимо определить тип объектов, который будет храниться в массиве. Затем следует выбрать структуру данных, которая будет использоваться для хранения этих объектов. Это может быть стандартный массив, динамический массив или структура данных, поддерживающая дополнительные операции, такие как вставка и удаление элементов.

**Инициализация и заполнение массива**

Инициализация массива классов включает создание объектов каждого типа, которые должны быть добавлены в массив. Это может происходить статически (при определении массива) или динамически (в процессе работы программы). Пример кода на языке C++:

**Проектирование структуры массива классов**

Проектирование структуры массива классов связано с выбором типа коллекции данных, а также решением, как будет происходить доступ к элементам массива. Это может быть обычный статический массив, динамический массив или более сложные структуры данных, такие как списки, очереди или стеки. Проектирование структуры должно учитывать:

1. Размер массива - фиксированный или изменяемый.

2. Способ доступа - индексация, итераторы или другие методы.

3. Алгоритмы работы с данными - сортировка, фильтрация, поиск.

**Оптимизация работы с массивом**

При работе с массивом классов важно учитывать, что производительность может значительно зависеть от размера массива и используемой структуры данных. Для оптимизации работы с массивами классов можно использовать различные техники, такие как:

1. Использование динамических структур данных для уменьшения накладных расходов при изменении размера массива.

2. Оптимизация доступа с помощью индексов или хэш-таблиц.

3. Уменьшение копирования данных - использование ссылок или указателей вместо копирования объектов.

**Преимущества и недостатки классов**

Преимущества:

1. Упрощение управления данными - массив классов позволяет хранить множество объектов в одном месте, что облегчает доступ и управление ими.

2. Гибкость - возможность использования различных методов для работы с данными, таких как сортировка, поиск, фильтрация.

3. Модульность - классы могут быть использованы в различных частях программы, что улучшает читаемость и поддержку кода.

Недостатки:

1. Производственные затраты - использование динамических массивов или сложных структур данных может потребовать дополнительных вычислительных ресурсов.

2. Затраты памяти - хранение множества объектов может значительно увеличивать потребление памяти.

3. Сложности с управлением памятью - особенно при использовании указателей и динамического выделения памяти.

**Создание массивов**

**Массив объектов в C++**

Подход 1: Использование стандартных массивов

Стандартные массивы (T[]) могут использоваться для хранения объектов класса. Однако их размер фиксирован на этапе компиляции, что делает их менее гибкими.

Пример:

class Rectangle {

public:

int width, height;

Rectangle(int w, int h) : width(w), height(h) {}

int getArea() { return width \* height; }

};

int main() {

Rectangle rectangles[3] = {Rectangle(4, 5), Rectangle(6, 7), Rectangle(8, 9)};

for (int i = 0; i < 3; ++i) {

std::cout << "Area: " << rectangles[i].getArea() << std::endl;

}

return 0;

}

Подход 2: Использование vector

Контейнер vector из стандартной библиотеки C++ предоставляет динамические массивы, которые могут изменять размер во время выполнения программы.

Пример:

#include <iostream>

#include <vector>

class Rectangle {

public:

int width, height;

Rectangle(int w, int h) : width(w), height(h) {}

int getArea() { return width \* height; }

};

int main() {

std::vector<Rectangle> rectangles = {Rectangle(4, 5), Rectangle(6, 7), Rectangle(8, 9)};

for (const auto& rect : rectangles) {

std::cout << "Area: " << rect.getArea() << std::endl;

}

return 0;

}

Преимущества vector:

* Возможность добавления и удаления элементов в массиве.
* Автоматическое управление памятью.

**Массив объектов в Python**

Python предоставляет встроенные динамические списки (list), которые могут содержать объекты любого типа, включая экземпляры классов.

Подход 1:Создание массива классов в Python

Пример:

class Rectangle:

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def get\_area(self):

return self.width \* self.height

# Создание массива объектов

rectangles = [Rectangle(4, 5), Rectangle(6, 7), Rectangle(8, 9)]

for rect in rectangles:

print(f"Area: {rect.get\_area()}")

Списки Python позволяют легко добавлять или удалять объекты:

rectangles.append(Rectangle(10, 12))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | Характеристика | | |  | | --- | | C++ (vector) | | |  | | --- | | Python (list) | |
| Гибкость размера | Требует контейнеров типа vector | Встроенная динамическая структура |
| Типизация | Статическая | |  | | --- | |  |  |  | | --- | | Динамическая | |
| Простата | Требует явного управления памятью | Простое создание и управление |
| Производительность | Высокая, подходит для оптимизации | Умеренная, подходит для быстрого прототипирования |

**Экспериментальное исследование**

Для сравнения возьмем два языка Python и C++.

Эксперимент 1: Создание и вывод массива объектов

В данном эксперименте так же замеряется время потраченное на создание и вывод данного массива. В С++ использовался массив *VECTOR*,в Python динамический

Цель: Проверить базовые возможности создания массива объектов в обоих языках и их вывод и сравнить их в таблица 1.

Ожидаемый результат:

- Оба кода должны корректно создать массив объектов (массив прямоугольников) и вывести площади каждого объекта.

|  |  |  |
| --- | --- | --- |
| Таблица 1-эксперемент 1 |  |  |
|  | С++ | Python |
| Использование памяти | Эффективное управление памятью, благодаря статической типизации | Более высокая нагрузка на память из-за динамической типизации |
| Время выполнения | 35-50 мс | 200-300 мс |
| Гибкость массива | Статическая, требует определения типов данных | Динамическая, облегчает создание объектов |

Эксперимент 2: Изменение свойств объектов в массиве

Цель: Проверить, как изменения в элементах массива объектов влияют на состояние самих объектов в Python и C++ и сравнить их в таблица 2.

Ожидаемый результат:

- Радиус объектов должен измениться на 1, и его площадь должна измениться в соответствии с новым значением.

|  |  |  |
| --- | --- | --- |
| Таблица 2-эксперемент 2 |  |  |
|  | С++ | Python |
| Использование памяти | Более эффективное управление памятью | Более высокая нагрузка на память из-за динамической природы |
| Время вывода | 249 мс | 20 мс |
| Время выполнения | 16-22 мс | 160-220 мс |
| Гибкость массива | Статическая, требует определения типов данных | Динамическая, облегчает создание объектов |

Эксперимент 3: Производительность при большом количестве объектов

Цель: Оценить производительность при работе с массивом объектов с большим количеством элементов (например, 100 000 объектов) и сравнить их в таблица 3.

Ожидаемый результат:

- В обоих случаях программа должна корректно создать массив из 100 000 объектов и вычислить суммарную площадь всех объектов. Также будет выведено время выполнения операции.

|  |  |  |
| --- | --- | --- |
| Таблица 3-эксперемент 3 |  |  |
|  | С++ | Python |
| Использование памяти | Более эффективное управление памятью за счет статической типизации и vector. | Более высокая нагрузка на память из-за динамической типизации |
| Время выполнения | 800-1200 мкс. Вычисления оптимизированы за счет статической типизации. | 2500-4000 мкс. Интерпретатор Python снижает |
| Гибкость массива | Статическая, требует определения типов данных | Динамическая, облегчает создание объектов |

Эксперимент 4: Инкапсуляция данных

Цель: Проверить использование инкапсуляции для работы с данными внутри классов и доступ к этим данным через методы и сравнить их в таблица 4.

Ожидаемый результат:

- Программа должна правильно инкапсулировать данные (имя и возраст), а вывод должен показывать правильную информацию о каждом объекте.

|  |  |  |
| --- | --- | --- |
| Таблица 4-эксперемент 4 |  |  |
|  | С++ | Python |
| Использование памяти | Эффективное управление памятью, благодаря статической типизации | Более высокая нагрузка на память из-за динамической типизации |
| Время вывода | 1051мс | 90мс |
| Время выполнения | 35-50 мс | 200-300 мс |
| Гибкость массива | Статическая, требует определения типов данных | Динамическая, облегчает создание объектов |

Экспериментальные тесты показали, что в обоих языках Python и C++ можно эффективно использовать массивы объектов для различных задач. В Python работа с массивами объектов упрощена благодаря динамическому типизированию и встроенным функциям, тогда как в C++ требуется больше внимания к производительности и памяти, но язык предлагает больше возможностей для оптимизации.

**Заключение**

Массивы классов являются мощным инструментом для работы с большими объемами данных в объектно-ориентированном программировании. Их использование позволяет улучшить организацию данных, облегчить управление объектами и повысить гибкость программ. Однако важно учитывать как преимущества, так и потенциальные недостатки, такие как затраты на память и производительность при работе с динамическими структурами данных.

**Список используемой литературы**

1. Б.Ш. Геккель, «Объектно-ориентированное программирование», М.: Высшая школа, 2009.

2. Л. Бауэр, «C++: Полное руководство», М.: Вильямс, 2013.

3. Д. Кнут, «Искусство программирования», том 1, М.: Наука, 1978.

4. К. Ларсен, «Python для профессионалов», СПб.: БХВ-Петербург, 2021.

.